

Design and evaluation of software framework that improves the management of reactive dependencies in development of object-oriented applications

Mijač, Marko

Doctoral thesis / Disertacija

2021

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **University of Zagreb, Faculty of Organization and Informatics / Sveučilište u Zagrebu, Fakultet organizacije i informatike**

Permanent link / Trajna poveznica: <https://urn.nsk.hr/urn:nbn:hr:211:897057>

Rights / Prava: [In copyright](#) / [Zaštićeno autorskim pravom](#).

Download date / Datum preuzimanja: **2024-11-16**



Repository / Repozitorij:

[Faculty of Organization and Informatics - Digital Repository](#)





University of Zagreb

FACULTY OF ORGANIZATION AND INFORMATICS

Marko Mijač

**DESIGN AND EVALUATION OF SOFTWARE
FRAMEWORK THAT IMPROVES THE
MANAGEMENT OF REACTIVE
DEPENDENCIES IN DEVELOPMENT OF
OBJECT-ORIENTED APPLICATIONS**

DOCTORAL THESIS

Varaždin, 2021.



Sveučilište u Zagrebu

FAKULTET ORGANIZACIJE I INFORMATIKE

Marko Mijač

**OBLIKOVANJE I EVALUACIJA
SOFTVERSKOG OKVIRA ZA
UNAPRJEĐENJE UPRAVLJANJA
REAKTIVNIM OVISNOSTIMA U RAZVOJU
OBJEKTO-ORIJENTIRANIH APLIKACIJA**

DOKTORSKI RAD

Varaždin, 2021.



University of Zagreb

FACULTY OF ORGANIZATION AND INFORMATICS

Marko Mijač

**DESIGN AND EVALUATION OF SOFTWARE
FRAMEWORK THAT IMPROVES THE
MANAGEMENT OF REACTIVE
DEPENDENCIES IN DEVELOPMENT OF
OBJECT-ORIENTED APPLICATIONS**

DOCTORAL THESIS

Supervisors:

Full Prof. Vjeran Strahonja, PhD

Assoc. Prof. Antonio García-Cabot, PhD

Varaždin, 2021.

ACKNOWLEDGMENTS

While I was writing this thesis, I often tried to imagine what would it be like to write the final sentence. What would I feel at that, very distant and seemingly out-of-reach moment? Joy? Thrill? Satisfaction? Pride? Well, it turns out that for me, the correct answer was relief and gratitude. Before I surrender to piled up and overdue obligations that are common side effect of the process of writing doctoral thesis, this whole experience deserves a bit of reflection, and gratitude above all. I owe it to the people who are in my life, I owe it to myself.

Looking back, the process of writing this thesis was one of those self-discovering journeys, in which person really becomes aware of his strengths, but even more his weaknesses and limitations. Battling the constant uncertainty and the barrage of issues that kept coming until the very end of writing thesis, has really put to test both my abilities and my character. Being able to cope with that required strong motivation from my part. Luckily, I had the best motivation one could possible have - a family. I would like to start expressing my gratitude by first thanking my beloved wife Silvija, who put her own ambitions to hold in order to make us, and keep us a family. I am also thankful for my children Klara, Jakov and Rafael, who were the cutest distractions from my daily work and worries. My deepest thanks go to my parents, whose sacrifices, support and unconditional love allowed me to become who I am today, both personally and professionally. Thanks also to my brothers, to whom I knew I could always count on.

No less gratitude I owe to many people who had important roles in shaping my thesis and my career in general. First of all, great thanks to my mentors prof. Strahonja and prof. García-Cabot for being patient and believing in me. Thank you also goes to my coworkers and friends, especially Zlatko and Boris, who often sought to relieve me of a lot of our duties and give me enough time to finish my thesis. Finally, to all focus group and technical action research participants, who invested their personal time and expertise to help me evaluate results of my thesis - Thank you, I greatly appreciate it.

ABSTRACT

In object-oriented (OO) applications objects collaborate through message passing, which results in these objects being coupled and mutually dependent. These dependencies can be of a reactive nature, i.e. such that, for example, change in state of one object, requires other, (directly or indirectly) dependent objects to update their state. The examples of such reactive dependencies can be found in various software systems, including rich graphical interfaces, spreadsheet systems, animation, robotics etc. Unlike the reactive paradigm, OO paradigm lacks abstractions and mechanisms for management of reactive dependencies. Instead, developers are left to handle this manually, e.g. by implementing Observer or similar design patterns. However, for large and complex dependency graphs it takes huge effort to avoid errors, redundancies, and to understand and maintain dependency graphs.

In this dissertation we address the problem of managing reactive dependencies in OO applications, by introducing reactive capabilities into OO programming languages in a form of software framework. We do this by conducting pragmatic, problem solving research called design science, which advocates building useful artifacts and applying them to problem context. The artifacts we propose and build are the *model* and *instantiation* of REFRAME software framework. All efforts and activities directed at building these artifacts are conducted and documented with necessary scientific rigor. In first process activity, by analyzing literature and utilizing personal experience, we explicate the stated problem and challenges it brings, and also reflect on its causes and effects. Through literature review we were also able to legitimize software frameworks as potential solutions for the problem. The second activity started with outlining the basic characteristics of REFRAME software framework, and then proceeded to create Software Requirements Specification document with detailed requirements for the framework. In accordance to these requirements, aforementioned *model* and *instantiation* of REFRAME software framework were designed and implemented. Through evaluation activities that followed we provided evidence of software framework's worth in the context of managing reactive dependencies. First, by building and testing REFRAME software framework we showed that such an artifact is technically feasible, that it works under certain assumptions, and that it has potential to solve or mitigate the stated problem. Demonstration on several illustrative scenarios provided further support for this. Finally, focus group and technical action research showed that potential users perceive REFRAME software framework as useful artifact.

PROŠIRENI SAŽETAK

Objektno-orijentirana (OO) paradigma je dominantna paradigma za modeliranje i razvoj velikih i kompleksnih softverskih sustava. Kao što njeno ime implicira, centralni pojam ove paradigme je "objekt" - entitet koji učahuruje podatke (atribute, svojstva) i pridruženo ponašanje (metode) koncepata problemske domene. Objekti u OO aplikaciji međusobno surađuju kako bi zajednički ostvarili cilj aplikacije, i na taj način postaju ovisni jedni o drugima. Ovisnosti unutar i između objekata mogu biti okarakterizirane kao *reaktivne*, tj. takve da, na primjer, promjena stanja jednog objekta zahtjeva ažuriranje stanja ostalih (izravno ili neizravno) ovisnih objekata. Drugim riječima, ovisni objekti bi trebali moći prepoznati da se nešto bitno za njih dogodilo, i reagirati na odgovarajući način. Primjere takvih reaktivnih ovisnosti možemo naći u različitim softverskim sustavima, kao što su sustavi sa bogatim grafičkim sučeljem, tablični kalkulatori, animacije, sustavi za modeliranje, razvojna okruženja i sl.

Objektno-orijentirana paradigma izvorno ne sadrži apstrakcije za upravljanje reaktivnim ovisnostima. Stoga se kao najčešće rješenje koristi Observer - jedan od najpoznatijih uzoraka dizajna u objektno-orijentiranom programiranju. Njegova svrha *"Definirati jedan-prema-više ovisnost između objekata tako da kada jedan objekt promijeni stanje svi ovisni objekti budu o tome obaviješteni i automatski ažurirani"* dobro odgovara ideji reaktivnih ovisnosti. Međutim upotreba ovog uzorka je izvorno zamišljena za odvajanje i sinkroniziranje podataka i grafičkog sučelja u MVC (engl. Model-View-Controller) arhitekturi. U slučaju kada reaktivne ovisnosti tvore velike i složene grafove ovisnosti, ovaj pristup zahtjeva ogroman trud da bi se izbjegle greške i redundancije, te da bi se razumjele i održavale reaktivne ovisnosti. Reaktivno programiranje s druge strane izvorno sadrži apstrakcije za upravljanje reaktivnim ovisnostima u obliku signala i događaja, međutim predstavlja jednu sasvim drugu paradigmu, različitu od OO programiranja. Stoga je jedan od glavnih pravaca istraživanja u ovom području pomirenje reaktivne i OO paradigme.

Ovo istraživanje je motivirano praktičnim problemom upravljanja reaktivnih ovisnosti u OO aplikacijama, koje se još uvijek dominantno odvija neadekvatnim pristupima, uzrokujući brojne probleme u kvaliteti, performansama i održavanju takvih aplikacija. Uz to, i pregled znanstvene literature ukazuje na prisutnost tog problema, kao i na nastojanja znanstvenika da ponude različite pristupe rješavanju tog problema. Slijedeći trendove istraživačke zajednice, ali i potreba iz prakse, ovo istraživanje također ima za cilj približavanje reaktivne i OO paradigme,

i to na način da se određeni koncepti i ideje iz reaktivne paradigme ali i iz drugih relevantnih područja kao što je AOP, ugrade u OO aplikacije i unaprijede rad sa reaktivnim ovisnostima. Stoga je formalni cilj ovog istraživanja: *Unaprijediti i olakšati upravljanje reaktivnim ovisnostima u objektno-orijentiranim aplikacijama oblikovanjem i evaluacijom softverskog okvira REFRAME, koji će omogućiti specificiranje, propagaciju, vizualizaciju i analizu reaktivnih ovisnosti.* U svrhu ostvarenja ovog cilja, postavljeno je glavno istraživačko pitanje (MRQ), 6 sporednih istraživačkih pitanja (RQ1 do RQ6), te jedna hipoteza (H1).

U ovoj disertaciji se bavimo problemom upravljanja reaktivnim ovisnostima u OO aplikacijama na način da uvodimo reaktivne mogućnosti u OO programski jezik u obliku softverskog okvira. Pri tome se vodimo pragmatičnom, na rješenje usmjerenom istraživačkom paradigmom zvanom znanstveno oblikovanje (engl. design science). Za nju je karakteristična izrada korisnih artefakata i njihova primjena u problemskoj domeni. Kao što je navedeno u samom cilju, u okviru ove disertacije mi predložimo i izgrađujemo model i instancu softverskog okvira REFRAME za upravljanje reaktivnim ovisnostima.

Sam proces znanstvenog oblikovanje je proveden kroz 5 aktivnosti odabranog metodološkog okvira. Prva aktivnost, je podrazumijevala provođenje detaljne analize relevantnih istraživanja, što je uz iskustvo samog istraživača rezultiralo **razjašnjavanjem problema**, uključujući prepoznavanje izazova koje problem sa sobom nosi, uzroka problema i njegovih posljedica. Navedeno, opisano prvenstveno u poglavljima 2 i 4, nam je omogućilo bolje razumijevanje prostora problema, te potvrdu praktične i istraživačke relevantnosti. Uz to, pregled literature na temu softverskih okvira nam je omogućio potvrdu softverskog okvira kao valjanog kandidata za rješenje, te sukladno tome početak tranzicije prema prostoru rješenja. Kroz prvu aktivnost je i odgovoreno na istraživačka pitanja RQ1 i RQ2. Nakon što smo analizirali problem, te identificirali potencijalno rješenje, mogli smo započeti sa aktivnošću **definiranja zahtjeva**. Na početku poglavlja 5 ocrnali smo osnovne budućeg softverskog okvira REFRAME, te smo osmislili 5 zahtjeva više razine. Ti zahtjevi su zatim detaljizirani u obliku dokumenta Specifikacija zahtjeva za softver (SRS), koji je sadržavao ukupno 34 funkcionalna zahtjeva i 4 nefunkcionalna zahtjeva. Na taj način smo izravno odgovorili na istraživačko pitanje RQ3. Na temelju zahtjeva iz SRS dokumenta, u sklopu treće aktivnosti **oblikovani i izrađeni** su već spomenuti *model* i *instanca* softverskog okvira REFRAME. U poglavlju 6 je detaljno opisana ova visoko-iterativna aktivnosti, koja je uključivala korake prikupljanja ideja, isprobavanje i oc-

jenu tih ideja, i na kraju odabir i implementaciju ideja koju su ocijenjene kao prikladne. Dok je model softverskog okvira dokumentiran u poglavlju 6, implementacija softverskog okvira je dostupna na službenom GitHub repozitoriju (<https://github.com/MarkoMijac/REFRAME.git>). Dokumentiranjem procesa izrade, ali i karakteristika samog softverskog okvira, pružili smo vrijedno znanje i iskustvo, i na taj način doprinijeli odgovoru na istraživačko pitanje RQ4.

Znanstvena **evaluacija** (aktivnosti 4 i 5) je provedena u obliku 4 međusobno komplementarne evaluacijske epizode. U prvoj epizodi, kroz prototipiranje i obilno testiranje (oko 1000 jediničnih testova), ponudili smo dokaz da je softverski okvir REFRAME tehnički izvediv (engl. technical feasibility), i da predstavlja efikasno rješenje za upravljanje reaktivnim ovisnostima (engl. efficacy). Na taj način je prva epizoda doprinijela odgovoru na istraživačko pitanje RQ4. Tehnička izvedivost i efikasnost je dodatno potvrđena demonstriranjem uporabe softverskog okvira na 15 ilustrativnih scenarija. I ova epizoda je također doprinijela odgovoru na istraživačko pitanje RQ4, ali je uz to i u potpunosti ponudila odgovor na pitanje RQ5. Kako bismo ponudili dokaz da je izrađeni softverski okvir koristan u realnom uvjetima, u sklopu evaluacijskih epizoda III i IV, provedeno je istraživanje fokus grupom, te tehničko akcijskog istraživanje. Sudionici obje epizode su bili jasni u ocjeni REFRAME-a kao korisnog za upravljanje reaktivnim ovisnostima, što nam je omogućilo da odgovorimo i na posljednje sporedno istraživačko pitanje (RQ6).

Konačno, odgovaranjem na 6 sporednih istraživačkih pitanja dovelo nas je u poziciju da ponudimo sljedeći odgovor na glavno istraživačko pitanje: Upravljanje reaktivnim ovisnostima u razvoju objektno orijentiranih aplikacija možemo unaprijediti oblikovanjem i implementacijom softverskog okvira, koji nudi dedicerane apstrakcije i mehanizme za specificiranje pojedinačnih reaktivnih ovisnosti, izradu grafa ovisnosti, te provođenje procesa ažuriranja grafa. Također, mogućnosti takvog softverskog okvir se mogu obogatiti izradom pratećih alata, koji bi omogućili analizu i vizualizaciju grafova ovisnosti, te generiranje dijelova ponavljajućeg koda. S obzirom da smo u disertaciji potvrdili praktičnu i znanstvenu relevantnost postavljenog problema i izrađenog rješenja, te da smo opisani proces proveli rigorozno i uz uporabu znanstvenih metoda, možemo i potvrditi postavljenu hipotezu H1: *Oblikovan i implementiran softverski okvir (REFRAME) za upravljanje reaktivnim ovisnostima u razvoju objektno-orijentiranih aplikacija će ispuniti zahtjeve relevantnosti i rigoroznosti znanstvenog oblikovanja.*

KEYWORDS

Reactive dependencies, Reactive programming, Object-oriented programming, Software frameworks, Design science

CONTENTS

Acknowledgments	
Contents	V
List of Figures	VIII
List of Tables	XI
1. Introduction	1
1.1. Problem identification	1
1.2. Research motivation and contribution	4
1.3. Research goal and questions	5
1.4. Methodology overview	6
1.5. Dissertation structure	7
2. Literature review	8
2.1. Managing reactive dependencies	8
2.1.1. Reactive programming	8
2.1.2. Event-based programming	14
2.1.3. Other approaches	19
2.2. Software frameworks	22
2.2.1. Software frameworks as a reuse technique	22
2.2.2. Relation with other reuse techniques	25
2.2.3. Types of software frameworks	28
2.2.4. Software framework structure	30
2.2.5. Framework-involved processes	33
3. Method	42
3.1. Explicate problem	43
3.2. Define requirements	43
3.3. Design and develop artifact	44
3.4. Demonstrate and evaluate artifact	45

4. Explicate Problem	49
4.1. Define Precisely	49
4.2. Position and Justify	51
4.3. Find Root Causes	51
5. Define Requirements	55
5.1. Outline artifact	55
5.2. Elicit requirements (SRS)	57
5.2.1. Introduction	57
5.2.2. Overall Description	59
5.2.3. Specific Requirements	65
6. Design and develop artifact	78
6.1. Imagine and brainstorm	78
6.1.1. Feature 1: Specify reactive dependency	78
6.1.2. Feature 2: Construct dependency graph	84
6.1.3. Feature 3: Perform dependency graph update process	85
6.1.4. Feature 4: Analyze dependency graph	92
6.1.5. Feature 5: Visualize dependency graph	95
6.1.6. Feature 6: Generate boilerplate code	96
6.2. Assess and select	98
6.2.1. Feature 1: Specify reactive dependency	98
6.2.2. Feature 2: Construct dependency graph	102
6.2.3. Feature 3: Perform dependency graph update process	103
6.2.4. Feature 4: Analyze dependency graph	108
6.2.5. Feature 5: Visualize dependency graph	111
6.2.6. Feature 6: Generate boilerplate code	112
6.3. Sketch and build	114
6.3.1. ReframeCore	115
6.3.1.1. Reactive nodes	115
6.3.1.2. Reactive dependencies and dependency graphs	121
6.3.1.3. Dependency graph update process	123
6.3.1.4. Reactor - a higher level interface for ReframeCore	127
6.3.2. ReframeTools	130
6.3.2.1. Analyzer tool	130
6.3.2.2. Visualizer tool	141
6.3.2.3. Code generation	144
6.3.3. High-level design of REFRAME	147
6.4. Justify and reflect	152

6.4.1.	Underlying design principles	153
6.4.2.	Documenting design decisions	164
7.	Artifact Evaluation	176
7.1.	Episode I - Prototyping and testing	176
7.1.1.	Prototyping	176
7.1.2.	Testing	177
7.2.	Episode II - Demonstration	183
7.2.1.	Base demonstration example	184
7.2.2.	Illustrative scenarios	185
7.3.	Episode III - Focus group	198
7.3.1.	Research problem	198
7.3.2.	Sample frame	199
7.3.3.	Moderator	200
7.3.4.	Questioning route	201
7.3.5.	Focus group session	203
7.3.6.	Data analysis	205
7.3.7.	Results	207
7.4.	Episode IV - Technical action research	213
7.4.1.	Research context	214
7.4.2.	Research problem	214
7.4.3.	Research design and validation	215
7.4.4.	Research execution	221
7.4.5.	Data analysis	222
8.	Discussion	231
8.1.	Reflection on the research	231
8.2.	Answering research questions	232
8.3.	Contributions	238
8.4.	Research limitations	244
8.5.	Future research	245
9.	Conclusion	247
	Bibliography	252

LIST OF FIGURES

Number	Figure caption	Page
Figure 1.	Observer pattern (simple) [61]	17
Figure 2.	Observer pattern (advanced) [61]	17
Figure 3.	Observer pattern (revisited) [46]	18
Figure 4.	Event notification pattern [122]	18
Figure 5.	Propagator pattern [53]	19
Figure 6.	Framework vs other reuse techniques	27
Figure 7.	Framework metamodel (code elements), adapted from [118]	33
Figure 8.	Design science methodological framework	42
Figure 9.	Evaluation strategy	46
Figure 10.	Example of reactive dependencies	50
Figure 11.	Example of more complex dependency graph	50
Figure 12.	Problem tree	52
Figure 13.	Solution tree	54
Figure 14.	Framework perspective	60
Figure 15.	Use case diagram describing REFRAME's features	66
Figure 16.	Different DAG representations: a) adjacency matrix, b) incidence matrix and c) adjacency list	85
Figure 17.	Overall architecture of REFRAME	115
Figure 18.	Essential members of Node class	117
Figure 19.	Hierarchy of reactive nodes	119
Figure 20.	Essential members and associations of NodeFactory class	120
Figure 21.	Methods for manipulation of predecessors and successors	122
Figure 22.	Essential members of DependencyGraph class	123
Figure 23.	Essential members of Scheduler class	124
Figure 24.	Scheduling nodes for update	124

Figure 25.	Strategy pattern in the context of Sorter class	125
Figure 26.	Essential members and associations of Updater class	126
Figure 27.	Example of object interaction required for performing update	127
Figure 28.	Essential members and associations of Reactor class	129
Figure 29.	Example of object interaction required for creation of Reactor object . . .	130
Figure 30.	Client feature residing in REFRAME tools	132
Figure 31.	Server feature residing in end-user application	132
Figure 32.	Inter-process communication between Analyzer tool and end-user appli- cation	133
Figure 33.	AnalysisGraph structure and node hierarchy	134
Figure 34.	Analysis graph and analysis node factories	135
Figure 35.	Node abstraction levels in filtering by affiliation	136
Figure 36.	Filtering by affiliation design	137
Figure 37.	Filtering by role and association	138
Figure 38.	Update analysis classes	140
Figure 39.	Analyzer GUI and related classes	141
Figure 40.	Hierarchy of VisualGraph classes	142
Figure 41.	Most important interfaces and classes in Visualizer	143
Figure 42.	Fluent interface implementation in REFRAME	145
Figure 43.	Implementation of code snippet for Let->Depend statement	148
Figure 44.	End-user application perspective	149
Figure 45.	REFRAME Tools perspective	151
Figure 46.	Most important interfaces and classes in ReframeCore	154
Figure 47.	Scheduler design with regard to sorting algorithm: a) not conforming to OCP, b) conforming to OCP	158
Figure 48.	Core interfaces	162
Figure 49.	a) violating Dependency inversion principle, b) conforming to Depen- dency inversion principle	163
Figure 50.	Test Explorer panel in Visual Studio displaying written automated tests	179
Figure 51.	Simple unit test example	180
Figure 52.	Complex unit test example	182

Figure 53. Code coverage results	183
Figure 54. Class diagram of base demonstration example	184
Figure 55. Dependency graph of base demonstration example	185
Figure 56. Measurement scale for perceived usefulness (adapted from [41])	204

LIST OF TABLES

Number	Table caption	Page
Table 1.	Useful features of existing design patterns (adapted from Mijač et al. [107])	20
Table 2.	Software framework definitions	24
Table 3.	Framework classification [90]	28
Table 4.	Framework development activities as seen by various authors	34
Table 5.	Chosen evaluation properties	46
Table 6.	Roles of reactive nodes as represented by abstractions from relevant design patterns	82
Table 7.	Graph algorithms	94
Table 8.	Specialized graph-drawing solutions	96
Table 9.	Code generation techniques in .NET framework and Visual Studio IDE	97
Table 10.	Currently implemented pipe commands	131
Table 11.	Filtering by association options	138
Table 12.	REFRAME Components from the end-user application perspective	149
Table 13.	Components from REFRAME tools perspective	150
Table 14.	Summary of most important design decisions made in REFRAME development	165
Table 15.	Focus group participants	200
Table 16.	Focus group questions and topics	202
Table 17.	Final version of template	206
Table 18.	TAR client cycle participants	217
Table 19.	TAR interview questions	220
Table 20.	Final version of template	223

1. Introduction

The introductory chapter provides overall background for the conducted research. It starts by brief description of the research problem and the context in which the problem takes place. Then, the researcher's motivation is discussed, as well as the potential theoretical and practical contribution of the research. This is followed by formulation of research goals, questions and hypotheses, which consequentially defines the scope of the research. After that, overview of the research methodology is provided, and the structure of the dissertation presented.

1.1. Problem identification

Object-oriented (OO) paradigm is a dominant paradigm for modeling and development of large and complex software. It originated in 1960s as a response to former paradigm not being able to cope with ever increasing size and complexity of software systems. As the name implies, the central abstraction of this paradigm is "object", which encapsulates data (properties/attributes) and associated behavior (methods) of concepts in problem domain. Not only does such approach provide better correspondence to human perception of things in its environment, but it also raises the level of design and source code re-usability.

In OO application each object has its responsibility and is expected to contribute to overall goal and purpose of application. In this process objects rely on each other and collaborate. This collaboration usually comes in a form of message passing, i.e. invoking other object's behavior (methods) or fetching data of other objects. The result of this collaboration is objects being to greater or lesser extent coupled and dependent on each other.

Dependencies between objects formed in this manner can sometimes be characterized as *reactive*, i.e. such that for example method invoke or data change in one object, automatically triggers invoke of corresponding method or updates the data in all dependent objects. In other words, dependent objects will recognize the occurrence of something interesting happening (event), and they will react accordingly. The examples of such reactive dependencies can be found in a number of software systems, including rich graphical user interfaces, spreadsheet systems, animation, modeling, simulation, embedded systems, programming environments, etc. Perhaps the most prominent example cited in literature and the one that best suits to what we consider reactive dependencies to be is the mechanism of cells in spreadsheet [27].

Object-oriented paradigm lacks native support for managing reactive dependencies [95]. Instead, the well-known Observer [61] design pattern is mostly used as a solution. The stated intent and overall idea of Observer pattern is to *"Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically"* [61]. When looking at its base idea, Observer pattern may seem as a perfect match for managing reactive dependencies, however, in reality it has a number of shortcomings that made it heavily criticized as unfit for complex use (e.g. [95], [129]). Interesting statistics in line with these critics has also been offered by Parent [113] in his presentation at one of the official Adobe's conferences. He argued that a 30% of the code in Adobe's desktop applications is devoted to event handling logic, while at the same time 50% of all reported bugs are located in this very code. Even if we do not take these numbers as totally accurate, still at least two facts can be derived from them. Foremost, we can expect to write a significant amount of boilerplate code which will be in charge of handling dependencies between objects. This means a significant effort will be spent into developing support code, rather than focusing on application's core functionality. Also, disproportionately high number of bugs and errors in this part of the code indicates that managing reactive dependencies is quite difficult and error prone.

Indeed, manually managing large number of reactive dependencies between objects proves to be a challenging task, since dependencies may form quite large and complex dependency graphs, very hard to comprehend. A number of issues can arise here, such as failing to react and update dependent objects (updating too rarely), reacting and updating defensively and redundantly (updating too frequently), updating in wrong order and thus causing the incorrect states or temporary inconsistencies (so called glitches), inefficient propagation of updates, circular dependencies causing infinite loops etc. In addition, the adverse impact on overall quality of software can arise from too much coupling and pollution of business logic with the so called "boilerplate" code in charge of managing reactive dependencies between objects.

Observer design pattern is actually imperative implementation of the concept of implicit invocation [62]. Due to its limitations, other, *"more powerful"* design patterns, such as Observer revisited [46], Event-notification [122] and Propagator [53] pattern have been proposed to replace Observer pattern in more complex use cases. While object-oriented paradigm failed to provide native support for reactive dependencies, other paradigms tried to offer potential solutions. Aspect-oriented programming (AOP) [86], for instance, is focused on separating code in

charge of managing reactive dependencies from application's core functionalities. Proponents of AOP identified managing reactive dependencies as one of the crosscutting concerns, and proposed improved - "*aspectized*" versions of Observer pattern (e.g. [149]). Approaches based on constraint programming (e.g. [30]) specify reactive dependencies as constraints, which underlying constraint solver is supposed to maintain consistent. Reactive programming as approach dedicated for development of reactive systems has provided built-in abstractions and mechanisms to achieve automated management of reactive dependencies. Most of the research in this paradigm stems from representatives of functional-reactive languages for animation such as Fran [47], which is why reactive programming has strong functional/declarative flavor as opposed to imperative style of object-oriented programming. Recent trends, however, acknowledge advantages and disadvantages of both reactive and object-oriented programming, and aim at reconciling these two paradigms, trying to take the best from each of them. Boix et al [65] identify two distinct approaches for doing this, one of them taking reactive programming, and the other object-oriented programming as a base point. Their ROAM framework (experimental reactive object-oriented framework) implemented in AmbientTalk programming language is the example of the second approach. Conversely, REScala - reactive programming language proposed by Salvaneschi et al. [129] is the example of the first approach.

All aforementioned approaches offer certain improvement over the plain Observer design pattern, however, they also require some compromises to be made. The more sophisticated versions of Observer pattern collectively offer some good features and ideas, however, no single design pattern is suitable for handling complex use scenarios, as reported in comparison by Mijač et al. [107]. Proposed Aspect-oriented versions of Observer pattern offer improvements in reusability, understandability and maintainability of code by explicitly separating management of reactive dependencies as a crosscutting concern and core business logic. However, both *aspectized* and *non-aspectized* design patterns in general offer mainly design reuse and still place large burden on developer to implement reactive dependencies by following specified design guidelines. In addition, AOP solutions tend to introduce "magical" code difficult to see and debug. Reactive programming does offer built-in support for reactive dependencies, however most of solutions, because of their functional/declarative background, poses a challenge to OO developers, and may require quite a shift from traditional imperative way of programming. Also, rewriting already existing OO applications to another paradigm may prove to be too

difficult and time-consuming. Part of the solutions which are advertised as reactive, do not consider reactive dependencies in a way they are considered in this dissertation, but rather focus on temporal aspects, and see reactivity as working with continuous streams of data and signals. Other reactive solutions handle reactive dependencies by introducing frameworks which require special, adapted compilers. The advantages of such approach include more natural integration of reactive dependencies into host language, higher level of automation, and overall better performance. However, this usually comes at the price of being stuck with original author's implementation, often done using non-mainstream or experimental technology, with end solutions offering no or very limited possibility for adaptation. The authors of such solutions are often individuals or very small teams, so practitioners usually cannot expect too much help or customer support. We believe that for these reasons proposed reactive solutions find hard to get out of the academic, laboratory setting to practitioners and real software projects.

1.2. Research motivation and contribution

This research has been motivated by a practical problem of managing reactive dependencies in object-oriented applications. The author of dissertation has encountered this problem while for the past almost ten years being actively involved in several projects of developing commercial software applications. The most relevant example is software application "KI Expert Plus", which is used for calculating thermal properties of residential and non-residential buildings with the goal of assessing energy efficiency. The size and complexity of particular application domain dictated a large number of reactive dependencies between different data and calculation procedures within software application, thus, making it very difficult to develop, understand and maintain.

Preliminary literature review showed that this problem has also been recognized by others in software engineering practice, as well as in scientific community. One of the most obvious indicators for this is the fact that Observer patterns is perhaps the most well-known pattern to both practitioners and scientific community. The MVC (Model-View-Controller) and MVC inspired architectures, whose constituent part is Observer pattern, are de facto a standard for separating and synchronizing user interface and underlying data. Different paradigms, including event-driven programming, constraint programming and reactive programming, have addressed the issue of managing reactive dependencies by proposing solutions such as design patterns,

software frameworks, dedicated programming languages etc. Proposed solutions can be placed within the continuum, starting from more abstract, design solutions such as design patterns, software architectures and models, to more concrete solutions such as software frameworks and programming languages.

The contribution of the research in this dissertation encompasses both scientific and practical aspects. It is reflected in the fact that managing reactive dependencies in OO applications as a relevant, and non-trivial problem recognized in both scientific community and practice, will be addressed by building solution in a form of a model and instantiation of software framework. Such software framework will utilize existing design patterns, however, it will not offer only design guidelines, but also concrete implementation. It will not require compiler modifications, thus, it will be easier to customize and port to another technology. In addition, the software framework will provide useful tools for generation, visualization and analysis of reactive dependencies. The scientific contributions reflect in knowledge generation, at first in the very process of investigating required characteristics of future software framework, specifying requirements, prototyping, modeling and building the software framework. Then, when built, the software framework itself as a result of the number of design decisions inherently contains knowledge (embedded knowledge). Lastly, the process of successively applying software framework to problem context generates knowledge about the problem, solution and the relationships between them. The most obvious practical contribution can be seen in scenario where developers use the instantiation of software framework as is, and apply it in order to improve the management of reactive dependencies in their OO application. Other scenarios may include developers customizing the software framework to better suit their needs, or building their own software framework in their chosen technology according to the original or customized model.

1.3. Research goal and questions

The main goal of this dissertation is to: *Improve and facilitate the management of reactive dependencies in object-oriented applications by designing and evaluating REFRAME software framework, which will allow specification, propagation, visualization, and analysis of reactive dependencies.* In order to achieve this goal following questions are stated:

Main research question:

MRQ: How can we improve the management of reactive dependencies in the development of

object-oriented applications?

Research sub-questions:

- RQ 1: What makes the management of reactive dependencies in development of object-oriented applications a challenging task?
- RQ 2: What are the means we can use to support development of object-oriented applications in order to improve the management of reactive dependencies?
- RQ 3: What functional and non-functional requirements should REFRAME software framework meet in order to manage reactive dependencies?
- RQ 4: What prerequisites, constraints and other factors have to be met in order for REFRAME software framework to be designed and working?
- RQ 5: What are the typical scenarios of managing reactive dependencies that can be used to evaluate REFRAME software framework?
- RQ 6: How does the use of REFRAME software framework affect the management of reactive dependencies in development of object-oriented applications?

In addition, the following high-level hypothesis is stated:

Hypothesis:

- H 1: Designed and implemented software application framework (REFRAME) for management of reactive dependencies in development of object-oriented applications will fulfill both relevance and rigor requirements of design science.

1.4. Methodology overview

With the regard to research being motivated by a practical problem, and the nature of final result of the research (innovative artifact - model and instantiation of software framework), *design science* has been chosen as a base research paradigm. *Design* is a process of creating applicable solutions for specified problem, and it has long been accepted as research paradigm in engineering disciplines [116]. Recently, it is also increasingly being used in the field of information systems [73]. Design science is a pragmatic paradigm with the goal of solving real

problems by creating innovative artifacts. According to Hevner et al. [73] these artifacts can be characterized as: constructs, models, methods or instantiations. The scientific dimension of design science, other than creating artifacts, also requires generation of new knowledge through design and application of artifacts. Another determinant of design science is requirement for systematic approach and rigorous evaluation of the created artifacts. In order to guarantee that design science is conducted in scientifically rigorous way we will follow the methodological framework for design science research proposed by Johannesson and Perjons [78]. The framework proposes 5-activity process, with each of the activities using appropriate methods and techniques.

1.5. Dissertation structure

The remainder of the dissertation starts with the literature review (**Chapter two**) who's purpose is explicating the context around the research problem as well as providing ideas for a solution. This includes covering topics closely related to handling reactive dependencies, such as reactive systems, reactive programming, implicit invocation etc. The literature review also extensively covers software frameworks as an intended means of constructing the solution to posed problem.

In **Chapter three**, dissertation continues with the detailed description of its underlying research paradigm, the chosen methodological framework guiding the research, and the five specific steps through which the research will be conducted.

The main part of the dissertation can be found in chapters four to seven, which sum up the efforts of individual steps of the methodological framework. Thus, the **chapter four** explicates the problem, **chapter five** defines the requirements for the solution artifact, **chapter six** presents design and development activities, and finally demonstration and evaluation steps are jointly described in **chapter seven**.

The efforts and results of the dissertation are then discussed in **chapter eight** and conclusions offered in **chapter nine**.

2. Literature review

Literature review conducted in this dissertation has two main parts. The first part aims at identifying research which describes key concepts, approaches and paradigms related to managing reactive dependencies. This is, primarily, required for understanding the research problem itself and grounding our subsequent findings in existing knowledge base. However, it also provides us with useful information, ideas and proven practices for constructing the solution (e.g. domain vocabulary, solution requirements, design and implementation ideas etc.).

The second part of the literature review is more focused on a solution space. It deals with software frameworks - a type of software artifact and a reuse technique which we propose as a suitable solution for managing reactive dependencies. Since software frameworks are complex software systems, whose development differs from traditional software application development, a literature review has also been conducted in order to cover the most important aspects of software frameworks and their development.

2.1. Managing reactive dependencies

The concept of reactive dependencies, defined in previous section, has been recognized by both academy and industry, and is related to several fields of research. Reactive systems, reactive programming, event-driven programming, data-flow programming and constraint programming are examples of such related research fields. Not all of these fields, however, have the same motivation and goals, so they do not interpret the "reactivity" in a same way, nor do they put an emphasis on the same aspects of reactivity. Also, offered solutions differ in approach and scale, and take different forms, ranging from design patterns, user developed libraries and frameworks, specialized programming languages and language extensions.

2.1.1. Reactive programming

One natural starting point in researching reactive dependencies, at least from the perspective of nomenclature, was field of reactive systems. Most papers explicitly mentioning reactive systems adopt Harel and Pnueli's [70] notion of reactive system, who aimed at distinguishing easily-dealt-with systems from difficult systems, i.e. *transformational* from *reactive* systems respectively. According to them, a transformational system accepts inputs, performs required

transformations and produces outputs (e.g. compilers, assemblers, expert systems etc). On the other hand, reactive system is repeatedly prompted by environment, and its role is to continuously respond to external inputs. In other words, reactive system responds or reacts to external stimuli generated by the user, environment or simply time passing. As examples of reactive systems Wieringa [156] lists real-time systems, embedded systems, control systems, information systems, groupware systems, ERP systems, workflow management systems etc. It is evident that large number of today's software systems can fully or at least in some parts be recognized as being reactive.

Considering reactive systems are complex to develop and often have to manage critical tasks, it is no surprise that majority of papers explicitly mentioning reactive systems were dealing with the formal verification/testing (e.g. [40], [36], [121]) and security issues (e.g. [26]) of reactive systems. In parallel, a number of specification, modeling and visual techniques, tools and extensions were proposed to aid reactive system development (e.g. [70], [69], [32], [76], [160]). Although the research on reactive systems is well established, papers in this subsection did not entirely correspond to this dissertation's notion of reactive dependencies. The reactive dependencies are indeed part of reactive systems and can be related to the notions of external stimuli, signals, events and reactions, however, the focus of resulted papers was different. Most papers emphasized or exclusively considered only temporal aspects of reactive systems. On the other hand, reactive dependencies, as seen in this dissertation, are not taking time dimension into account, at least not in a sense of providing dedicated time abstractions.

A paradigm suited for development of different kinds of reactive systems, especially real-time, event-driven and interactive systems, is *reactive programming*. Bainomugisha et al. [27] in their extensive survey define it as a paradigm centered around the issues of *continuous time-varying values* and *propagation of change*. These issues are tackled by providing abstractions able to express and automate flow of time, as well as abstractions for expressing data and computational dependencies. Thus, they identify two distinguishing abstractions offered by reactive programming languages: *behaviors or signals* and *events*.

Behaviors or signals are abstractions which represent continuous time-varying values, i.e. the time itself, or some function of a time. This makes them targeting temporal aspects of reactive systems, which are not the subject and focus of this dissertation. Unlike behaviors, which change continuously over time, events are discrete values. They change at discrete points

in time, forming possibly infinite stream of value changes.

Events better relate to our concept of reactive dependencies, however, some distinctions should be made clear. One important aspect of event handling in reactive programming is handling event streams (data streams or streams of value changes). This includes performing operations such as filtering, selecting, combining and transforming streams. However, again, this is not the subject of this dissertation. Rather, what we tackle is the aforementioned issue of *propagation of change*. With this in mind, we see events as these very changes in program's state which cause other parts of program's state to update itself, and thus result in entire cascade of updates through out the software system. Having this in mind, reactive dependencies, as proposed in this dissertation, can be viewed as a mechanism for expressing and enforcing this causal relationship, i.e. the relationship between the part of the program's state which is being changed and its dependents. Therefore the topic and issue of reactive dependencies clearly positions itself within the field of reactive programming.

According to Bainomugisha et al. [27] today's applications are driven by all sorts of events, both internal (occurring within the application) and external (occurring in outside environment). The main task placed in front of application is to react to these events by updating program's state and displaying response to user. Unfortunately, manual management of state changes and data dependencies is complex and error-prone. This is especially true when this is done using traditional programming techniques such as design patterns, where programmer is required to manually update all dependent data after some change in state has occurred. Instead of that, reactive programming aims at embedding the spreadsheet-like model in programming languages, and automating the update of dependent data using the underlying execution model. The major issue here, according to Salvaneschi et al. [129], is to detect changes to input values (i.e. events) and then decide which cached values need to be invalidated or recomputed. The easiest approach would be to invalidate or recompute all cached values whenever any of the input values are changed. However, efficient but more complex solution would be to analyze actual dependencies between inputs and outputs, and after a change perform update of only those outputs that depend on initially changed value.

Before we dig into the solutions offered in the field of reactive programming, we can look at the simple illustrative example of differences between traditional imperative programming and reactive programming in terms of managing state changes.

Listing 2.1: Difference between imperative and reactive programming

```
a = 1;  
b = 2;  
c = a + b;           //Imperative: c = 3; Reactive: c = 3;  
a = 2;               //Imperative: c = 3; Reactive: c = 4;
```

Lets assume that the part of the program's state is defined and stored in variables a and b . The values of these variables can be changed (e.g. by the user). There is also variable c whose value should be calculated from the values of a and b . This exactly happens in the third line, i.e. after the variables a and b are set to their values, c is assigned with their sum. As one might guess, after executing this line of code, the variable c should hold the value of $c = 3$ in both imperative and reactive system. However, in line four, the value of variable a changes, and this is where the difference between imperative and reactive system kicks in. In imperative systems the variable c would still hold the value of 3 because subsequent change of variable a in no way affects already evaluated expression in line three. And this is something most traditional programmers actually expect. In order to update the value of c in imperative system, we would need to re-evaluate expression stated in line three by manually executing it again. However, in reactive systems variable c now holds the value of 4, because by specifying expression $c = a + b$ we instructed runtime engine that c is a function of variables a and b , i.e. that it is dependent on them. Whenever any of the two variables change, the value of variable c should be updated. Rather than doing this manually, updating dependent values in reactive programming is left to underlying reactive engine.

A number of reactive programming languages and language extensions for development of reactive systems have been proposed over time. Initially, those were representatives of functional-reactive programming intended for development of highly interactive systems and animations. Fran (Functional Reactive Animation) [47], for example, is a functional-reactive language implemented as a Haskell extension, intended to aid the construction of interactive multimedia animations. It provides abstractions for both behaviors and events. Yampa [26] is functional-reactive language based on Fran, but is specifically adapted for performance critical reactive systems. FrTime [37] is a functional-reactive language implemented as Scheme language extension, designed for development of interactive applications. It also provides abstractions for both behaviors and events. Frappe [39] is functional-reactive library for Java. It defines two Java

interfaces with methods for behavior and event abstractions. The concrete classes then have to implement these interfaces in order to manifest themselves as events or behaviors. Notable work on reactive programming has also been done by Meyerovich et al. [105] who presented Flapjax - a reactive programming language for web development implemented as a Javascript framework. It supports behaviors for describing values that change over time, and event streams for expressing streams of discrete values.

More recently, reactive programming as a paradigm is increasingly gaining attention from both academic community and practitioners. Different programming libraries, frameworks and language extensions are constantly emerging, and big companies such as Facebook, SoundCloud, Trello, GitHub, Microsoft, and Netflix are supporting and using this concept [151]. This is especially the case for reactive programming dealing with event and data streams. This popularity is understandable if we consider ever increasing requirements for interactive applications, as well as advances and increase in use of big data, IoT and cloud technologies. Indeed the recent spread of reactive paradigm to cloud, data-intensive and big-data applications, IoT and Complex Event Processing is reported by various authors [128], [125], [97]. Partial credit for popularization of reactive programming goes to The Reactive Manifesto [13]. This initiative promotes Reactive Systems as (1) *responsive* - responds in a timely manner, (2) *resilient* - stays responsive in the face of failure, (3) *elastic* - stays responsive under varying workload, and (4) *message driven* - relies on asynchronous message-passing. Another related initiative, which also puts working with streams of data (especially "live" data) to focus is Reactive Streams [14].

Popularization of reactive programming resulted in its ideas also penetrating mainstream programming languages. One of the most prominent examples of this is ReactiveX - a collection of ongoing open-source projects, which comprise efforts in bringing aspects of reactive programming to different languages [15]. Extensions for mainstream languages gathered around ReactiveX include extensions for Java, C#, C# Unity, C++, Javascript, Python, Ruby, JRuby, Groovy, Scala, Clojure, Kotlin, Swift, Netty, and Android, all of them being quite recently developed. They do not operate on continuous time-varying values (behaviors), but are rather focused on discrete values, i.e. streams (sequences) of events/data called Observables [15]. ReactiveX provides a number of operators to work with Observables, including operators for filtering, selecting, transforming, combining, and composing.

The perspective of reactive programming advocated by initiatives such as The Reactive

Manifesto and Reactive Streams, is followed by several popular solutions or families of solutions such as ReactiveX. However, as already stated, these solutions are focused on processing event/data streams, often in a form of real time, "live" data. While this represents very interesting aspect of reactive programming, it is not something we focus on in this dissertation.

Most of the solutions mentioned so far follow the functional/declarative style of programming. Conversely, as reported by [129] and [65], most of today's software applications are developed using imperative, object-oriented languages, with the serious doubt of it changing anytime soon. Reaping the benefits of reactive programming may require transitioning to not only new technology and programming language, but also to different programming style. This process may also be accompanied by the need to rewrite and refactor large quantities of existing imperative code. According to Salvaneschi et al. [129], this might not be something that traditional object-oriented developers would be inclined to do. The same authors also report that because of functional flavor, reactive programming may not fit very well with mutable objects; some computations may not be easily or clearly expressed; and functional style could induce some performance inefficiencies. Because of declarative style, Salvaneschi and Mezini [130] also see debugging as an issue in reactive programming, since current debuggers are suited for imperative programming. They, however, propose methodology for debugging, and also the Reactive Inspector - debugger tool for reactive programs which focuses on navigating dependency graph. Schuster and Flanagan [136] admit that reactive libraries can provide data structure such as signals, which support change propagation. However, they emphasize that this often requires user code to be *lifted* onto signals, which involves significant syntactic overhead. This is, for example, seen in Frappe [39], which in order to use behaviors and events, introduces a lot of boilerplate code tangled with core business functionality. Margara and Salvaneschi [97] argue that while existing reactive languages apply sequential computational model, reactive computations are in many cases independent, and introduction of parallelism could improve overall performance.

On one hand, we have the lack of native support for reactive programming in object-oriented languages, and on the other unwillingness to abandon all the benefits of object-orientation in favor of reactive programming. According to Boix et al. [65], this raises the question of reconciling reactive programming and object-oriented programming, i.e. combining reactive programming with "encapsulated mutable data structures" (objects). They recognize two fundamentally

different ways in which this can be done. The first one is *object-oriented reactive programming*, and it assumes starting with reactive concepts as a base, and then introducing object-oriented features. This approach is also referred to as "objectifying". The second one is *reactive object-oriented programming*, which centers on objects as a base technology and introduces reactive language features. This approach is referred to as "reactifying".

In their paper, Boix et al. [65] took the approach of "reactifying" and presented ROAM - an experimental reactive object-oriented framework implemented as an extension to AmbientTalk programming language. Schuster and Flanagan [136] also recognized the challenge of integrating reactive programming with imperative programming. They also opted for "reactifying" approach and introduced the concept of reactive variables which are used for modeling dependencies and change propagation. In order to facilitate adding reactive variables to imperative languages, they provided formal operational semantics. While, it avoids the need for lifting primitive computations and explicit definition of signals, their approach does not allow passing reactive variables as values, and also limits their dependents to lexical scope of their declaration. Frappe [39] is also tied to a traditional OO language (Java), however, besides producing a lot of boilerplate code, it also does not prevent the so-called glitches. The creators of FlapJax [105] claim that the conflict between declarative style exhibited by reactive programming and imperative style is needless. While they do encourage the use of declarative style, they see FlapJax programs as declarative specification over imperative data. Salvaneschi and Mezini [129] also made an effort to overcome some limitations of current reactive approaches and to support development of reactive applications in object-oriented setting. However, they took "objectifying" approach, and started from functional-reactive and dataflow programming perspective, Their efforts were directed at integration of object-oriented concepts with REScala reactive programming language. They based their work around advanced event system found in EScala [64], and are also influenced by the reactive abstractions found in Scala.React [95].

2.1.2. Event-based programming

Another related field that is being mentioned both in the context of reactive programming as well as an independent field is event-based programming. Margara and Salvaneschi [97], for example, recognize event-based programming together with reactive programming as most prominent approaches for development of reactive applications. *Event-based* or *event-driven*

programming is a paradigm specifically suited for development of event-based systems. According to Faison [50], a software system can be considered as event-based if its parts are primarily interacting through event notifications. Similarly, Hinze et al. [74] see event-based system as a software system in which observed events cause reactions in the system. Most of today's software systems, especially with graphical user interfaces, are event-based, as much of their development is about writing methods which handle occurring events (e.g. mouse click, key stroke, file downloaded, etc.). Large body of work done in the field of event-based programming is related to distributed systems and the interaction of their constituent parts. Event-based systems in such environments are usually called *publish-subscribe* systems. Eugster et al. [48] tackle this topic and offer thorough report on "many faces of publish-subscribe" systems. According to Hinze et al. [74], event processing has also become the paradigm of choice in many monitoring and reactive applications. They refer to these systems as *sense-respond* systems.

The notion of **Event** is at the core of the event-based paradigm. It can be defined, for example, as a "*detectable condition that can trigger notification*" [50], or "*significant change in the state of the universe*" [74]. It is important to see that while numerous conditions may occur both in the system and the system's environment, it is only detectable conditions that we consider as events. Also, it is only those detectable conditions that we deem to be potentially important or significant, that we choose to express interest in. Another, closely related concept, is the concept of **notification**. Faison [50] defines it as an "*event-triggered signal sent to a run-time recipient*". This signal is usually sent either by (1) *transferring data*, i.e. making data available to recipient using some shared resource (e.g. shared memory or network) or (2) *transferring execution control*, i.e. calling procedure on the recipient's side.

When talking about event-based interaction in software systems, Faison [50] describes two principal roles that are essential in order for this interaction to occur. The first one is the role of an entity being able to produce or detect event and then send notification. Such entity is most commonly referred to as *event publisher*, *event producer*, *event source* or just *sender*. The second one is the role of an entity being able to receive the notification. Such entity is referred to as *event subscriber*, *event consumer*, *event handler*, *target* or *receiver*. Sending notification is often called *firing the event*, while the act of establishing the link between sender and receiver is called *subscription* or *registration* [50]. In a similar manner, Hinze et al. [74] classify parts of the event-based system into: monitoring component, transmission mechanism and reactive

component. While monitoring component and reactive component correspond to event sender and event receiver roles respectively, transmission mechanisms describes how notifications are sent and received. If notifications are exchanged directly between sender and receiver, then we talk about decentralized, point-to-point systems. However, often a separate, third component is introduced in order to decouple senders and receivers, and route notifications throughout the system. Such approach is called centralized or middleware approach.

One of the ideas that is at the core of event-based programming is the idea of *implicit invocation*. According to Steimann et al. [143] implicit invocation can be considered as both an architectural style and programming paradigm, and is also known as event driven programming and publish-subscribe architecture. It has been introduced by Garlan and Notkin [62] and has been since frequently used to implement inversion of control and achieve loose integration of objects and components. While in traditional systems components usually interact by explicitly invoking each other's methods, implicit invocation relies on events instead. A component, acting as a sender, can publish one or more events as a means of informing other components that some *significant change* happened. Other components, acting as receivers, can express their interest in these events by subscribing to it. Whenever the sender component triggers event, all subscribed components execute their own methods which are in charge of handling that event. In this way the *implicit* invocation of methods is carried out. Now, instead of sender component being required to know its receivers, receivers are required to know the sender, thus the name *inversion of control*. Notkin et al. [111] state several advantages of implicit over explicit invocation, e.g. by separating invocation relationship, it makes it easier to add, modify and integrate new components without the need to change existing components. Garlan and Shaw [63] in their seminal work also recognize support for reuse and evolution of software as two most important benefits of implicit invocation.

Implicit invocation and event driven programming are based on imperative implementations of design patterns such as well-known Observer pattern (Figure 1) [61]. The stated intent and overall idea of Observer pattern is to "*Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically*" [61]. The original Observer pattern assumes objects taking the role of Subject (Observable) - an object whose state change can be observed, or Observer - an object who observes Subject's state and updates its state accordingly. That said, implicit invocation implemented through de-

sign patterns is at least at conceptual level a natural fit for managing reactive dependencies. However, solutions based on Observer pattern are often criticized as inadequate. Salvaneschi and Mezini [129] state that traditional solutions based on Observer pattern have numerous inconveniences, but programmers bear them in return for benefits of OO design. Syromiatnikov and Weyns [145] state that while Observer based synchronization promotes separation of concerns and decoupling, its implicitness makes hard to see and control the effects of notifying Observers. Maier et al. [95] state that Observer pattern is still a predominant approach in dealing with state changes, but using it is very hard and error-prone. Chaturvedi et al. [33] also add partial observation, inability for update prioritization and dangling references as Observer pattern problems.

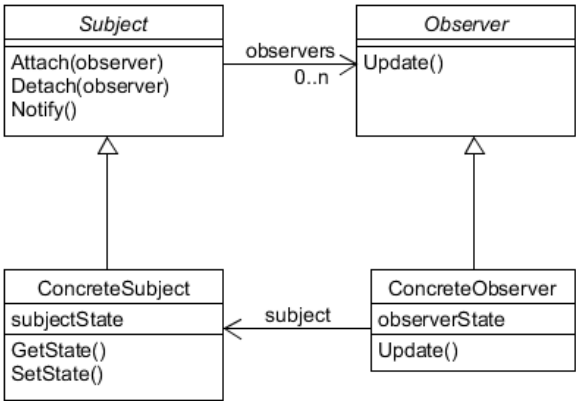


Figure 1: Observer pattern (simple) [61]

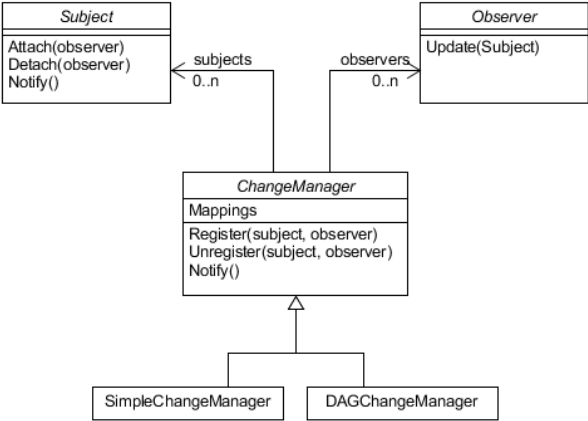


Figure 2: Observer pattern (advanced) [61]

Over time, design patterns similar to Observer have been proposed. Observer pattern revisited (Figure 3) [46] for example improves original Observer pattern by introducing Observer-

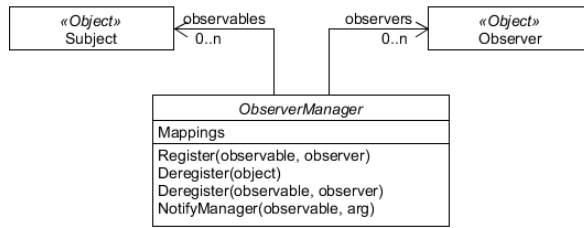


Figure 3: Observer pattern (revisited) [46]

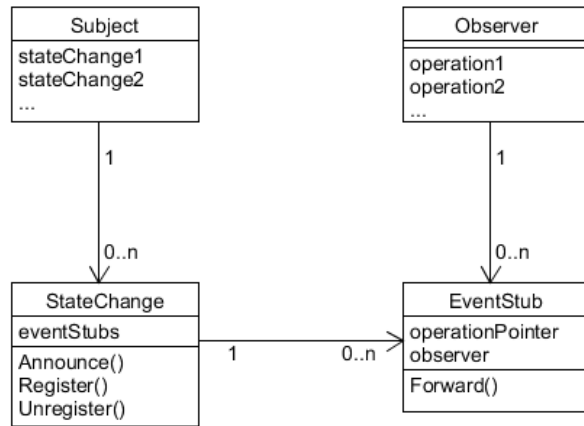


Figure 4: Event notification pattern [122]

Manager as central entity in charge of managing dependencies between Observer and Observable. This is along the lines of ChangeManager in more advanced version of Observer pattern (Figure 2) proposed by Gamma et al [61]. In Event-notification pattern (Figure 4) [122], on the other hand, dependencies are distributed by individual Subjects in dedicated StateChange objects, which together with EventStub objects allow finer event granulation (i.e. specifying multiple events per Subject and multiple update methods per Observer). Propagator pattern (Figure 5) [53] also manages dependencies in distributed manner, however it proposes mechanisms for handling acyclic and cyclic dependency graphs.

Mijač et al. [107] analyzed these patterns and compared their ability to handle complex cases of event propagation and dependencies between objects. They concluded that while individual design patterns exhibit some good features and ideas, no single design pattern is suitable for handling cases where dependencies between objects form large dependency networks. This is especially the case with plain Observer pattern, which is indeed usually mentioned in the context of decoupling user interface implementation and underlying data. E.g. in well-known MVC architectural pattern, an Observer pattern is used to decouple and synchronize Model, View and Controller [24], [145].

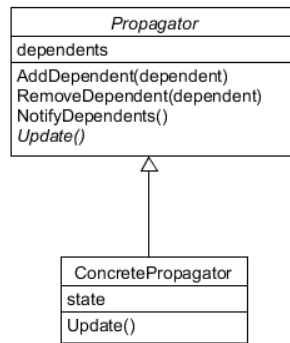


Figure 5: Propagator pattern [53]

In order to address the shortcomings of current Observer pattern implementations in handling complex dependency networks, Mijač et al. [107] identified several useful features of existing design patterns and proposed they should be combined. These features (see Table 1) will prove useful for defining requirements for a proposed solution in this dissertation.

2.1.3. Other approaches

Aspect-oriented programming (AOP) has been first described by Kiczales et al. [86] as an approach to implement design decisions which crosscut system's basic functionalities. While, for example, object-oriented programming assigns each class its own concern, proponents of aspect-oriented programming emphasize that not all concerns should be assigned to one particular class or one particular component. Rather, some concerns such as exception handling, managing security, logging etc., crosscut several classes, components, and even layers.

One of these crosscutting concerns, according to proponents of AOP, is managing dependencies between objects through design patterns such as Observer pattern (for example, Tennyson [149], recognizes data synchronization as a crosscutting concern commonly seen in software design). However, several drawbacks of traditional implementations of data synchronization and dependency management using Observer pattern are pointed out. Eales [46] claims that in traditional Observer pattern implementations concrete classes must implement Subject/Observer interfaces or inherit abstract classes, while in fact they are not real specializations of Subject or Observer. Not only does this disrupt natural inheritance hierarchy, but in existing systems with already established inheritance hierarchy it can be very challenging to introduce this kind of Observer pattern implementation. Similar is stated by Jicheng et al. [77] as they claim that Observer pattern features are tangled with object's core features, obscuring their primary con-

Table 1: Useful features of existing design patterns (adapted from Mijač et al. [107])

Feature	Description	Guideline
Dynamic dependency network	Dependencies should be formed and destroyed dynamically, at runtime.	All five patterns apply .
Dual roles	Object should be able to take the roles of both event emitter and event receiver at the same time.	<i>Subject</i> and <i>Observer</i> should be defined as class interfaces. Alternatively, see <i>Propagator</i> role in [53].
Centralized management of dependencies	Centralized (middleware) approach for managing dependencies should be allowed by providing a dedicated separate component in charge storing dependencies and managing update process.	See the role of <i>ChangeManager</i> [61] and <i>ObserverManager</i> [46].
Arbitrary "Update" method	Event should be handled and state updated by an arbitrary method assigned at runtime (as opposed to predefined Update method in original Observer pattern)	See implementation of <i>EventStub</i> objects in [122].
Multiple exposed events	Object should be able to emit multiple different events (as opposed to only one in original Observer pattern)	See implementation of <i>StateChange</i> objects in [122].
Additional data about emitter or event	Receiver should receive and/or be able to fetch data about event emitter or event itself.	Push additional data as parameter, or provide reference to object containing the data.
Decrease coupling between emitters and receivers	Dependent objects should be decoupled by employing implicit invocation principle and by introducing various levels of indirection.	All five patterns employ implicit invocation. Further indirection can be introduced through centralized approach for managing dependencies and <i>EventStub</i> and <i>StateChange</i> objects in [122].
Handling acyclic graphs	Dependencies forming acyclic graph should be handled in order to ensure proper order of propagation, and thus avoid redundant updates and glitches.	Apply breadth-first order of propagation or perform topological sorting of dependency graph. This is mentioned in [61] and [53].
Handling cyclic graphs	Dependencies forming cyclic graph should be handled in order to prevent infinite loops.	Apply techniques mentioned in [53], such as graph marking (maintaining the list of visited objects), topological sorting and "smart propagation", to break up the loop and avoid redundant updates.
Performance optimization	Where possible, performance optimizations should be applied in order to prevent or lower the occurrence of unnecessary and redundant updates.	Apply cut-off [53] propagation step to avoid unnecessary event firing. Keep dependencies stored and organized in a way which allows different analysis of dependency graphs to be conducted. Handle acyclic and cyclic behaviors.
Events composition and filtering	It should be possible to form new event by composing two or more existing events, and also to apply filters to both event emitting and receiving.	Ideas for this can be taken from declarative and reactive approaches (e.g. [95]).

cern. This results in core system's functionality being more difficult to develop, understand and maintain. The same thing happens also to code in charge of managing dependencies. On one hand, it is being scattered all over the system, and on the other it is tangled with the code implementing core functionalities. According to Tennyson [149], with every case of crosscutting class cohesion decreases while coupling between classes and code scattering increases.

The solution to these problems is seen in using aspects. Noda and Kishi [110] use the concept of separation of concerns in order to implement design patterns more flexibly. They argue that design patterns and core application's functionality are different concerns and need to be separated. In his paper Tennyson [149] presents a novel, *aspectized* variant of Observer pattern with the same intent as original design pattern, but with eliminated or reduced crosscutting

concerns. A number of similar variants of Observer pattern based on aspects are proposed, and demonstrated in AspectJ programming language: [77], [117], [29], [68]. All these implementations recognize Observer pattern related code as a crosscutting concern. They extract it from core classes and put it in aspect, keeping core classes not only clean, but completely oblivious of their role as Subject or Observer.

In order to manage reactive dependencies, some authors used AOP in combination with other approaches. For example, Axelsen et al. [25] combined AOP features with package templates - mechanisms for instantiating ordinary but customized package. Such approach was intended to produce reusable variant of Observer pattern which would minimize the amount of *glue code*. Zhuang and Chiba combined the ideas from event-driven programming, aspect-oriented programming and reactive programming in order to expand event-driven systems to support reactive programming. They first proposed a prototype of new language extension DominoJ with a construct named method slot - an abstraction which integrates methods, events and advices [163], [164]. Then, they expanded DominoJ to ReactiveDominoJ [165], [162] which automatically infers dependencies and creates bindings between events and handlers, thus supporting reactive dependencies.

Several other approaches and paradigms have been used to develop reactive systems, especially real-time systems. Bainomugisha et al. [27] name *synchronous programming* paradigm as earliest proposed paradigm, *dataflow programming* paradigm, and the combination of the two - synchronous dataflow paradigm. Another approach is *constraint programming*, which is a form of declarative programming with the idea of solving problems by declaratively specifying constraints which must be satisfied in a computer program [22]. The central part of this approach are constraints, which, according to Freeman-Benson and Borning [57], present "*declarative statements of relations among elements of language's computational domain...*". Rather than imperatively specifying concrete steps in which the problem will be solved, constraints are used to describe the problem and let the underlying solver find the solution. For example, by defining constraint in a form of e.g. $c = a + b$ we oblige language's embedded constraint solver to always maintain that constraint. Whenever any of the elements in constraint change, other elements have to be updated so the constraint statement remains valid. We can see that the constraints, as a concept, are closely related to reactive dependencies.

Demetrescu et al. [43] took constraint programming and dataflow programming approach in

managing reactive dependencies. They presented a general-purpose framework DC (C++) for specifying one-way dataflow constraints as an equation of the form $y = f(x_1, \dots, x_n)$. Whenever any of the variables x_i changes, expression on the right side is re-evaluated and assigned to y . Their approach is based on two key elements: reactive memory locations and constraints. Freeman-Benson and Borning [57] set a goal to integrate declarative constraints with imperative object-oriented concepts, and directed their efforts into development of Kaleidoscope - an object-oriented constraint imperative programming language. Similar, but more recent language presented also by Borning is Wallingford [30], which again uses constraints to achieve reactivity between objects's states.

Similar approach to constraint programming is taken by Heron [72]. He discusses the use of definitive scripts for maintaining dependencies between values. Definitive scripts contain a list of definitions describing dependencies between objects in form of $t = f(s_1, s_2, s_3, \dots, s_n)$, with t being target variable, $s_1 \dots s_n$ being source variables, and f being expression for computing target variable from source variables. Definitive system is in charge of handling definitions and ensuring target variables are recomputed when any of the source variables change.

2.2. Software frameworks

2.2.1. Software frameworks as a reuse technique

In every aspect of our lives we tend to reuse our personal or someone else's knowledge, past experiences, and various artifacts that we found in nature or built by ourselves. This allows us to better cope with existing and new problems, which are becoming larger and more complex every day. We behave very similar when we are developing software, with one important difference - the very nature of software, which is non-physical in its core, makes reuse more capable (although not necessarily easier). That is why, from the very beginnings of software development, reuse was one of the key forces influencing evolution of software development approaches, development processes, technologies, programming languages and other aspects of software development.

Software reuse can be described as the process of using existing software and software knowledge to construct new software [155]. It can be *opportunistic*, i.e. applied ad-hoc to particular situation and usually not planned in advance. More favorable, though, is *systematic*

reuse, i.e. intentional and managed process of creating software artifacts with their reuse in mind, and reusing these artifacts as many times as possible. Schmidt and Buschmann [134] emphasize the need for systematic reuse of software models, designs and implementations that have been already developed and tested in order to increase software productivity and quality.

Various reuse techniques were proposed over time, differing in scale, level of abstraction and complexity of reused software artifacts [23]. They range from pure implementational artifacts such as source-code components, to pure design artifacts such as highly abstract architectures and patterns. One of today's most common reuse techniques, positioned in the middle of aforementioned *implementational vs design* artifact continuum, are **software frameworks**, which are at the same time being able to deliver high levels of design and code reuse as well. With frameworks, reuse is beneficial within individual applications as well as across several applications [52]. Frameworks are both the most frequently created reuse artifact, and also the most frequently reused artifact in application development. It is easy to find a large number of software frameworks of different flavors, created for different domains and purposes, large or small, commercial, free or open-source. Historically, frameworks originated in the domain of graphical user interfaces (GUIs), starting with pioneers such as MacApp, X-windows, MFC, Java Swing and others. Their success inspired and fueled broader adoption of frameworks in other domains.

Software frameworks today have decisive role in building software systems with ever increasing size and complexity. They are acknowledged by both academy and industry as one of the most powerful and widespread reuse techniques. No serious development is today done using bare programming languages, by taking the "reinventing the wheel" approach. Instead, modern software development is largely framework-based or framework-driven, no matter what technology is used. One would have a hard time finding non-trivial software application which does not involve at least one software framework. New software frameworks constantly emerge, proposed by both researchers and practitioners, with practitioners seeing them as mechanism to improve their software process and software product, and researchers recognizing them as an essential software artifact worth researching.

According to Merriem-Webster dictionary [5], framework in general can be defined as "*a basic conceptual structure*" or "*a skeletal, openwork or structural frame*". Similarly, Oxford dictionary [6] defines framework as "*a basic structure underlying a system, concept, or text*".

These definitions clearly imply framework being a skeleton or a backbone of some kind of a system, which determines (or at least heavily influences) its base structure and form. Not surprisingly, when we talk about software frameworks, we talk about backbone of the software system.

Definitions of software frameworks are numerous. Some of the most commonly found ones are listed in Table 2. While these definitions are not necessarily in contradiction, they for sure provide different views and perspectives on software frameworks. We can see that in these definitions, software frameworks are described using different, but still valid, concepts, such as: abstract and reusable design and application, reuse technology, customizable system, application skeleton, semi-defined application etc.

Listed definitions are also giving a brief glimpse on how the software framework could be structured internally. Here, the influence of object-orientation as an originating technology for software frameworks is clearly visible as frameworks are described to consist of set of cohesive design and implementation elements, namely patterns, templates, packages and collaborating classes.

Table 2: Software framework definitions

Definition	Concept	Structure	Purpose
"set of classes that embodies an abstract design for solutions to a family of related problems" [81].	Abstract design	Set of classes	Provides common solutions
"reusable design of all or part of a system that is represented by a set of abstract classes and the way their instances interact" [80].	Reusable design	Set of abstract classes and their instances.	Reuses design
"a technology for reifying proven software designs and implementations, which is capable of reducing cost and improving quality of software" [51]	Reuse technology	-	Offers reducing costs and improving quality.
"system that can be customized, specialized, or extended to provide more specific, more appropriate, or slightly different capabilities" [114]	Customizable system	-	Provides customizable solutions.
"powerful technology for developing and reusing high-quality middleware and application software" [134]	Reuse technology	-	Reuses high quality software artifacts.
"application generators that are directly related to specific domain, i.e. family of related problems" [98]	Application generators	-	Generates solutions for family of related problems.
"powerful object-oriented reuse technique that typically emphasizes the reuse of design patterns and architectures" [88]	Reuse technology	-	Reuses design patterns and architectures.
"package that contains model elements that specify a reusable architecture for all or part of the system". Typically include classes, patterns, or templates. [18]	Package	Package with classes, patterns or templates	Reuses architecture.
"reusable, semi-defined application that can be specialized to produce custom applications". [23]	Semi-defined application	-	Produces custom applications.
"skeleton of an application that can be customized by application developer". [80]	Application skeleton	-	Provides customization capabilities.
"large and abstract applications aimed at particular domain that can be tailored to suit individual applications". [31]	Abstract application	-	Building individual applications.
"large structure that can be reused as a whole to construct new systems". [31]	Large structure	-	Construct new systems.

While the concepts and structural elements used to describe software frameworks may differ, there is no ambiguity about their purpose. Software frameworks aim at facilitating development of end-user applications by providing customizable and extendable design and implementation solutions.

Another interesting way to view frameworks is through the commonality/variability lenses [118]. We can say that frameworks aim at capturing commonalities and variabilities of specific domain, by providing implementation for common elements, and localizing variabilities at *variation points* [92]. They act as an extension to base programming languages, by utilizing commonalities in design and implementation [118]. These commonalities may emerge from the domain developers work in, development practices they carry out, or applications they develop. In any case, the choice of commonalities built into the framework and the variabilities enabling the framework extension determines the reuse potential of the framework.

2.2.2. Relation with other reuse techniques

In order to understand the role and significance of software frameworks, it is important to position them with regard to other reuse techniques. Reuse techniques differ in several aspects, but perhaps two of the most important differentiating characteristics are the *size/complexity* of reuse technique and the *level of abstraction*.

There is no doubt that software frameworks fall into large and complex reuse techniques. Polančič et al. [118] even see software frameworks as the most complex reusable structures, made up of different design and implementation parts. They add that unlike other reuse techniques, software frameworks aim to reuse larger-grained components and higher-level designs. Schmidt and Buschmann [134] and Zhang and Kim [161] also emphasize the role of software frameworks as technique aimed at larger scale reuse.

With regard to *level of abstraction*, some reuse techniques take the low level approach - by offering reuse of implementation (code), and other take the high level approach - by offering reuse of design. Software frameworks take stand in the middle of these approaches, promoting at the same time both design and implementation reuse [80]. Most will agree, however, that the very design reuse is the most important thing software frameworks offer.

In research literature, reuse techniques most often compared and associated with software frameworks are: components, libraries, architectures and design patterns. In contrast with com-

ponents, Johnson [80] sees frameworks as more customizable and powerful option, but also an option with more complex interfaces, which makes frameworks more difficult to learn. Unlike components, which emphasize code reuse, frameworks tend to reuse both code and design. Froehlich et al. [60], see frameworks as a solution to larger-grained problems than components.

Class libraries are also large reuse structures, containing a number of individual components. However, Johnson [80] indicates that frameworks reuse high-level design, while libraries reuse implementation. Also, in frameworks inner components necessarily collaborate, while in class libraries this is not the case. Krajnc and Heričko [90] add that frameworks usually do not allow individual classes to be reused, and they control the flow of application execution (so-called inversion of control). Sparks et al. [139] differentiate frameworks and libraries by the way they are reused. When reusing frameworks we need to inherit from framework's classes and specialize them by overriding existing methods, or implementing abstract methods. Contrary, when reusing libraries we usually call functions and pass required parameters.

Design patterns represent common solutions to repeating problems [61]. Compared to software frameworks they are smaller, and also more abstract, i.e. they promote pure design reuse, while frameworks also provide implementation [90]. Because of that, according to Johnson [80] frameworks tend to be more concrete, easier to reuse, but also less flexible than patterns. Architectural styles promote pure design reuse on a larger scale than design patterns promote, however, frameworks again differ with their reuse of implementation.

One additional reuse technique closely related to frameworks are *framelets*. They are proposed by Pree and Koskimies [120] who describe them as flexible and reusable building blocks of applications. Unlike frameworks, framelets are small in size (less than 10 classes), they do not assume to hold main control of execution, and they have clearly defined simple interface. Although smaller in scale, framelets still share similarities with frameworks, such as that they too offer software architecture reuse, and implicit invocation principle. However, instead of complex and large frameworks, authors suggest offering family of related framelets for particular domain which represent the framework. Frameworks may often be monolithic, while framelets promote modularity.

Figure 6 graphically depicts relation between frameworks and several related reuse techniques with regard to their size and the level of abstraction. We can see that by offering both design and code (implementation) reuse, frameworks hold middle ground between predomi-

nantly design reuse techniques and code (implementation) reuse techniques.

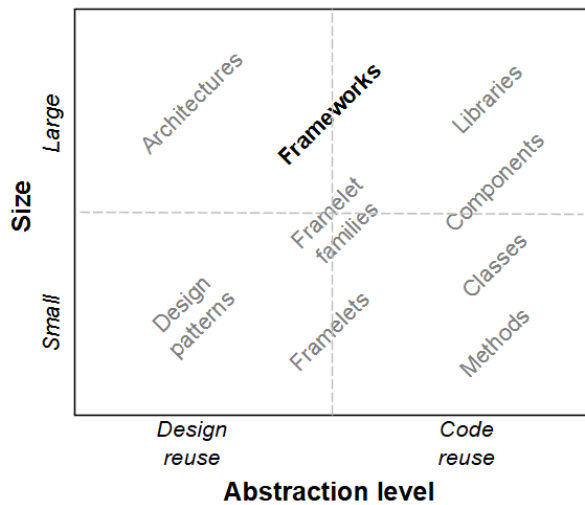


Figure 6: Framework vs other reuse techniques

In order to achieve large-scale reuse, software frameworks combine and compose other reuse techniques and structures of various scale and abstraction level. At first, software frameworks most certainly prescribe particular software architecture as a large-scale design reuse technique, and impose it on an end application. Software architecture may be subdivided into smaller-scale design concepts, i.e. design patterns. In this way, design patterns can be seen as micro-architectural parts of the framework [161]. Schmidt and Buschmann [134] also see frameworks as concrete realizations of group of patterns. According to Johnson [80], frameworks are highly interconnected with design patterns. At first, when developing framework, developers tend to apply large number of design patterns. Secondly, if applied many times, the framework itself (i.e. design characteristics the framework promotes) may become a sort of higher level design pattern. Lastly, the use of framework by developers results in identifying frequent new problems and solutions, which are often formalized as design patterns.

Design patterns are also often used as a communication medium, i.e. to communicate ideas between different stakeholders (developers, managers, end-users etc.) in the various stages of software framework development process. Froehlich et al. [60] emphasize in particular the role of design patterns as a common vocabulary between framework developers and framework users. Similarly, Srinivasan [140] sees documenting frameworks as a set of design patterns as an effective way to achieve good communication between framework developer and framework user. This is made easier given the fact that both participating sides in this communication are necessarily software developers.

2.2.3. Types of software frameworks

Due to a large number of existing frameworks and their differentiating characteristics, number of authors attempted to categorize them, guided by different criteria. Krajnc and Heričko [90] combined these categorizations, and proposed the most extensive 7-criteria classification presented in Table 3.

Table 3: Framework classification [90]

Criteria	Type
Extensibility	Whitebox frameworks Blackbox frameworks Graybox frameworks Glassbox frameworks
Scope	System infrastructure frameworks Middleware integration frameworks Domain frameworks Enterprise frameworks Business collaboration frameworks
Approach	Object-oriented frameworks Component frameworks Service-oriented frameworks Aspect-oriented frameworks
Standardization	Standardized frameworks Semi-standardized frameworks Unstandardized frameworks
Granularity	Fine-grained frameworks Coarse-grained frameworks
License	Commercial frameworks Free frameworks
Format	Logical specification Physical design Source code Binary code

Extensibility tackles the question of how and at what degree can the framework be extended, i.e. actually be used. The *white-box* frameworks heavily rely on well-known object-oriented mechanisms for extension, such as inheritance, overriding and dynamic binding. Such frameworks are generally harder to use because they expect application developer to understand framework internals and to provide most of the concrete behavior, but are at the same time easier to develop for framework developers. White-box frameworks are also referred to as *architecture-driven* or *inheritance-focused* frameworks [102]. *Black-box* frameworks, on the other hand, use composition as a means to provide extensibility, i.e. they specify interfaces through which components can be plugged into the framework. This makes them easier to use given that a lot of concrete behavior is already present and the complexity of framework internals is hidden. However, this means that they are harder to develop. Mattsson [102] refers to them as *data-driven* or *composition-driven* frameworks. In reality few frameworks can be characterized as pure white-box or pure black-box frameworks. Rather they combine the char-

acteristics of both categories, so, where convenient, part of their features are offered through e.g. inheritance, and other part of them through composition. It is often the case that the framework starts off as a white-box framework, and then during its lifetime shifts towards black-box framework. Lastly, the framework is referred to as *glass-box* framework if its implementation is available for framework user to see, but not to change.

The **scope** determines framework's primary reuse target, i.e. what is framework trying to reuse. It places frameworks into one of the five categories. *System infrastructure frameworks* are usually company's internal frameworks intended to simplify development of system infrastructure. Similarly, *middleware integration frameworks* are also internal frameworks, but they are aimed at integrating distributed applications. *Domain frameworks* capture the expertise of a particular domain, and do not cover other aspects of applications. *Enterprise frameworks* are large frameworks which embody architecture for entire application, and may provide both infrastructural as well as domain-specific features. Lastly, *business collaboration frameworks* as their name implies, aim at supporting development of applications for collaboration of businesses by integrating their services and exchanging data.

Frameworks can also be classified according to **approach** used to build them. Object-oriented paradigm and its reuse mechanisms fueled the emergence of software frameworks, which became known as *object-oriented frameworks*. They, in turn, led to emergence of *component frameworks*. Lopes et al. [93] infer that typically object-oriented frameworks are white-box, and component-oriented frameworks are black-box, although, they do not reject the possibility of other combinations. *Service-oriented frameworks* are successors of object-oriented and component frameworks, emphasizing the contractual nature of framework's behavior. Lastly, *aspect-oriented frameworks* aim to utilize features of aspect-oriented paradigm.

Standardization plays major role in acceptance of particular technology, including software frameworks. Thus, Krajnc and Heričko [90] identify three levels of framework standardization: (1) *standardized frameworks* are the ones widely recognized and supported by the international standards body; (2) *semi-standardized frameworks* are the ones supported by a group of important vendors in the field; (3) *unstandardized frameworks* are the ones defined by the single vendor. During their lifetime, software frameworks may change the level of standardization.

Granularity indicates the size and complexity of the frameworks. *Fine-grained frameworks* are usually small frameworks with small number of closely defined features. On the other hand,

coarse-grained frameworks or *monolithic frameworks* are usually large frameworks, consisting of large number of features with powerful but complex interfaces. Froehlich et al. [58] note that both approaches have their benefits and drawbacks, but building fine-grained frameworks is favorable in most cases due to them being less complex, easier to maintain and more modular.

According to **license**, frameworks can be regarded as either commercial frameworks or free frameworks. *Commercial frameworks* require some form of payment to use the framework and access other framework related services (e.g. technical support, updates, education etc.). *Free frameworks* are, on the other hand, free to use with or without restrictions. When discussing free frameworks we should also mention *open-source frameworks*. Not only they are free to use, but usually their whole source code is available for viewing, changing and extending. As with standardization, frameworks can also change license model during their lifetime.

Format defines different forms the frameworks can take during each stage of their development cycle. Software framework starts as a *logical specification*, containing descriptions of features the framework must provide and constraints under which the framework must operate. *Physical design* details this specification, including specification of components, classes, hierarchies, interfaces, methods and other required elements. Lastly, implementation of the framework results in a *source code* or *binary code* of the framework.

2.2.4. Software framework structure

Being one of the reuse techniques, software frameworks in its base revolve around capturing a specific domain's commonalities and variabilities [94]. They do so by designing and implementing common domain elements into software framework, and providing an architecture which will localize domain variations. Coplien et al. [38] define commonality as an assumption held uniformly across all objects of a given set, contrary to variability which stands for an assumption held only for a subset of these objects. Both are essential to framework-driven reuse - commonalities determine the reuse potential of frameworks, and variabilities enable realization of that potential.

Common parts in frameworks represent invariant parts of the framework domain, the parts which ideally are going to be reused in as much software applications as possible. They are often referred to as *frozen spots* - already coded software pieces to be reused, which provide architectural backbone, communication, data exchange and synchronization mechanisms [114].

Since frameworks need to generate applications for entire domain, they are required to have some points of flexibility and variability [98]. These points are known as framework's *hot spots*, and they allow common parts of the framework to be reused in different contexts. They are means of adjusting frameworks to concrete application's needs [114], means for application developers to add their own application-specific code [118]. Two categories of methods are generally involved in hot spot implementation, namely *hook* methods and *template* methods [133]. Hook methods represent placeholders for variable parts of the framework, an actual points of adaptation and extension (e.g. abstract methods, parametrized methods, callback methods, configurations...). Template methods, on the other hand, define abstract behavior and generic interactions, and they invoke individual hook methods.

Frozen spots as representatives of framework commonalities, and hot spots as representatives of framework variabilities are indispensable part of the framework. They constitute what is frequently called a *kernel* [98], *central backbone* [114], or a *core design* [31] of the framework. This core part of the framework specifies key abstractions of the domain, and how these abstractions relate and interact. Implementation-wise, this means that the framework backbone consists of set of abstract and concrete classes [31]. Frozen spots are largely implemented as concrete classes, although, in order to support frozen spot hierarchy, some of them may also be implemented as abstract classes. In either case, classes implementing frozen spots are made invisible to framework user, i.e. they are not part of the framework's interfaces. This is because they are not meant to be tackled by anyone other than framework developer, and even then with a great caution. On the other hand, hot spots are, according to [31], implemented mostly as an abstract classes, which are made visible and available to framework users as a part of framework's interface. It should be noted, however, that inheritance mechanism is not the only way frameworks implement hot spots. According to Lopes et al. [92], variation points can at least be classified as white-box (inheritance-based, abstract classes) or black-box (composition-based). In addition to inheritance, Polančič et al. [118] also list dependency injection, template methods and closures as a means to implement hot-spots. They mention *dynamic binding* (also known as *inversion of control* and "*Hollywood principle*") as a base underlying technique implemented through these mechanisms. This technique assumes that exact code to be executed is determined at runtime, which allows framework to call applications code. Inversion of control mechanism is one of the main determinants of software frameworks. As opposed to that, Lopes et al. [92]

introduce the notion of *call points* - framework code which is called by application.

Besides core part of the framework, consisted of frozen spots and hot spots, frameworks often contain what we call *auxiliary* part. This part is not essential for basic functioning of the framework, but instead aims at providing means to empower the framework, speed-up and facilitate the use of the framework. One of these mechanisms which increase the utility of the framework are basic components, which are according to [114] mandatory part of the framework. These components are usually built internally by either framework or component developer. They offer typical and common implementations of hot spots which can be immediately plugged into framework backbone by framework users. In this way, framework user can build a large part or even entire application simply by combining and composing framework and its built-in components. In order to be more usable, frameworks are often accompanied by whole libraries containing these predefined components. Bosch et al. [31] refer to these libraries as *framework internal increments*. Polančič et al. [118] also list framework class libraries as one of the framework's most common constituent code elements, which offer concrete predefined components ready to be used by application developers with little or no modification. If we consider framework backbone to be primary reuse mechanism in frameworks, then accompanying libraries can be referred as secondary. Auxiliary part of the framework can also offer various tools [60]. These tools often act as a mediator between framework and its users, by offering easier access to framework's features, automating set of tasks, generating pieces of code etc. This helps framework users to master the framework faster, and to be more productive.

While software frameworks are primarily intended to be reused by framework users, framework developers can also benefit from reuse when developing software frameworks. One way to do that is by using already existing frameworks and building our own framework on top of them [81]. Therefore, besides libraries, hot spots and frozen spots as a smaller scale elements, software frameworks can also contain other, perhaps smaller and more specific frameworks within. Interesting suggestion comes in a form of *framelets* [120] - a very small framework (up to 10 classes) with clearly defined interfaces, which can be one approach to introduce additional modularity and build frameworks from other frameworks.

The framework elements we mentioned, and their relation to framework and also each other is shown in the framework metamodel in Figure 7.

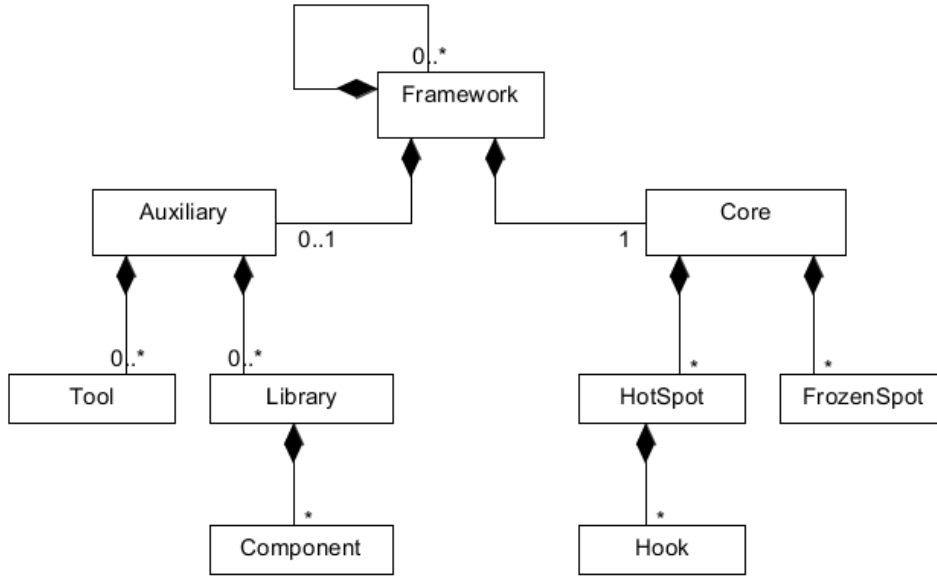


Figure 7: Framework metamodel (code elements), adapted from [118]

2.2.5. Framework-involved processes

Software reuse in general involves two main processes: *building* reusable software artifacts, and repeatedly using (i.e. *reusing*) these artifacts. The same goes for software frameworks, so we are going to be interested in the processes of (1) *software framework development*, and (2) *software framework usage*. Bosch et al. [31] see these processes as phases in, what they call, a *framework-centered/framework-based* software development. Along with framework development and framework usage phase, they also introduce third one - *evolution and maintenance* phase, which explicates the process of framework maturing and evolving over time to fit changing requirements. Because of iterative nature of framework development all three phases are mutually related and entwined, so it is not easy to discuss them separately. However, for the sake of clarity, in this section we will present software framework development process and software framework usage process separately. On the other hand, evolution and maintenance phase will be considered as an activity within software framework development process.

Software framework development process

Software framework development process is a set of mutually related activities which result in producing software framework and other accompanying software and non-software artifacts (specifications, models, tests, tools, documentation etc.). Often terms such as framework design or framework building are used interchangeably with framework development. In any case,

when we talk about overall process of designing/building/developing framework, we actually talk about framework development in a *broader sense*. This, however should not be confused with often used designing/building/developing/implementing framework in a *narrow sense*. In a narrow sense, these terms refer to individual activities within framework development process, which result in creating framework models and/or framework source code implementation. We will always designate framework development in a broader sense as a *process*, and framework development in a narrow sense as an *activity*.

The commonality/variability abstractions as lenses can offer us a high-level view at the framework development process. The framework development would first involve specifying and implementing commonalities, and then specifying variabilities which are going to be implemented and filled-in later, in the framework usage process. For a more detail we turn to general, software development meta-process. According to Sommerville [137], no matter what software development process we practice, and what software product we develop, we are involved in four fundamental activities, namely: software *specification*, software *design and implementation*, software *validation* and software *evolution*. We use this generalized view to structure and describe framework development processes suggested by various authors (see Table 4). Parsons et al. [114] and Markiewicz and Lucena [98] include also the activities of *additional components development*, and *framework instantiation*. However, we argue these activities refer to the process of using frameworks, rather than developing frameworks.

Table 4: Framework development activities as seen by various authors

Meta-activities	Parsons et al. [114]	Markiewicz and Lucena [98]	Sparks et al. [139]	Bosch et al. [31]
Framework specification	1. Domain and role definition	1. Domain analysis	1. Define scenarios and use cases	1. Domain analysis
Framework design and implementation	2. Backbone development 3. Basic components development - -	2. Framework design - - -	2. Design framework 3. Build framework - -	2. Architectural design 3. Framework design 4. Framework implementation 5. Framework documenting
Framework validation	-	-	4. Test framework	6. Framework testing
Framework evolution	-	-	5. Deploy framework	7. Framework maintenance and evolution
-	4. Additional components development	3. Framework instantiation	-	-

Framework specification

Framework specification activity involves understanding and defining what features and characteristics the future framework will provide, and under what constraints it will operate.

Parsons et al. [114] emphasize the need to identify domain which will be covered by the framework and also the roles the framework users will take. Markiewicz and Lucena [98] recognize the importance of domain analysis as a way to discover current and future framework requirements, and at least partially uncover frozen spots and hot spots of the framework. The results of domain analysis are according to Bosch et al. [31] packed in a form of domain analysis model, which contains domain description, concepts, relations and requirements for the framework.

In order to successfully analyze target domain, and extract framework requirements, several techniques and resources may be of aid. Domain's key abstractions and variations may be identified by utilizing personal experience ([98]), domain experts' knowledge ([31]), and previously published experiences and standards ([98], [31]). As a one way of becoming domain expert, Froehlich et al. [58] advises framework developers to build an application(s) within domain. However, domain knowledge is frequently gained also by examining existing applications within domain ([98], [31], [58]). Johnson and Foote [81] for example find it worthwhile to examine a nearly-complete projects to see if new abstract classes and frameworks can emerge, and be reused later. This can be seen as a form of recycling and also cleaning up existing design. Another technique for analyzing framework's domain is to identify framework scenarios and use cases [139], which again helps us gaining insight into framework's future abstractions and the ways framework will interact with end-applications. When domain analysis has provided us with sufficient knowledge about the target domain, its concepts and abstractions, its scope and vocabulary, framework requirements can be captured and specified. Various techniques from requirements engineering can be used in this phase.

Main participants in this activity are domain experts, framework developers and application developers, which in many cases will be the same group of people or individuals.

Framework design and implementation

Software framework design and implementation activity results in distinct but highly related artifacts, namely design and implementation of the framework. Sparks et al. [139] and Bosch et al. [31] for example, clearly separate these two activities in their respective framework development processes. However, due to their mutual relation and iterative nature, framework design and implementation are often seen as inseparable and therefore as one activity. For example, under the term "design", Markiewicz and Lucena [98] imply both design and implementation.

For the sake of clarity we will describe design and implementation separately, with the remark that in practice these steps are indeed often conducted iteratively with no clear boundaries.

Design as an activity involves modeling structural and behavioral characteristics, static and dynamic properties of the software system. If that system is software framework, then the modeling comes down to defining structure and behavior of common part and variable part of the framework, i.e. frozen spots and hot spots. Parsons et al. [114] wanted to emphasize that, so they even named whole design and implementation activity as a backbone development activity. Framework design starts with *architectural design* of the framework, where framework developer, in accordance to domain analysis model and specified requirements from the previous phase, decides on suitable architectural style and produces top-level design of the framework [31]. According to Landin et al. [91], architectural design of the framework can be accomplished by performing following tasks: (1) refining the analysis object model, (2) assigning system responsibilities to specific objects, (3) analyzing object collaborations and (4) refining inheritance hierarchies and collaborations.

Top-level architectural design is further refined in *detail design* step [31] by adding details about the structure and behavior of the framework. This involves specifying exact attributes and methods the framework classes will implement, with regard to characteristics and constraints of the chosen technology and language. It also implies defining collaborations and relationships between classes in a more precise manner. Special attention is devoted towards modeling framework's hot-spots and interfaces. As opposed to an architectural design which focuses on high level design decisions, detail design step deals with lower level design decisions i.e. micro-architectural decisions.

Both high and low level framework design will benefit the most from framework developers' personal experience in framework design and design of software in general. Additional knowledge can be further obtained from external experts, existing papers and reports (lessons learned, best practices, models, guidelines, design patterns, architectural styles, etc.), and also similar existing frameworks.

While models permeate all activities of software and framework development, design is perceived as perhaps the most model-intensive phase. Design models tend to be a prescriptive ones, i.e. they prescribe how the original (framework) should look like and behave when built, thereby guiding developers through implementation. In this sense, *framework implementation*

can be seen as a process of transferring from design models to a working software framework, by coding, debugging, and testing framework elements, and integrating them into framework. While it was at certain level possible (and desirable) to omit technology-specific aspects of framework development in the previous activities, in implementation activity this is not the case. Framework implementation resides in concrete technology space, therefore it involves using particular programming language (e.g. C#, Java, etc.) and tools (IDEs, testing tools, debugging tools, etc.), and depends on particular runtime environment and platform (.NET, JRE, etc.).

Framework implementation starts by implementing framework backbone, i.e. abstract and concrete classes which form frozen-spots and hot-spots. This is followed by implementing individual components ready to be plugged into the framework's hot-spots. These components can then be organized into framework's internal increments (libraries). Lastly, framework tools are to be developed in order to mitigate complexities of using framework.

Framework validation

According to Sommerville [137], software testing is intended to do two things: show that software does what is intended to do (i.e. that it meets its requirements), and to discover software defects before it is put to use (i.e. find incorrect or undesirable behavior). The first thing is referred to as a validation testing and the second one as a defect testing. Both of these are essential activities in framework development. According to Bosch et al. [31], they determine whether the framework provides the intended functionalities, and whether it is usable. Sparks et al. [139] advocate thorough testing and assuring high test coverage of the framework's code in order for frameworks to be safe for use in application development.

There are several reasons why framework testing should be rigorously conducted before framework is put to real use. One of the reasons is that by intentionally reusing the framework, we are also unintentionally reusing each framework's defect [139]. As much as catastrophic a defect in particular end-user application may be, a defect in a framework is reused in each application using the framework. By being multiplied with each framework reuse, framework defects usually have much larger impact than individual application errors. Frameworks should be well tested also because framework users expect to be able to rely on the framework. After all, one of the key aspects of reuse is avoiding to repeatedly validate the same common software

artifacts [134].

In the framework ecosystem there are three distinct points in which defects can occur: framework, framework-application interface, and application [59]. Framework developers and testers are responsible for testing framework internals and framework interface towards applications, while application developers' responsibility is testing end-application. Sparks et al. [139] argue framework testing to be much easier in isolation. When the framework is already applied, any error that occurs may be result of defect in either framework code or application code. Discerning the real source of the error is usually a hard task [51].

However, according to Froehlich et al. [58] the only true and definitive framework test involves using the framework to build applications. Zhang and Kim [161] agree with that and report framework testing to often be a byproduct of end-application testing, i.e. by testing end-applications frameworks are iteratively validated. Testing framework in such way does not have to necessarily involve building real or complete applications by application developers. Rather, this can be done by framework developers implementing number of examples of using framework. These examples can vary in size and granularity, and can aim to test entire framework, particular module within the framework, or only individual hot-spots.

Different methods for testing software are applicable also to framework testing. Internal framework classes can for example be individually tested in isolation by writing unit tests. In this way, each application which reuses that framework, also reuses testing. Some integration testing should also take place during framework development, with to goal to test how internal framework components integrate, and also to test interfaces between framework and application. Hooks, as a part of framework interface, are according to Froehlich et al. [60] important points of access to the framework, so their evaluation should be among high priority quality activities. Integration testing can be conducted by, for example, implementing already mentioned concrete examples and applications.

Framework evolution

Processes of software development and software evolution (maintenance) were historically seen as separate processes [137]. While the software development resulted in a working software system ready to be used, software evolution focused on adapting and evolving software systems in order for them to remain working and remain being useful. Modern software de-

velopment, however, implies coping with domain and requirements instability, and constant changes throughout the whole software system lifetime. That is why the boundary between software development and software evolution is increasingly fading, and, at the beginning two distinct processes, are increasingly being seen as one.

This is especially the case with frameworks. Most authors see iteration as one of the main determinants of framework development process. Sparks et al. [139] for example stress that the framework is deployed in several iterations. Fayad et al. [52] also see iteration as one of the most common characteristics of framework design process. Froehlich et al. [58] state that frameworks are not built during a single pass through framework development process, but rather require iterative and cyclic approach. Because of this iteration, it is often impossible to discern framework development from framework evolution.

Mattson [102] reports evolution and maintenance of software frameworks can have following goals: (1) *corrective* - bringing framework into more correct state (e.g. fixing failures and bugs); (2) *perfective* - improving framework's overall quality (e.g. performance, maintainability, ease of use, etc.); and (3) *adaptive* - evolving the framework to meet changing requirements (e.g. introducing new features, changing or removing existing ones).

According to Bosch et al. [31], initial versions of the frameworks tend to be white-box. They start out small, as a few classes and interfaces generalized from a few applications in the domain. The framework design changes frequently and inheritance is used as a main mechanism of reuse. Johnson and Foote [81] see such white-box frameworks as a natural stage in the evolution of the framework. However, over time, requirements are stabilizing and the framework becomes more mature. Frequent use scenarios emerge, which results in predefined components being developed and added to framework library. Inheritance as a reuse mechanism is replaced by composition, which allows predefined components to be plugged into the framework backbone. All this leads to framework becoming black-box through gradual evolution.

Software framework usage process

Software framework usage process is a set of mutually related activities which result in producing software applications using software frameworks. This process is also often referred to as framework instantiation or application development (framework-based). Artifacts resulting from this process are called framework instances, framework specializations or simply applications. While application development and applications are common terms when talking about

software development and software products, they do not reveal the full role of frameworks. On the other hand, framework instantiation and framework instances as a terms, aim to emphasize the central role the frameworks often have in application development, and the view on end-applications as mere realizations and extensions of the framework. The main participants in the framework usage process are application developers, i.e. framework (re)users, who use frameworks to develop end-user applications.

Taking the commonality/variability abstractions as lenses to view the framework usage process, we can see that this process involves reusing commonalities previously specified and implemented in the framework, and implementing (concretizing) variabilities which are required, but only specified by the framework. The process of instantiating framework, i.e. building application is the process of assigning variabilities with concrete values and programming code. According to Bosch et al. [31], framework users use already existing core framework design and framework internal increments (components), and develop application-specific increments in order to finalize end-user application. Similarly, according to Murray et al. [109], in order to instantiate framework, framework users have to provide concrete components which the framework expects, and which conform to framework interfaces.

Lopes et al. [92] indicate two fundamentally different forms of framework reuse in framework-based software development: *anticipated reuse* and *unanticipated reuse*. Anticipated reuse refers to reuse that framework provides through its variation-points, which the framework user adapts by providing application specific code. This is typical for black-box frameworks. Unanticipated reuse, on the other hand, happens when the framework user wants to make adaptations which are not part of the framework's predefined variation-points, i.e. the user must make changes to framework internals. Since it requires access to framework internals it is typical for white-box frameworks. While anticipated reuse is natural, intended and aimed to be convenient, the unanticipated reuse is more problematic. It requires greater knowledge about the framework internals, it is more error prone, it creates problems in framework evolution and maintenance, and may sometimes simply not be possible.

Although using frameworks assures high level of reuse when developing software applications, frameworks may be very large and complex artifacts. This means that, in order to be productive with the framework, framework user has to spend a lot of time and significant effort to master the framework. It is therefore natural to seek the means to facilitate the framework

usage process and make the framework user productive as soon as possible. Sparks et al. [139] note that one of the characteristics of reusing frameworks is fairly large amount of what they call *stereotypical code*. This code stems from the specific way the application uses framework, and is usually highly repetitive. This opens possibilities for utilizing code generators to speed up the application development and framework reuse. Indeed, this is often seen in practice, where, in order to make the process of using frameworks easier, frameworks are often accompanied by various tools. Other authors have also recognized the benefits of this approach. Lopes et al. in [94] and [93], recognize frameworks to be hard to understand and difficult to reuse. They note that, in order to minimize the effort required to reuse frameworks, application construction should be guided by tools that aid in framework instantiation, and preferably automate some of the steps in this process. Froehlich et al. [60] suggest using automation tools which should aid application developer in using frameworks. Coplien et al. [38] see here the role of software generators to provide parametrized variability of software. Johnson and Foote [81], associate software frameworks with toolkits - a collection of high level tools which allow framework users to interact with the framework in order to configure and construct new applications. According to Van Grup and Bosch [153], composition and configuration of components in black-box frameworks can be supported by tools and made easier to use the framework.

3. Method

With the regard to research being motivated by a practical problem, and the nature of final result of the research (innovative artifact - model and instantiation of software framework), design science has been chosen as a base research paradigm. Design is a process of creating applicable solutions for specified problem, and it has long been accepted as research paradigm in engineering disciplines [116]. Recently, it is also increasingly being used in the field of information systems [73]. Design science is a pragmatic paradigm with the goal of solving real problems by creating innovative artifacts. According to Hevner et al. [73] these artifacts can be characterized as: constructs, models, methods or instantiations. The scientific dimension of design science, other than creating artifacts, also requires generation of new knowledge through design and application of artifacts. Another determinant of design science is requirement for systematic approach and rigorous evaluation of the created artifacts. In order to guarantee design science is conducted in scientifically rigorous way we will follow the methodological framework for design science research proposed by Johannesson and Perjons [78]. The framework proposes 5-activity process, with each of the activities using appropriate methods and techniques.

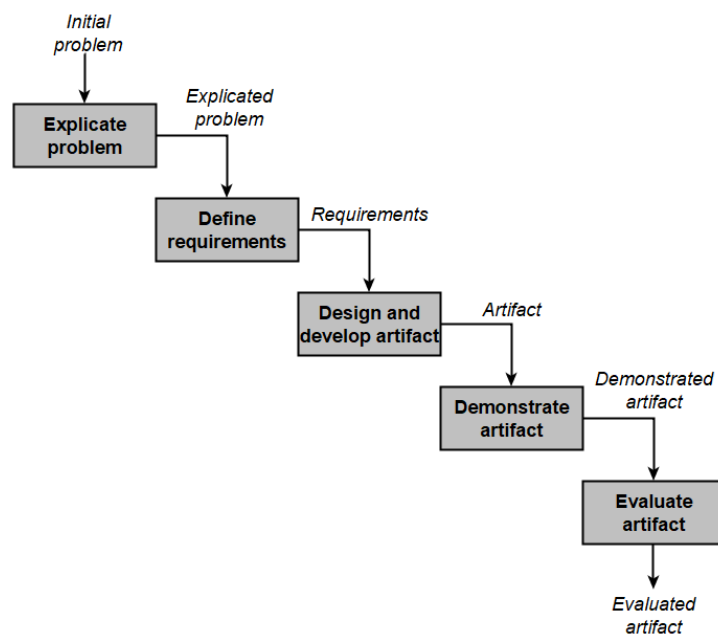


Figure 8: Design science methodological framework

3.1. Explicate problem

Management of reactive dependencies in object-oriented applications as a problem recognized by both practitioners and researchers will in this activity be clearly defined, placed in a context and shown as relevant. In order to achieve this, the problem goes through 3 sub activities. The first one (**Define Precisely**) is in charge of defining constructs needed for formulating problem and design space (e.g. the notion of reactive dependency, dependency graph, update process etc.), defining the problem itself and the scope of the research. In the second sub activity (**Position and Justify**) the problem will be placed in the context of both real scenarios in practice and relevant research. The third sub activity (**Find Root Causes**) will utilize the techniques of root cause analysis in order to systematize the main causes of the problem and determine which of them can be eliminated or mitigated by the REFRAME.

For the purpose of this activity a detailed literature review will be conducted in the fields of design patterns, reactive systems, event-driven systems, software frameworks and other related fields. This activity will provide the answers to the research questions **RQ1** and **RQ2**.

3.2. Define requirements

Explicated and clearly defined problem and its causes are prerequisite for the second activity. This activity aims at explicitly defining requirements which the artifact (REFRAME) has to fulfill, and is carried out in two parts. The first sub activity (**Outline Artifact**) outlines the base concepts of the future artifact, i.e. what kind of artifact is to be made, what are its most important characteristics. In our case there are two artifacts: *prescriptive model* of software framework (REFRAME) and the *instantiation* based on this very model. The second sub activity (**Elicit Requirements**) involves making the detailed specification of functional and non-functional requirements, which REFRAME needs to meet in order to eliminate or mitigate the causes of the problem identified in the previous activity. The main sources for requirements will be review of relevant literature, researcher's own experience, and the use of different techniques for creative thinking and prototyping.

While detailed requirements are still to be made, they will most certainly include the following: specification of reactive nodes, specification of reactive dependencies between reactive nodes, reactive dependencies between reactive nodes with the cardinality many-to-many, build-

ing the dependency graph, conducting update process, identification of possible parallel update etc. In addition to requirements for the core part of the REFRAME framework, requirements for some helper tools will also be specified. The code generation tool will enable developers to generate part of the boilerplate code required for managing reactive dependencies. The tool for visualization will aid developers in perceiving and understanding reactive dependencies by showing them visually in a form of directed dependency graph. The tool for analysis will also provide the basis for better understanding reactive dependencies using different graph analysis techniques. This activity will provide answers to the research question **RQ3** by providing the *Software Requirement Specification document*.

3.3. Design and develop artifact

In third activity the model and instantiation of REFRAME which meet the specified requirements and eliminate or mitigate identified problem are designed and developed. The first sub activity (**Imagine and Brainstorm**) collects the ideas for design and development of artifact. From the perspective of the model and instantiation of REFRAME this primarily refers to reasoning about structural and behavioral characteristics of the model, architectural decisions, quality properties (e.g. reusability, maintainability, customizability, performance etc.), implementation techniques, applied technology and other important characteristics of framework to be made. In the second sub activity (**Assess and Select**) collected ideas are assessed, and the set of ideas is selected as a starting point for design and development of REFRAME. From that starting point, iteratively, through incremental improvements we come to a satisfactory solution. In the third sub activity (**Sketch and Build**) we concretize selected ideas and we build the prescriptive model of REFRAME by defining its structure (using for example UML class diagram) and behavior (using for example UML activity, sequence and state diagram). Finally, based on the model the instantiation of REFRAME is developed in chosen OO technology (e.g. C# .NET or Java). The fourth sub activity (**Justify and Reflect**) aims at documenting design decisions, the reasons behind them, considered alternatives and compromises (design rationale). The whole third activity is characterized by the internal iterative process of reasoning about possible architectural, technological and implementation options, trying out these possible solutions, and choosing appropriate solutions (prototyping). This activity will provide answers to research question **RQ4**.

3.4. Demonstrate and evaluate artifact

Evaluation in design science research is a key activity, and has to be carefully planned and conducted, because it adds the scientific component to design. For that purpose evaluation strategy for this research has been designed, with its most part being conducted in the fourth (Demonstrate artifact) and fifth (Evaluate artifact) activity of methodological framework. In order to rigorously design evaluation strategy, different guidelines and frameworks have been reviewed and used. The efforts and gained experience in this activity also resulted in our own high-level guidelines for evaluation in design science [106].

As a base approach for designing evaluation strategy, the framework for evaluation in design science (FEDS) [154] is used, with its four steps being: (1) explicate the goals of evaluation, (2) choose the evaluation strategy or strategies, (3) determine the properties to evaluate, (4) design the individual evaluation episodes.

The **goals** (1. step) of our evaluation strategy are in accordance with the goals specified by FEDS framework, which implies scientific rigor to be assured. In the context of this research this means we need to show that the use of REFRAME improves the management of reactive dependencies in development of OO applications and that REFRAME is useful in real scenarios. In addition, we see no special ethical implications regarding evaluation strategy, and we see it as feasible with regard to resources available to researcher (time, money, participants...).

Out of four main **evaluation strategies** (2. step) which are proposed by FEDS framework, we chose *Technical Risk & Efficacy* as most appropriate base evaluation strategy. The reasons for this are namely the facts that REFRAME as software framework is large and complex socio-technical artifact and that the technical aspect is more emphasized. The *Technical Risk & Efficacy* strategy emphasizes artificial formative (early) and artificial summative (late) evaluation in order to establish efficacy of the artifact.

Total of 3 **evaluation properties** (3. step) have been chosen from the list proposed by Prat et al. [119] with regard to evaluation goals, the type of the artifact, evaluation strategy and the representation of these properties in similar studies.

Individual **evaluation episodes** (4. step) are in line with trajectory of evaluation strategy, which means that most formative and summative episodes are conducted in artificial setting. Each evaluation episode pairs the evaluation properties with appropriate evaluation methods.

Table 5: Chosen evaluation properties

Property	Description
Efficacy	Shows the artifact is working and that has potential to solve stated problem.
Technical feasibility	Answers the following questions: Is it possible and in which way to design and develop the artifact? What are the capabilities and constraints of the artifact? Which technical aspects should be considered? Which external aspects should be considered (e.g. technology maturity, alternatives, community size, popularity, support, tools etc.)?
Usefulness	Shows the effect of artifact's use as perceived by the users.

The process of choosing evaluation methods was guided by taxonomy of evaluation methods in design science proposed by Prat et al. [119].

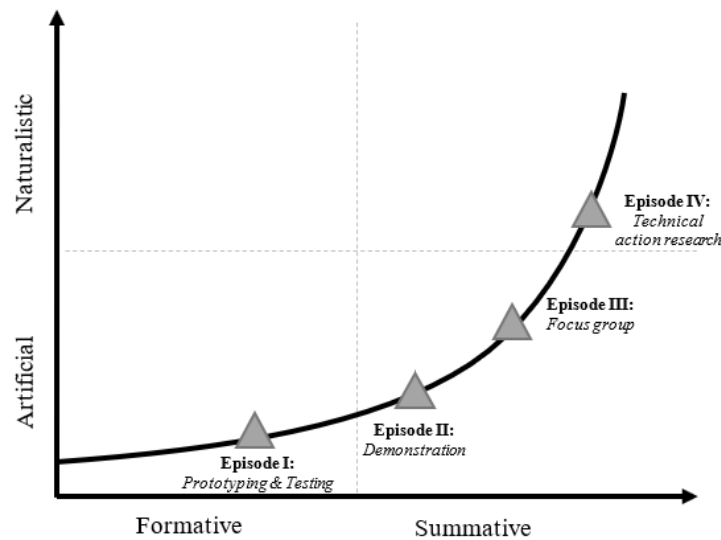


Figure 9: Evaluation strategy

Episode I - Prototyping and testing

The first evaluation episode is formative, which means its purpose is obtaining the feedback, trying out and testing alternative ideas and approaches, and incremental improvement of REFRAME. It is conducted in artificial, laboratory setting, and is suitable for early identification of technological and efficacy risks. Therefore in this episode *technical feasibility* and potential *efficacy* will be evaluated. Evaluation methods used in this episode include prototyping, testing and informed argument. Sommerville [137] describes prototyping as building initial version of software with the purpose of demonstrating concepts, trying out design options, getting additional information about the problem and possible solutions. The contribution of prototyping in

evaluation is acknowledged by numerous authors, e.g. [35], [115], [138] etc.

Testing has two distinct goals [137]: demonstrate that software meets the functional and non-functional requirements; and to discover errors, incorrect or unwanted behavior, incorrect results, performance problems etc. As well as prototyping, the testing is also acknowledged as relevant evaluation method, e.g. [73], [131]. Because of its formative purpose, this evaluation episode is conducted largely during the third activity of the methodological framework. Consequently, it contributes to answering research question **RQ4**.

Episode II - Demonstration

The purpose of the second evaluation episode is summative, i.e. it is conducted with the purpose of assessing overall value and utility of the developed artifact. It is placed within 4. activity of methodological framework. As well as the first evaluation episode, it is conducted in artificial, laboratory setting, and it evaluates *technical feasibility* and *efficacy*. However, contrary to the first episode, here the evaluation is done summatively.

The evaluation method bears the same name as the episode - Demonstration, which according to Prat et al. [119] is the most frequently used evaluation method in design science. In this episode we will define several illustrative scenarios for management of reactive dependencies in development of OO applications. These scenarios will be implemented using REFRAME, which will demonstrate that the artifact with required characteristics can be built (technical feasibility), and that the artifact has the potential to solve the problem (efficacy). This evaluation episode contributes to answering research questions **RQ4** and **RQ5**.

Episode III - Focus group

Focus group is one of the most popular qualitative methods. In design science it is frequently used in the processes of problem explication, requirement specification, and artifact evaluation [78]. Tremblay et al. [150] in their article adapted focus group method for use in design science research, and suggested its use for artifact improvement (explorative focus group) and for artifact evaluation (confirmatory focus group). Focus groups have also been proposed as techniques for requirements elicitation and evaluation in software engineering [89]. In this episode with the purpose of evaluating REFRAME's usefulness (research question **RQ6**) the confirmatory focus group shall be conducted according to protocol created according to guidelines from [150].

Episode IV - Technical action research

Technical action research (TAR) is proposed by Wieringa and Morali [157] as one of the methods for evaluation of artifacts created through design science. The main intent of TAR is to transfer the artifact from laboratory setting to practice, and to evaluate the artifact according to these more realistic conditions. It should be emphasized that in TAR the researcher participates in 3 logically separated roles: in artifact design and development, in application of artifact on real problem, and lastly in answering knowledge questions. In the context of this research, the researcher has the main role in designing and developing REFRAME framework (this is done prior to TAR within 3rd activity of methodological framework). Application of artifact on real problem involves using REFRAME to develop few modules of real software application (KI Expert Plus). Besides researcher, in this step two more KI Expert Plus developers will participate. After applying REFRAME to develop real software application the researcher will collect the feedback from participants on their experiences in using REFRAME, its usefulness and possible improvements. Finally, researcher has a leading role in answering knowledge questions about REFRAME. Here, the collected qualitative data is analyzed in order to provide answer to research question **RQ6**. TAR is going to take place as the final (IV) evaluation episode within 5. activity of methodological framework. Like the III episode it will also evaluate usefulness, but in the more realistic conditions. TAR is going to be conducted according to the protocol created by following the guidelines from [158].

4. Explicate Problem

Explication takes place from the very first moment the researcher becomes aware of the problem. At first, this activity may be implicit in the researcher's mind, sometimes even unconsciously. In later stages, explicit and systematic activities are performed in order to better formulate the problem, and investigate its causes and effects. Large part of the efforts of problem explication are traditionally captured in Introduction (Chapter 1) and Literature review (Chapter 2). In this chapter these efforts will be summarized, supplemented and structured to fit the chosen methodological framework.

4.1. Define Precisely

When developing OO applications software developer creates objects (unit of decomposition in OO programming) which collaborate in order to realize the purpose of application. As a result of this collaboration, dependencies between objects are formed. Sometimes these dependencies are of a reactive nature, which means that when one object changes its state or invokes some method, its dependent objects need to react accordingly by updating their own state or invoking their own methods.

Simple example of this can be seen in Figure 10. Attribute Z (*ClassB*) is defined as a function f of attributes X and Y (*ClassA*), effectively making Z dependent on its parameters X and Y . When either X or Y change their values, in order for it to still be valid, we have to update Z 's value. Dependencies such as the ones between Z and X , and Z and Y we refer to as *reactive dependencies* and the attributes X , Y and Z with respect to these dependencies we refer to as *reactive nodes*. As presented, reactive nodes may become interconnected with reactive dependencies and construct graph-like structures called *dependency graphs*.

Although fairly simple, the example in Figure 10 witnesses how a developer has to carefully consider what attributes has to update after some change, and also in what order this update has to be performed. If, for example, attribute X changes its value, we can see from dependency graph that attributes Z and Q as dependents of X need to be updated. But, since Q is also dependent on Z it is important to first update attribute Z and only then attribute Q . Otherwise, Q would be updated with old Z 's value. Also, Q should not be updated twice, i.e. once because of its dependence on X and once for its dependence on Z . The process of bringing dependency

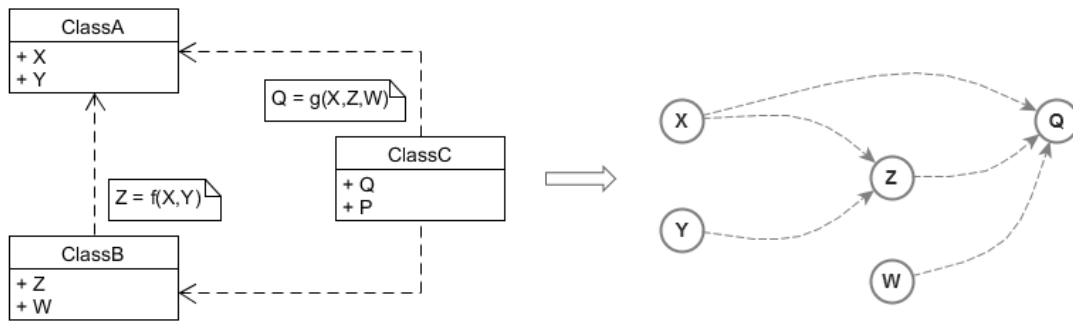


Figure 10: Example of reactive dependencies

graph and all of its affected elements (nodes) into correct state after a change has occurred we refer to as *dependency graph update process*.

Object-oriented paradigm does not provide native support for specifying reactive dependencies, forming dependency graphs and performing update process. Instead, it rather relies on developers manually implementing aforementioned Observer or similar design pattern. This does not only involve writing a lot of so called "boilerplate" code in order to enable this kind of reactive behavior, but it also requires careful construction of dependency graph and manual management of update process. When dependency graphs start becoming larger and more complex (more reactive nodes and more reactive dependencies), such as the one illustrated in Figure 11, manual management of reactive dependency graphs proves to be significantly more challenging, time-consuming and error-prone. At some point, it may become impossible to understand and manage vast web of interwoven reactive dependencies.

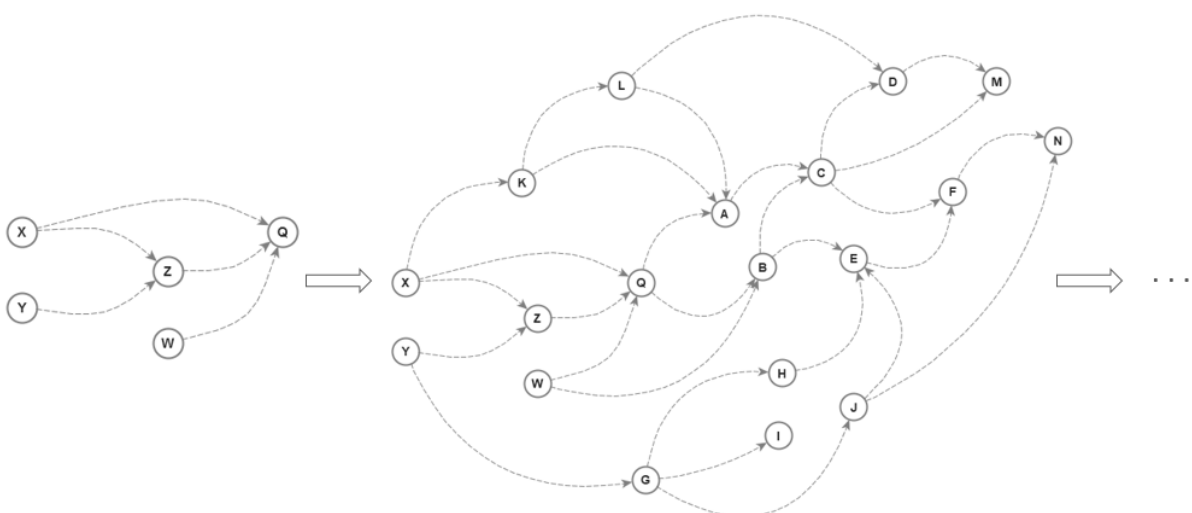


Figure 11: Example of more complex dependency graph

4.2. Position and Justify

Management of reactive dependencies has been identified in Introduction (chapter 1) as a practical problem, relevant to practice the author has been involved in, and also to large number of existing and yet to be developed OO applications. Literature review in Chapter 2 showed that a number of design and implementation-level solutions were proposed to tackle some aspect of the problem. These propositions came from both practitioners and researchers arising from different backgrounds. Further proof that this problem is pervasive in development of OO applications and therefore significant for practitioners, is the fact that Observer [61] - design pattern intended to deal with this particular problem, is one of the most well-known and most represented OO design patterns. Literature review in Chapter 2 also showed the scientific significance of this problem, as it is represented as a topic in the research on OO design patterns, event-driven programming, aspect-oriented programming, reactive programming, constraint programming, and other similar fields. The stated demonstrates that managing reactive dependencies in OO applications as a research problem holds both practical and scientific relevance, which is one of the requirements set by Design science.

4.3. Find Root Causes

In order to model problem space in a more detail, problem tree analysis has been performed (Figure 12). This allowed us to separate the effects and symptoms of the problem from the causes of the problem. As a focal problem we specified that the management of reactive dependencies in OO applications is difficult and error-prone. This results in some adverse effects on the development process, quality of code and the end product.

Traditional handling reactive dependencies in a form of Observer pattern results in significant amount of so called "boilerplate" code which tangles with the core functionalities and polluting the classes. Furthermore, this may introduce negative impact on overall code quality, making the code harder to maintain and understand, especially for newly introduced team members. Finally, this may result in worsening the quality of end software product itself.

The difficulty of handling reactive dependencies in OO applications necessarily results in developers making more errors. Not being able to grasp sheer number of dependencies and their relation, developers may fail to implement all required updates, which results in objects'

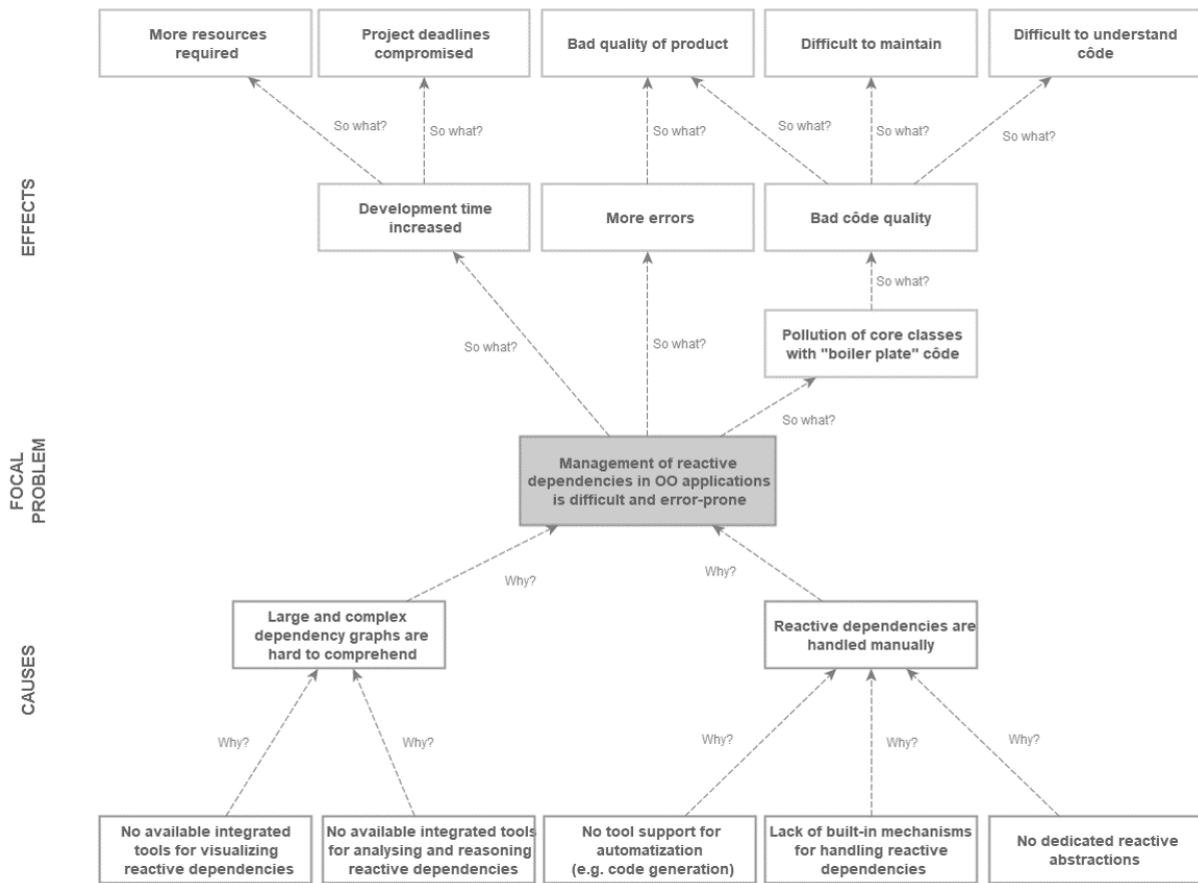


Figure 12: Problem tree

state being inconsistent. To avoid this problem, developers may go to another extreme - to act defensively and implement redundant updates. Over time, these redundant updates tend to accumulate, which at some point can make the performance problems so severe, the application may become unusable. One of the frequent issues are so called "glitches" - temporary inconsistencies caused by performing updates in a wrong order. This could happen when in example from Figure 10, we would perform update sequence in following order $X- > Q- > Z- > Q$ rather than $X- > Z- > Q$. Another possible issue is occurrence of circular dependencies, which can cause infinite loops, wrong results and program crashes.

Difficulties in handling reactive dependencies can mean more time is needed for developing software product, which project management may try to compensate by increasing the efforts and spending more resources. At worst, project deadlines may be compromised.

After we analyzed the symptoms (effects) of focal problem, it remains to investigate what exactly causes the problem. We can identify two high level reasons. The first is that reactive dependencies form dependency graphs, which are inherently complex structures with potentially

large number of elements and relationships between them. Human brain, due to its natural limitations, simply has hard time comprehending and understanding such structures. When faced with such problems people resort to the use of different tools and techniques which could help them to overcome limitations in their cognitive abilities and gain better understanding. Unfortunately, in this case there are no available integrated tools to use, which would support developer by e.g. visualizing, analyzing and reasoning about dependency graphs. The second high level reason is that, although the management of reactive dependencies is difficult, error-prone, time consuming and involves significant amount of "boilerplate" code, it is still mostly done manually. This is caused by the lack of dedicated reactive abstractions and mechanisms for specifying reactive dependencies, forming dependency graphs and appropriately handling update process of dependency graphs. In addition, the lack of integrated tools and support (e.g. code generators, modelers, debuggers, etc.) which would further automatize and facilitate the management of reactive dependencies also contributes to this undesirable state of affairs.

Now that we obtained firm understanding of the research problem, its effects and causes, we can start transitioning towards proposed solution. In order to model potential solution space we will reverse negative statements from a problem tree (Figure 12) and with the resulting positive statements construct the solution tree (Figure 13).

We can see that the central goal is to see management of reactive dependencies in OO applications improved. Achieving this goal would cause some positive outcomes. Firstly, it would result in "cleaner" core classes, easier to maintain and understand, and better overall quality of code and the product. Improved process of managing reactive dependencies would also mean less errors and improved overall development process, leading to decreased development time and less required resources.

In order to achieve the set goal, we need to offer two fundamental things: make dependency graphs easier to comprehend, and increase the level of automation in handling reactive dependencies. Part of the solution to help software developers in better understanding dependency graphs, could involve tools for visualization of reactive dependencies integrated into development environment. Additionally, integrated tools for automatic analysis and reasoning on dependency graphs could also be provided. The second part of the proposed solution could address automation issue by providing dedicated abstractions for expressing reactive dependencies. Once we have defined abstractions for reactive dependencies, the solution can provide

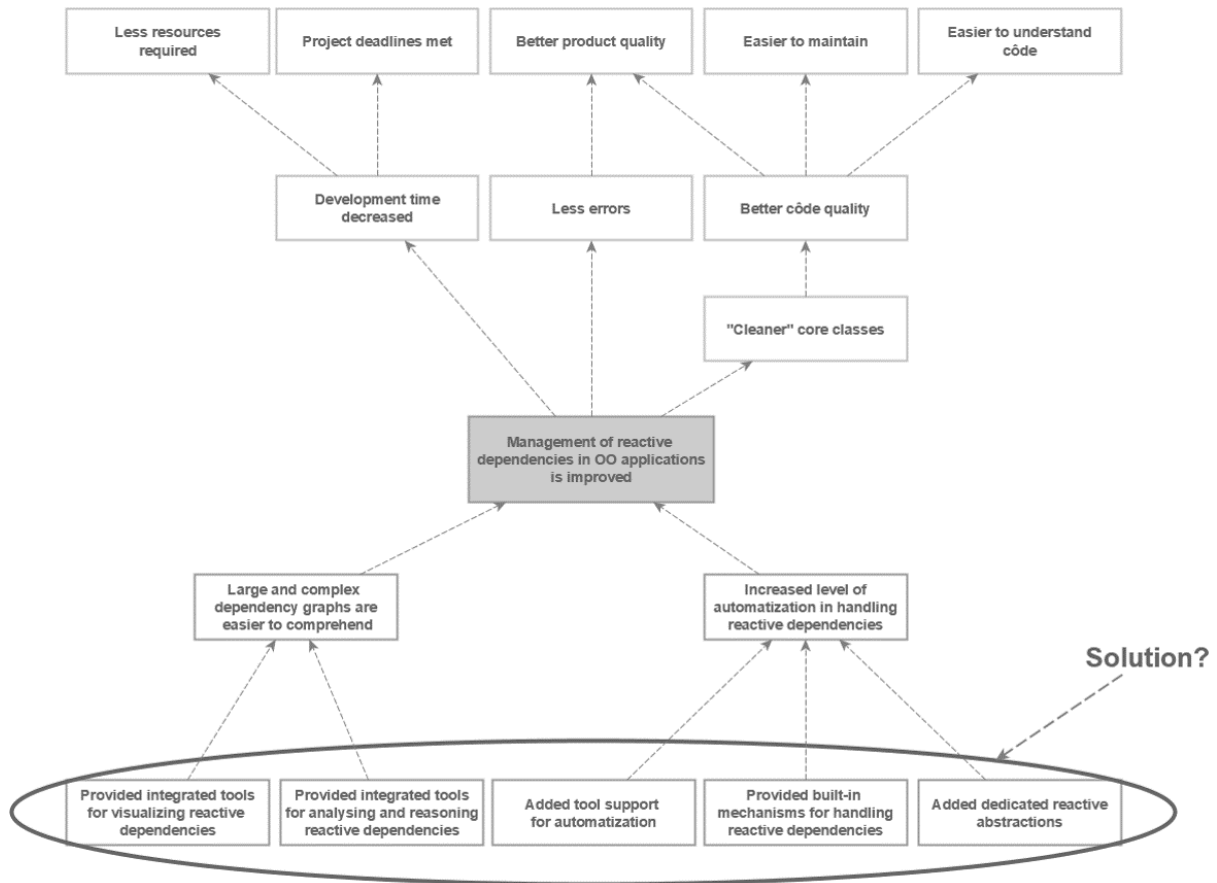


Figure 13: Solution tree

mechanisms for specifying reactive dependencies, building dependency graphs from them, performing update process etc. Providing abstractions and mechanisms for managing reactive dependencies significantly decreases amount of boilerplate code involved in this task, however, further automation may be achieved by providing integrated tools for code generation.

5. Define Requirements

5.1. Outline artifact

In previous chapter (Explicate problem) we analyzed the problem, and more importantly identified its possible causes. This allowed us to reason about possible solution to the problem, and to identify its general characteristics and concepts. By looking at the solution tree we see that the solution, i.e. design science artifact to be made, should be based on providing application developers with dedicated abstractions and operations to handle reactive dependencies, accompanied also by a set of tools which facilitate the use of artifact. In this way, domain knowledge and the problem-solving process engraved into the artifact at design time, is through this artifact made available to application developers for repeated use (i.e. reuse). As already stated in previous chapter, one of the software reuse techniques that are fit to represent this kind of artifacts are software frameworks. They allow capturing abstractions from problem domain, imposing application design, and providing implementations and tools for developers to use in application development.

Going back at solution tree we can devise a set of high-level requirements that the framework should fulfill.

1. *Framework shall provide dedicated abstractions for expressing concepts related to reactive dependencies in application development.*
2. *Framework shall provide built-in mechanisms for performing basic required operations related to reactive dependencies.*
3. *Framework shall automate parts of the reactive dependencies' handling process.*
4. *Framework shall provide means to visualize reactive dependencies.*
5. *Framework shall provide aids for analyzing reactive dependencies.*

In addition to this high-level statement of what framework is going to offer to application developers, we can further outline the framework by positioning it in software frameworks' ecosystem. For this purpose we will utilize framework classification reported in section 2.2.3. It should be noted, however, that framework's characteristics do not always take the form of one particular discrete framework type. Rather, the framework may have characteristics of multiple types, or may fit somewhere "in between".

The fact that the framework is novel and that it will be used fairly short amount of time during dissertation development, may result in significant changes and extensions to be required after the dissertation, when more experience in using the framework is gained. In order to allow the maturing process of the framework, flexibility needs to be assured, so inheritance as a *white-box* extension mechanism will be allowed. However, with ease of use, better productivity and control over the framework in mind, it can be expected for composition as a *black-box* extension mechanisms to be applied where possible. That being said, from the point of **extensibility**, our framework will have both *white-box* and *black-box* characteristics, and could thus be described as *gray-box* framework. Although we do not feel that, according to **scope** criteria, any of the suggested framework types is perfectly describing what our framework is trying to reuse, the closest fit in our opinion are the *system infrastructure frameworks*. They aim at simplifying development of application infrastructure, which we believe a proper management of reactive dependencies can do. *Object-oriented programming* will be taken as a lead **approach** in building the framework, although some ideas concerned with handling reactive dependencies will be taken also from other paradigms such as *reactive programming* and *aspect-oriented programming*. By being a completely novel framework, from the **standardization** point of view, our framework can be considered as an *unstandardized* framework. It will focus on supporting fairly specific and narrow domain, so in terms of **granularity** it will be best described as a *fine-grained framework*. According to **license** criteria, our framework will be *free* to use, and also *open-source* for other developers to be able to modify it and extend it. Regarding the **format**, it will be available in both *binary* and *source code* version. Binary version is aimed at those who just want to use the framework, while source-code version will be available for those who want to better grasp the internals of the framework and possibly make adaptations outside of those possible through white-box and black-box hot spots.

Implemented software frameworks are working systems that can be applied in application development. Therefore, from the *design science* point of view, they can be considered as an *instantiation* artifact. However, although the framework implementation may be the ultimate software artifact resulted from framework development process, other artifacts also arise from this process. Some of them can also be considered as a valid design science artifacts. Such is the case with design model of the framework, which prescribes, in a higher and more abstract level, the structure and behavior that the future framework instantiation artifact will possess.

Such model corresponds to design science *model* artifact.

That being said, the main resulting design science artifacts in this dissertation are *prescriptive model* and *instantiation* of software framework for managing reactive dependencies in object-oriented applications. It should be noted however, that there is an unfortunate clash of terminology in design science and software frameworks literature that should be clarified. In design science, *instantiation* is a type of artifact which represents implemented and working system, so instantiation in this context refers to implemented and working software framework. In software frameworks literature, however, instantiation is used either as a verb, to refer to a process of using a framework to build end-user applications, or as a noun to refer to an end-user application built using the framework. When using this expression in the dissertation, we will precisely indicate whether we talk about the artifact type or a process.

It is a common practice to give a name to artifacts being built, so in the rest of the dissertation our framework will be referred to as **REFRAME** (coming from REactive FRAMEwork). The next section will bring more detailed requirements that are going to be placed in front of REFRAME.

5.2. Elicit requirements (SRS)

In previous section we clearly stated that the artifact to be produced will be software framework named REFRAME, and we outlined its base characteristics. This section aims at eliciting detailed requirements for REFRAME and wrapping them into a specification document. Following subsections follow adapted document structure and practices from IEEE 830-1998 [17], and represent REFRAME's Software Requirement Specification (SRS) document.

5.2.1. Introduction

Purpose

This document represents a Software Requirement Specification (SRS) containing elaborate description of required functionalities and characteristics of software framework named REFRAME. This framework is the target artifact and the final output of the research project carried out as a design science research. The second activity of the methodological framework [78] we used for conducting design science research deals with defining requirements of the aforementioned artifact, and the SRS covered in this document is the main output of this

activity.

Primarily, the SRS will serve as a central place for researcher to document the core features and characteristics of the proposed software framework. Then, the SRS will act as a discussion ground and communication medium for researcher and other stakeholders (mentors, research committee, and other interested practitioners and researchers) for expressing their opinions and interest on the proposed features and characteristics, as well as for assessing its originality and significance. Lastly, the SRS document will serve as a contract which the researcher will need to fulfill in successive activities in order to deliver the proposed framework. The requirements in the document will also help in designing test cases in testing phase.

Scope

As outlined in previous section, REFRAME's scope encompasses providing application developers with proper abstractions and functionality for handling reactive dependencies in OO application development. In addition, REFRAME will contain set of tools which will aid application developers in using the framework.

We can describe *reactive dependency* as an expression $y = f(x)$, where y and x are values (usually describing the state of the object), and f (computation or any other sort of transformation) denotes dependency y has on x . However, what makes f a reactive dependency is the characteristic that whenever x changes, the value of y as its dependent value must automatically update in order for dependency to be consistent. By using dedicated abstractions built into the framework, application developers will be able to implement individual reactive dependencies found in application domain. However, reactive dependencies in application domains are seldom found as single isolated cases. Rather, they form larger structures with arbitrary number of reactive dependencies chained together. This is why REFRAME will allow application developers to chain individual reactive dependencies into larger and possibly very perplexed graph-like dependency structures, which we refer to as *reactive dependency graphs*. Mechanisms implemented in the framework will then be in charge of identifying and automating updates in these dependency graphs in order to ensure their overall consistency. In addition, REFRAME will be accompanied by three tools in order to facilitate the use of the framework and the process of handling reactive dependencies. One of the tools will generate parts of boilerplate code in order to support framework instantiation process and increase the level of automation. The other two tools will serve as a means to help application developers to understand and visualize

dependency graphs, which due to their sheer size and complexity can be hard to grasp. In this way, proposed framework will provide improvements over currently dominant approaches for managing reactive dependencies using Observer and other similar design patterns. REFRAME aims to make the management of reactive dependencies within object-oriented applications easier, faster, with less *code* and errors, and more transparent.

For the sake of clarifying REFRAME's scope, it should be noted that *reactive dependencies* as defined above, are similar with signals and events described in areas of reactive programming, event-driven programming and aspect-oriented programming, as well as with the overall intent of different design patterns used in object-oriented programming. However, the scope of the proposed framework is to work with reactive dependencies in object-oriented setting only in a form of events, by providing improvements over currently dominant approaches inspired by Observer design pattern. The framework will not consider signals, which are abstractions for addressing temporal aspects of reactive systems, or working with streams of data/events. Also, the framework will not consider distributed systems, performance or security critical systems.

Overview

Up until now, we delineated the purpose and the audience of the SRS document itself, and introduced the scope and base characteristics of the target artifact - software framework REFRAME. The following, second section of the SRS, provides overall description of the proposed framework, and lays the foundations for defining specific requirements. The third section details the functional and non-functional requirements for the proposed framework in order to enable design, development and testing of the framework. Here each requirement is separately addressed and elaborated.

5.2.2. Overall Description

Product Perspective

REFRAME is a software framework which is going to be built using object-oriented programming language, and will necessarily reside in chosen technology's specific ecosystem. There are several elements in this ecosystem that REFRAME will relate to and collaborate with, namely: *end-user applications; integrated development environments (IDE); base frameworks, libraries and services; other custom frameworks and libraries; and runtime environment.*

By being software framework, REFRAME's code and design aspects, as well as its tools

will be used during development of end-user applications (design time). The resulting applications cannot be executed without REFRAME, so they are also bound at run-time. It is therefore justified to say that most important REFRAME's interactions are with *end-user applications* developed using it. During application development REFRAME's features and tools are accessed through *IDE*, so this is another system REFRAME needs to collaborate and integrate with. Most OO programming ecosystems include some sort of *fundamental frameworks and libraries*, which offer common, general purpose abstractions in order to make a shift from the bare constructs of OO programming language. REFRAME will reuse abstractions from these frameworks where possible and appropriate, and add a layer of higher level abstractions for handling reactive dependencies. In addition, REFRAME may reuse other, *custom frameworks and libraries* in order to fill the gap left by base frameworks and libraries, or in order to additionally raise the level of abstraction. This can be seen as a REFRAME participating in framework layering/stackin process.

Both framework and application code is developed and executed on top of *runtime environment*, which abstracts low-level operation system services and hardware specifics, and makes the development process easier. Among many things, runtime environment provides and/or supports compilers, debuggers, profilers and other tools. Graphical representation of *system interfaces* between REFRAME and other mentioned systems in its environment are depicted in Figure 14.

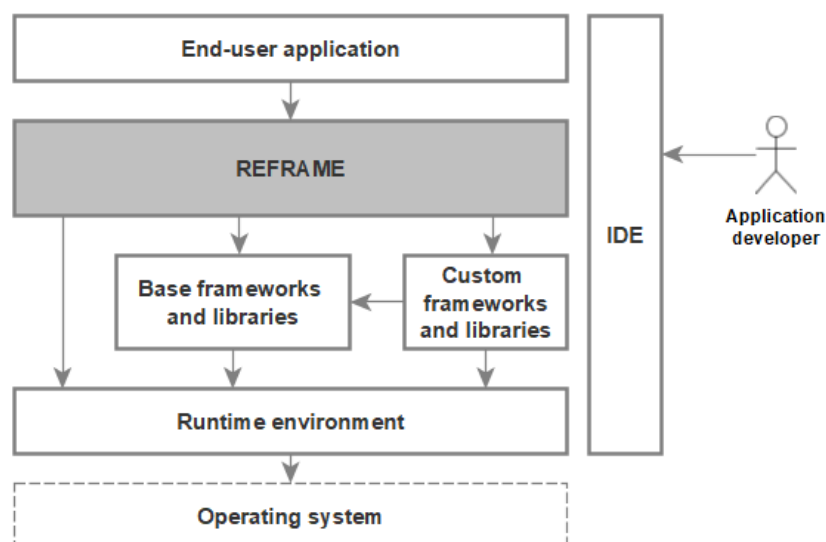


Figure 14: Framework perspective

Primary *user interface* through which the framework users (i.e. application developers) can interact with REFRAME is provided by existing IDE. In order for REFRAME's features and services to be available, particular application project should reference the framework. After that, framework user can invoke REFRAME's core features and services by writing programming statements in IDE. Also through IDE, but in a more visual style, framework user will be able to interact with REFRAME's tools (e.g. in order to display parts of dependency graph), which also have to be beforehand plugged into IDE.

REFRAME is a pure software system, it does not directly involve communication with hardware, and therefore does not specify any *hardware interfaces*. As with other similar frameworks and software systems, this part is abstracted by runtime environment and operating system. In addition, the framework also does not rely on any specific *communication interface*.

This leaves REFRAME directly interacting with exclusively software systems through *software interfaces*. However, these interfaces are mostly implicit, i.e. they are enabled by the underlying technology. For example, interaction between REFRAME and IDE is allowed by the fact that REFRAME is written in technology which IDE understands and is compatible with. However, while no explicit effort by framework developer needs to be invested for IDE to support framework's abstractions, integration of REFRAME's tools into the IDE will require explicit attention.

The main activity of the REFRAME happens during runtime of end-user application which uses it, and depending on the size and complexity of dependency graph some *constraints* on primary memory may apply. This can be especially the case if end-user application which uses REFRAME is deployed on mobile or embedded systems with limited hardware resources. Secondary memory may be used by REFRAME to store reactive dependency specifications and other configuration data. However, here we are looking at fairly small amount of data (measured in kilobytes or megabytes at most), so this should not pose a problem to any modern system.

REFRAME's perspective shown in Figure 14 is a generic one, fairly ignorant to specific technology. However, while this might be enough for framework's conceptual model, which among other things abstracts some of the technological specifics, framework implementation is tied to particular technology. We chose .NET programming platform for REFRAME implementation, due to it being one of the most popular and powerful platforms, as well as it being the platform the author is most acquainted with. This includes using C# object-oriented pro-

gramming language to develop REFRAME through the Visual Studio IDE. REFRAME Tools are also going to be provided through Visual Studio. .NET platform has a number of huge frameworks and libraries which are going to be utilized by REFRAME where appropriate and needed. Runtime environment supporting all this is called *common language runtime* (CLR). Finally, end-user applications which are to use REFRAME also have to be developed using .NET platform in order to be compatible with the framework.

Product Functions

The main REFRAME's function is to facilitate the handling of reactive dependencies. Thus, if we take bottom up approach, the first thing REFRAME has to provide is a means to construct individual reactive dependencies. This would include specifying two important things: two participants involved in reactive dependency, and their respective roles in this reactive dependency. If we were to blindly follow previous descriptions of reactive dependency, suitable participants in the context of OO paradigm would be *properties* - class members which capture the state of the object. However, our intention is to allow methods, i.e. class members which capture the behavior of the object, to also engage in reactive dependencies. Whatever the type of participant, its role in reactive dependency determines whether the participant is dependent one, or the one at which other participant depends on.

When end-user application is run, REFRAME steps-in to process individual reactive dependencies, and ties them to form larger structures - dependency graphs. These can hold arbitrary number of reactive dependencies. New reactive dependencies can be added and existing removed from dependency graph. Also, when dependency graph changes, REFRAME checks its validity and existence of possible problems (e.g. circular dependencies), and warns the framework user about them.

Dependency graph is brought to inconsistent state either by change in its structure itself (e.g. by adding or removing reactive dependencies) or by activating reactive dependency (e.g. when the change is triggered in participant). REFRAME's responsibility is then to analyze dependency graph, validate it, and determine dependent participants that are affected by the change. If dependency graph is valid, the exact order of propagation of changes through the graph, i.e. exact order of updates of participants, is determined. Update of individual participants affected by the change is achieved by running operations associated with each participant respectively.

Although abstractions and mechanisms built into REFRAME will offer both design and

code reuse, handling reactive dependencies requires writing certain amount of *boilerplate* code. Therefore, REFRAME provides a code generator tool, which will generate the stereotypical parts of this code, in order to speed-up the process. This primarily involves inserting pieces of code which will be responsible for constructing reactive dependencies, triggering changes and possibly other infrastructure and configuration code required for handling reactive dependencies.

Since reactive dependencies may form fairly large and complex dependency graphs, framework users may have hard time comprehending and understanding them. In order to aid framework user in understanding complex dependency graphs, REFRAME provides tool for visualizing the current state of the entire dependency graph or only some contextual part of it (e.g. only participants which are dependent on some other participant, or only predecessor participants which influence some particular dependent participants). Similarly, a tool of analyzing dependency graphs will offer basis for reasoning about constructed reactive dependencies.

Graphs are a suitable way to express individual reactive dependencies as well as more complex dependency structures. In mathematics, graph theory studies graphs as a mathematical structures which model pairs of related objects. We will borrow some concepts from graph theory to form a vocabulary of constructs for defining framework requirements.

Graph can be defined as an order pair $G = (V, E)$, where V is a set of *vertices* or *nodes*, and E is set of *edges* or *arcs*. Each edge e from E associates two vertices x and y from V , so we can define edge e as a pair of vertices (x, y) . Since reactive dependencies do imply direction, edges e from set E are directed, i.e. the pairs of vertices (x, y) are ordered pairs, where x is called predecessor and y successor. Therefore, in the context of reactive dependencies we are talking about *directed* graphs. When particular REFRAME feature is specified, related definitions and clarifications of main constructs from REFRAME (e.g. reactive node, reactive dependency, dependency graph) will be expressed with the help of the concepts from graph theory.

User Characteristics

REFRAME framework is intended to be used by the software developer population when developing object-oriented end-user applications. While the framework itself requires no particular education level, developer is expected to have basic skills in software development (writing code, debugging, using IDEs...), and some experience in framework-based object-oriented programming. These are by no means restrictive requirements, after all, most of today's software

development has these characteristics. Therefore, we believe REFRAME to be suitable for wide range of both novice and experienced developers.

Constraints

REFRAME is open to use, modify and extend by interested developers. Its model is designed to suit object-oriented programming paradigm, but offers flexibility in terms of particular implementation decisions. However, concrete instance of REFRAME is not technology agnostic but implemented in .NET ecosystem. This places constraints on the use of REFRAME and mandates end-user applications to match chosen technology. This means that the end-user application and the instance of REFRAME have to be technology compatible. If this is not the case, developer needs to either switch the technology of end-user application to the one REFRAME instance is implemented in, or one needs to re-implement REFRAME's model in desired technology. When re-implementing the model it should be noted that availability and the characteristics of certain implementation options may significantly vary in different technologies. It is also worth to note, that REFRAME will offer no security or performance warranties, especially in this early stages of development and use, so using it in any kind of critical applications should be done with utmost caution.

Assumptions and Dependencies

REFRAME assumes the availability of programming language and environment in which the framework will be used. That said, if parts of the technology REFRAME depends on changes in a way that affects requirements stated in this document, requirements will have to adapt. This effectively means that if for example externals of .NET framework and Visual Studio change, this may affect REFRAME. Also, the requirements for REFRAME will be discussed with other stakeholders, and the changes may arise in order to satisfy their suggestions.

Apportioning of Requirements

As is the case with other frameworks, and software in general, REFRAME is a dynamic, ever-changing system. The initial version of the framework, which is a result of this this dissertation, is by no means the final version. Arbitrary number of future releases will be the result of framework's continuous evolution and maintenance efforts, driven by *corrective*, *perfective* and *adaptive* goals. Some of the features and characteristics of REFRAME not planned for this

version, but considered for future releases, are as follows:

- Tool for debugging reactive dependencies.
- Tool for visual specification and modeling reactive dependencies.
- Additional means of automating the process of handling reactive dependencies (especially specifying reactive dependencies).
- Additional predefined analyses and visualizations.
- User-created, ad-hoc analyses.
- Support for unit test generation based on reactive dependencies.

5.2.3. Specific Requirements

External Interface Requirements

As is already mentioned in subsection 5.2.2, REFRAME's external interfaces are mostly implicit. Main user interface which will provide access to REFRAME's features is Visual Studio IDE. Through it, when developing end-user applications, framework user can reference compiled files containing REFRAME's components and then proceed to instantiate and invoke framework features by writing code in IDE's code editor. REFRAME's tools will also integrate into Visual Studio as IDE extensions, and in a more visual style provide framework users ability to generate parts of boilerplate code, visualize and analyze structure of dependency graphs.

REFRAME's interfaces with other systems are inflicted by the chosen .NET platform. Interface and interaction between REFRAME and end-user application is based on REFRAME's compiled files being referenced in end-user application, and REFRAME's components being used, configured and customized within end-user application's code in order to help realize some of application's functionalities. The necessary prerequisite for this to be possible, is that both end-user application and REFRAME are built using .NET platform. The same prerequisite stands true for REFRAME to be compatible with Visual Studio IDE.

In order for REFRAME's tools to be tightly integrated with Visual Studio IDE, they have to be developed as Visual Studio IDE Extensions using Visual Studio SDK (Software Development Kit) or some other types of developer aids available in Visual Studio (e.g. code snippets, quick actions). Extensions are add-ons that allow developers to customize and enhance Visual

Studio IDE by adding new features or tools. In our case, the aim is to support the process of using REFRAME in application development.

The nature of interfaces between REFRAME and .NET frameworks, and any other custom frameworks and libraries, is similar with the one with end-user applications. However, in this case, REFRAME references other frameworks' compiled files and uses their features to realize some particular functionality. Again, the prerequisite for this is that REFRAME and other frameworks are .NET compatible.

Other than these mentioned interfaces, REFRAME requires no additional external interaction with other systems. Any possible interaction with hardware or communication protocol and devices are abstracted by CLR and operating system.

System Features

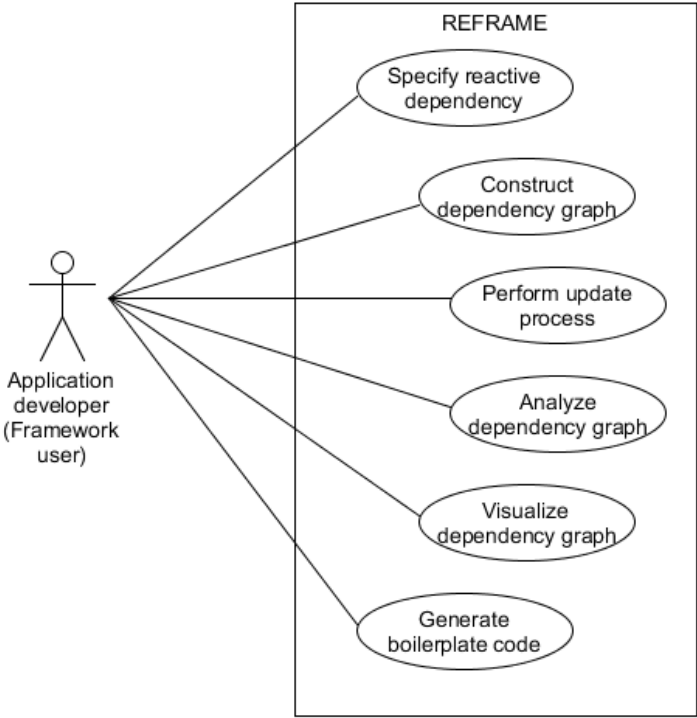


Figure 15: Use case diagram describing REFRAME's features

Feature 1: Specify reactive dependency

Specifying individual reactive dependencies is REFRAME's central and fundamental feature - a prerequisite to all other features. As already described, it allows us to set up a special association between two participants, where one participant depends and reacts on what is happening to the other participant.

Before we are able to specify reactive dependencies, we have to specify the construct representing the very participants involved in dependencies. We will start by naming these participants (*reactive nodes*), which is consistent with terminology from graph theory.

- **Reactive node** n is an entity which encapsulates state/behavior of an object designated to participate in reactive dependencies. It is represented by a node (vertex) in a graph.

Now that we have defined *reactive node* construct, we can also describe reactive dependency as an association of exactly two different reactive nodes, where one reactive node acts as a predecessor node (event emitter), while the other takes the role of successor node (event receiver). Reactive node's specific role as predecessor or successor node, is relative to its responsibility in particular reactive dependency. Since reactive node can participate in multiple reactive dependencies, in different reactive dependencies it can take different specific roles. So in one reactive dependency it can represent a predecessor node, while in other it can represent a successor node. The basic constructs related to reactive dependencies are defined as follows:

- **Predecessor node** is a reactive node p which acts as a *predecessor* to some other node s in reactive dependency d . In a graph it is represented as a starting node of an edge e .
- **Successor node** is a reactive node s which acts as *successor* to some other node p in reactive dependency d . In a graph it is represented as an ending node of an edge e .
- **Reactive dependency** d is an ordered pair (p, s) , where p acts as a predecessor node and s acts as a successor node. Reactive dependency d has a characteristic that whenever predecessor node p changes, successor node s automatically updates. Reactive dependency is represented by an edge e in a graph. Predecessor node p and successor node s forming reactive dependency are considered to be adjacent nodes with respect to edge e , and edge e representing reactive dependency is said to be incident with nodes p and s . Predecessor node p and successor node s cannot be the same node.

- **Outgoing reactive dependency** of node n is an ordered pair $o = (n, s)$, where n acts as a predecessor, and s as its successor node.
- **Ingoing reactive dependency** of node n is an ordered pair $i = (p, n)$, where n acts as a successor node, and p as its predecessor node.

Associated requirements:

- F1-REQ1** Each reactive node shall be uniquely identifiable.
 - F1-REQ2** Reactive nodes referring to the same state/behavior of the same object are considered to be the same reactive node.
 - F1-REQ3** Reactive node shall be able to take the roles of both event emitter and event receiver at the same time (see *Dual roles* in Table 1).
 - F1-REQ4** Multiple reactive nodes can be defined for one target object, allowing finer granularity (see *Arbitrary "Update" method* and *Multiple exposed events* in Table 1)
 - F1-REQ5** Reactive node shall be able to encapsulate additional data about event emitter/receiver (see *Additional data about emitter or event* in Table 1).
 - F1-REQ6** Individual reactive dependencies shall be specified by providing one predecessor and one successor node.
 - F1-REQ7** Reactive dependencies with the same respective predecessor and successor reactive nodes are considered to be the same reactive dependency.
 - F1-REQ8** Reactive dependency cannot have the same reactive node as both its predecessor and successor node, i.e. direct circular dependencies are not allowed.
-

Feature 2: Construct dependency graph

The ability of reactive node to participate at the same time in multiple reactive dependencies with different roles, opens up the possibility for interconnecting reactive nodes and chaining reactive dependencies to form more complex dependency structures - dependency graphs. Since individual reactive dependencies imply direction, dependency graphs formed out of them can be considered as directed graphs.

We provide following definition of dependency graph, related concepts and types of reactive nodes with regard to their position and role in dependency graph:

- **Dependency graph** is a structure expressed as an ordered pair $G = (N, D)$, where N is a set of reactive nodes and D is a set of reactive dependencies. Each reactive dependency d from D associates a pair of reactive nodes (p, s) , where p is a predecessor node, and s is a successor node.
- **Valid dependency graph** is a graph D containing at least one reactive dependency, and no cycles appearing in any of the paths through the graph.

Associated requirements:

- | | |
|----------------|--|
| F2-REQ1 | Dependency graph shall be formed by specifying one or more reactive dependencies. |
| F2-REQ2 | Each reactive dependency shall belong to one and only one dependency graph. |
| F2-REQ3 | Dependency graph structure shall be altered during runtime by adding new and removing existing reactive dependencies. |
| F2-REQ4 | Multiple dependency graphs can be constructed within the same application. |
| F2-REQ5 | Within dependency graph, reactive node shall be able to participate at the same time in zero or more reactive dependencies as a predecessor node, and in zero or more reactive dependencies as a successor node. |
| F2-REQ6 | Validity of dependency graph shall be checked and possible problems reported, whenever graph structure is altered. |
-

Feature 3: Perform dependency graph update process

The main goal of handling reactive dependencies is to keep dependency graph, i.e. its reactive nodes consistent. Inconsistent dependency graph means that at least one, but possibly more or even all of the graph's nodes require update. This means that they have out-dated state that should be updated, or that they are supposed to invoke specified behavior. Typical occurrences that bring dependency graph into inconsistent state are: (1) the change in the structure of dependency graph, i.e. introducing new or removing existing reactive nodes or dependencies, (2) setting or changing object's state represented by reactive node, or (3) invoking particular object's behavior represented by reactive node. We refer to such occurrences as *triggering change*, and this corresponds to terms such as triggering/firing/raising events in the context of event-driven programming.

- **Triggering change** is an occurrence of something happening in individual *source reactive node* or dependency graph itself, which results in a graph becoming inconsistent and requiring update.

When dependency graph becomes inconsistent, it is REFRAME's responsibility to bring it back to consistent state. We refer to this process as a *dependency graph update process*, or simply *update process*. During this process, inconsistencies have to be resolved in its entirety, but REFRAME also needs to avoid performing redundant or unnecessary updates of dependency graph and its individual reactive nodes. This means we have to identify the cause of the graph's inconsistency, all reactive nodes that are affected by it, and determine the order in which to resolve inconsistencies of individual reactive nodes. In doing that, REFRAME has to consider how to properly handle acyclic and cyclic graph structures.

- **Update process** is the process of bringing dependency graph into a consistent state by: (1) determining what caused the inconsistency (i.e. what triggered the change), (2) determining which reactive nodes are affected by the change and have to be updated, (3) determining the exact order of update, and finally (4) updating each individual node in that particular order.
- **Reactive node update** is the process of bringing particular reactive node into a consistent state by updating its out-dated state or invoking specified behavior.

In some cases, it can be possible and also beneficial in terms of performance and/or responsiveness to update dependency graph in parallel. This, however, heavily depends on the structure of dependency graph, and the exact reactive nodes that are required to be updated. In order to enable parallel update process, REFRAME needs to be able to identify groups of reactive nodes that can be updated in parallel, update them in parallel and establish synchronization.

Depending on the structure of dependency graph, and the exact programming logic in end-user application code, duration of update process can be expected to significantly vary. Also, the update process controlled and executed by the framework, can fail due to a number of framework or application related reasons. It is therefore expected of REFRAME to report the progress and status of ongoing update process, and provide error details in the event of update process failing.

Associated requirements:

- F3-REQ1** Triggering change can be manually invoked at arbitrary points in end-user application code.
 - F3-REQ2** Contextual data related to triggering change shall be provided, including the cause of inconsistency.
 - F3-REQ3** The exact reactive nodes of dependency graph that are made inconsistent by triggering change shall be determined.
 - F3-REQ4** The exact order in which update of inconsistent reactive nodes is to be performed shall be determined.
 - F3-REQ5** Redundant and unnecessary updates of dependency graph and its individual reactive nodes shall be avoided.
 - F3-REQ6** Acyclic graphs shall be properly handled during update.
 - F3-REQ7** Cyclic graphs shall be properly handled during update.
 - F3-REQ8** Update process can be performed sequentially or in parallel.
 - F3-REQ9** Contextual information about the progress and status of the ongoing update process shall be provided.
-

Feature 4: Analyze dependency graph

Once constructed, dependency graph can be quite large (contain large number of reactive nodes) and complex (contain large number of reactive dependencies interconnecting these nodes). One of the goals of REFRAME is to provide tools which would aid comprehension and understanding of dependency graph's structure and update process. In order to do that, REFRAME and its tools have to be able to examine runtime state of end-user application, identify existing dependency graphs and perform different analysis on these graphs and their nodes. In addition, REFRAME should monitor the update process and report its status.

While showing an entire dependency graphs and all of their reactive nodes can be useful, due to potential size and complexity of dependency graph, this could hold too much information for developer to comprehend. Therefore, our analyses will be based on some sort of information reduction. This reduction can be (1) *horizontal* - showing one part of the graph and hiding the other, or (2) *vertical* - showing dependency graph at different levels of abstraction. An example of horizontal reduction would be viewing only source reactive nodes, or only predecessors of certain reactive node. On the other hand, an example of vertical reduction would be aggregated view of dependency graph, where reactive nodes are classes or components instead of state of individual objects. Following concepts are important for understanding analyses of dependency graphs:

- **Degree of a reactive node** $deg(n)$ represents a number of edges e incident with reactive node n , i.e. number of edges going in or out of the reactive node. It determines a total number of reactive dependencies that the reactive node participates in within particular dependency graph, either as a predecessor or successor.
- **In-degree of a reactive node** $deg^-(n)$ represents a number of edges e going into the reactive node n , i.e. the number of reactive dependencies d in dependency graph in which reactive node n acts as a direct successor node.
- **Out-degree of a reactive node** $deg^+(n)$ represents a number of edges e going out of the reactive node n , i.e. the number of reactive dependencies d in dependency graph in which reactive node n acts as a direct predecessor node.
- **Leaf reactive node** represents a reactive node n which has either in-degree $deg^-(n) = 0$

or out-degree $deg^+(n) = 0$. In other words it is a node which solely acts either as a predecessor node or successor node in all reactive dependencies it is involved.

- **Source reactive node** is a leaf reactive node n with $deg^-(n) = 0$ and $deg^+(n) > 0$. This means that reactive node n is successor to no other reactive node, but it is a predecessor to at least one other reactive node. It acts solely as a predecessor in each reactive dependency it is involved.

Usually, these are reactive nodes which are triggered by occurrence of something happening (e.g. user enters some data).

- **Sink reactive node** is a leaf reactive node n with $deg^-(n) > 0$ and $deg^+(n) = 0$. This means that reactive node is successor to at least one other reactive node, but is not a predecessor to any reactive node. It acts solely as a successor node in each reactive dependency it is involved.

Usually, these are reactive nodes which represent final results to be calculated or final operations to be performed.

- **Intermediary reactive node** is a reactive node n which unlike leaf reactive node has both in-degree $deg^-(n)$ and out-degree $deg^+(n)$ greater than zero. In other words, it is a node which acts as a predecessor in at least one reactive dependency, and also acts as a successor in at least one reactive dependency.

These reactive nodes act as a connective tissue of dependency graph, which allows triggering change in source reactive nodes to result in update propagation all the way to sink reactive nodes.

- **Orphan reactive node** is a reactive node n with $deg(n) = 0$. It is a node which is not involved in any reactive dependency. In the context of managing reactive dependencies, such node does not have a purpose.
- **Direct predecessors** of reactive node n are reactive nodes p_1, p_2, \dots, p_i which are incident with reactive node n , i.e. for which there is ingoing reactive dependency $d = (p_i, n)$.
- **Indirect predecessors** of reactive node n are reactive nodes p_1, p_2, \dots, p_i which are not incident with reactive node n , but a path of predecessors can be formed between n and p_i .

- **Direct successors** of reactive node n are reactive nodes s_1, s_2, \dots, s_i which are incident with reactive node n , i.e. for which there is outgoing reactive dependency $d = (n, s_i)$.
- **Indirect successors** of reactive node n are reactive nodes s_1, s_2, \dots, s_i which are not incident with reactive node n , but a path of successors can be formed between n and s_i .
- **Direct neighbours** of reactive node n are reactive nodes g_1, g_2, \dots, g_i which are incident with reactive node n , i.e. for which there is ingoing reactive dependency (g_i, n) or outgoing reactive dependency (n, g_i) .
- **Indirect neighbours** of reactive node n are reactive nodes g_1, g_2, \dots, g_i which are not incident with reactive node n , but a path of predecessors or successors can be formed between n and g_i .

A large number of potentially useful analyses can be performed on graph structures, and it is not possible to list them all as a mandatory requirement in this document. Instead, we provide list of general requirements which serve as a guide to form specific analyses that are relevant and useful for understanding dependency graphs.

Associated requirements:

- F4-REQ1** Analysis shall be performed in order to identify and obtain basic information on dependency graphs existing in end-user application during run-time.
 - F4-REQ2** For each identified dependency graph a list of reactive nodes can be obtained at the base abstraction level (i.e. state of individual objects).
 - F4-REQ3** In addition to base abstraction level, aggregated views of dependency graph shall be provided on a different levels of abstraction (e.g. class-level).
 - F4-REQ4** For each dependency graph a separate list of graph's *leaf*, *source*, *sink*, *intermediary* and *orphan* reactive nodes can be obtained.
 - F4-REQ5** For each reactive node a separate list of its (direct and indirect) predecessor, successor and neighbour reactive nodes can be obtained at different depth level.
 - F4-REQ6** For each reactive node a separate list of its leaf, source, sink, and intermediary reactive nodes can be obtained.
 - F4-REQ7** For each conducted update process basic information can be obtained, including: success status, cause, duration and list of involved reactive nodes.
-

Feature 5: Visualize dependency graph

In addition to different analyses, understanding of structures and processes around dependency graph can be further improved by visually presenting parts or entire graph. This can help developers gain better understanding of how reactive nodes are interconnected, and what paths through the graph update process will take. Various graph visualizations will be offered in line with the results of conducted dependency graph analyses.

Associated requirements:

- | | |
|----------------|---|
| F5-REQ1 | Dependency graphs can be visualized in its entirety or in part, relying on the results of conducted analyses. |
| F5-REQ2 | Update process can be visualized by displaying involved reactive nodes and basic information about them in the context of update process. |
-

Feature 6: Generate boilerplate code

Using frameworks usually involves writing a fair amount of repetitive and stereotypical code which is often referred to as *boilerplate code*. This opens up the possibility to utilize code generation techniques to, at least partially, remove burden of manually writing such code from developer itself and generating that code instead. However, specifying detailed requirements for this feature is not something that can be done without delving into design and implementation details of the rest of the framework. This is something that makes generating code different from other features of the framework. While other features of the framework aim at mitigating domain problems, this feature, instead of adding new domain-related functionality, focuses on the problems related to framework use. For us to know what exact code should be generated and how it should be done, we first need to have that code, i.e. we need to have the framework designed and implemented. Coupling software requirements phase with such details is not convenient at this point, as we want to remain on a higher level of abstraction and also ignorant to technological, design and implementational aspects of the framework. However, we can try to anticipate tasks in using framework which are most likely to be repetitive, and set them as candidates for generating code. In the context of REFRAME we can expect such repetitive tasks to be associated with defining large number of reactive nodes and specifying large number of reactive dependencies. Therefore, these are our primary targets for utilizing code generation.

Associated requirements:

- F6-REQ1** Writing boilerplate code in charge of defining reactive nodes shall be aided by code generation.
- F6-REQ2** Writing boilerplate code in charge of specifying reactive dependencies shall be aided by code generation.
-

Design constraints

Although an effort is placed to make REFRAME's requirements and model as portable as possible to other technologies within object-oriented paradigm, REFRAME is essentially implemented framework. Therefore, some design constraints are technology-oriented, i.e. they stem from using particular technology to develop REFRAME. Specifically, REFRAME is developed using C# programming language on top of .NET platform and its constituent frameworks and libraries. Any relevant feature or characteristic that .NET platform or C# programming language posses or lack, do impose some constraints on REFRAME's design. Similarly, REFRAME's tools are to be integrated into Visual Studio IDE, so their design is constrained by Visual Studio's extensibility options.

Software system attributes

In addition to a number of functional requirements, specification documents also host other requirements related to non-functional/quality attributes of software systems and their use. For example, ISO/IEC 25010 standard for software quality model [11] prescribes eight quality properties of software system itself, and five characteristics related to its use. Most of these properties are further decomposed into several quality characteristics, which ends up in this particular quality model having 40 characteristics in total. While it is certainly desirable for a software system to manifest as many as possible of such properties and characteristics, this is often not feasible within limited resources. Therefore, we list subset of these properties that should be taken into consideration during development of REFRAME. Some of these properties are particularly relevant to software frameworks in general and some are related to evaluation activity of design science process in this dissertation.

- **Functional suitability** - "*degree to which a product or system provides functions that meet stated and implied needs when used under specified conditions*" [11]. This refers to system being (1) *functionally complete*, (2) *functionally correct*, and (3) *functionally*

appropriate. The intent of this quality property corresponds in part with *efficacy* property from our evaluation activity.

- **Maintainability** - "*degree to which a product or system is composed of discrete components such that a change to one component has minimal impact on other components*". This refers to system being (1) *modular*, (2) *reusable*, (3) *analyzable*, (4) *modifiable*, and (5) *testable*.
- **Satisfaction** - "*degree to which user needs are satisfied when a product or system is used in a specified context of use*." This refers to system manifesting (1) *usefulness*, (2) *trust*, (3) *pleasure*, and (4) *comfort*. The usefulness quality property corresponds to the usefulness evaluation property from our evaluation activity.

Associated requirements:

- NF-REQ1** REFRAME shall be functionally suitable for managing reactive dependencies in OO applications.
 - NF-REQ2** REFRAME shall be maintainable in terms of making possible the natural evolution of the framework, through the processes of correcting, perfecting and adapting the framework.
 - NF-REQ3** REFRAME shall be maintainable in terms of making possible the reuse of framework in different contexts and domains.
 - NF-REQ4** REFRAME shall be satisfiable in terms of being perceived as useful (by application developers) for managing reactive dependencies in OO applications.
-

6. Design and develop artifact

Previous chapter resulted in providing requirements specification for REFRAME. In this chapter, as the name implies, specified requirements shall be concretized in a form of a prescriptive model and code implementation. The activities in this step are performed back and forth, in an iterative manner. Realistically capturing the dynamics of iterative thought process is not something that can easily be done. Throughout this chapter we will try to convey the sense of iterative process, but in a simplified way in order to not sacrifice the clarity and dissertation structure. Here we essentially describe ideas on different possible design and implementation options for specified requirements, assess these options, try out these options, and abandon or accept them as a part of the solution.

6.1. Imagine and brainstorm

The main goal of this section is to capture the set of main ideas related to possible design and implementation options for REFRAME. These ideas will be presented with respect to particular feature stated in SRS document in Chapter 5. Here we will try to restrain from making final judgments about the presented options and alternatives, and leave that for sections that follow.

6.1.1. Feature 1: Specify reactive dependency

As already stated in SRS document, specifying reactive dependencies is a fundamental feature of REFRAME. However, in order to be able to specify reactive dependencies, we first have to discuss their constituent parts - reactive nodes. One of the fundamental questions that arise here is what can be a reactive node in OO application? What can trigger a change in OO application, and what should be able to react and respond to that change? In order to answer that, we look at the constituent parts of the object - a fundamental building block of OO applications. The object, regardless of particular technology, has two distinct parts: state and behavior. The object's state is represented by data members (known as fields and attributes), each capable of holding certain data in accordance to its specified data type. Object's behavior, on the other hand, is represented by methods containing programming logic. Depending on the programming language, other member types may exist, but we will limit our consideration to aforementioned fields and methods, as they are universal across OO languages.

Most literature sources consider reactive dependencies as a means to synchronize objects' state. Therefore, allowing object's fields, as representatives of that state, to act as a reactive node, may seem as obvious thing to do. Application developers would then be able to, for example, set up reactive dependency between two fields of the same or different objects, and update the field represented by a successor node whenever the field represented by predecessor node changes. It should be noted that this requires object fields to be publicly available, which is in direct conflict with one of the core OO principles - *data hiding*. Through the use of encapsulation technique programmers usually declare all fields as private, thus making them unavailable for everyone to see or change except the owner object. A common way to use these private fields from outside of the owner object is through public methods called *getters* (*accessors*) and *setters* (*mutators*). As their names imply, getters enable us to get the field value, while setters provide means to set or change it. Through these methods we can employ arbitrary programming logic to dictate how fields are allowed to be used. This practice is in fact so common that a lot of programming languages and editors provide direct support to generate getters and setters, or in some other way facilitate their use. Some programming languages (such as Java) express getters and setters as any other method, with the prefix *get/set* in the method name being an indicator of their special role. While nothing prevents us from implementing plain getter and setter methods in .NET languages also, there is an alternative in a form of a special language construct called *property*. Properties in .NET fall into category of so-called *syntactic sugar* features, which offer alternative, presumably better syntax that masks already existing language features. Properties are just camouflaged getter and setter methods which can be easily verified by inspecting *intermediate language* (IL). Therefore, an alternative to allowing object's fields to act as a reactive node is using getter and setter methods, or special constructs such as properties instead.

While the need to support reactive dependencies between objects' state seems fundamental, the support for reactive dependencies involving objects' behavior may not be so apparent. Although the support for constructing reactive dependencies between methods as reactive nodes would certainly require additional effort, there may be several cases when this would be useful. One of the cases, as we already stated, would be when methods are used as a "proxy" to represent fields. So, if we already allow getters and setters to act as reactive nodes, it might not require too much work to allow this for methods in general. Also, even if we chose fields

to act as reactive node, in order to update the successor after the predecessor was triggered, usually some method containing the field's update logic has to be executed in the background. Sometimes it might be more convenient to bypass the fields, and just construct reactive nodes and dependencies from these update methods. Other examples where method reactive nodes could be useful are cases when instead of updating one particular field, we want to group update multiple fields, but without creating reactive dependency for each field. It can be seen as a sort of cumulative update. Finally, there are cases when instead of state, we want to focus on behavioral aspects of the application, and to just schedule arbitrary number of method calls to do some processing in specific order.

Whatever type of members we choose to allow to act as reactive nodes, we need to offer appropriate means to express reactive nodes in programming language. One of the approaches may be to extend the programming language with new features, and develop custom compiler to support operations with them. For example, we could introduce new built-in keywords and data types which would be used as a part of declaration statement for data and method members. Fairly recent example of this in .NET environment, is introducing new *async* and *await* keywords which are used in the context of asynchronous programming [1].

More conservative approach would be to avoid making changes to compiler and instead extend the capabilities of language by utilizing existing features. We could, for instance, utilize meta-programming capabilities of the language, and use meta-data to mark members as reactive nodes. *Meta-attributes* (*annotations* in Java) are extensively used in frameworks to provide additional, class level meta-data for different code elements (assemblies, classes, class members etc.). For example, unit testing frameworks assign special meta-attribute to methods containing unit tests. This enables testing tools to automate recognition and execution of unit tests. Similarly, Entity Framework - Microsoft's official Object-Oriented Mapping (ORM) framework, uses meta-attributes for multiple purposes, including: mapping class properties with table attributes, specifying constraints on particular class properties, etc. In order to be able to read this meta-data, we must turn to meta-programming technique called *reflection*. Reflection can be described as an ability of a software application to observe and possibly modify its structure and behavior [144].

Another possible option is to introduce new 'reactive' types by inheriting common data types, and extending them to behave as reactive nodes. Alternatively we can implement reac-

tive nodes as a generic, separate class, unaware of the member's type, and then find a way to associate it with particular member.

Listing 6.1: Possible ways to express reactive nodes

```
public reactive int A; //Introducing new keyword 'reactive'

[ReactiveNode] //Using meta-data
public int A;

public ReactiveInt A; //Introducing 'reactive' data types

public int A; //Introduce standalone classes for reactive nodes
ReactiveNode nodeA = new ReactiveNode("A");
```

Reactive nodes have no particular purpose other than being constituent parts of reactive dependencies. Each reactive dependency is expressed by exactly one reactive node assigned with the role of predecessor, and exactly one node assigned with the role of successor. Therefore, in order to be able to create reactive dependency, it is essential to discuss how reactive node roles can be represented.

Whenever we come across some issue in OO context, it is always a good idea to consult existing design patterns, and try to identify relevant ones modeling useful abstractions and relationships between these abstractions. As reported in Chapter 2 the most famous design pattern whose idea resembles the idea of reactive dependencies is Observer pattern. However, other variants of Observer pattern as well as patterns with similar intent do exist. In previous research [107] we identified and compared five such patterns: simple Observer pattern, advanced Observer pattern, Observer pattern revisited, Event-notification pattern and Propagator pattern. While analyzed patterns do share common idea and intent, they also differ in some important aspect and thus offer useful insight into possible design aspects of reactive dependencies.

Table 6 shows how abstractions from these design patterns represent reactive nodes with respect to the role they play in reactive dependency. We can see that reactive nodes with different roles are predominantly treated as distinct concepts implemented by separate abstractions. For example, in Observer pattern, reactive nodes with the role of predecessor are represented by

Subject class, and the ones with successor role as Observer class. Only in Propagator pattern are both roles contained within the same abstraction, i.e. with the Propagator class. With regard to implementation mechanisms, reactive node's roles are usually expressed by inheriting an abstract class or realizing interface. Here, different combinations can occur: both roles implemented as abstract classes, both roles implemented as interfaces, or one role implemented as an abstract class and the other as an interface. Although nominally distinguishing the roles of predecessor (Observable) and successor (Observer), Observer pattern revisited uses base Java Objects stripped from any behavior specific to reactive node's roles. Rather, such behavior is moved to separate (ObserverManager) class. Therefore in this case it is hard to talk about real inheritance. Event notification pattern is an exception with regard to implementation mechanism. Instead of inheriting abstract class or realizing interface, it uses composition to implement reactive node's roles.

Table 6: Roles of reactive nodes as represented by abstractions from relevant design patterns

Pattern	Predecessor role	Successor role	Mechanism
Observer (simple) [61]	Subject	Observer	Inheritance
Observer (advanced) [61]	Subject	Observer	Inheritance
Observer revisited [46]	Observable	Observer	Inheritance
Event notification [122]	StateChange	EventStub	Composition
Propagator [53]	Propagator	Propagator	Inheritance

Other than aforementioned design patterns, some OO languages have already built-in features that try to mimic the idea of Observer pattern. For example, .NET languages include built-in language construct called *event*, which can be described as a "*message sent by an object to signal the occurrence of an action*" [8]. Events as a feature in .NET could be used to implement reactive nodes. An event, with its ability to inform about a change, would correspond to the role of a predecessor node. A special method, usually referred to as *event handler*, which reacts on this event, would represent successor node. A similar feature is offered also in Java in a form of *events* (predecessor nodes) and *event listeners* (successor nodes) [10], primarily aimed for use in graphical user interfaces. However, unlike in .NET, implementation of this feature in Java is closer to design offered by Observer pattern.

Individual reactive dependencies, as we see them, are in fact ordered pairs of one predecessor reactive node and one successor reactive node. One design point to consider is whether we want to express them as implicit or explicit concepts. If we decide to do it in an implicit manner,

we rely on the reactive node abstraction, and treat reactive dependency as just an association of two reactive node instances. On the other hand, defining reactive dependency as an explicit concept requires creating separate abstraction. This separate abstraction would encapsulate association of two reactive nodes, but would also allow us to capture additional data for reactive dependencies (e.g. weight, priority). If we take a look at the related design patterns, but also built-in features in programming languages such as C# and Java, we can see that prevailing approach is the one seeing reactive dependency as an implicit concept of association of reactive nodes. In such case, construction of reactive dependencies can be viewed as simply pairing two reactive nodes, with a clear indication which one takes the predecessor and which one takes the successor role.

In simple Observer pattern, Event notification pattern and Propagator pattern, this is done in a way that each predecessor takes charge of managing reactive dependencies in which it is involved. In order to do that, predecessor declares a collection capable of storing multiple instances of successor reactive nodes, and provides methods to add (i.e. *Attach*, *Register* or *AddDependent*) or remove (i.e. *Detach*, *Unregister*, *RemoveDependent*) elements of that collection. In other words, predecessor forms an ordered pair, i.e. reactive dependency, with each successor node in its collection. Similar situation is also with C# and Java built-in event handling features. In Java, an object implementing *Listener* interface (successor role) is added to internal collection of an object with a predecessor role. In C#, in a process called *event subscription*, a reference to a method (successor role) is added to so called *invocation list* of an event (predecessor role). This approach could be characterized as the one described by Hinze et al. [74] as *decentralized approach*.

Observer pattern advanced and Observer pattern revisited take a slightly different approach. In addition to abstractions representing reactive nodes and their two roles, these patterns prescribe additional, central entity in charge of constructing and storing reactive dependencies. Here, it is the manager class (*ChangeManager*, *ObserverManager*) that declares collection storing ordered pairs of predecessor and successor reactive nodes, and provides operations to modify that collection. Each pair in these collections represent one reactive dependency. As opposed to decentralized approach, this approach could be described as centralized or middleware approach [74].

6.1.2. Feature 2: Construct dependency graph

In addition to expressing individual reactive dependencies, another design point worth consideration is how do we express dependency graph as a set of interrelated reactive dependencies. Again, a question is whether we need dependency graph as an implicit or an explicit concept. As we discussed in previous subsection, part of design patterns and built-in language features propose making predecessor nodes in charge of constructing and storing reactive dependencies they are involved in. No graph-like abstraction that manages reactive dependencies is prescribed. Rather, reactive dependencies that potentially form a dependency graph are scattered across individual predecessors. On the other hand, Observer pattern advanced and Observer pattern revisited do prescribe separate abstraction that is in charge of storing and managing reactive dependencies. Although this abstraction is referred to as a *Manager*, its responsibilities correspond to the ones of dependency graph.

In deciding which approach to take, it is also beneficial to consult options for designing and implementing graph structures as concepts from graph theory. For example, directed-acyclic graph (DAG) seems as a data structure fit to represent group of reactive nodes interconnected with reactive dependencies. Implementation-wise, according to Goodrich and Tamassia [66], graphs are commonly represented by two data structures, namely: *adjacency matrix* and *adjacency list*. Although less commonly, *incidence matrix* is also mentioned in this context in developers community and forums.

Adjacency matrix is basically two-dimensional array of size $N \times N$, where N is a number of nodes in a graph. If A is adjacency matrix, then $A[p][s] = 1$ indicates that there is reactive dependency between node p as a predecessor and node s as a successor. Similarly, incidence matrix is also two-dimensional array, but with size of $N \times D$, where N is number of nodes in graph and D is number of reactive dependencies. If A is incidence matrix, $A[p][d] = -1$ indicates that node p acts as a predecessor in reactive dependency d , while $A[s][d] = 1$ indicates that node s acts as a successor in reactive dependency d . In adjacency list, each node is associated with collection (e.g. arrays, lists, hash tables) of its neighboring nodes. A particular node n may keep track of only collection S of its successors, or collection P of its predecessors, or both. Of course, mentioned data structures are canonical DAG implementations, often adjusted in real applications to fit particular use.

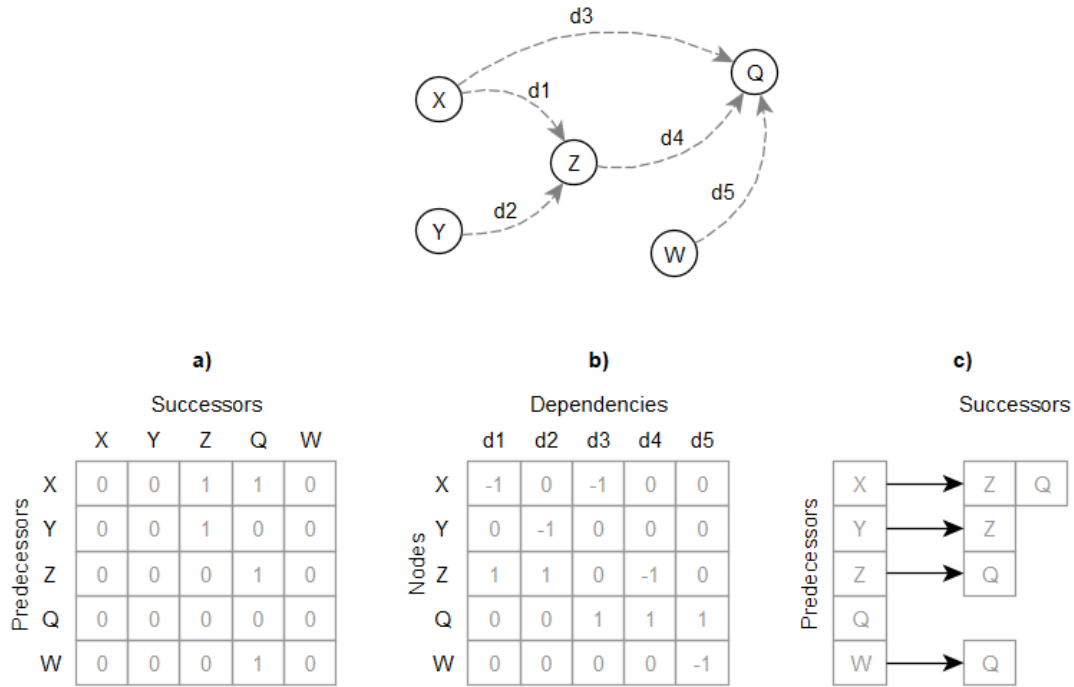


Figure 16: Different DAG representations: a) adjacency matrix, b) incidence matrix and c) adjacency list

6.1.3. Feature 3: Perform dependency graph update process

As stated in SRS document, the main goal of handling reactive dependencies is to keep dependency graph and its reactive nodes consistent, i.e. ensure that all state members are up-to-date and all methods are invoked when required. The inconsistency arises as a result of a process we called *triggering change* which happens when dependency graph structure is changed, or when particular reactive node triggers change.

A change in the structure of dependency graph happens when reactive nodes are added or removed from dependency graph, as well as when reactive dependencies are formed or dismantled. Here we should carefully consider which of these actions should indeed trigger the change. For example, when reactive node is just added to dependency graph, it does not participate in any reactive dependency, and therefore triggering change in that case would make no difference to other reactive nodes. Similarly, when reactive node is removed from dependency graph, its reactive dependencies are also removed, so there is no need to trigger the change in all these cases.

Triggering the update process in case of change in dependency graph structure, may at first seem fairly straightforward. After all, adding and removing reactive nodes and reactive de-

dependencies is conducted by calling dedicated methods of the framework, so raising event or invoking update process within these methods is a simple thing to do. As this code is part of the framework, it is in the hands of framework developers, and completely transparent to framework users. However, in reality, multiple reactive nodes and dependencies are usually going to be added or removed together in sequence. If we would trigger update each time this happens, update process would be unnecessarily performed multiple times. In order to back up this claim, we need to consider the usual points in application runtime, that reactive nodes and dependencies are most likely to be added or removed: (1) loading data and instantiating objects during application startup procedure, (2) instantiating and destroying objects as a response to user action, (3) destroying objects as a part of application closing procedure. In a first (application startup) and third case (application closing) it is apparent that due to large number of objects being instantiated or destroyed, also potentially large number of reactive nodes and dependencies are going to be simultaneously added or removed. This would cause huge number of unnecessary updates. Similarly, if an object, which is with its members involved in multiple reactive dependencies, is instantiated or destroyed as a response to user action, these multiple reactive nodes and dependencies are going to be added or removed, and again cause multiple updates.

When we talk about particular reactive nodes triggering change, we talk about cases when underlying state member changes its value which requires other states to be updated accordingly, or when the particular method is invoked which requires other methods to be invoked too. Unlike with dependency graph structural changes, the problem here is that regardless of performance and redundancy issues, state members and methods which should trigger the update process are not part of framework's code. Instead they are part of application's code, the code which simply does not exist at the time of framework development. Because of that, we need and explicit input from application developer to know which members should behave in this (reactive) manner and trigger the update process. Full automation of this feature would require compiler modifications. For example, framework could introduce new language keywords (see section 6.1.1) to use within member declaration, and compiler would then had to be modified to understand them. Whenever state member declared in this way would change its value or when a method member would be invoked, compiler would trigger the update process automatically instead of the application developer. Such *compiler-dependent* approach is indeed taken

by reactive programming languages and some frameworks. Another approach to achieving certain level of automation in triggering update process is meta-programming. Although it does not require compiler customizations, its techniques are often interacting with compiler. Various meta-programming techniques are at different levels supported by various programming languages. Techniques available specifically for C# programming language include [71]: Reflection, Text Template Transformation Toolkit (T4), CodeDOM, Reflection.Emit, Expression trees, IL Rewriting, Domain specific languages (DSLs), .NET Compiler Platform (Roslyn API), etc.

As we discussed in Chapter 2, AOP principles have been used to address certain shortcomings of traditional Observer pattern. In AOP context, managing reactive dependencies would be treated as a cross-cutting concern, and would be extracted from domain classes and placed into *aspects*. AOP allows us to specify points in the application code (usually methods) called *JoinPoints*, and then use *PointCuts* to assign *Advice* to these points in code. The Advice determines what method is going to be performed when the specified point in code is reached. In the context of triggering particular reactive node, we could use AOP to insert code (at runtime) that triggers update process at the end of e.g. property setters and other methods. In order to achieve that, AOP frameworks rely on frameworks providing code injection capabilities, such as Cecil [49].

If above mentioned techniques for automating the process of triggering particular reactive node are not suitable, or are not feasible at this stage, REFRAME could still try to make this as straightforward for framework user as possible. Using OO techniques to provide clean and clear interfaces with as much complexities as possible hidden behind them can still be acceptable solution.

After the occurrence of *triggering change*, the dependency graph can be characterized as inconsistent and it should be updated. The first step in achieving that is identifying the very cause of dependency graph's inconsistency. In case of graph update process being triggered by a change in a state or invocation of method this is clear. The source reactive node which caused the inconsistency is the very reactive node that was triggered. However, in case of changes in dependency graph structure, a predecessor node of the added or removed reactive dependency is the one which could qualify for source reactive node.

As we also pointed out, multiple structural changes to dependency graph very often occur

simultaneously. Similarly, simultaneous triggering of multiple reactive nodes can also occur, either programmatically or initiated by the user. Since this would obviously lead to performing update process multiple times, we may decide to e.g. temporarily disable update process until all changes are done, and then manually invoke update only once. However in this case, we would have more than one reactive node that was triggered and it would be more difficult to find out the exact reactive nodes which are to be updated and in which order it should be performed. The simple solution for this would be to invoke update process of entire dependency graph. This would update dependency graph and bring it into consistent state, but it would also perform unnecessary updates of some parts of dependency graph. Other, more complex solution involves tracking all reactive nodes that were triggered during the time update was disabled, and then introducing temporary predecessor to these nodes and triggering it.

When we identify reactive node which triggered the change and acts as an initial node in update process, we need to determine which reactive nodes are going to be updated, and in which order. In the context of directed-acyclic graph (DAG) structure (which dependency graph is based on) this means we have to find *topological sorting* or *topological ordering* of the graph (or the part of the graph). Feiler and Tichy [53] also recognized this as a means to ensure proper order of updates in their Propagator pattern. Topological sorting of DAG can be defined as a linear ordering of its vertices such that for every directed edge (u, v) from vertex u to vertex v , u comes before v in ordering [21]. Translated to the context of REFRAME, this means that topological sorting will give us a linear ordering of reactive nodes, such that for every reactive dependency $d = (p, s)$ from predecessor node p to successor node s , p comes before s in update process. It is important to note that if a directed graph is acyclic (i.e. it does not contains cycles), it necessarily has at least one, but possibly multiple topological orderings. If there indeed are multiple topological orderings, all of them are perfectly correct to use in performing update process.

There are known algorithms in graph theory used for topological sorting, with most commonly discussed being Kahn's algorithm [83], and Tarjan's algorithm [148]. Kahn's algorithm is based on *breadth-first search* (BFS) algorithm [108] for traversing and searching graph structures. The baseline idea here is that vertices (nodes) not having incoming edges are safe to be added to ordering. Therefore, the algorithm starts with finding such vertices, gradually traverses graph in layers, i.e. immediate neighbors first, and removes the edges to create more vertices

without incoming edges.

Listing 6.2: Pseudocode of Kahn's algorithm

```
let sorted is List
procedure sort()
  let pending is List
  foreach vertex V in graph
    if V has no incoming edges
      add V to pending
    endif
  endfor
  while pending has elements
    V = pending.next()
    add V to sorted
    neighbors = vertices pointed by edges going from V
    foreach neighbor in neighbors
      E = edge from V to neighbor
      remove E from graph
      if neighbor has no incoming edges
        add neighbor to pending
      endif
    endfor
  endwhile
end
```

Unlike Kahn's algorithm, Tarjan's algorithm is based on *depth-first search* (DFS) algorithm for traversing and searching graph structures. This means that, instead of traversing graph in layers by first processing all immediate neighbors, we process nodes as deep as possible in the graph structure before we backtrack to the next immediate neighbor. Because of that, Tarjan's algorithm gives inversed topological sorting, i.e. the first nodes given by the algorithm should be the last ones in topological ordering.

Listing 6.3: Pseudocode of Tarjan's algorithm

```
let sorted is List
let visited is List

procedure Sort()
    foreach vertex V in graph
        visit(V)
    endfor
end

procedure visit(V)
    if V is not contained in visited
        add V to visited
        neighbors = vertices pointed by edges going from V
        foreach neighbor in neighbors
            visit(neighbor)
        endfor
        add V to sorted
    endif
end
```

When performed on dependency graph, topological sorting will give us list of graph's reactive nodes, linearly ordered in a way they need to be processed (updated). After that, the simplest way to perform update process is to iterate through the list and update each individual reactive node. We refer to this as *sequential update process*. Sometimes, however, we would be better off by introducing concurrency and parallelism. Depending on the size of graph, and the complexity of computations associated with updates of individual reactive nodes, graph update process can be time-consuming. If in such cases update is run in the main thread, together with Graphical User Interface (GUI), update process will be blocking the thread from performing any other operations for some significant time, and the application will appear as unresponsive to the user. In order to make application usable and prevent blocking, we need to perform update process concurrently. This can be done by letting main thread remain in charge of GUI, and placing update process to new thread. Now, the CPU scheduler is responsible to assign

slices of CPU's time to each thread, effectively making an illusion of update process and GUI running simultaneously, thus making the application responsive and usable.

While making graph update process run in separate thread may improve usability of the application, it does not improve the overall performance of the update process. Not only it may not result in improvement of performance, but update performed concurrently may require even more time to complete. For example, the time we need to perform graph update process is roughly equal to sum of times required for update of individual reactive nodes (we will ignore the overhead introduced by the framework). If the graph update process is performed in the same thread as GUI, the thread's portion of CPU's time is dedicated to update process solely. When the update starts it will not be interrupted by the GUI operations. However, if we decide to perform update process concurrently with the GUI, update process will get interrupted by the GUI operations, and may take more time to complete (CPU scheduler does thread interleaving in a nondeterministic fashion, so we can not fully anticipate how CPU's time is going to be distributed). In any case, the goal of concurrency here is not to offer best update process performance, but instead to assure usability of application during the update process.

If we want to address performance of graph update process, we need to introduce parallelism, i.e. perform graph update process itself by multiple CPUs or CPU cores at the same time. In order for this to be possible, we need to examine topologically ordered sequence of reactive nodes, and extract at least two subsequences that are independent, and can be run in parallel. An algorithm that we found as potentially useful in this context is Coffman-Graham's [44] algorithm for graph layering. The algorithm assorts graph nodes to layers in a way that nodes belonging to particular layer are mutually independent, and can therefore be processed in parallel. It is performed by following these three steps: (1) *transitive reduction*, (2) *topological ordering* and (3) *graph layering*.

Concurrency mechanisms are common features of modern OO programming languages. In .NET ecosystem these have been present from the very beginnings, although significantly evolving in time. Significant number of technologies are at disposal to developers when developing concurrent applications in .NET. They can be broadly categorized as: *asynchronous programming*, *multithreaded programming* and *parallel programming*. Asynchronous programming has two distinct goals: (1) maintain responsiveness of user interface, and (2) enable scalability of server side applications [34]. In achieving these goals, asynchronous programming tries to

mimic concurrency, i.e. dealing with multiple tasks at once, but, if possible, without introducing additional threads. There are three different patterns we can resort to when doing asynchronous programming: (1) Task-based asynchronous pattern (TAP), (2) Event-based asynchronous pattern (EAP) and (3) Asynchronous programming model (APM) [19]. On the other hand, multithreaded programming, as the name implies, aims at enabling concurrency using multiple threads. Through *System.Threading* namespace it offers mechanisms and abstractions such as *Thread*, *ThreadPool*, *BackgroundWorker*, *Semaphore* and others for creating, starting and canceling threads, performing synchronization between threads etc. Finally, parallel programming extends the concept of multithreading by trying to take advantage of multiprocessor or multi-core systems to run more than one thread simultaneously. Cleary [34] reports on two forms of parallelism we can encounter: (1) *data parallelism* (simultaneously processing a number of independent pieces of data) and (2) *task parallelism* (simultaneously processing a number of independent pieces of work). Both forms of parallelism are supported in .NET through the use of dedicated libraries such as Parallel LINQ (PLINQ) or Task Parallel Library (TPL) [20].

6.1.4. Feature 4: Analyze dependency graph

The base goal of performing analyses on dependency graphs is to facilitate developers' understanding and comprehension of dependency graph structure and the update process. The pivotal questions that arise here are (1) what particular analyses can contribute to this goal, and (2) how can they be made an integral part of the framework?

The main source for defining relevant analyses is researcher's experience in developing end-user applications with reactive dependencies. Some of these analyses were suggested as requirements in Define Requirements chapter 5. For example, for each existing dependency graph we should be able to obtain its basic properties, such as total number of nodes, number of nodes playing particular role (e.g. source node), number of defined reactive nodes and dependencies, graph density, etc. These properties can help us gain quick insight into structural aspects of dependency graph, such as size and density.

A more detailed information on dependency graph can be provided by listing all reactive nodes contained in the graph, along with information associated with these nodes (such as name, type and degree). Since we are, at some particular moment, rarely interested in all reactive nodes, it is useful to enable developers to fetch only subset of graph's nodes (i.e. horizontal

reduction). One of the criteria for defining these subsets that is relevant for the context of reactive dependencies, can be the role of reactive node. So, for example, by fetching only graph's *source* nodes we can find out what are the input parameters of some calculation model. Similarly, by fetching orphan nodes we can search for unnecessary parameters or the ones which are mistakenly left completely unconnected with others. Additionally, we should be able to fetch subsets of nodes which are in some manner related to particular node in a graph. For example, by fetching source nodes of some particular node, we can see which input parameters directly or indirectly influence our node of interest.

Since dependency graphs are basically acyclic directed graphs, a wide variety of existing algorithms written for these data structures can apply. Therefore, it is also useful to examine well-know algorithms traditionally used in graph theory and supported disciplines such as social network analysis (SNA). However, not all of these algorithms are meaningful and suitable in the context of analyzing dependency graphs with reactive dependencies. While conducting full-scale scientific analysis of graph and network algorithms was not planned to be conducted within this dissertation, we will briefly go through widely known categories and representatives of these algorithms (see Table 7).

Since reactive nodes represent a pair consisted of runtime object and its member (e.g. property), dependency graphs can be quite large structures. Therefore, it could be useful for users to be able to view dependency graph on a different levels of abstraction (vertical reduction). For example, instead of original dependency graph, set on an object-member level of abstraction, we could be interested in seeing aggregated graph version showing dependencies between objects or classes. In this sense we identified 6 levels of abstraction that can be implemented: (1) object-member level, (2) object level, (3) class-member level, (4) class level, (5) namespace level, and (6) assembly level. These abstraction levels could be combined with above mentioned analyses producing significant number of different analyses.

In addition to analyzing dependency graph structure, analysis of performed update process can be extremely useful. This does not only include listing reactive nodes which took part in update process, but also reporting underlying cause of update, whether the update was successful or not, update duration, state changes caused by update, etc.

Table 7: Graph algorithms

Algorithm	Description
Graph traversing algorithms	Allow traversing nodes in graph or tree structures, which makes them useful for different goals (searching, sorting, finding paths etc.). Well-known representatives are Breadth-first search (BFS) and Depth-first search (DFS) algorithms.
Cycle detection algorithms	Allow us to detect the existence of cycles in graph, i.e. to determine whether there is a path p which starts and ends at the same vertex v .
Topological sorting algorithms	Allow us to obtain list of graph nodes in which node is always ordered before all of its successor nodes and after all of its predecessor nodes. Tarjan's and Kahn's topological sorting algorithms are common representatives.
Spanning tree algorithms	Allow us to find subgraph S of graph G which contains all the vertices of graph G but with minimum number of edges required to stay connected. A minimum spanning tree extends the definition towards weighted graphs and assumes also minimum possible total edge weight.
Shortest path algorithms	Allow us to find a shortest path between two vertices.
Maximum-flow algorithms	Allow us to find maximum possible flow rate between vertex denoted as <i>source</i> and vertex denoted as <i>sink</i> in a weighted graph. The well-known representative is a Ford-Fulkerson's algorithm.
Graph connectivity algorithms	Allow us to inspect connectivity of graph and its nodes. A number of algorithms with different goals can be listed in this category, such as: checking if there is a path between two vertices, finding all paths between two vertices, determining strongly connected graphs or graph components, finding non-reachable nodes, etc.
Centrality measures	Allow us to identify most important (central) vertices within a graph, with importance being defined differently depending on the problem we're analyzing. Some of the commonly used centrality measures are: (1) <i>Degree centrality</i> , (2) <i>Closeness centrality</i> and (3) <i>Betweenness centrality</i> . Degree centrality refers to determining the degree of the node, i.e. the number of its immediate neighbours. Closeness centrality aims at calculating topological distance between node and all other nodes. Betweenness centrality is based on calculating the number of times a node finds itself on the shortest path between two other nodes.

The analyses we choose to implement aim at helping application developers to better understand complex dependency graphs. While they can certainly be very useful, these analyses do not represent the core aspect of the framework, but rather have the supporting role. Nevertheless, they have to be integrated into the framework. One of the common ways to do that, would be to offer analysis functionality as a framework tool, which can be optionally used by

application developers. This tool could be in a form of extension of the Visual Studio IDE, standalone application, or a combination.

In order to perform any kind of analysis, we have to obtain the data. In the context of REFRAME this means we have to provide our analyses with the state of dependency graphs (all its reactive nodes and dependencies) at any time during end-user application runtime. The way we can do that depends on whether the analysis tool is integrated and operated from end-user application, whether it is integrated into IDE, or whether it is a standalone application. It includes options such as: (1) directly accessing runtime state of end-user application, (2) shared memory/file and (3) inter-process communication, etc.

6.1.5. Feature 5: Visualize dependency graph

As its name implies, this feature extends the framework by offering visual representation of dependency graphs in a form of *Visualizer* tool. Some of the early questions for this feature that were contemplated during initial conceptualizations and prototyping sessions, were: (1) how do we feed Visualizer with dependency graph data, and (2) what exact technology will be used for drawing graphs.

With regard to providing the dependency graph data to Visualizer, the situation is very similar to providing the data to Analyzer. On one side we have dependency graph data sitting in end-user application's runtime, and on the opposite side we have a tool tasked to perform some operations on that data. Solution for exchanging data between these two parties will probably be applicable for both Analyzer and Visualizer. Visualizer should also be able to display results of Analyzer tool, therefore data exchange should be also possible between two tools. Lastly, we want to keep Visualizer focused solely on its core responsibility (visualizing), and avoid any programming logic which would manipulate externally obtained data. This will allow us to decouple Visualizer as much as possible from other parts of the framework, and preserve the possibility of replacing visualization technology and implementation specifics.

Once the process of obtaining dependency graph data is decided, a question of choosing visualization technology arises. We considered taking following fundamental approaches: (1) using general low-level drawing libraries, or (2) using specialized graph-drawing libraries. Both options may have their advantages. For example, using low-level libraries may allow one to be more flexible, build a solution tailored to its specific needs, and also avoid being dependent

on some third-party solution. On the other hand, using third-party solutions may significantly speed-up the development, require less resources and offer richer feature set.

The representative of the first approach is the set of .NET’s libraries that can be found in *System.Drawing* namespace. These libraries allow us to utilize GDI+ (Graphical Device Interface) to draw different kinds of shapes in windows applications. On the other hand, there are quite a few specialized graph-drawing libraries, available for different programming languages and different environments (both web and desktop). However, in Table 8 we list several prominent solutions that could be used in .NET environment. In next section, these solutions will be further examined and compared by taking into account following aspects: maturity, active maintenance, ease of use, viewer availability and license.

Table 8: Specialized graph-drawing solutions

Library	Description
yFiles.NET	Commercial library for .NET Windows Forms, which allows creating and viewing graphs.
GraphViz	Open-source graph-drawing library written for C programming-language. There is no recent compatible viewer for Windows operating system.
QuikGraph	Open-source graph-drawing library for .NET, successor of deprecated QuickGraph. There is no dedicated graph viewer.
MSAGL	Open-source graph-drawing library for .NET. It also provides viewer.
DGML	XML-based file format (<i>Directed Graph Markup Language</i>) for specifying directed graphs. There is .NET library which can be used to create DGML graphs, and also an integrated viewer for Visual Studio IDE.
Graph#	Open-source graph layout framework for .NET. Includes GUI control for WPF applications.

6.1.6. Feature 6: Generate boilerplate code

In chapter 5 we presented code generation feature as a way to, at least partially, remove the burden of manually writing framework-related boilerplate code. As a specific requirements we offered to provide code generation for two places where we anticipated most of the code repetition would happen, namely: (1) defining reactive nodes, and (2) specifying reactive dependencies. Since the code generation naturally relies on the code it is supposed to generate, in the early phases of REFRAME development we were not able to set the definitive course

of action in this matter. Rather, we had to wait to see how the core APIs of the framework will end-up looking, and how will the framework be used. However, we used this early phase to investigate technological possibilities and options for code generation in .NET environment. In this way we would be ready to choose appropriate code generation solution when the core APIs are ready. In addition, we would also be aware of what can and what cannot be done, and use this information to perhaps shape the framework in a way which would enable us to utilize code generation more easily. Table 9 lists most prominent code generation techniques that are available in .NET environment and Visual Studio IDE. In next section, these techniques will be further examined and possibly tried-out in order to see which one would fit best to our needs.

Table 9: Code generation techniques in .NET framework and Visual Studio IDE

Technique	Description
Text Template Transformation Toolkit (T4)	T4 templates use a mixture of fixed text blocks and control logic to generate text string or a file at design or run time. The resulting, generated text can be text of any kind, including source code.
CodeDOM	CodeDOM is a collection of classes which represent common types of source code elements, which can be used to generate source code.
Reflection.Emit	Reflection.Emit is a collection of classes that allow compiler or tool to emit meta-data and IL (Intermediate Language) at run time.
Expression trees	Expression trees are another way to generate code in .NET, but without having to dive into IL. It allows developers to use tree-like data structures to represent and generate code.
IL Rewriting	IL Rewriting refers to the process of altering already compiled IL code, stored in an assembly.
Domain-specific languages (DSLs)	DSLs are created on top of general purpose languages in order to specialize for particular problem domain.
.NET Compiler platform (Roslyn API)	Allows developers to access information generated in different phases of compiler pipeline. This means that it can be used to analyze and make modifications to source code, analyze semantic model, work with the project structure etc.
Code snippets	Visual Studio IDE feature that allows us to define small blocks of reusable code which can be easily inserted into code file.
Quick actions	Visual Studio IDE feature that allows us to refactor, generate or modify source code.

6.2. Assess and select

In this section we aim at assessing design and implementation ideas that were generated in section 6.1, and selecting those ideas that are not only promising adequate results but are also feasible to implement. In practice, this involved searching through both scientific and professional literature, and also prototyping and trying out different options. These efforts will be categorized according to requirement feature stated in SRS document.

6.2.1. Feature 1: Specify reactive dependency

As discussed in previous section, when it comes down to question what data members are we going to allow to act as reactive nodes, we can identify two approaches. The first, direct approach, is to allow traditional data members, such as *fields*, to act as reactive nodes. The second, indirect approach, is to represent fields with "proxy" method members, and allow them to act as reactive nodes. These "proxies" usually come in a form of *getter* and *setter* methods, or some of the language-specific syntax alternatives such as .NET *properties*.

We mentioned that first approach has some adverse effects in a form of breaking *data hiding* principle. This is in fact serious implication, because data hiding is performed in order to keep the complexity of system under control, and also to prevent unintentional or intentional malicious changes. Since we are reluctant to force developers break basic principles of OO programming in favor of using REFRAME, second approach seems preferable. Although both are acceptable solutions, the question of whether to use getter and setter methods, or properties to represent fields still remains. One possible disadvantage of getter and setter methods may be the fact that they are methods and they clearly look like methods. While developers are perfectly aware of their "proxy" role, the impression of working with methods when we actually want to work with data members, still remains. Properties, on the other hand, also do the job of protecting fields, but at the same time allow developers to retain the sense of working with real data members, i.e. state. Therefore, we will take advantage of the fact that REFRAME is being developed in C# .NET, and choose properties as data members for reactive nodes. It should be noted, however, that in other programming languages which do not provide properties as a feature, we can still resort to using fields or getter and setter methods.

In previous section we presented several arguments and scenarios where allowing method members to be used as a base for reactive nodes may be beneficial. This includes using methods

as proxies to data members, using methods to accumulate multiple state updates, or scheduling methods as some form of task processing. Unlike with data members, with method members there are no proxies, but only one, direct approach. Therefore, as suggested in requirement specification, we will also allow method members to act as reactive nodes.

Extending programming language by adding new features and keywords, and implementing custom compiler supporting them would certainly provide developers with smoother experience in handling reactive nodes. However, implementing custom compiler is complex undertaking, and is seldom justified. Framework users who would try to adapt or extend such low-level framework would face much larger challenge than if the framework would be at higher level. Using custom compilers also means that we diverged from official compiler, which begs the question of keeping compiler updated. It can also be argued that such extension of the standard language constructs may result in language syntax becoming over-saturated over time. However, instead of implementing full-scale custom compiler, in one of the prototyping iterations we tried-out .NET compiler platform SDK (Roslyn APIs) [12], as possibly useful tool. Provided APIs allow developers to access information generated in different phases of compiler pipeline. This means we can use it to analyze and make modifications to source code, analyze semantic model, work with the project structure etc. Unfortunately, .NET Compiler Platform SDK does not provide a way to extend the compiler. However, the features it does provide may be of great use in framework development, especially for developing tools which accompany the framework.

As previously discussed, meta-attributes are frequently used in frameworks to provide additional (meta) data for code elements. In one of the prototyping iterations we attempted to apply them to explicitly designate the role of reactive node for class members and classes themselves. However, we found meta-attributes not suitable for this particular purpose. Meta-attributes are usually very simple classes, with small number of members, if any. On the other hand, even in the early phase of REFRAME development it was quite obvious that reactive node is not going to be a trivial class, but will surely have several state members and accompanying methods. Passing large number of parameters to meta-attribute to fill its state is fairly cumbersome. A more important reason is the fact that meta-attributes operate on a class level, i.e. they are assigned to a class not an object. This means that it was not possible to associate different parameter values for different objects of the same class. However, even though this

particular technique did not come to life, trying it out spurred ideas on possible uses of other meta-programming techniques, such as *reflection*.

One of the usual ways frameworks extend language capabilities, is by offering new abstractions in a form of classes to express certain domain concepts. This does not require compiler modifications, and guarantees more developers will be able to understand the framework and adapt it if required. It also means that, unlike with meta-attributes, we are able to have a separate instance of reactive node for each object and its member. Therefore we chose to implement reactive nodes simply as new classes. However, since reactive dependencies and related features are not built into the programming language itself, this approach certainly involves more boilerplate code in end-user application.

Implementing reactive node classes in a form of reactive types, which inherit or aggregate common types and extend them with reactive behavior, was also one of the ideas. However, there is very large number of built-in types, and it is inconvenient to extend them all. And even if we managed to do this, there are still custom classes created by application developers which we cannot anticipate. This approach might work only if we wanted to introduce reactive behavior for a fixed and limited data types. Instead, we wanted to have more general solution, i.e. reactive nodes being able to support members of any type. Therefore, we chose to implement reactive nodes unaware of the underlying member type.

As discussed before, in order for reactive dependencies to be able to form acyclic graphs, individual reactive nodes have to be able to play both predecessor and successor roles. Not being able to do that would be a serious limitation. Whether listed design patterns express reactive node's roles as one or two separate abstractions is in direct relation to this issue. For example, if predecessor and successor roles are expressed as separate abstract classes, no concrete class can be both predecessor and successor (due to lack of support for multiple inheritance). Such design decision would obviously prevent us from meeting aforementioned requirement. Alternative design, frequently applied in such cases, could involve exposing one or both reactive node's roles as interfaces. Now the class representing reactive node can implement both interfaces (or interface and abstract class) and thus support both predecessor and successor roles. Choosing to implement at least one of the roles as an abstract class offers us possibility to reuse implementation common to specific role. However, as discussed in Chapter 2, concrete classes inheriting roles as abstract classes are not real specializations, i.e. there is no real *is-a* relationship. Such

inheritance would then disrupt natural class hierarchy, or it may be very hard to introduce into already existing hierarchy. On the other hand, we may choose to represent both reactive node's roles as interfaces, which seriously limits implementation reuse. Another approach to this is offered by Observer revisited pattern. As already discussed, the classes implementing predecessor role (Observable) and successor role (Observer) are treated and manipulated as base (Java) objects, and the implementation in charge of handling reactive dependencies is delegated to another class. In this way we would avoid artificial inheritance and would gain additional flexibility, but the question of how to enforce proper behavior of these plain objects would still remain. That being said, we argue that the best solution to support reactive node having both roles, is to implement both roles as one abstraction. This in fact means that abstraction for reactive node in general is sufficient, and we do not need separate abstractions for reactive node's roles. This is in line with the design proposed in Propagator pattern.

Now, the question whether we are going to use inheritance or composition also has some important implications with regard to level of granularity at which we can define reactive nodes. Design proposed in all three versions of Observer pattern, and in Propagator pattern as well, relies on inheritance and treats whole domain object as one reactive node. That is in contrast with our fundamental premise - that reactive nodes can be defined at the level of object's individual state and behavior, i.e. properties and methods. One of the possible solution that were considered was to parametrize reactive nodes. So, for example, while entire object would be considered as only one reactive node, additional parameter would be passed along this reactive node indicating exact member that was triggered, or that has to be updated. However, although this approach would introduce finer granularity in triggering and updating individual reactive nodes, it would alter the notion of reactive node and make it difficult to construct and manage dependency graphs. A potentially fitting design idea is proposed in Event notification pattern. Here a composition is used to allow any class to have arbitrary number of StateChange members, which take the role of predecessors, and also arbitrary number of EventStub members with the role of successors. This allows controlling the level of granulation at which reactive nodes are specified. The downside is that StateChange and EventStub are separate classes, so we would have to join them into one as in Propagator pattern.

The idea behind .NET events is very similar to idea of Event notification pattern. Since events are just another type of class members in .NET, for particular class we can declare as

many events as we want. Similarly, event handlers are methods, so naturally there can be arbitrary number of them also. That means that we can have multiple predecessor nodes and multiple successor nodes for each object, and thus control the level of granulation. One of the problems of .NET events is fair amount of boilerplate code. For each predecessor node we have to implement additional class member of type event, and for each successor node we have to implement event handler method. In addition, as with Event notification pattern, important downside is that events as representatives of predecessor role and a handler methods as representatives of successor role are separate concepts. Java event model combines the shortcomings of Observer pattern and .NET events, and is also not suitable for expressing predecessor and successor roles in REFRAME.

Although reactive dependencies are key concept in REFRAME, at this point we will not express them as an explicit, separate abstraction. Rather, we are going to express them as an association of two reactive nodes. Such view of reactive dependencies as a concept determined by ordered pairs of reactive nodes is firmly grounded in graph theory. This is further justified by the fact that no additional data is planned to be associated with individual reactive dependencies.

6.2.2. Feature 2: Construct dependency graph

Contrary to reactive dependencies, with dependency graph we advocate the use of explicit abstraction. There are several reasons for this. Firstly, in REFRAME we want to allow coexistence of multiple dependency graphs. This implies there is a need to uniquely identify each dependency graph. Also, there are other possible members describing the state of dependency graph, that may be required. Apart of the dependency graph state, there is whole range of behavior related to constructing, storing, fetching and updating reactive dependencies that naturally fits into dependency graph abstraction. In addition, most implementations of directed acyclic graph have graph as an abstraction. With regard to related design patterns, this means that we side with design options proposing a manager abstraction as a central entity in charge of dependencies. They present it as a more advisable option when dealing with complex and large dependency graph. This is also an option that Hinze et al. [74] describes as centralized or middleware approach.

While all three mentioned data structures used for implementing DAGs can successfully support basic DAG operations, we find the idea of adjusted list as the most fitting option in our

context. For example, adjacency matrix is cumbersome when there is a need to dynamically add or remove nodes from graph - something that frequently happens in REFRAME usage scenarios. Also, adjacency matrix stores information about dependencies for each possible combination of existing nodes, although, in reality dependencies exist for only small subset of these combinations. Some search and traversing operations, such as finding nodes associated with particular node, also require more processing in adjacency matrix. Finally, in adjacency matrix we use indexes to refer to particular reactive nodes, which is not natural when nodes are implemented as full-scale objects. Incidence matrix shares shortcomings of the adjacency matrix, with addition of being more efficient in terms of space in case of sparse graphs, but also being less efficient in terms of performance. Generally speaking, matrix structures are seen as inefficient in terms of space for all except the most dense graphs. Adjacency list, on the other hand, is more space efficient, can better support adding new nodes to graph dynamically, and is better suited to work with nodes as full-scale objects. In order to allow easier traversal through the graph, each reactive node will contain separate collections of its predecessor nodes as well as successor nodes.

6.2.3. Feature 3: Perform dependency graph update process

When we talk about performing graph update process, the first thing we need to determine is whether the *triggering change* will be invoked automatically by the framework or manually by the application developer. In case of dependency graph structural changes, triggering change automatically when reactive nodes and dependencies are added or removed poses as a most transparent and straightforward option from the perspective of application developer. However, as we mentioned in subsection 6.1.3, most of the time this would lead to huge number of unnecessary updates performed. Unfortunately, since the code invoking dependency graph structural changes is application code authored by application developers, framework cannot automatically guess or anticipate how many of these changes will be simultaneously introduced. This part of the application code is fairly arbitrary, and even if we would set framework to somehow analyze application code, it would be extremely hard for REFRAME to infer what part of the application code makes a coherent unit after which update process should be performed. So, in order to prevent redundant updates from happening, it makes more sense in this case to request application developer to explicitly invoke update process.

Triggering change due to state member change or a method invocation suffers from some of the same issues as the triggering change due to graph structural changes. For example, state changes and method invocations are also part of the application code, which framework cannot easily interfere with to make automatic invocations of graph update. Compiler-based approaches could possibly achieve that, but we decided not to side with these approaches because we want to be dependent solely on official compiler. Alternative is to insert (inject) code that triggers change into application code where state members are changed or methods are invoked. The simplest way from framework developer's point of view is to leave that for application developer to do manually. As far as this feature is concerned we will opt for this strategy. However, unlike in the case of graph structural changes, application code in charge of state member changes or method invocations will most likely be more uniform. For example, code that triggers change due to state change may be inserted as last line of code in property setter. Analogously, triggering change due to method invocation may be inserted as a last line of code in invoked method. This opens up possibility to automatically add these lines of code, either during design time or (post) compile time. We will consider using these options when describing feature related to code generation tool.

In order to identify what parts of dependency graph are inconsistent and have to be updated we have to find source reactive node. As we discussed, in case of triggering individual reactive node due to state change or method invocation, the situation is clear - the source node is reactive node being triggered. However, in cases where graph inconsistency cannot be reduced to single source reactive node (multiple structural changes, multiple reactive node's triggering simultaneously), REFRAME will provide two techniques in order to prevent multiple updates: (1) *updating entire dependency graph*, (2) *introducing temporary source node*. The first technique is fairly straightforward, and contains following steps: (1) suspend graph update process, (2) make changes that would originally cause performing update process multiple times, (3) resume graph update process, (4) perform update process of the entire dependency graph. This technique will likely result in some parts of the graph updating unnecessarily, however, depending on the size of graph and extent of changes previously done, this may not be a large part of graph. Sometimes, however, we have multiple source reactive nodes, but their combined effect does not justify updating entire, potentially large dependency graph. In such cases the second, more complex technique may be better option. Since tracking parts of the graph influenced by

multiple source reactive nodes is very complex, we will reduce it to a case with one source reactive node. This will be done by introducing temporary source reactive node as a predecessor to all source reactive nodes.

Listing 6.4: Introducing temporary source reactive node

```
let G is dependency graph
let sourceNodes is List of triggered nodes
procedure PerformUpdate()
  let T is temporary node
  add T to G
  foreach node in sourceNodes
    add dependency from T to node
  endfor
  PerformUpdate(temporaryNode)
  foreach node in sourceNodes
    remove dependency from T to node
  endfor
  remove T from G
end
```

Determining the nodes which have to be updated and the exact order in which this should happen requires performing topological sorting (ordering) of the dependency graph. Depending on the situation, topological sorting algorithm will need to be capable of sorting entire graph or only part of the graph which is directly or indirectly dependent on one or more source reactive nodes. As we discussed, two algorithms are commonly used for topological sorting, namely: Kahn's algorithm and Tarjan's algorithm. We found both algorithms to be suitable in the context of REFRAME, with their subtle differences being associated with underlying traversal algorithms (BFS and DFS respectively). Their advantages and disadvantages tend to manifest with regard to particular structural properties of dependency graphs. Since there is no particular dependency graph type that we can say will dominate in REFRAME use, it is hard to argue which algorithm will be better for REFRAME. Therefore, we choose to implement Tarjan's algorithm based on DFS traversal algorithm as a default topological sorting algorithm in REFRAME. However, the framework will be designed in a way to allow alternative algorithms

to be implemented and plugged in instead of default one.

In order to implement parallel graph update process, we analyzed and adapted Coffman-Graham's algorithm in one of the prototyping sessions. Transitive reduction, as a first step of the algorithm, aims at removing transitive dependencies from the graph in order to minimize the graph's width. In the algorithm, the width corresponds to number of processors in multiprocessor systems assigned to processing the nodes. However, we will omit this step, since nodes are going to be processed in threads, and underlying CPU scheduler will decide how these threads are going to be assigned to CPU cores. Also, we do not want to alter original graph because dependencies between nodes carry important semantics. The second step is topological sorting, and this is something we already have to implement regardless of parallelization. Therefore, idea in the third step (graph layering) of the Coffman-Graham's algorithm is the only thing we will apply in order to enable parallelization.

Graph layering divides graph into arbitrary number of layers, with each layer containing at least one, but usually more than one node. Nodes belonging to the same layer are independent, and can therefore be processed in parallel. When all nodes from one layer are processed, we can proceed with the next layer, up until all layers are processed.

Listing 6.5: Pseudocode of graph layering algorithm (third step of Coffman-Graham's algorithm)

```
let sortedNodes is topologically sorted List of nodes
procedure AssignLayers()
    foreach node in sortedNodes
        predecessors = get predecessors of the current node
        if predecessors is empty
            assign node with layer = 1
        else
            maxLayer = determine maximal layer of predecessors
            assign node with layer = maxLayer + 1
        endif
    endfor
end
```

With regard to performance, parallelization of graph update process should be used with caution, because, while it will result in performance increases in some cases, in other cases may make performance even worse than with sequential update process. For example, let say we have dependency graph with nodes $n_1, n_2, n_3, \dots, n_N$, and time required for updating each individual node being $t_1, t_2, t_3, \dots, t_n$ respectively. If we would perform entire graph update process sequentially, the total amount time required would equal $T = \sum_{i=1}^N t_i$, i.e. the sum of all update times of individual nodes. In case of parallel update, however, when individual nodes are assorted to layers $l_1, l_2, l_3, \dots, l_M$, the time required would equal $T = \sum_{j=1}^M \max(t)_j$, i.e. the sum of update times of longest individual nodes for each layer. If the total number of layers is significantly less than total number of nodes ($M < N$) it is reasonable to expect performance improvement. However, this improvement can easily be annulled or even surpassed with the overhead costs resulting from spawning and managing threads. Also, the number of nodes in layer that can be simultaneously processed depends on the number of CPU cores. If the number of nodes in particular layer exceeds the number of CPU cores, it will require more than $\max(t)$ time to process that layer. Situation may vary depending on many factors, however, the general rule would be that if the update processes of individual reactive nodes are performance-intensive, then it makes sense to try-out the parallel update process. Otherwise, overhead costs of dealing with threads will probably be higher than potential performance improvements.

Responsiveness of user interface, as one of the goals of asynchronous programming, aligns well with features we want to have in REFRAME. As we already discussed, performing graph update process may take some time, and it is important not to block the main thread and make GUI unresponsive. In .NET asynchronous programming, this is ideally done without using additional threads to separate GUI processing from performing lengthy asynchronous operation. However, this is possible only when asynchronous operation is I/O bound operation, i.e. when it is handled by external system (e.g. operation system, network device, web service, human user etc.). In case the asynchronous operation is CPU bound, i.e. there is no waiting for external system to respond, we have to resort to using additional threads if we want to assure responsiveness. This exactly is the case in REFRAME, since operations contained in update process are expected to be CPU bound. Therefore, in cases where the goal is to ensure responsiveness of end user application, REFRAME will offer execution of update process in separate thread to avoid blocking the GUI thread. With regard to patterns of asynchronous programming, as we

have reported, three patterns are in use. However, official .NET documentation describes EAP and APM as legacy patterns, and advises the use of Task-based asynchronous pattern (TAP) for new applications [19]. Traditional abstractions (e.g. *Thread*) for multithreaded programming that can be found in *System.Threading* namespace are nowadays considered to be low-level abstractions. Official .NET documentation advises against their use, and instead recommends using higher-level abstractions such as *Task*, available in Task Parallel Library.

The overall graph update process in REFRAME consists of a number of update processes of individual reactive nodes, that should be executed in proper order. Each of these node update processes can be seen as a separate **task**. Therefore, in REFRAME we are concerned with task parallelism, while data parallelism is out of the framework's scope. In .NET, task parallelism is supported by Task Parallel Library set around *Task* class. Primary benefits TPL brings is: (1) more efficient and scalable use of system resources, and (2) more programmatic control than it is possible by using lower-level *Thread* abstractions [19]. This includes utilization of existing threads from *ThreadPool* instead of creating new ones, algorithms for load balancing, and rich set of Task-supporting APIs. These benefits allow TPL to form not only the basis of parallel programming, but also multithreaded and asynchronous programming (TAP). As a comprehensive solution, TPL library will be used in REFRAME.

6.2.4. Feature 4: Analyze dependency graph

As discussed in *Imagine* and *Brainstorm* section, there is a large number of algorithms with different goals that can be used to analyze graph structures. Some of these algorithms are clearly useful for analyzing dependency graphs in the context of REFRAME. Some, however, would not produce meaningful results, or their suitability should yet be determined subsequently after some time spent using the framework. Therefore, in this initial version of the framework (in accordance to requirements set in SRS) we will include subset of analyses which we consider to be suitable for the context of reactive dependencies. A framework will also be designed in a way to be easily extended with other analyses.

The process of analyzing structure of obtained dependency graph will start by choosing one of the 6 abstraction levels at which we want to analyze the graph (vertical reduction). After that, complete graph at chosen level can be listed, or we can decide to fetch only a subset of graph nodes (horizontal reduction). Following mechanisms for defining graph subset are chosen to be

implemented: (1) *filtering by affiliation*, (2) *filtering by role*, and (3) *filtering by association*. Filtering by affiliation is closely related to graph abstraction levels, and assumes being able to show only reactive nodes which belong to some particular parent node. For example, if we choose to list dependency graph on an object-member abstraction level, we can choose to list only reactive nodes that belong to some particular assembly, namespace, class or an object. Filtering by role assumes being able to show only reactive nodes with some particular role in a graph (e.g. graph's source nodes). Finally, filtering by association enables us to list reactive nodes which are associated with some particular node of interest. In this way we can fetch all of node's predecessors, successors and neighbors, or only part of them (e.g. predecessors which are source nodes). Since these mechanism can be combined, a fair number of different analyses can be produced.

With regard to graph algorithms shown in Table 7 we can see that some of them we already intend to use. For example, Graph traversing algorithms and topological sorting algorithms are key for assuring proper order of updating individual reactive nodes within REFRAME's update process. This is also the case with cycle detection algorithms which help us assure dependency graph is kept acyclic. On the other hand, spanning tree algorithms are intended to be used on undirected graphs or directed multigraphs, while dependency graphs are directed acyclic graphs. A spanning tree devised from dependency graph would have compromised structure in terms of missing reactive dependencies, so it would not only be useless but also misleading. Similar situation is also with maximum-flow algorithms, which only make sense for weighed graphs. Shortest path algorithms can be run on dependency graphs, however, at this point we do not see how these would be meaningfully interpreted. Therefore, we will omit them from this version of framework.

In case of connectivity algorithms, for example, checking and finding paths between vertices would allow us to see if two reactive nodes are directly or indirectly dependent. Furthermore, algorithm for finding non-reachable nodes would allow us to find orphan nodes. On the other hand, algorithms for determining strongly connected graphs or components are not useful in the context of REFRAME. These algorithms try to find path between each pair of vertices in a graph or a component, which in REFRAME's directed acyclic graph will practically never be the case. Together with connectivity algorithms, centrality measures bear the largest relevance to REFRAME's dependency graphs. They may be useful for finding the most influential reactive

nodes in the graph (e.g. parameter which is central to some particular calculation model). In this version of framework we will implement degree centrality measures as they are most straightforward for users to understand and interpret.

Aforementioned analyses are going to be made available for application developers in a form of framework's accompanying tool. The first idea for this tool was to be completely integrated into IDE as an Visual Studio Extension [16]. However, after a few prototyping sessions, several problems became evident, most important being: (1) very difficult development of tool's GUI in a form of extension, and (2) extension's inability to access runtime state of end-user application.

In an attempt to resolve problems with GUI development, we decided to use traditional *Windows Forms* for tool's user interface development. Visual Studio Extension would then only be used as a means to invoke analysis tool from Visual Studio IDE. This should not only be removing restrictions on GUI development, but it should also make easier to transition analysis tool to standalone external tool, easily used also by end-users.

Accessing runtime state of end-user application which is needed for obtaining dependency graph data on which analyses would be performed, proved to be a bigger problem. We dismissed the option of executing our tool within end-user application very early in development phase. Instead, we opted for external tool, independent of end-user application. This is because we did not want to require application developers to alter the end-application code just to be able to run our tool. After a few prototyping sessions, it became apparent that with Visual Studio Extension we cannot access a runtime state of running application. One of the potential solutions that arose, was to utilize IDE's own *debugger* to inspect the runtime state. However, after making reasonable effort, no suitable APIs capable of providing that were found in debugger. In addition, we also considered implementing our own custom debugger, but after examining fairly scarce documentation for customizing debugger, we realized this would require too much effort with uncertain results.

At this point we were stuck with a problem of being required to inspect runtime state of end-user application from a tool which is run in a completely separate process. Therefore, the next option to try-out in prototyping session was *inter-process communication* (IPC). IPC can be achieved in a number of different ways, ranging from simple file sharing to using Windows Communication Foundation (WCF) framework. Sharing dependency graph data as a file was not an option, as it would require invoking some kind of export feature from the end-user appli-

cation and then importing data into analysis tool. This would mean altering end-user application's code, and also possible performance issues in case of large dependency graphs. Although WCF is a suitable solution, it is a whole framework, and we wanted to avoid additional dependencies. Instead, we decided to use Named Pipes [9] as a fast, reliable and compact solution for achieving IPC in a client-server style. Unlike WCF, Named Pipes are part of standard *System.Core* assembly, so no additional assemblies are required. In this configuration, end-user application would act as *server*, and would respond to requests sent by analysis tool which took the role of a *client*. Exchanged requests and responses will use XML format.

6.2.5. Feature 5: Visualize dependency graph

In section 6.1 we concluded that both Analyzer and Visualizer tool share the same issues related to obtaining dependency graph data from end-user application's runtime. This opened-up the possibility for these tools to also share the same solution. As discussed in the section devoted to Analyzer tool, we opted for inter-process communication implemented using Named Pipes as a solution. However, instead of both Analyzer and Visualizer tool communicating via pipes with end-user application, it seemed reasonable to make only Analyzer do that. Analyzer would then be in charge of storing dependency graph data in an appropriate data structure (on a tool's side) and performing different analyses. On the other hand, Visualizer would only need to take data from the Analyzer and display it. Whether this data represents original ("intact") dependency graph, or a result from one of analyses, to Visualizer this would make no difference. In this way we have a possibility to make Visualizer highly decoupled component, which is easy to maintain and even replace.

With regard to chosen technology, although it may seem as a legitimate option, using low-level drawing libraries such as the ones found in .NET's *System.Drawing* namespace, proved to be inadequate. While the drawing of individual shapes is not particularly problematic, drawing graphs comes with a whole set of issues, a lot of which were the subject of thorough scientific work in the last several decades. The example of such complex issues is determining the layout of nodes and edges which ensures proper readability and comprehension of graph. Since resolving such low-level issues falls out of the scope of this dissertation, it makes more sense to search for existing graph-drawing library and focus on specifics of visualization in REFRAME.

As hinted in previous section, solutions listed in Table 8 were examined and compared. In

the end we decided to go for DGML option, as it proved to be a stable and mature solution, offering most of the things we required out-of-the-box, and for free. Building graphs with DGML was fairly straightforward thanks to included .NET library (Microsoft.VisualStudio.GraphModel namespace), while viewing graphs was possible using Visual Studio's built-in graph editor. Using other mentioned solutions would also be a viable option, but we would have to carefully weigh-in some of the adverse aspects. For example, yFiles.NET is a very able and feature-rich library, however we did not want to make REFRAME (open-source) dependent on commercial solution. On the other hand, while being open-source solutions, GraphViz and QuikGraph did not offer a dedicated graph viewer. Finally, MSAGL and Graph# offered a complete set of features, however it seems they are no longer actively developed and maintained. Nevertheless, choosing DGML as a graph visualization technology is not a definitive and immutable decision. Depending on the future needs of the framework, we may want to replace it with technology which would require more effort to adapt, but would also offer more flexibility and more features.

6.2.6. Feature 6: Generate boilerplate code

In this section we discuss how individual code generation techniques (listed in 6.1) respond to exact needs for code generation in REFRAME. In order to do that, we had to know how REFRAME's core APIs are going to look like from the perspective of application developer (framework user). This is why activities described in this section were carried out a bit latter, i.e. after core features were already designed. The prototyping sessions in which we tried-out the REFRAME's core APIs showed that (as we anticipated) most of the repetitive boilerplate code deals with: (1) defining reactive nodes, (2) specifying reactive dependencies between them, but also (3) triggering change. Code statements which perform these tasks are added by application developer at appropriate places in existing code structures (classes). This means that if we want to support these tasks by code generation, we have to make possible to inject arbitrary code into existing classes at appropriate places (e.g. existing method or setter bodies).

While number of code generation techniques such as T4 templates, CodeDom, Reflection.Emit and Expression trees do allow us to generate code (design, compile or even runtime), non of these techniques are particularly suitable for changing content of existing code. Code injection, on the other hand, is possible through *IL rewriting* (at runtime) and through *.NET*

Compiler platform (at design time). IL rewriting is usually done using specialized third-party libraries due to extreme complexity and the lack of direct support in .NET framework. One of the widely used mature libraries that supports code injection, and that also happens to be free is Cecil [49]. In examining Cecil's code injection features, several observations were acquired: (1) By using Cecil, we are introducing additional, third-party library, and make our framework depend on it, (2) using Cecil to generate reactive nodes and reactive dependencies would not be justified since we would have to define dependency graph in some other way (e.g. through XML specification or through GUI), (3) injecting *triggering change* statement could be a viable option but it is questionable if this is worth introducing the whole framework, (4) code injection results in creation of new version of assembly which cannot be reloaded at runtime, (5) debugging process can be impaired due to difference between source code and IL.

Runtime code injection as a feature in .NET compiler APIs has been requested by developers in .NET community for quite some time. But although the public discussions (visible on the project's GitHub pages) are still ongoing, no implementation of the feature is at sight. However, as mentioned, .NET compiler APIs allow developers to manipulate syntax trees of existing source code, including injecting arbitrary lines of code into properties or methods. We managed to try this out during one of the prototyping sessions. The positive side of .NET compiler APIs is that we stick with the official .NET APIs and a product which has been constantly evolving. Also, altering syntax trees results in altering source code, so there is no "hidden" statements like in the case of IL rewriting which could impair the process of debugging. However, in order to inject code in charge of specifying reactive nodes and dependencies, and triggering change, application developers would have to provide required information in some other way.

One of the ways we tried to provide this information is using meta-attributes. During one of the prototyping sessions we assigned custom meta-attributes to properties we wanted to inject with triggering change statement. Then, using .NET compiler APIs, syntax tree of the class is analyzed and all properties decorated with this particular meta-attribute were injected with required code. While this approach showed potential, the process of assigning meta-attribute to class members was no easier than adding the very code we wanted to inject. The similar situation was also with generating code statements in charge of specifying reactive nodes and dependencies. In our prototyping sessions we considered using XML-based structure for specifying meta-data required for generation of reactive nodes and dependencies. However, even

using GUI tool which we created for specifying this meta-data, required the same or more effort as manually adding the code we wanted to generate.

The decision we made after trying out different options, was to (at least in this iteration) abandon the "heavy-weight" solutions and start with something simpler. The first step would be to try simplifying framework's APIs in order to make framework easier to use and code more readable. This will be done by creating DSL layer on top of REFRAME's core APIs. DSLs are frequently used for this, e.g. by extending existing languages in order to better support and express concepts for a specific problem domain. In the second step, we would offer option to generate the skeleton of individual code statements, in which application developers would fill-in the variable parts. For this we considered utilizing Visual Studio's *code snippets* or *quick actions* features. After trying both options, we found code snippets to be more convenient as they offered everything we needed and were also much easier to implement.

Despite choosing combination of DSL and code snippets as a solution for code generation for this iteration of framework, we remain open for future alternative solutions. We are particularly interested in new and developing technologies which constantly emerge in .NET Compiler API, such as *source generators* (still a preview option at the time of writing).

6.3. Sketch and build

In this section we utilize SRS document, as well as ideas collected and selected in previous sections, in order to design and implement REFRAME. Features 1, 2 and 3 from SRS document deal with the "core" part of the framework, i.e. programming abstractions and operations required to allow developers to express reactive dependencies and perform update process in end-user application. This part of the framework is absolutely essential and at the same time sufficient for framework to be usable, so we refer to it as *ReframeCore*. The efforts in designing and implementing ReframeCore will occupy the first part of this section. On the other hand, features 4, 5, and 6 deal with the part of the framework which is optional for use, and aims at increasing productivity and comprehension in working with reactive dependencies. This second part of the framework is referred as *ReframeTools*, and consists of auxiliary tools for analyzing and visualizing dependency graphs, as well as generating parts of boilerplate code. The efforts in designing and implementing this part of the framework will be described in second part of this section. Finally, in the third part of this section, we will distance ourselves from the de-

tailed design aspects and take a high-level perspective in describing overall architecture of the framework.

Figure 17 shows conceptual architecture of REFRAME and its surroundings. We can see that REFRAME consists of two related parts: (1) *ReframeCore* and (2) *ReframeTools*. When developing end-user applications with reactive dependencies, developers can use abstractions and mechanisms of *ReframeCore* to express reactive nodes, reactive dependencies and dependency graphs. This requires end-user application to hold a reference to *ReframeCore*. Optionally, application developers can also use *ReframeTools* to enhance the use of *ReframeCore* in terms of analyzing and visualizing dependency graphs, and generating part of the boilerplate code.

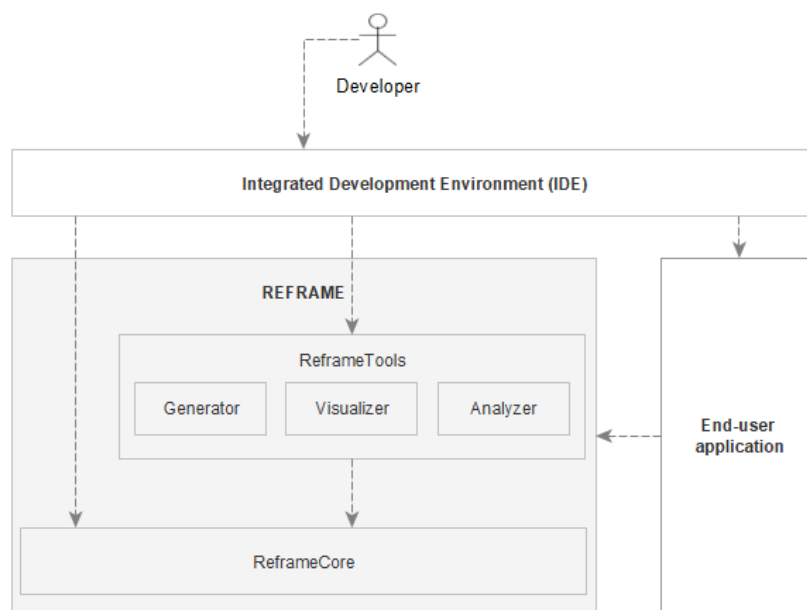


Figure 17: Overall architecture of REFRAME

6.3.1. ReframeCore

6.3.1.1. Reactive nodes

Reactive node is a strong and fundamental concept in REFRAME's problem domain. It is used as a building block for expressing other concepts such as reactive dependencies and dependency graphs. Therefore, it is reasonable to model it as an explicit abstraction in solution space. We begin this by assigning a class *Node* to represent reactive node, and then continue describing desired characteristics of reactive nodes by adding state and behavior to this class. As we will see, iterative refinement will result in forming node hierarchy.

One of the things that are evident from SRS document is that reactive node needs to point at the class member it represents, for example by adding a dedicated property to Node class which would refer to a class member. This can be done by: (1) storing member name (i.e. method name or property name) as a text string, or (2) using meta-programming techniques to store meta-data member representations, such as *PropertyInfo*, *MethodInfo*, or *MemberInfo* objects in .NET, obtained through reflection. In order for REFRAME's core design to be more general and more easily applicable to different technologies, we will try to favor technology-agnostic design decisions. In this case, it means that we will store only plain member names in *MemberName* property. Reflection is indeed going to be used to fetch other meta-data when required, however, this functionality is going to be placed in helper library classes, in order to keep core classes "clean".

In addition to a member, reactive node also has to point at the exact instance of a class owning that member. This will be implemented through *OwnerObject* property. One of the questions that arose in this context was the declaring type of the *OwnerObject* property. The possible options included: (1) using .NET base *Object* type, (2) introducing new "reactive" type, or (3) using .NET generic types. Prototyping sessions showed that by introducing new "reactive" type we would interfere with class hierarchies of business objects implemented by application developers. Using generics also proved to be impractical, and also not necessary. Therefore, again we opted for simple, and most general solution of using .NET base *Object* type to store owner object.

With regard to reference to owner object, there was one additional issue related to resource deallocation that should be mentioned. When you create an instance of an object in .NET in a conventional manner, we say that application holds a *strong reference* to an object. Strong references have an implication that as long as there is even one strong reference to it, an object cannot be collected by garbage collector. This means that if we would pass strong references of owner objects to reactive nodes, the existence of reactive nodes would prevent deallocation of their respective owner objects. Even in the case of removing all strong references from application's code, the strong references might still be existent in framework's code. Solving this issue from end-application's code would place too much burden on application developers, since it would require checking for references in reactive nodes whenever an object is removed in application. Doing that on the framework's code side is also not possible since we are not

able to obtain information on the exact number of strong references to particular object. The remaining option was to use so-called *weak references* instead of strong references to hold an owner object in reactive nodes. The weak references still provide access to an object, however their defining characteristic is that they do not prevent garbage collector from collecting objects. This means that if strong references of an object are removed from application, weak references in reactive nodes are not going to get in a way of garbage collector.

<i>Node</i>
+ Identifier : uint
+ Member : string
+ OwnerObject : object
- WeakOwnerObject : WeakReference
- UpdateMethod : Action
+ Node(owner:object, member:string)
- GenerateIdentifier() : uint
+ Update() : void

Figure 18: Essential members of Node class

Another important aspect of reactive nodes relevant to manipulation of individual reactive nodes, understanding of reactive dependencies and dependency graphs, and performing correct update process, is being able to uniquely identify reactive node. When we identify or compare objects in OO setting, we usually do this by examining references of these objects. For example, two objects, *a* and *b*, are equal if their references are the same, i.e. their references point to the same location in memory. With reactive nodes this is not the case. We can easily make several separate instances of Node class, and as long they refer to the same owner object and the same member, they are to be seen as the same reactive node. There are ways in C# programming language to override default implementations of *equality operator* "==" (performs reference comparison in case of reference types) and *Equals* method (performs value comparison) in order to alter comparison rules for certain types. However, instead of that, we will assign reactive node with explicit and unique identifier. This identifier would then be used to differentiate between different reactive nodes during usual operations within REFRAME, but also as a handle to fetch reactive nodes from dependency graphs. Having explicit identifier can also be useful as a human-readable reference to reactive node in logging, reporting, and other tools. Therefore, we need to add another property called *Identifier* to Node class, and also assure it being really unique for reactive nodes referring to the same owner object and member. This was done by automatically generating identifier based on combination of hash values of *OwnerObject* reference and *MemberName*.

Listing 6.6: Generating identifier for reactive nodes

```
protected uint GenerateIdentifier(object owner, string member)
{
    uint id = 0;
    if (owner != null && member != "")
    {
        id = (uint)(owner.GetHashCode() ^ member.GetHashCode());
    }
    return id;
}
```

Finally, each node should be capable of invoking arbitrary behavior on *OwnerObject*, which would make individual reactive node update during overall, dependency graph update process. To make this very convenient, *Node* class will provide parameterless *Update* method, which returns no result (i.e. void). Invoking this method will make individual reactive node updated. While this makes interface of *Node* class clean and simple, the *Update* method is just a proxy method of a real *OwnerObject*'s method that is actually executing in the background. In order to store that real method we can use .NET constructs called *delegates*, which can be described as types that hold a reference to a method. In particular, we are going to use the *Action* delegate, which encapsulates a method with no parameters and no return value.

Of course, not all reactive nodes need to have specified update *Action*. For example, source reactive nodes (as described in SRS document) act as a triggering point of dependency graph, and are likely to be "updated" by e.g. user entering some data. On the other hand, intermediary and sink reactive nodes depend on source reactive nodes and other intermediary nodes, and need to have update *Action* defined in order to be updated.

Since reactive nodes were aimed to represent different types of members (properties and methods) and also different types of owner objects (singular objects and collections), differences started to appear in design and implementation between these node variations. This was first attempted to address by introducing *Type* field in a *Node* class. However, since differences in implementation continued to emerge, a class hierarchy was introduced in order to handle these differences using polymorphism.

It is important to note that the ending hierarchy (see Figure 19) was a result of several iter-

ations. In each iteration multiple design decisions and multiple refactorings were applied. For example, initial version of class hierarchy was introduced by performing *Replace Type Code with Subclasses* [56] refactoring. This resulted in creating new classes, with all of them inheriting the base *Node* class. *PropertyNode* and *MethodNode* represented reactive node referring to singular owner object and a property or method member respectively. On the other hand, *CollectionPropertyNode* and *CollectionMethodNode* referred to their counterparts with collection owner object. In order to reuse common implementation from *CollectionPropertyNode* and *CollectionMethodNode*, we applied *Extract Superclass* [56] and introduced new *CollectionNode* superclass. Since the sole purpose of both *Node* and *CollectionNode* class was to reuse common implementation by their subclasses, they were declared as *abstract* classes.

In order to further decrease coupling between classes representing reactive nodes and other parts of the ReframeCore, *INode* and *ICollectionNode* node interfaces were introduced. The idea behind this is that no class (other than their immediate child classes) should know about existence of concrete or abstract reactive node classes. Instead, all interaction with reactive nodes is going to be conducted through the *INode* or *ICollectionNode* interfaces. This is in line with generally accepted *Dependency Inversion Principle* (DIP), postulated by Martin [99]. In practice, dependencies directed solely to *INode* and *ICollectionNode* interfaces mean that we can rewrite node hierarchy as we see fit, by e.g. replacing implementations of existing reactive nodes, or by introducing new reactive node subclasses.

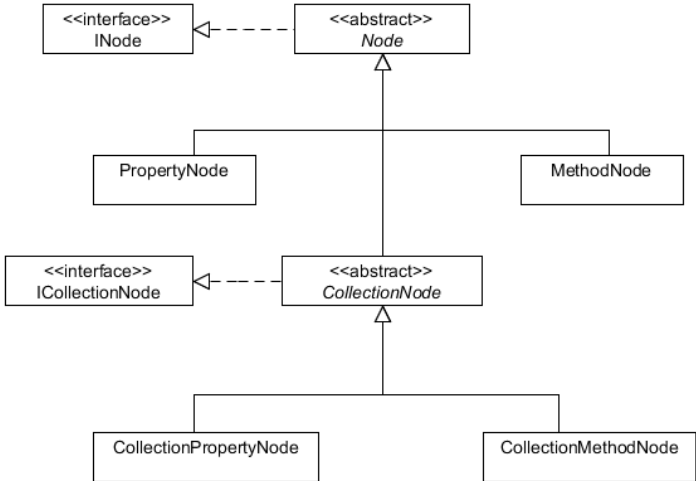


Figure 19: Hierarchy of reactive nodes

While in the rest of the framework code and the application code we can store reactive node references in variables of interface type *INode*, there is one more thing that hampers total decoupling of reactive node classes from other parts of the code. In the process of instantiating reactive nodes we still need to specify which exact concrete class we want to instantiate. Although there is no way to completely avoid this, there are solutions which tend to minimize the effect. For example, Gamma et al. [61] propose *creational patterns* as a category of design patterns which try to abstract the process of class instantiation. They attempt this by: (1) encapsulating knowledge on the exact concrete class which is to be instantiated, and (2) encapsulating knowledge on how exactly the instantiation is conducted. To put it another way, instead of scattering instantiation logic and coupling concrete reactive node classes to all the places in code which need to instantiate them, we localize this logic and make one part of the code responsible for it. This additionally reduces coupling and also improves code quality in terms of maintainability, understandability, reusability etc.

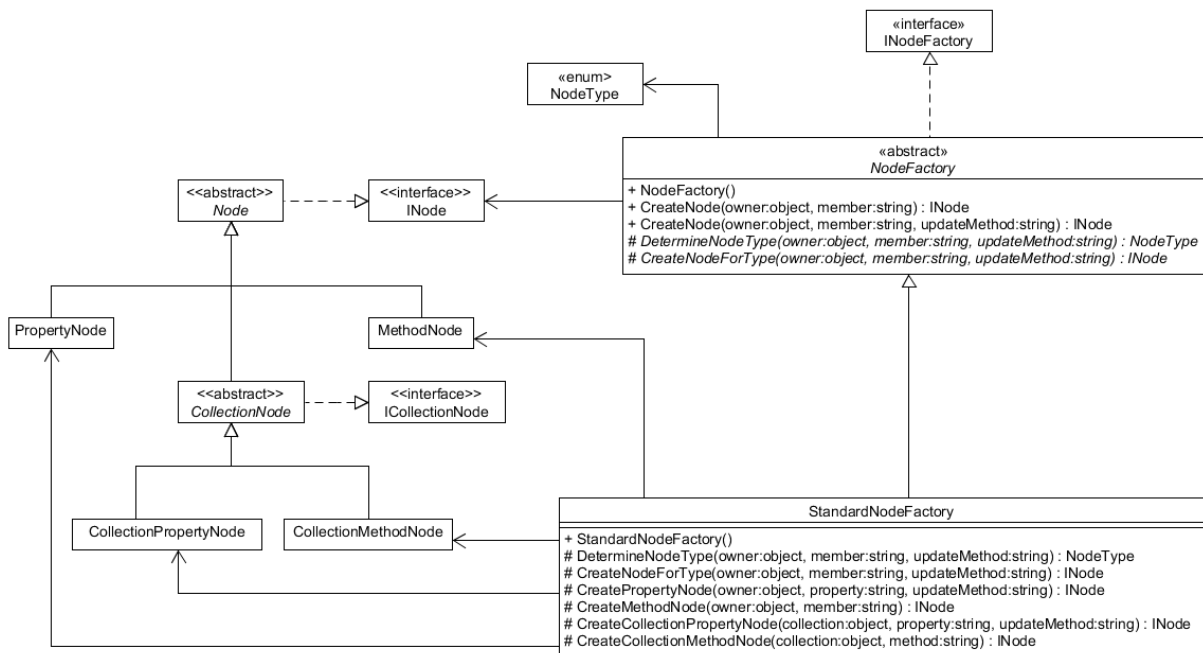


Figure 20: Essential members and associations of NodeFactory class

In order to achieve this, we applied a variant of *Abstract Factory* [61] design pattern. This is also suggested by Martin [99], as he claims that one of the coding practices emerged from *Dependency Inversion Principle* ("Don't refer to volatile concrete classes"), puts constraints on creation of objects and enforces the use of *Abstract Factories*. Here, *Abstract factory* is represented by an abstract class *NodeFactory*, which implements two overloads of the public method

CreateNode (see Figure 20). This method prescribes simple algorithm for creating reactive nodes whose main steps involve invocation of abstract methods. Method *CreateNode* in fact represents a Template Method [61] design pattern, and captures commonality which all node factories should adhere. Abstract methods, on the other hand, have to be implemented by concrete subclasses, and represent a way to introduce variability in the process of creating reactive nodes.

While abstract factory class is useful for specifying commonalities in reactive node creation process and also specifying points where variability can occur, we still need concrete factory class to do the real work. REFRAME provides class *StandardNodeFactory* - a concrete factory able to create currently supported types of reactive nodes. Extending reactive nodes creation process can be done either by implementing new concrete class from abstract factory class, or by extending *StandardNodeFactory*. As with reactive node hierarchy, we introduced *INodeFactory* interface through which other parts of ReframeCore can interact with node factory objects.

6.3.1.2. *Reactive dependencies and dependency graphs*

Although they are central concept in REFRAME, reactive dependencies are not modeled as an explicit abstraction. This is because we are not associating with reactive dependencies any information other than which reactive nodes are constituting them. Therefore, instead of explicit abstraction, reactive dependency is implicitly expressed as an association of two reactive nodes - predecessor and successor. If the need would arise to associate additional information to reactive dependencies, then they would need to be represented by an explicit abstraction. In this case, core algorithms (such as sorting, scheduling and update algorithms) would require certain adjustments.

While REFRAME is going to have a central entity representing dependency graph, individual reactive dependencies are not going to be centrally stored, but rather distributed across individual reactive nodes. Each reactive node will keep record of reactive dependencies it is involved with, either as predecessor or successor. This is implemented by each reactive node maintaining two separate lists of reactive nodes - the *Predecessors* list depicting its predecessor nodes, and the *Successors* list depicting its successor nodes. *Outgoing reactive dependencies* of reactive node n can then be described by an ordered pair (n, s) , where n is a current node (acting as a predecessor), and s is any reactive node from n 's *Successors* list. Similarly, *ingoing reactive dependencies* of reactive node n can be described as an ordered pair (p, n) where p is

any reactive node from n 's Predecessors list, and n is a current node (acting as a successor). Maintaining two separate lists does require more memory space due to each reactive dependency being represented twice (in predecessor node's list of successor nodes, and successor node's list of predecessor nodes). However, this is beneficial to performance, because instead of traversing all possible reactive dependencies for a given reactive node, we can directly assess all of its immediate neighbors. Other than Predecessors and Successors lists, reactive nodes through *INode* interface expose methods for basic manipulation of these lists (see Figure 21).

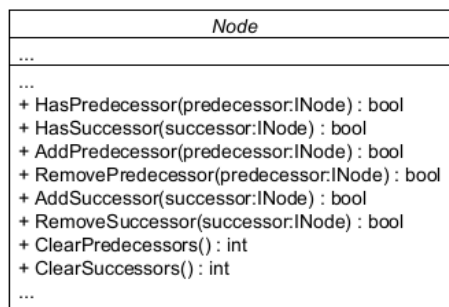


Figure 21: Methods for manipulation of predecessors and successors

Since most of what REFRAME is doing revolves around dependency graphs, they are the most important higher level concept in REFRAME, and therefore require special attention to design them. They will appear as an explicit abstraction, i.e. class *DependencyGraph*. Being the central concept of the framework creates a natural tendency for *DependencyGraph* class to become overloaded with functionalities, starting with graph instantiation process, managing the graph structure, scheduling and performing updates, providing update progress information etc. Such large concentration of functionalities within individual class makes the class "know too much" and "do too much", which is generally considered harmful, especially in terms of understandability and maintainability. In literature, such classes are often referred to as "god classes" or "monster classes". Prominent authors such as Fowler [56] and Martin [100] label this phenomenon as "large classes" and "too much information" respectively, and consider it to be a bad smell in code. Since "god classes" fairly frequently emerge in code, lot of principles and practices are aimed at preventing and eliminating them, such as *Single Responsibility Principle* (SRP) popularized by Martin [99]. As a part of SRP guidance, several refactorings documented by Fowler [56] were applied in order to eliminate "large class" bad smell, namely: *Extract Class*, *Extract Subclass* and *Extract Interface* refactorings.

These efforts eventually led to a decomposition of one huge responsibility initially attributed

to *DependencyGraph* class, into several smaller-scale and more cohesive responsibilities. The one natural responsibility left for *DependencyGraph* class was to manage the graph structure. Firstly, it involves providing appropriate data structures for storing the graph, i.e. for storing the very reactive nodes which constitute dependency graph. As we previously discussed, we identified adjacency list as an appropriate data structure for this. On an implementational level, reactive nodes constituting dependency graph are stored in *Nodes* collection in *DependencyGraph* class. However, reactive dependencies are distributed and kept within their respective reactive nodes, i.e. in their aforementioned Predecessors and Successors lists. This means that adjacency list containing dependency graph spans two classes - *DependencyGraph* and *Node*.

In addition to providing data structure for storing graph, *DependencyGraph* class also provides a set of basic methods for manipulating graph structure (see Figure 22). This structure is shaped by adding reactive nodes to dependency graph and forming reactive dependencies between them.

DependencyGraph
+ Identifier : string + Nodes : IList<INode>
+ DependencyGraph(identifier:string) + GetNode(node:INode) : INode + AddNode(node:INode) : INode + RemoveNode(node:INode, forceRemoval:bool) : bool + ContainsNode(node:INode) : bool + ContainsDependency(predecessor:INode, successor:INode):bool + AddDependency(predecessor:INode, successor:INode):void + RemoveDependency(predecessor:INode, successor:INode):bool ...

Figure 22: Essential members of *DependencyGraph* class

6.3.1.3. *Dependency graph update process*

Major responsibility that was seceded from *DependencyGraph* was performing dependency graph update. Successive attempts to model dependency graph update resulted in further decomposition of this responsibility (in accordance to SRP [99]). From this effort, two main steps of update process emerged, namely: (1) determining update schedule, and (2) performing the update. Although mutually related, both of these steps were recognized as separate responsibilities, and therefore modeled as separate abstractions. This resulted in two new classes, conveniently named: (1) *Scheduler*, and (2) *Updater*.

Determining update schedule refers to the issue of finding out which nodes from the graph are required to be updated, and also in which order is this going to happen. And this is ex-

actly what *Scheduler* class is doing. The complexities of determining the update schedule are hidden behind two overloads of the method *GetNodesForUpdate* (see Figure 23). For a given dependency graph, the first method overload returns collection of all reactive nodes from the graph, sorted in order they should be updated. The second method overload offers determining the partial update schedule, and for a given initial node it returns properly ordered collection of only those reactive nodes that are directly or indirectly dependent on the initial node.

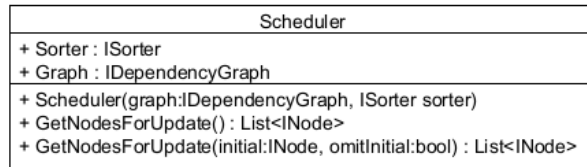


Figure 23: Essential members of Scheduler class

In order to provide scheduling functionality offered through *GetNodesForUpdate* method, a few things are happening internally in the class. This includes making temporary adjustments to graph (such as redirecting reactive dependencies), topologically sorting dependency graph, assigning update-related metadata to reactive nodes and logging update schedule for testing and reporting purposes. Again, the question of treating some of these sub-functionalities as separate responsibilities arise.

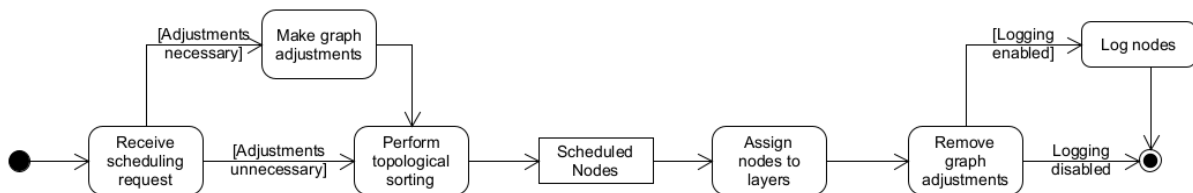


Figure 24: Scheduling nodes for update

Logging, for example, has been traditionally considered as a cross-cutting concern, which should be removed from core behavior. Indeed, logging implementation was extracted to a *NodeLog* class, not only to separate responsibilities, but also to enable the reuse of logging functionality in other parts of the code. Topological sorting is another functionality which can qualify as a separate responsibility. However, there is an additional reason why it would be convenient to move topological sorting to a separate class. As we already discussed, topological sorting, as a fundamental part of determining the update schedule, can be implemented using different variants of algorithms. While we opted for Tarjan's [148] algorithm based on

Depth-first search, we also emphasized the importance of being able to easily replace that with some other algorithm (such as Kahn's [83] algorithm based on Breadth-first search). This scenario is a textbook example of a well-known *Strategy pattern*. According to Gamma et al. [61] Strategy's intent is to "*Define family of algorithms, encapsulate each one, and make them interchangeable...*". As one of the applications of the pattern, authors also mentioned "*... need for different variants of algorithm... reflecting different space/time trade-offs...*". Therefore, in order to separate topological sorting from scheduling responsibility, and also to make algorithms interchangeable, we implement the Strategy pattern (see Figure 25). The *Strategy* role is represented by *ISorter* interface, which was implemented by *DFS_Sorter* acting as a *Concrete strategy* role. Lastly, the *Context* role is played by *Scheduler* class. Since *Scheduler* is coupled only to *ISorter* interface, any future *Concrete strategies* implementing that interface can be interchangeably used by the *Scheduler*.

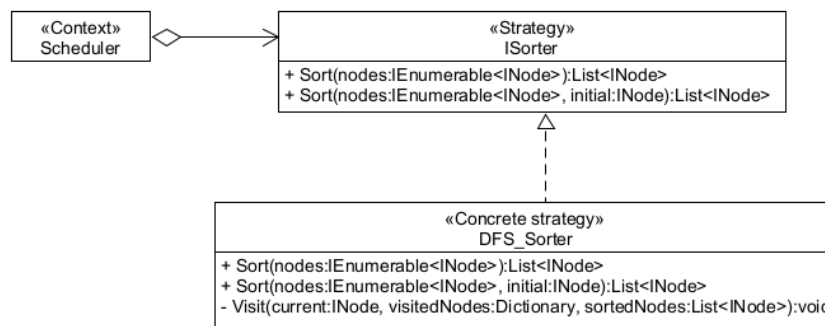


Figure 25: Strategy pattern in the context of Sorter class

When we determined the update schedule, i.e. which exact nodes have to be updated, and in what particular order, the very update of these nodes can be executed. This responsibility has been assigned to an *Updater* class (see Figure 26), and it involves configuring update options, performing the actual update, and tracking, logging and reporting update status. The configuration options include: (1) choosing update strategy, (2) suspending update (methods *SuspendUpdate* and *ResumeUpdate*), and (3) enabling logging of updated nodes. Update strategy provides three options for performing update process, namely: (1) *Synchronous* (default), (2) *Asynchronous*, and (3) *Parallel*. The very update process is issued by invoking one of the overloads of *PerformUpdate* method. The first overload performs update process on the entire dependency graph, while other overloads provide different ways to specify initial node and perform update process of only part of the graph.

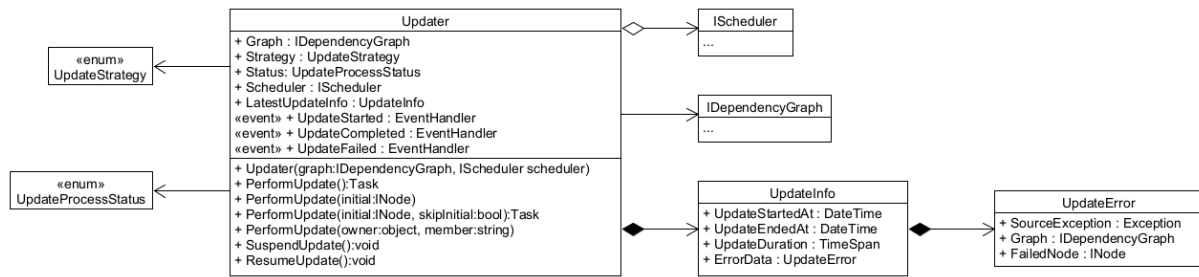


Figure 26: Essential members and associations of Updater class

When performed *synchronously*, update process is executed in a sequence with other steps of the calling application code, usually within the same thread. Updating individual reactive nodes within the update process is also done in sequence, one reactive node after another. This is a reasonable option if the update process is not a lengthy one, and it does not disrupt responsiveness of the end-user application. It is also a right option when the application developer wants to manage concurrency strategy by himself within the end-user application. However, in case of lengthy update process running in the main thread, a synchronous option will result in main thread being blocked and GUI being unresponsive. This can be avoided by using *asynchronous* option, in which update process is performed in separate thread, rather than on the GUI thread. This prevents GUI thread from being blocked and achieves responsiveness of end-user application, even in cases of lengthy update process. Finally, *parallel* update process, aims at improving overall performance of graph update process, by utilizing multiple threads to update independent individual reactive nodes in parallel. It is important to emphasize, that all that the application developer needs to do to switch from one update strategy to another, is to change the Strategy property of the Updater instance. This makes it very easy to experiment how end-user application is influenced by different update strategies, without making any actual changes to application code.

For implementation of both asynchronous and parallel update process we lean on .NET official Task Parallel Library and its core concept represented by *Task* class. While the *Task* represents concept on a higher level of abstraction than the class *Thread*, making implementation of concurrency strategies easier to achieve, we still needed to specify how update schedule will be parallelized. In previous section, we discussed Coffman-Graham's [44] algorithm for graph layering, and how its third step can be used to form layers of reactive nodes. This particular step was in fact implemented in a *Scheduler* class by assigning each scheduled reactive

nodes with the number of the layer it belongs to. What remained for *Updater* to do was to fetch one layer at the time, and since its reactive nodes were mutually independent, execute them in parallel.

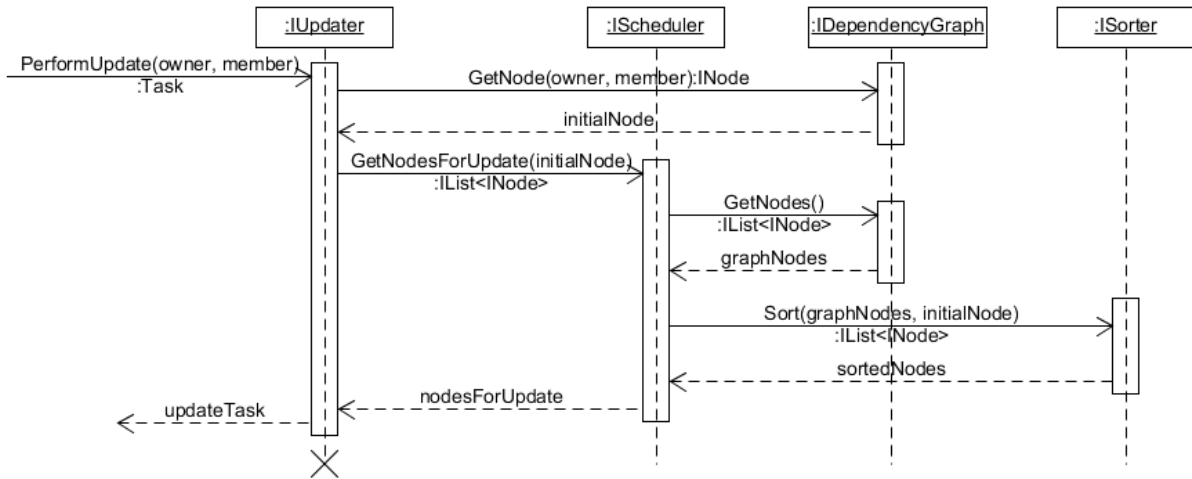


Figure 27: Example of object interaction required for performing update

As discussed in previous section, suspending and resuming update is a mechanism for optimizing the number of update processes which are performed in cases of e.g. multiple structural changes or multiple reactive nodes triggering simultaneously. The implementation includes *SuspendUpdate* and *ResumeUpdate* methods, which application developer can use to wrap application code that would otherwise result in multiple update processes. Any attempt to perform graph update between invocations of *SuspendUpdate* and *ResumeUpdate* methods is going to be ignored.

6.3.1.4. Reactor - a higher level interface for ReframeCore

Decomposing ReframeCore into a number of classes, with each of them being in charge of their own distinct responsibility, helps managing frameworks complexity and makes the framework easier to understand, maintain, reuse and extend. However, this may come at cost of making framework more challenging to use, as application developers necessarily have to deal with more instances of different classes, and manage relationships and interaction between these instances. This can result in a steep learning curve, and discourage potential framework users from giving framework a try. What we want instead, can be described by well-known quote from computer scientist Alan Kay, which states "*Make simple things simple, and complex things possible*". Translated to a REFRAME context, this means we want to provide simpler way for

application developers to interact with the core features of the framework, especially in simple and common usage scenarios. For more complex and less frequent scenarios, we want application developers to still be able to utilize all the power of dealing with individual components of the framework and even extend and customize them.

One of the common solutions for providing unified, higher-level interface in order to simplify use of some complex system is a *Facade* [61] design pattern. According to Gamma et al. [61], simplified view of the complex system provided by *Facade* tends to be good enough for most clients. It abstracts the structural details of the system, and shields users from unnecessary details. However, *Facade* does not encapsulate the details in a sense that it makes them unavailable to application developers. All of the individual components are still accessible for those application developers who need them.

In one of earlier iterations, one of the idea was to make *DependencyGraph* take the role of *Facade*. However, this would result in increase coupling between *DependencyGraph* class and classes such as *Updater*, *Scheduler* and *Sorter*. Therefore, instead, we introduced additional class which would serve as a *Facade*, and we named it *Reactor*. As it is common for *Facade* classes, *Reactor* does not implement any particular programming logic itself. Rather, *Reactor* instance only forwards the method calls to instances of other classes, in this case *DependencyGraph* and *Updater* directly, and *Scheduler* and *DFS_Sorter* indirectly. In this way *Reactor* becomes a simple gateway through which application developers can manipulate structure of underlying dependency graph (e.g. add and remove nodes and dependencies, fetch nodes etc.) and perform graph update without being required to bother with the underlying details. It is interesting to note that design of *Reactor* class (and the use of *Facade* pattern in general) contradicts the SRP principle, however, this is done knowingly, and with a clear purpose.

Since *Reactor* represents a facade for an arbitrary configuration of *DependencyGraph*, *Updater*, *Scheduler*, and *DFS_Sorter* instances, one of the design decision that have to be made is where should we put the responsibility of constructing *Reactor* instances and setting up these configurations. During application runtime, it may well be opportune to have multiple different configurations, i.e. multiple different *Reactor* instances, which are going to be used at multiple places throughout the application code. Therefore, in addition to handling construction logic, we also need a way of keeping track of constructed reactors and accessing particular reactor.

The simplest option from framework developer's perspective would be to trust application

developers with this. Application developers would use Reactor's parametrized constructors to set up particular configuration. They would also assure that created Reactor instances are globally accessible, either as individual variables or in some sort of collection, so that they can be used where needed. However, instead of relying entirely on application developers to handle this, we want REFRAME to provide consistent way of doing this. In order to achieve it, we need to have three goals in mind: (1) encapsulating knowledge with regard to Reactor construction logic, (2) keeping track of constructed Reactor instances, (3) assuring global access to already constructed Reactor instances.

Encapsulation of construction logic is a trait of creational design patterns such as already mentioned *Abstract Factory* pattern [61]. As opposed to constructor methods of Reactor class we find placing complex construction logic into separate class as a more reasonable option. However, at this point we do not see the need for flexibility offered by a full-scale abstract factory. If the framework use should indicate otherwise, design can be changed afterwards. Keeping track of Reactor instances and assuring their global access aligns well with *Registry* pattern [54]. According to Fowler *Registry* pattern describes "a well known object that other objects can use to find common objects and services" [54]. That being said, in REFRAME, we will introduce a separate class named *ReactorRegistry*, which will adopt the *factory* and *registry* roles. While the *registry* role is explicit in both the implementation and the naming of the class, *factory* role is more implicit, i.e. it does assume responsibility of constructing *Reactor* instances but without full-scale abstract factory implementation. Therefore, application developers can use *ReactorRegistry* class for creating new and obtaining existing, already "registered" Reactor instances.

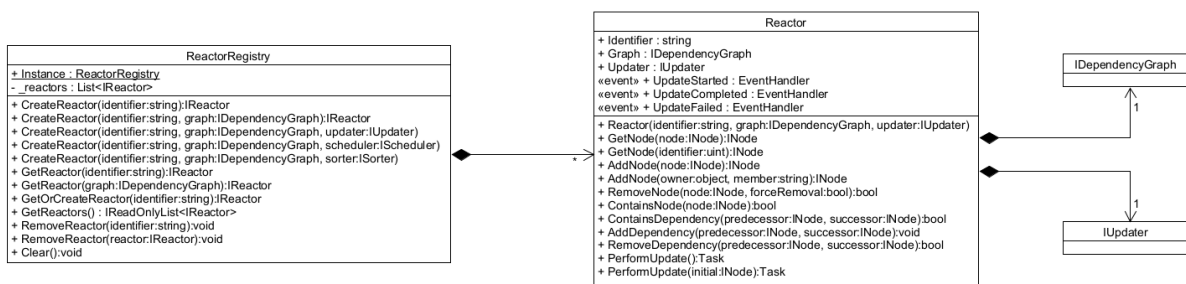


Figure 28: Essential members and associations of Reactor class

As both stated in our third goal, and the Fowler's definition of *Registry* pattern, instance of *ReactorRegistry* should be globally accessible in application code. However, it also makes sense

for only one instance of `ReactorRegistry` class to exist. Both of these requirements can be met by either proclaiming the class as *static*, or by implementing *Singleton* design pattern [61]. The rationale behind using static class is that static classes cannot be instantiated, so the class itself emulates the notion of only one object existing. Each member of a static class is also static, and is conveniently accessible directly from the globally accessible class itself. On the other hand, *Singleton*'s intent, as defined by Gamma et al. [61], is to "ensure a class has one instance, and provide a global point of access to it". It involves creating non-static class, but artificially limits the ability for creating more than one instance of a class by hiding the constructor. This one instance is available through static property or a method of *Singleton* class. In this way it also becomes globally accessible. With regard to `ReactorRegistry` implementation we prefer the *Singleton* implementation, since it is more flexible than static class implementation. For example, contrary to static classes, *Singleton* class can implement interfaces, it can be derived from (and therefore extended), it can be passed around as a parameter etc.

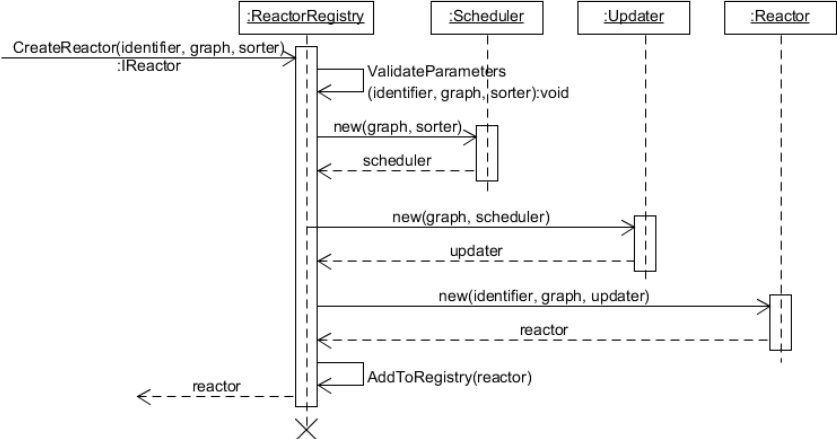


Figure 29: Example of object interaction required for creation of `Reactor` object

6.3.2. ReframeTools

6.3.2.1. Analyzer tool

Obtaining dependency graph data

Before going into design and implementation details related to analyzing dependency graphs, it is reasonable to discuss how the *Analyzer* tool will obtain the very data it is supposed to work with. In previous sections of this chapter we concluded that dependency graph data originally resides in end-user application, which is executed in a separate process. Getting hold of this

data by *Analyzer* tool meant we need to establish inter-process communication between end-application and *Analyzer* tool. As we previously announced, our solution will take a form of well-known *client-server* architectural pattern with the goal of abstracting the communication details related to underlying *named pipes* technology.

The client role, played by the *Analyzer* tool, assumes performing two usual tasks: (1) requesting dependency graph data by sending command to a server, and (2) receiving a raw XML response. The Figure 30 shows abstractions required for carrying out these tasks. The *PipeCommand* allows us to specify the command name, optional parameters, and the identifier of the component that is supposed to handle the command on the server. Since all communication via *pipes* is done by exchanging textual data between client and server, *PipeCommand* has *Serialize* method, which translates command data into XML content. The *PipeClient* abstract class handles the general, low-level mechanics of using *pipes* to send a command to a server, and receive the response. On the other hand, concrete class *ReframePipeClient* defines and sends REFRAME-specific commands, which are exposed in a form of *GetRegisteredReactors*, *GetReactor*, and *GetUpdateInfo* methods.

Table 10: Currently implemented pipe commands

Method	Command	Handler	Description
GetRegisteredReactors	ExportRegisteredReactors	CoreHandler	Requests the list of reactors registered in end-user application.
GetReactor	ExportReactor	CoreHandler	Requests details (including entire dependency graph) of particular reactor.
GetUpdateInfo	ExportUpdateInfo	CoreHandler	Requests information related to performed update process in particular reactor.

The *server* role in our inter-process communication is played by end-user application. Its primary role is to respond to client's commands by sending requested data. This is achieved by server being able to: (1) understand client's commands, and (2) send runtime dependency graph state as XML content. The Figure 31 shows design perspective of the server role. The *PipeServer* abstract class is in charge of low-level mechanics dealing with server status and the use of *pipes* to establish communication with a client. The concrete *ReframePipeClient* is only left to initialize specific handlers which are going to handle particular commands coming

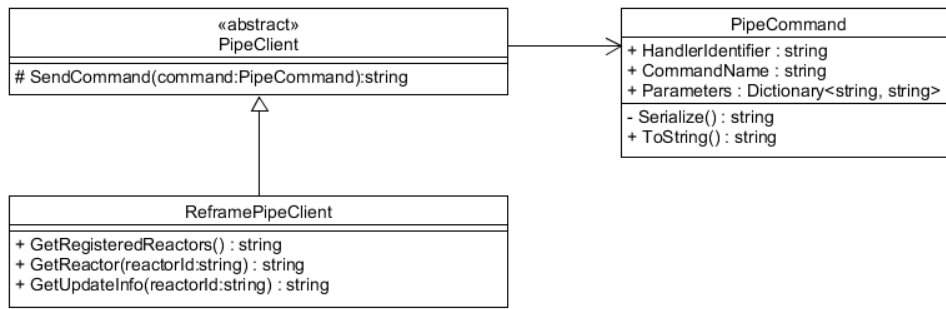


Figure 30: Client feature residing in REFRAME tools

from a client. A *handler* can be any class conforming to *ICommandHandler* interface, i.e. having an *Identifier* and a *HandleCommand* method. Potentially reusable parts of *handler's* implementation (including mandatory *Identifier* and *HandleCommand*) are placed into abstract *CommandHandler* class. The concrete *CoreHandler* class was only required to implement abstract method *GetExporter* responsible for delegating XML export process to any component conforming to *IExporter* interface. Since we wanted *CoreHandler* to be able to respond to three aforementioned client's commands (sent by *ReframePipeClient*), we paired each of the commands with appropriate concrete realization of *IExporter*, namely *XmlReactorsExporter*, *XmlReactorDetailExporter*, or *XmlUpdateInfoExporter*.

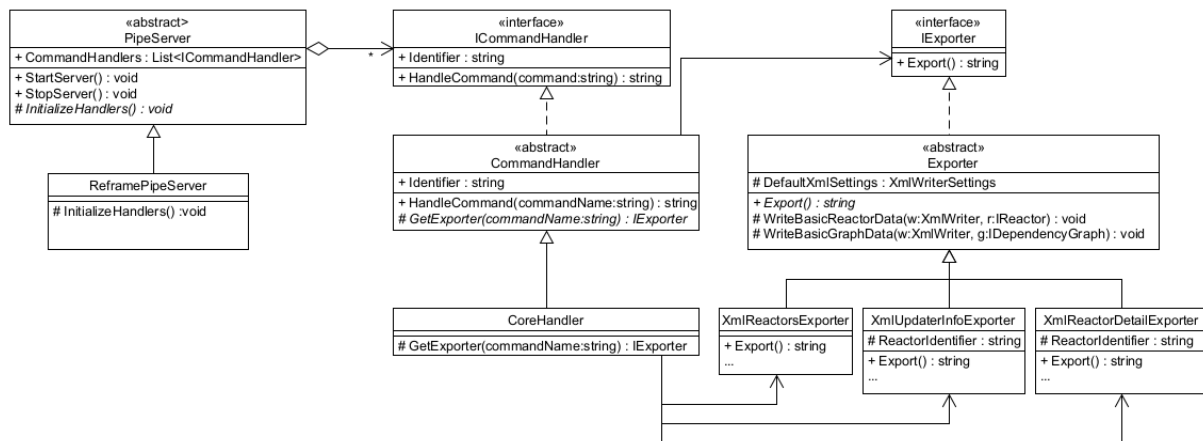


Figure 31: Server feature residing in end-user application

The chronology of entire inter-process communication between *Analyzer* tool and end-user application is depicted in Figure 32. On the left side we see the *client* role played by *Analyzer* tool, which requests the data from the server. On the right side we can see the *server*, residing in end-user application, which packs the reactor data in a form of XML content, and sends it to the client. It is worth to mention, that although the server functionality falls into end-user

application, the entire functionality is already implemented as a part of REFRAME framework. The only thing that application developer is required to do in order to enable the server, is to reference needed server assemblies and call the *StartServer* method.

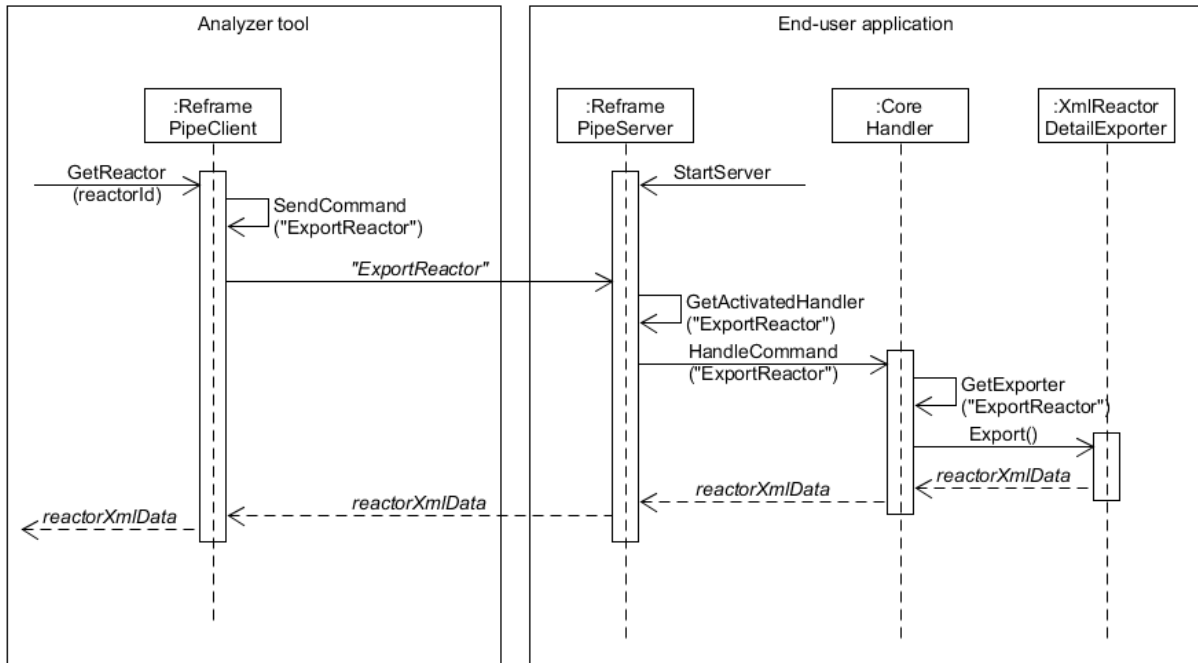


Figure 32: Inter-process communication between Analyzer tool and end-user application

Creating analysis graphs and nodes

Since dependency graph data which server packed and sent to a client is in a form of XML, we decided to parse this content into an explicit graph-like structure in order to make manipulation and analysis easier. Although initially considered, the idea of simply reusing *DependencyGraph* and node hierarchy from core part of REFRAME was abandoned due to differences that arose (especially ones in node hierarchy and their structure). Since the reuse benefit was also not particularly large, it seemed a better option to leave *analysis* and *core* graph structures separate in order to avoid possible clashes in the future. Additionally, this would allow Analyzer and Visualizer to also be used with alternative implementations of the core framework features. Indeed, as long as the server-side provides properly formed XML document, Analyzer and Visualizer will do their job.

This separation was realized by introducing the concepts of *analysis graph* and *analysis node*. When looking at Figure 33 we can see that to a large extent these concepts imitate basic graph-enabling data structures that can also be seen in REFRAME core. For example,

AnalysisGraph provides a means to store and manipulate graph nodes, while *AnalysisNode* allow us to represent characteristics of individual nodes and realize dependencies between them. Unlike in REFRAME core, however, here the node hierarchy is not driven by the need to capture different class members (e.g. property, method) that the node represents. Rather, the hierarchy is introduced to handle specifics of six different abstraction levels (vertical reduction) that the analysis graph and nodes can be at. However, after several iterations of refactoring, analysis nodes ended up having specifics related to these abstractions levels, while analysis graph did not. This resulted in having one concrete analysis node class for each abstraction level, and only one general class (i.e. *AnalysisGraph*) for analysis graph. So for example, for a class-member level of abstraction we would have an instance of *AnalysisGraph* with assigned *ClassMember* to *AnalysisLevel* property, and any number of *ClassMemberAnalysisNode* instances contained in *Nodes* property.

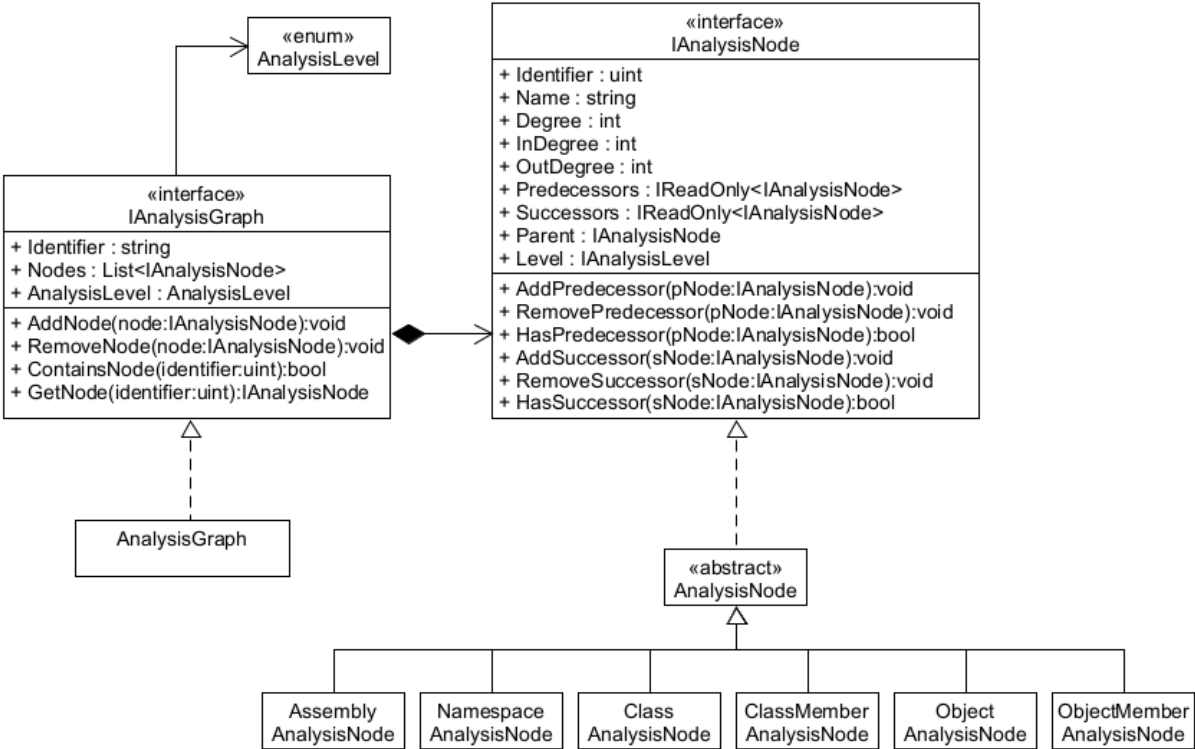


Figure 33: *AnalysisGraph* structure and node hierarchy

Following the good practices employed when designing core part of the framework, details of concrete implementations of analysis graph and nodes are also hidden behind interfaces, namely *IAnalysisGraph* and *IAnalysisNode*. In order to ensure no client class references directly any of the concrete classes, even in the process of analysis graph and nodes instantiation,

an *abstract factory pattern* was applied. Besides sheer decoupling from client classes, there was another very important reason in support of introducing factories in this particular case. With factories, we were able to separate responsibility of creating graph and nodes from *AnalysisGraph* and *AnalysisNode* classes. This creation process was very demanding, since it involved: (1) parsing possibly large XML content obtained from end-user application, (2) extracting necessary data from it and (3) constructing the graph and nodes. Finally, by allocating all dealings with XML content to factory classes, we also made analysis graph and nodes independent of the underlying data transfer technology, effectively making the framework more modular. In Figure 34 we can see *AnalysisGraphFactory* abstract class, which sets up the general steps for creating any *IAnalysisGraph* compatible graph through its *CreateGraph* template method. The creation of nodes, however, is delegated to accompanying *AnalysisNodeFactory* abstract class. Naturally, a creation process for each concrete analysis graph and node, is implemented by concrete factories. For example, *AnalysisGraph* at an assembly level and *AssemblyAnalysisNode* instances are created using *AssemblyAnalysisGraphFactory* and *AssemblyAnalysisNodeFactory* respectively.

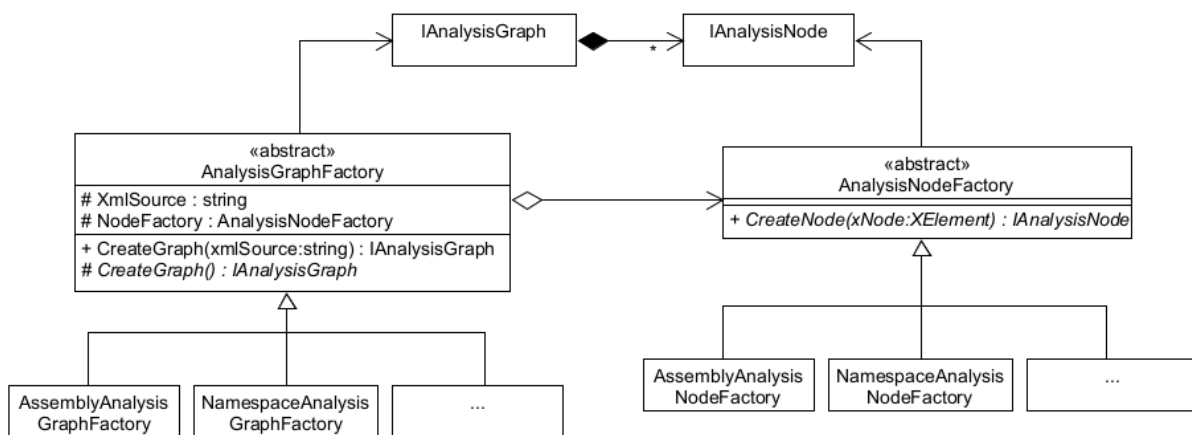


Figure 34: Analysis graph and analysis node factories

Graph structure analysis

As we discussed in previous subsection, a raw XML content obtained from end-user application contains all the data related to runtime dependency graph. Since this runtime graph is always at the *object-member* level, we found it convenient to offer application developer an option to alter the level of abstraction when analyzing and displaying the graph. *Vertical reduction* was realized through the use of factory and node hierarchies, with each of the abstraction

levels handled by its own concrete factory and node class. For example, original XML graph data obtained by *ReframePipeClient* can be passed to concrete *AssemblyAnalysisGraphFactory*, which will result in creating analysis graph at an assembly-level of abstraction containing nodes of *AssemblyAnalysisNode* type.

When we have *AnalysisGraph* at desired level of abstraction, we can decide to list the entire graph, i.e. all of its nodes, or only a subset (*horizontal reduction*). In previous section we discussed three different filtering mechanisms that are going to be available to choose from. The first, *filtering by affiliation*, is tightly related to analysis graph abstraction levels. It enables application developer to list only those nodes from graph that have a child-parent relationship with a given node. As can be seen in Figure 35, at the lowest level of this child-parent hierarchy is *ObjectMemberAnalysisNode* node, whose parent is an *ObjectAnalysisNode*. *ObjectAnalysisNode* on the other hand has *ClassAnalysisNode* as its parent, and so on. What this implies is that when we have analysis graph at certain level of abstraction, we can filter its nodes with regard to *affiliation* to all of their direct and indirect parent nodes. For example, if we have an analysis graph at an assembly or namespace level of abstraction, we are not going to have any filtering options since its nodes are at the top of the child-parent hierarchy (i.e. they do not have a parent). The other extreme would be a graph at an object-member level of abstraction, whose nodes can be filtered by *ObjectAnalysisNode*, *ClassAnalysisNode*, *NamespaceAnalysisNode* and *AssemblyAnalysisNode* nodes, i.e. all of its direct and indirect parents.

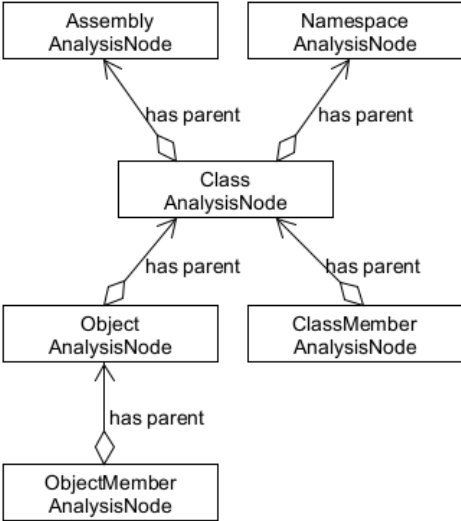


Figure 35: Node abstraction levels in filtering by affiliation

Filtering by affiliation is designed as shown in Figure 36. We can see two interacting hierarchies, the left one depicting graphical user interface classes, and the right one the classes with filtering logic itself. Depending on the analysis graph level of abstraction, the concrete GUI form is displayed with its concrete *AnalysisFilter* handling all the filtering logic. Parent classes *FrmAnalysisFilter* and *AnalysisFilter* allowed us to not only reuse part of the code, but also to hide implementation specifics, effectively enabling *AnalysisController* to operate with any current and future *filtering by affiliation* implementation.

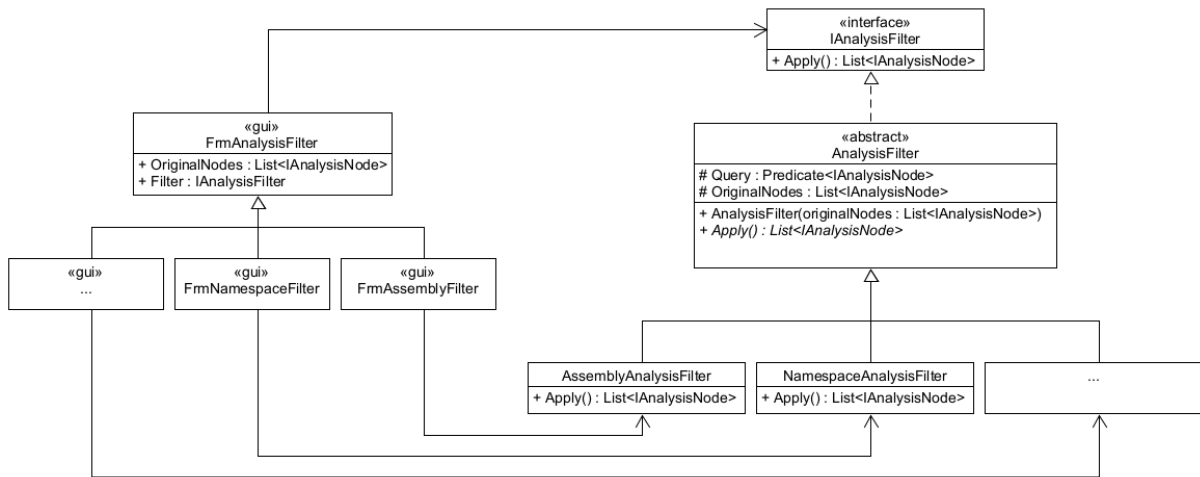


Figure 36: Filtering by affiliation design

Filtering by role as a second mechanism for horizontal reduction, can also be applied at analysis graph of any level of abstraction. Currently, there are total of 5 filtering options available for this category, showing graph's: (1) source nodes, (2) sink nodes, (3) leaf nodes, (4) orphan nodes and (5) intermediary nodes. These options are self-explanatory and in line with definitions provided in chapter 5. User selecting these options through the graphical interface (represented by *FrmAnalysisView* class hierarchy), results in invoking appropriate method in *AnalysisController*. This in turn leads to executing the very filtering implementation, located in *Analyzer* class (Figure 37).

Finally, *filtering by association* aims at displaying analysis nodes associated in different ways with chosen analysis node. There are total of 9 filtering options at our disposal, listed and described in Table 11. As with filtering by role options, these options are also available to user through *FrmAnalysisView* hierarchy. The user's selection is handled by *AnalysisController*, while the filtering itself is handled by *Analyzer* class (see Figure 37).

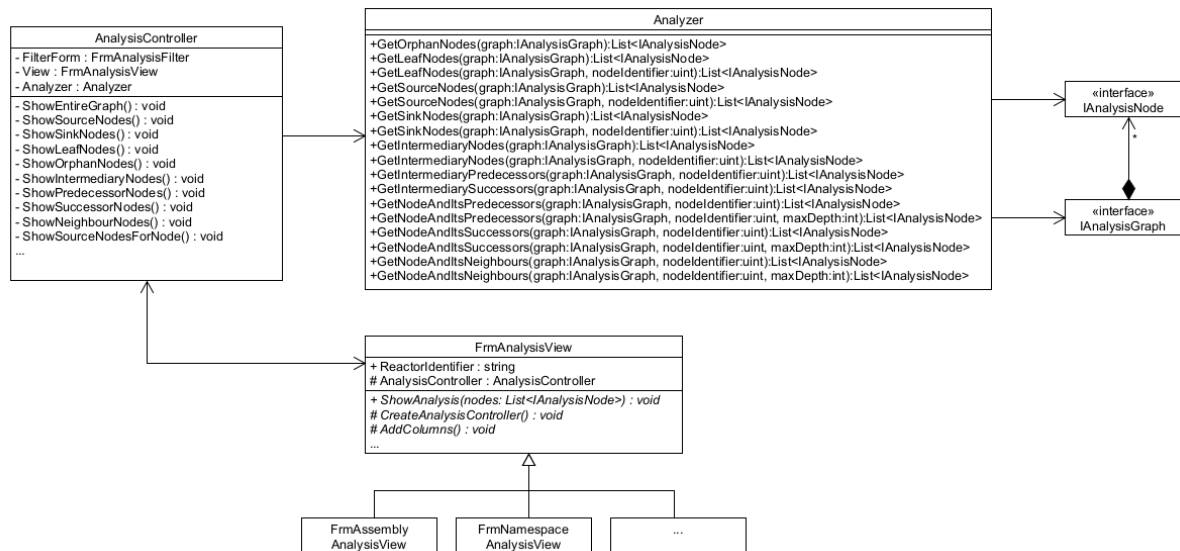


Figure 37: Filtering by role and association

Table 11: Filtering by association options

Filtering option	Description
Show predecessors for node	Shows all (direct and indirect) predecessors of a given node at desired depth level. It can be useful in showing us what reactive nodes does our given node depend on. Note: <i>Depth level of 1 indicates only direct predecessors will be shown, level 2 indicates direct predecessors and their direct predecessors will be shown, and so on.</i>
Show successors for node	Shows all (direct and indirect) successors of a given node at desired depth level. It can be useful in showing us what reactive nodes depend on our given node.
Show neighbors for node	Shows all (direct and indirect) neighbors (both predecessors and successors) of a given node at desired depth level. It can be useful in showing us what reactive nodes depend on, or are depended on by given node.
Show source nodes for node	Shows all nodes which have a source node role in a graph, and which have a path between them and a given node. This can show us, for example, what input parameters does a given node depend on.
Show sink nodes for node	Shows all nodes which have a sink node role in a graph, and which have a path between them and a given node. This can show us, for example, what output parameters or results depend on a given node.
Show leaf nodes for node	Shows all nodes which have a leaf node role in a graph, and which have a path between them and a given node. This can show us, for example, what input and output parameters are associated with a given node.
Show intermediary nodes for node	Shows all nodes which have an intermediary node role in a graph, and which have a path between them and a given node. This can show us what non-input and non-output parameters in a graph are associated with a given node.

Table Table 11 – *Filtering by association options*

Filtering option	Description
Show intermediary predecessors for node	Shows all nodes which have an intermediary node role in a graph, and are predecessors of a given node. This can show us what non-input predecessors are associated with a given node.
Show intermediary successors for node	Shows all nodes which have an intermediary node role in a graph, and are successors of a given node. This can show us what non-output successors are associated with a given node.

Update process analysis

When designing update process analysis one of the things we tried was to fit update analysis classes into existing Analyzer tool infrastructure. We were able to achieve that by treating results of the update process (i.e. the list of updated nodes) as any other analysis graph. This allowed us to reuse common design and implementation located in *AnalysisGraph* and *AnalysisNode* hierarchies, while specifics related to update process were handled in *UpdateAnalysisGraph* and *UpdateAnalysisNode* concrete classes. As can be seen in Figure 38, these specifics predominantly include additional properties describing different aspects of update process.

UpdateAnalysisGraph was assigned with properties containing general information about update process. Perhaps the most important information is whether the update process was successful or not. If the update process was unsuccessful, additional information about the node which failed is provided. Furthermore, *UpdateAnalysisGraph* also contains information about the cause of the update, which can be either that (1) complete update was requested, or that (2) update was triggered by individual node. In the latter case, additional information about the triggered node is provided, including previous and current value. Finally, in order to be able to assess update process performance, information on update start-time, finish-time and duration is also available. In order to decouple *UpdateAnalysisGraph* from its clients, all of these update-related properties are exposed through *IUpdateGraph* interface.

Properties assigned to *UpdateAnalysisNode* contain node-specific update information. For example, *UpdateOrder* allow us to reconstruct the order in which the nodes are updated. *CurrentValue* and *PreviousValue* make it possible to assess whether some particular parameter did change as a result of update process. Finally, *UpdateStartedAt*, *UpdateCompletedAt*, and *UpdateDuration* can be used to examine performance of individual nodes. As with *UpdateAnalysisGraph*, node's update-related properties are also accessed through *IUpdateNode* interface.

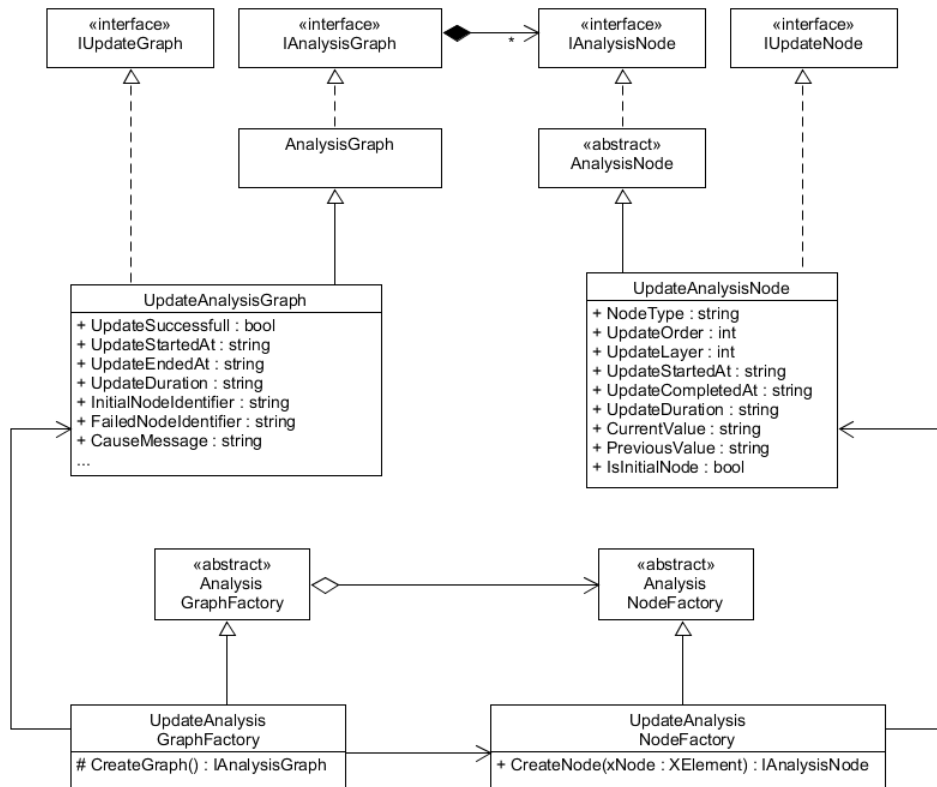


Figure 38: Update analysis classes

In addition to fitting update graph and node classes into *AnalysisGraph* and *AnalysisNode* hierarchies, their creation process was assigned to factory classes as well, namely: *UpdateAnalysisGraphFactory* and *UpdateAnalysisNodeFactory*. This again allowed us to restrict the use of concrete analysis graph and node classes only to factory classes, while the client classes depend on interfaces.

Analyzer tool GUI

The final important thing to discuss with regard to Analyzer tool is its graphical user interface. In previous sections we already mentioned that we chose Windows Forms technology due to its flexibility and familiarity from both developers' and users' perspective. As can be seen in Figure 39, class design for Analyzer GUI is fairly simple. The entry point for REFRAME Tools is *FrmRegisteredReactors* form, which, as its name implies, displays the list of registered reactors. The form itself contains only the code responsible for displaying elements of graphical interface and simple *event handler* methods. Entire process of getting raw data for registered reactors, parsing the data into appropriate data structures and displaying the data is coordinated and handled by *RegisteredReactorsController* class. As we have seen previously in this section,

the responsibility of finding registered reactors and providing raw XML data from end-user application is taken by *ReframePipeClient* class.

By selecting one of the offered reactors listed in *FrmRegisteredReactors* form, we can proceed to other forms. For example, choosing options from *Graph structure analysis* menu will take us to forms from the *FrmAnalysisView* hierarchy. On the other hand, choosing *Update process analysis* option will result in displaying *FrmUpdateProcessInfo* form. Class design in these cases replicates the one with *FrmRegisteredReactors* form. Again, we have forms containing only the code relevant for graphical user interface, while the controllers delegate individual steps of the business logic to different components and coordinate the whole process.

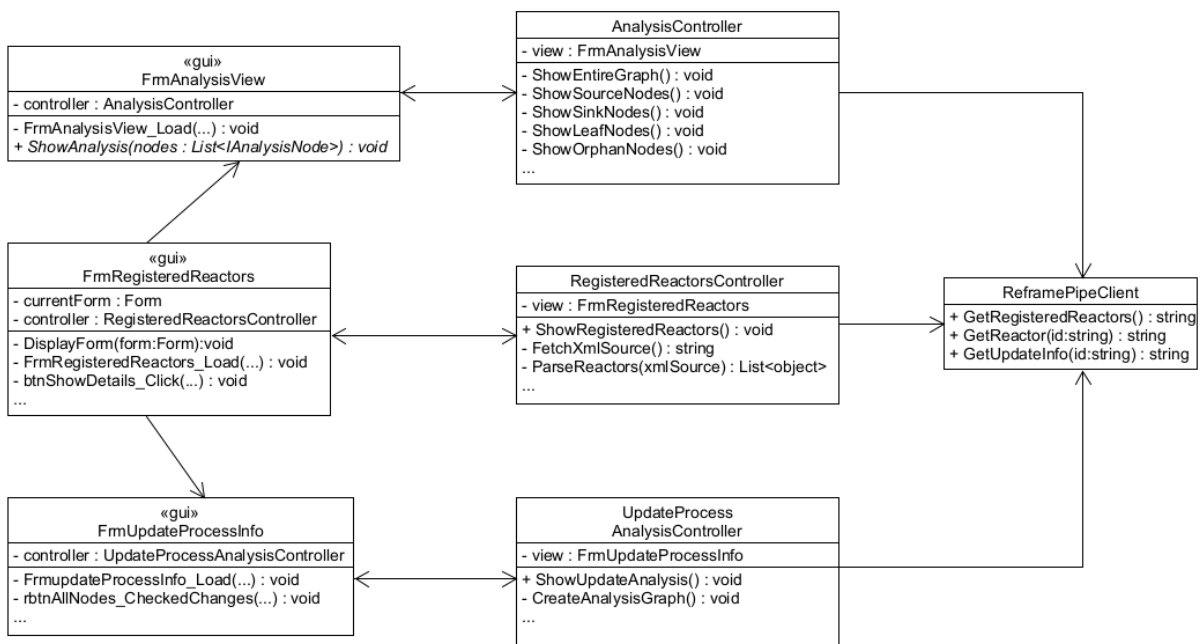


Figure 39: Analyzer GUI and related classes

6.3.2.2. Visualizer tool

Visualizer tool has a sole purpose to visually present dependency graph data obtained from Analyzer tool. In simple terms, this means putting the Analyzer data into DGML file, and then opening that file in Visual Studio's DGML viewer. Despite apparent simplicity, a fair number of classes were involved in this task, each having its own responsibility.

The first thing we wanted to address was translating the graph data forwarded in a form of *IAnalysisGraph* structure into a DGML compatible graph structure (*Graph* class found in *Microsoft.VisualStudio.GraphModel* namespace). This turned-out to be a non-trivial task, so

we assigned this responsibility to a new *VisualGraphDGML* class. However, it quickly became apparent that this *IAnalysisGraph*->*DGML graph* translation significantly varied depending on the analysis level of provided *IAnalysisGraph*. The obvious solution to handle this variability was to mimic the hierarchy under *IAnalysisGraph*. If we take a look at Figure 40, we can see that *VisualGraphDGML* class became an *abstract* class holding reused implementation, with the most notable part being *template method* named *CreateGraph*. This method specifies simple algorithm of four fixed steps for creating the graph: (1) adding custom properties which reflects particular graph’s data attributes, (2) adding nodes to a graph, (3) adding dependencies to graph, and (4) and painting the graph. These steps are implemented as either abstract or virtual methods, so that six concrete classes can agree on the default implementation or offer their own.

In addition to this hierarchy, an *IVisualGraph* interface was introduced in order to avoid potential client classes being coupled to implementation details. One of such details is *Graph* class, which closely related to DGML visualization technology. Since we wanted to leave our options open to replace current with some other visualization technology, *IVisualGraph* prescribes technology-neutral *SerializeGraph* method. This method only requires from concrete classes to serialize the graph into textual representation, with no mention of particular format whatsoever.

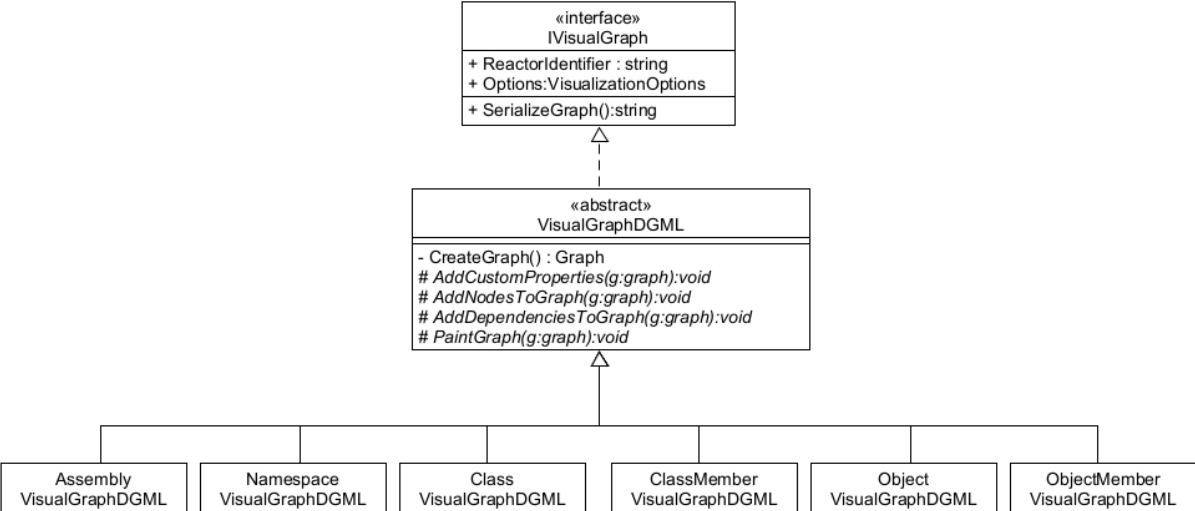


Figure 40: Hierarchy of VisualGraph classes

In order to open serialized graph in DGML Viewer, we have to save graph’s in-memory textual representation into a file. This responsibility can be assigned to any class implement-

ing *IGraphFileCreator* interface (see Figure 41). Its sole mandatory method is *CreateNewFile* which is obliged to take any graph implementing *IVisualGraph* interface and save it to a file. This allows us to replace implementation details related to output file, such as naming, extension, format, location etc., without rest of the framework being aware of that. So far, the only realization of this interface is a *DGMLFileCreator* which holds the concrete implementation for creating *.dgml* files.

As with representation of graph hierarchies in *ReframeCore* and *Analyzer*, here in *Visualizer* we also wanted to further eliminate dependencies on concrete *VisualGraphDGML* classes that would arise in graph instantiation process. Like before, we employed *abstract factory* pattern, which resulted in hierarchy of *factory* classes, with each concrete *VisualGraphDGML* class having its own concrete factory. In Figure 41 we can see the example of interaction between the roles of "abstract factory" and a "product" played by *IVisualGraphFactory* and *IVisualGraph* respectively, and their concrete counterparts *AssemblyGraphFactoryDGML* and *AssemblyVisualGraphDGML*. The same relationship can be depicted for all other levels of analysis (e.g. namespace level, class level etc).

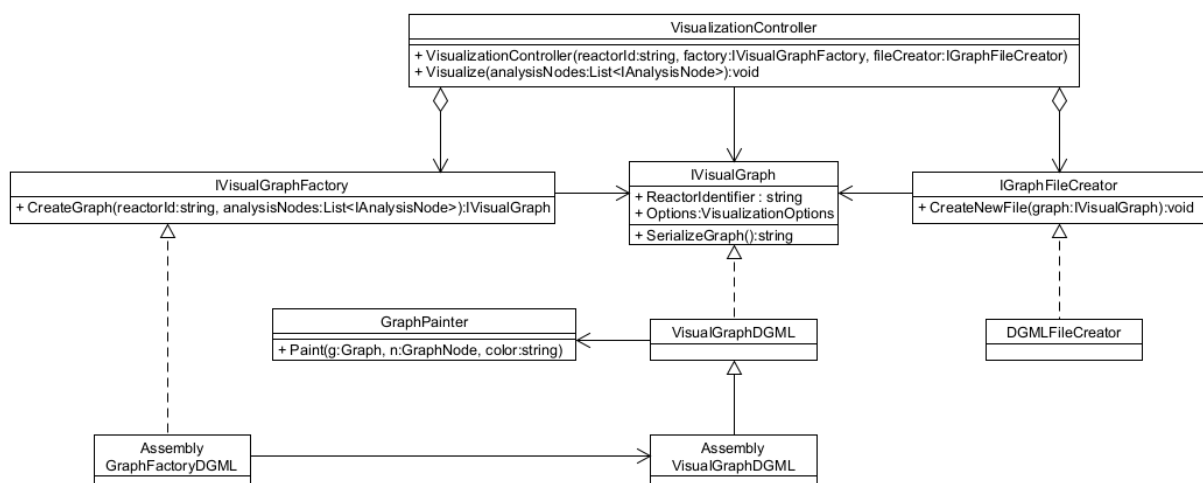


Figure 41: Most important interfaces and classes in Visualizer

The class responsible for orchestrating the whole visualization process is *VisualizationController* (see Figure 41). Its *Visualize* method takes the list of nodes which represent the result of performed analysis, uses this nodes to create the *IVisualGraph*, prompts user for any visualization options, saves the graph into a file, and finally opens the file in a dedicated viewer. While performing this process, *VisualizationController* delegates important steps to *factory* and *file creator* objects, which were provided using *dependency injection* technique. In addition to

separation of concerns, the intent here was to make *VisualizationController* class resistant to changes introduced in concrete *IVisualGraph* realizations and chosen visualization technology.

6.3.2.3. Code generation

Making REFRAME's APIs clean and simple was one of the design goals from the very beginning. In addition to overall efforts in framework design, the most explicit contribution to this goal was offering the access to common framework features through *IReactor* interface (an example of *Facade* pattern). However, as a first, preparational step to offering code generation in REFRAME, we invested effort to additionally simplify most repetitive tasks by building small DSL layer on top of *IReactor* interface.

One of the ways to create DSLs in .NET environment is using Boo [2] programming language. However, after examining this technology, it became apparent that effort required for creating full-scale DSL would not be justified in the context of REFRAME due to its fairly small number of domain concepts. Instead, equally fitting but more lightweight approach called *fluent interface* syntax [55] proved to be a better option. Fluent interface tries to imitate the look and feel of working with DSLs, and enables code to be read similar to natural language. A number of frameworks, inside and outside of .NET use fluent interfaces to increase readability and ease of use (e.g. Entity Framework).

Implementation-wise, fluent interfaces assumes adjusting the naming of the methods and allowing method chaining. Since we did not want to alter the code part of the framework for fluent interface implementation, we used a special technique called *Extension methods* [7], which allowed us to extend REFRAME's APIs without changing its original classes. This technique is frequently used when adding new functionality to existing libraries and frameworks (e.g. LINQ operators in .NET) in order to prevent problems caused by changing frameworks APIs. It includes implementing static class with static methods which are then at compile time associated with the classes we want to extend.

In listing 6.7 we demonstrate the idea behind fluent interface in the context of REFRAME. The first part shows very simple scenario where we used *IReactor* interface to add three reactive nodes and then establish two reactive dependencies between them. The second part uses fluent interface to do the exact same thing, but in an abbreviated and more readable form. The whole fluent statement can be read like this: *use reactor instance to let some reactive node depend on some other reactive node(s)*.

Listing 6.7: Adding reactive dependencies with Reactor and fluent interface

```
//Using reactor object directly
var node1 = reactor.AddNode(objA, nameof(objA.PA1));
var node2 = reactor.AddNode(objB, nameof(objB.PB1));
var node3 = reactor.AddNode(objC, nameof(objC.PC1));
reactor.AddDependency(node2, node1);
reactor.AddDependency(node3, node1);

//Using Fluent interface
reactor.Let(() =>objA.PA1).DependOn(() =>objB.PB1, () =>objC.PC1);
```

As can be seen in Figure 42, fluent interface implementation is located in *ReactorExtension* class, which offers three distinct methods: (1) *Let*, (2) *DependOn*, and (3) *Update*. While *Let* and *DependOn* methods are always paired together to replace the use of *AddNode* and *AddDependency* methods, *Update* method is used separately as an alternative to *PerformUpdate* method.

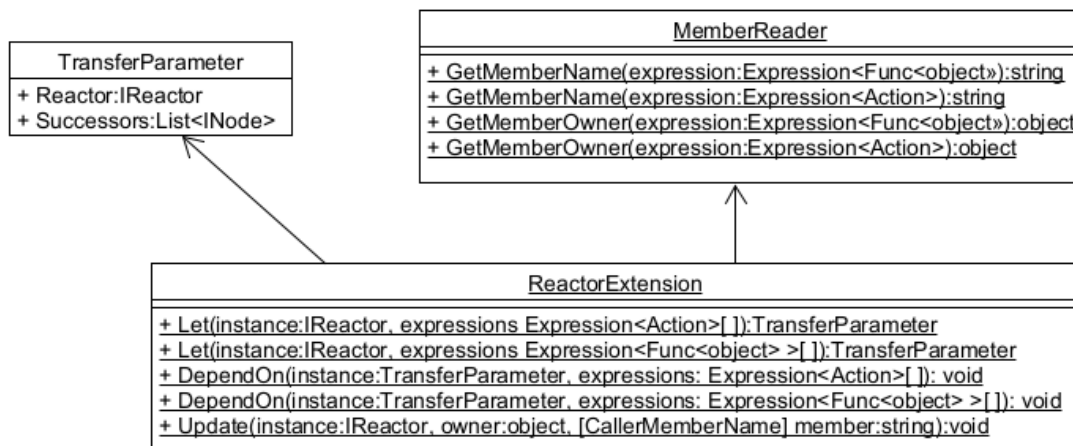


Figure 42: Fluent interface implementation in REFRAME

Behind a bit complex signature, *Let* method essentially allows us to specify the successor node for reactive dependency. Apart from implicit *IReactor* parameter, it accepts one parameter - a lambda expression pointing at the exact property or method member which represents a reactive node. Using lambda expression for this purpose allows us to utilize code completion features of the IDE, and reduce the risk of type errors and the issues with subsequent renamings. Behind the scenes, *Let* method deconstructs provided lambda expression and extracts the

information about the owner object and the class member. These are then used to create reactive node and add it to dependency graph.

The method chaining is realized through *TransferParameter* object, which is returned by *Let* method and accepted as an implicit parameter by *DependOn* method. Through *TransferParameter* (inspired by Data Transfer Object pattern [54]) we pass active *Reactor* instance and successor reactive node to *DependOn* method. In addition, *DependOn* method also accepts one or more lambda expressions pointing at members which will represent reactive nodes. After constructing predecessor nodes from provided lambda expressions, *DependOn* method now has everything it needs to establish reactive dependencies.

The Update method from fluent interface wraps reactor's PerformUpdate method. In order to simplify the method call, we used the fact that the Update is usually going to be invoked within the object and member which define the triggering node. This means that, while we always have to specify reactor instance, the owner object parameter can be replaced with *this* keyword. Also, in order to avoid the need to explicitly pass the member name, we can decorate this parameter with *CallerMemberName* meta-attribute. By doing this, we instructed compiler to automatically pass the name of called property or a method.

Listing 6.8: Triggering change with Reactor and fluent interface

```
//Using reactor object directly
reactor.PerformUpdate(this, nameof(PA1));

//Using Fluent interface
reactor.Update(this);
```

After we additionally simplified framework's APIs that are going to be used repetitively, in the next step we can approach the very code generation. As we pointed out in the previous section, we will be using code snippets to generate parts of the repetitive code. In order to do that we have to determine what part of the framework APIs statements is fixed and can be reused through code generation, and what part is variable and needs to be filled-in manually by application developer. If we, for example, take a look at the *Let->DependOn* method pair, we can identify three points of variability, namely: (1) the instance of reactor, (2) successor reactive node, and (3) one or more predecessor nodes. That means that the code snippet for *Let-*

>*DependOn* statement will generate the skeleton of the statement with default values for these variability points, but will allow us to easily fill-in our own values. Listing 6.9 shows these variability points (surrounded by curly brackets) for code statements we supported by code snippets. We can see that we provided code snippets for 3 frequently repeated code statements: *GetReactor*, *Let->Depend*, and *Update*. These snippets are easily invoked by typing in *refget*, *refdep*, and *refup* shortcuts respectively.

Listing 6.9: Variability points in fluent interfaces

```
//GetReactor statement
var {reactor} = ReactorRegistry.Instance.GetReactor({"default"});

//Let->Depend statement
{reactor}.Let(()=>{successor}).DependOn(()=>{predecessor1},...);

//Update statement
{reactor}.Update(this);
```

The code snippet implementation is actually a XML-based specification, which consists of a header and snippet definition. The header contains snippet's meta-data such as title, author, description and shortcut. On the other hand, snippet definition contains the template describing the code statement which will be generated and the description of replacement parameters (variability points). The example of snippet implementation for *Let->Depend* code statement is shown in Figure 43.

6.3.3. High-level design of REFRAME

In order to understand high-level design of REFRAME we will show how the framework is partitioned into components (.dll libraries), and how these components are mutually related. We will start by examining *end-user application perspective* of design (see Figure 44), by showing components which application will require in order to access features of the framework. There are total of three different framework use models to chose from. The first, and the most basic use model assumes using only two components from REFRAME, namely: *ReframeCore* and *ReframeBaseExceptions*. These are mandatory components which allow end-user application to access all essential features of REFRAME, i.e. to construct dependency graphs and perform

```

<CodeSnippet Format="1.0.0">
  <Header>
    <Title>LetDepend</Title>
    <Author>Marko Mijač</Author>
    <Description>Inserts Let->Depend command</Description>
    <Shortcut>refdep</Shortcut>
  </Header>
  <Snippet>
    <Code Language="CSharp">
      <![CDATA[
        $reactor$.Let(()=>$successor$).DependOn(()=>$predecessor$);
      ]]>
    </Code>
  <Declarations>
  <Object Editable="true">
    <ID>reactor</ID>
    <Type>ReframeCore.IReactor</Type>
    <ToolTip>Reactor instance which handles reactive dependency.</ToolTip>
    <Default>reactor</Default>
  </Object>
  <Object Editable="true">
    <ID>successor</ID>
    <Type>System.Object</Type>
    <ToolTip>Specifies member to serve as a successor node.</ToolTip>
    <Default>successor</Default>
  </Object>
  <Object Editable="true">
    <ID>predecessor</ID>
    <Type>System.Object</Type>
    <ToolTip>Specifies member to serve as a predecessor node.</ToolTip>
    <Default>predecessor</Default>
  </Object>
  </Declarations>
</Snippet>
</CodeSnippet>

```

Figure 43: Implementation of code snippet for Let->Depend statement

update process.

The second use model introduces optional *ReframeFluentAPI* component, which as its name implies contains implementation of Fluent interface. This allows us to, in addition to traditional imperative style, also use alternative, more declarative approach when specifying individual reactive dependencies between nodes.

Finally, the third use model is only relevant if we want to use REFRAME tools. It allows us to setup inter-process communication between end-user application and REFRAME tools. By introducing *IPCServer*, *ReframeServer* and *ReframeExporter* components, we can start the server in end-user application which will respond to requests from the Analyzer tool (client) and send dependency graph data. Since integration of server components into end-user application and starting the server is a trivial task, application developers can switch to and from this use model very easily.

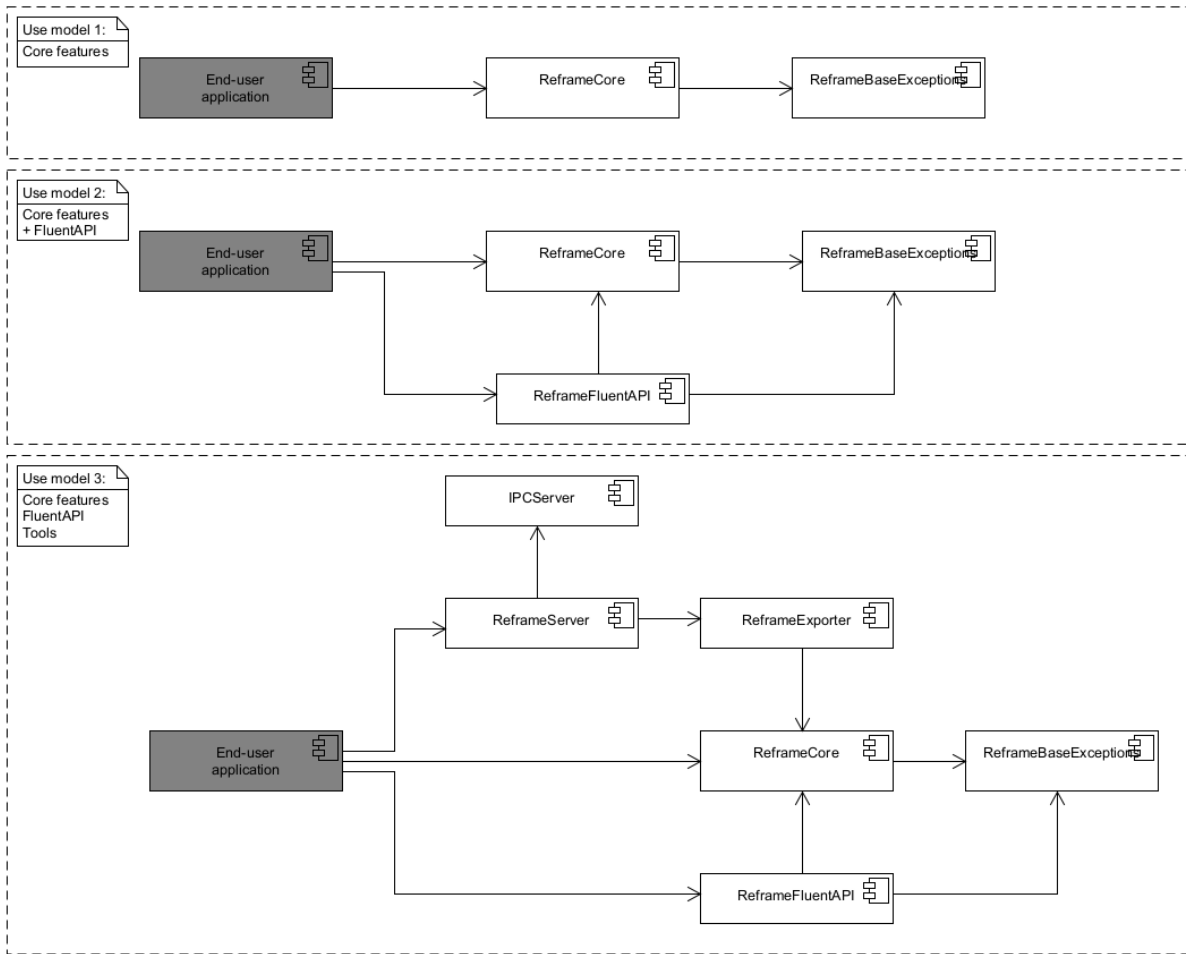


Figure 44: End-user application perspective

Table 12: REFRAME Components from the end-user application perspective

Component	Description
ReframeCore	The core component of the framework which enables end-user application to construct dependency graphs and perform update process. It contains abstractions representing members of reactive node hierarchy, dependency graph, update, scheduler, reactor and other essential parts of the framework.
ReframeBaseExceptions	Contains the definition of the root REFRAME exception which is inherited by specific exceptions in other components.
ReframeFluentAPI	Contains classes with extension methods which allow us to use reactor in a declarative style.
IPCServer	Contains interfaces and abstract classes with reusable part of the server side of inter-process communication.

Table Table 12 – *REFRAME* Components from the end-user application perspective

Component	Description
ReframeServer	Contains concrete classes with REFRAME-specific implementation of the server side of inter-process communication.
ReframeExporter	Contains classes responsible for exporting dependency graph data and update process data in a form of XML content.

The second perspective on the use of REFRAME is relevant only if we want to use REFRAME tools. As can be seen in Figure 45, the starting point of REFRAME tools is *ReframeToolsGUI* component. This component defines joint graphical user interface, through which application developer can access tools' features. In order to fetch dependency graph data from end-user application, REFRAME tools need to play the client role in inter-process communication. This is accomplished using *IPCClient* and *ReframeClient* components. *ReframeAnalyzer* component is responsible for interpreting fetched data and performing different analyses, while *ReframeVisualizer* and *VisualizerDGML* are in charge of visualizing the results of these analyses.

Table 13: Components from REFRAME tools perspective

Component	Description
ReframeToolsGUI	Contains graphical user interface classes and coordinates the rest of the components.
IPCClient	Contains interfaces and abstract classes with reusable part of the client side of inter-process communication.
ReframeClient	Contains concrete classes with REFRAME-specific implementation of the client side of inter-process communication.
IPCServer	Contains interfaces and abstract classes with reusable part of the server side of inter-process communication.
ReframeAnalyzer	Contains abstractions representing members of analysis graph and analysis node hierarchies, factories for creating graph and node objects, filter specifications, analysis and metrics implementations, and other parts related to graph and update analysis.
ReframeBaseExceptions	Contains the definition of the root REFRAME exception which is inherited by specific exceptions in other components.

Table Table 13 – *Components from REFRAME tools perspective*

Component	Description
ReframeImporter	Contains utility classes which interpret XML content fetched from end-user application.
ReframeVisualizer	Contains interfaces and abstract classes with reusable and technology-independent part of the visualizer implementation.
VisualizerDGML	Contains concrete classes implementing visualizer using DGML technology.

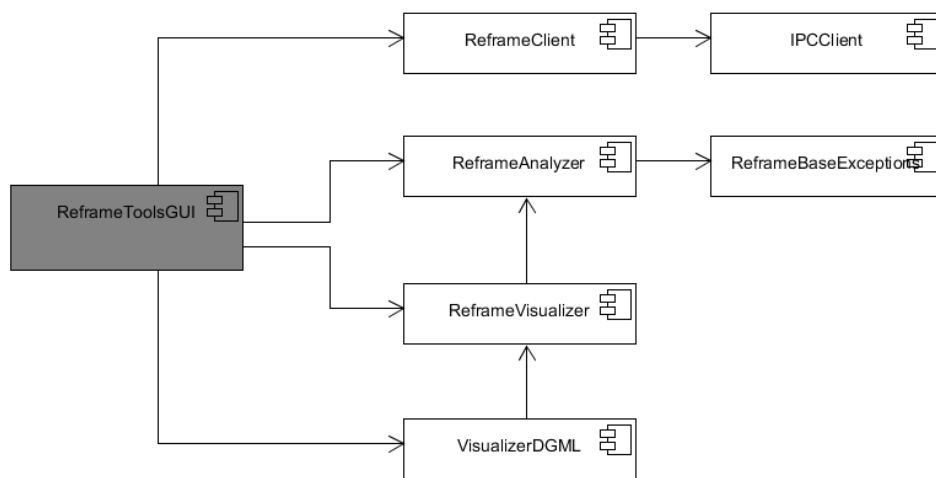


Figure 45: *REFRAME Tools perspective*

It is worth to mention that the current layout of components is the result of multiple refactoring sessions, in which responsibilities were transferred between classes, and classes were transferred between components. The primary goal was to mold components into logically consistent units and reduce coupling between them. Low coupling between components promotes modularity, and makes replacing whole components possible with minimal changes to other parts of framework. In addition, this allows us to use components selectively, and import into our project only the ones we really use.

Direction of coupling should also be such to make components with core domain functionality as stable as possible, i.e. with minimal number of dependencies towards other components. In this way, the only reason to change core domain components would be the very change in some aspects of domain.

The effects of such design can be seen in throughout REFRAME. For example, in Figure 44,

we can see how *ReframeCore* component, which contains the most important part of the framework, is coupled only to *ReframeBaseExceptions* component. Since *ReframeBaseExceptions* component contains only the most general REFRAME exception specification, it is unlikely to change. This not only makes *ReframeCore* very stable component, but it also allows the first use model from end-user application perspective. *ReframeFluentAPI* component depends only on stable *ReframeCore* and *ReframeBaseExceptions*, which means it will change only if core domain concepts change or in case we want alter something in the fluent syntax. At the same time, the component has no incoming dependencies from framework components, which makes it easy to replace. Finally, in case of components implementing server-side of inter-process communication (*ReframeServer* and *ReframeExporter*), we can provide replacement components without any effect to other components in framework.

Similar situation can be seen in Figure 45. For example, changes to tool's graphical user interface in no way affect the rest of the framework. Replacing components implementing client-side of inter-process communication will require only changes to *ReframeToolsGUI* which coordinates the interaction between tool's components. *ReframeAnalyzer* does have incoming dependency from *ReframeVisualizer*, which implies visualizer components may be affected by the change in analyzer component or by its replacement. However, this dependency is at the level of interface specification and not concrete implementation, which means *ReframeVisualizer* will not be affected by mere implementation changes. Lastly, while dependency graphs are currently visualized using DGML technology, this can be changed by replacing *VisualizerDGML* component with alternative implementation, and calling this new implementation in *ReframeToolsGUI*.

6.4. Justify and reflect

In this section we will look back at efforts made in design and implementation of REFRAME, and summarize and rehearse the most important ones. Up until now, we discussed these efforts in the context of individual aspects of *ReframeCore* and *ReframeTools*, such as structural and behavioral characteristics of classes, relationships between classes, and their placement into components. While we tried to elaborate these decisions on the spot by placing them in the context of established good practices and design patterns, in the first part of this section we will cover the general design principles which underlined the whole design process.

The second part of this section, however, will document and elaborate REFRAME's *design rationale* by listing most important design decisions.

6.4.1. Underlying design principles

A lot of different principles have been proposed as guidelines and inspiration for design rationale. One of the most commonly used sets of such design principles are best known under acronym SOLID. As is the case with most principles, patterns and practices in software development, SOLID design principles are not the work of one author, rather, their underlying ideas have been evolving for years, alongside the software development practice itself. However, they are most famously organized, reported and advocated by Martin in several of his books (such as [99]).

Before discussing the implications of each of the five SOLID design principles on REFRAME design, it should be noted that they are not meant to be applied in a large upfront design activity. Rather, they represent a lens through which we continually and critically observe our design in each development and maintenance iteration. It is equally important to view SOLID principles as really the principles, and not the strict rules we must adhere to at all cost. Indeed, finding the right balance between, often contradictory and opposing requirements, principles and constraints, may just be one of the most difficult things about software design. The five principles are not addressing discrete issues, but tend to work as a whole. Therefore, design decisions made in order to comply to one design principle, often result in complying to other design principles as well. Sometimes, however, they can also contradict each other.

SOLID principles are fairly general which allows them to be applicable to a wide range of domains and problems in software design. This, however, has unfortunate consequence of them being somewhat vague and open to various (mis)interpretations. Indeed, even brief analysis of developer forums show many requests for clarification of these principles and also many attempts in offering one. Sadly, offered clarifications often demonstrate the same vagueness as the principles themselves, and also often contradict one another. This is why, in addition to reporting how each of the principles was applied to REFRAME design, we also offered brief clarification of the principles. The intention was not to repeat what can be found in different books, blogs, forums or lecture materials, but rather to clarify interpretation we adopted, which is necessary to understand underlying rationale for design decisions that were made.

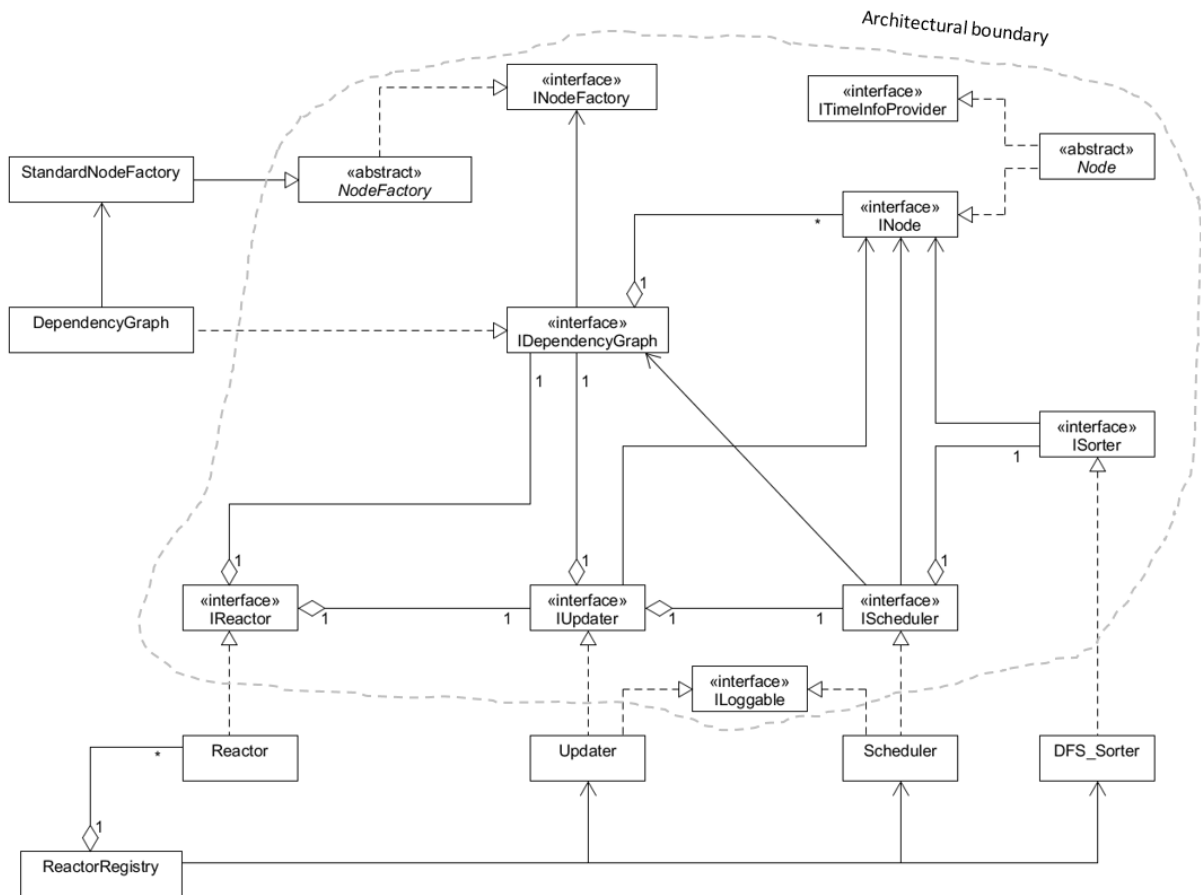


Figure 46: Most important interfaces and classes in ReframeCore

Single Responsibility Principle (SRP)

The first of the SOLID principles, *Single Responsibility Principle* (SRP) advocates one *responsibility* per class, i.e. one reason to change the class. This principle is inspired by similar principle called *Separation of Concerns* (allegedly introduced by Dijkstra), and the concepts of *coupling* and *cohesion*. The main idea is very reasonable - by isolating responsibilities we lower the coupling between classes and increase the internal cohesion within the classes, which altogether leads to increase in software quality.

While there is no exact recipe which guaranties framework design to be aligned with SRP, this was one of the guiding principles in our design process. We analyzed in advance the functionalities that had to be implemented in order to meet the requirements, and identified cohesive groups of functionalities that could potentially qualify as responsibilities. Identified responsibilities were throughout the development in several iterations refined and re-assigned to other existing or new classes. Throughout that process different refactorings [56] were applied

in order for code to adhere more to SRP.

In Figure 19 and Figure 46 we can see an example showing how the overall responsibility of *ReframeCore* to handle reactive dependencies is decomposed into a number of classes, each being assigned with a smaller, more cohesive responsibility. So, for example, the responsibilities of providing proper data structure for storing and manipulating reactive dependencies were assigned to *DependencyGraph* class and a hierarchy of *Node* classes. The range of responsibilities related to assuring consistency of dependency graphs, i.e. updating them, were divided across *Updater*, *Scheduler* and *Sorter* classes. Responsibility for bringing these components together, and exposing unified interface fell on the *Reactor* class. Finally, complexities involved in the process of creating reactive nodes and reactors were handled by *NodeFactory* and *ReactorRegistry* classes. There are of course also additional utility and helper classes which took over some part of the overall framework's responsibility. For example, *Reflector* is a static class representing a small library of methods, which facilitates the use of reflection in REFRAME by raising the level of abstraction.

Similar responsibility assignment efforts (see Figure 30 to Figure 42) also took place in *ReframeTools*. For example, dependency graph and node representations in *Analyzer* and *Visualizer* tool imitate the ones in *ReframeCore*. Whenever the creation process involved non-trivial tasks, this was seen as separate responsibility and specialized *factory* classes were introduced. Message and data exchange between end-user application and *ReframeTools* has been handled by a set of classes, with each having its own distinct role and responsibility. We have, for example, *PipeClient* responsible for sending a command, *PipeServer* listening for incoming commands, *CommandHandler* handling the command, and *Exporter* creating XML response. Furthermore, tasks such as performing analyses, visualizing graph, painting graph, creating graph file, displaying GUI forms, and others are all assigned to their own classes or even class hierarchies.

Although very simple in definition, SRP is often viewed differently by individuals in software development community. While the notion of a class being responsible for only one thing is agreeable to most, it is the understanding of what is to be considered a *responsibility* or a *reason to change* that is not so clear. This is why applying SRP has to be accompanied by applying common sense, because going into either of the extremes in determining granulation of responsibilities will have adverse effects on software design. On one hand, too large responsibility will

result in so-called "god classes" which impede the understandability and maintainability of software. On the other hand, treating each and every small functionality as a separate responsibility will result in flood of trivial classes, which could clutter overall design and break encapsulation. In fact, we should not only think about how many reasons for change has particular class, but also how many classes have to be modified in order to introduce simple change. In designing REFRAME we tried to find a middle ground between these extremes, and anticipate the main lines across which the reasons for change could come. That being said, the feedback from more extensive use of the framework may result in the need for refactoring current responsibilities, which is in line with the iterative perspective on applying SOLID principles.

Open-Closed Principle (OCP)

Martin [99] attributes the Open-closed principle (OCP) to Bertrand Meyer, another one of the classic authors, who stated that "*a software artifact should be open for extension, but closed for modification*" [104]. To put it another way, this means that we should be able to extend the behavior of the framework without modifying existing, already tested and working behavior.

The antagonism of the opposing forces in this principle is evident even in the very naming of the principle. How can something be both open and close? Indeed, as Martin and Martin [101] in their book say, OCP represents an ideal which cannot be achieved in its entirety. No matter how "closed" we design a class to be, there can always be a change we did not or could not anticipate, and which will require a modification of a class. Also, attempting to conform to OCP by anticipating in advance every, even remotely possible change and guarding against it, can prove to be extremely expensive and wasteful. However, OCP is a goal towards we should go and try to approach it as close as it makes sense. In reality, this means first closing the design against the changes that are most likely to occur and have the most adverse impact. Going further than that in early phases of development may be counterproductive, therefore it might be reasonable to wait and let the experience in use drive the need for design refactoring. Martin and Martin describe this as a "*fool me once*" attitude [101], which prevents us from introducing too much needless complexity into our design.

One of the main characteristics of OCP is that it enables us to add new behavior without it causing a cascade of changes all throughout the code. This proves to be extremely beneficial in terms of maintaining and adapting software to ever changing requirements, which makes an essence of what "*soft*" in software really means. While this makes OCP important for design of

software in general, it is especially relevant for software frameworks. For example, by applying OCP, framework developers benefit by framework being able to more easily evolve during its lifetime. Also, changing framework's existing classes by framework developers may cause code of any number of applications using the framework to become incompatible, broken and incorrect. What is even worse, framework developers and application developers may not even be aware of the issue before a lot of damage is done. From the perspective of application developers "closedness" allows framework to provide common and fixed parts of the design and implementation which can be reused. It can be even formally enforced if the framework is made blackbox. On the other hand, "openness" provides variability points of the framework, i.e. means to reuse design and implementation by extending the framework.

The principal underlying mechanism for achieving OCP is an abstraction. Software systems that conform to OCP are inevitably more abstract, i.e. they have higher number of abstract classes and interfaces. Whether it is introduced as an abstract class or interface, abstraction provides fixed design contract by specifying mandatory methods to be implemented. In addition, it also supports variability by omitting optional methods and concrete implementations, which is then deferred to a wide range of possible concrete classes.

In order to make a claim that REFRAME conforms to OCP at a sufficient extent, let's look for example at the class diagram in Figure 46. It shows core classes of the framework, the change of which could pose particularly large issue. What is easily noticeable, is that the system in the diagram contains proportionally large number of abstract classes and interfaces. Indeed, most of the concrete classes implement their respective interfaces. In order to show concrete example of OCP in action, let's examine the relationship and interaction of e.g. *Scheduler* and *DFS_Sorter* classes. The *Scheduler*, which is responsible for determining the update schedule for dependency graph, has to be provided with implementation of topological sorting algorithm. A variant of such algorithm, based on depth-first search, is implemented in *DFS_Sorter* class. Let's say, for the sake of discussion, that the *Scheduler* holds a direct reference to *DFS_Sorter*. If we would want to enable *Scheduler* to use some other topological sorting algorithm, such as e.g. Kahn's algorithm, in addition to implementing new class (e.g. *Kahns_Sorter*) holding the new algorithm, we would also need to modify the existing *Scheduler* class (and perhaps also some other classes), because it only knows how to use *DFS_Sorter*. This implies that the *Scheduler* does not conform to Open-closed principle, since in order to extend it to be able to

use different sorting algorithm, we have to modify it.

Scheduler can be made to conform with OCP by applying e.g. *Strategy* pattern. We first introduce abstraction, i.e. we extract the interface *ISorter*, which represents a *family of algorithms*, and which all existing and future concrete implementations of topological sorting algorithm must adhere to. Now, instead of *Scheduler* knowing, and thus depending on concrete classes, *Scheduler* holds reference to an instance of *ISorter*. This makes concrete implementation of topological sorting interchangeable in *Scheduler*, i.e. it makes it possible to change what sorting algorithm will be used without changing the *Scheduler* itself.

Similar claims of compliance with OCP can also be made for other classes across REFRAME. For example, in Figure 20 we can see how *Abstract Factory* pattern is used to manage instantiations of various types of reactive nodes. The abstract class *NodeFactory* implements *CreateNode* method which invokes abstract methods *DetermineNodeType* and *CreateNodeForType*. The *CreateNode* method represent a *Template method* pattern, a fixed part of the abstraction. On the other hand, methods *DetermineNodeType* and *CreateNodeForType* represent points of variability, and require from concrete factories, such as *StandardNodeFactory*, to provide implementations. In this way, *NodeFactory* class is able to be coupled only to *INode* reactive node generic interface, and to remain ignorant of any current or future concrete implementations of that interface. For example, if we would like to extend the system by adding new reactive node types, and also new concrete factory in charge of instantiating them, abstract *NodeFactory* would not need to change. Abstract node factory also has implications to *DependencyGraph* class (see Figure 46). Since *DependencyGraph* is coupled to abstract instead of concrete factory, it can also be seen as closed to change with regard to extending system with new concrete factories and nodes. Similar effect with abstract factories is also achieved in *Analyzer* (see Figure 34) and *Visualizer* tools (see Figure 41).

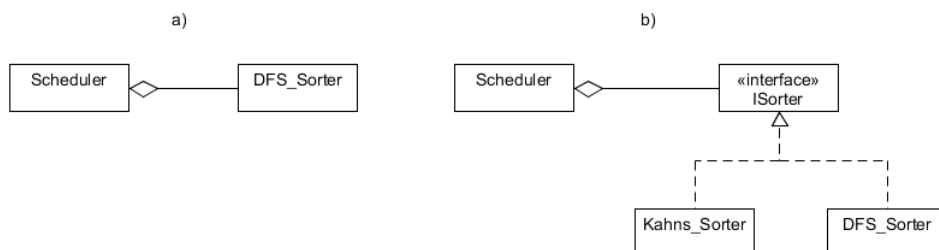


Figure 47: Scheduler design with regard to sorting algorithm: a) not conforming to OCP, b) conforming to OCP

Liskov Substitution Principle (LSP)

Liskov Substitution Principle (LSP) was initially about addressing the issue of forming proper inheritance hierarchies, i.e. it presented a guide for the use of inheritance. Later, it was extended to also address the proper use of interfaces. The principle was originally proposed by Barbara Liskov, however, Martin and Martin [101] offered refined and more concise definition for the principle: "*Subtypes must be substitutable for their base types*". What this means is that client classes which depend on the base class or an interface, have the right to assume that these can be substituted by any other class deriving from the base class or implementing interface, without compromising the expected behavior.

The underlying intent of the LSP is to assure that the client class will not have to change because of new derived class or interface implementation emerged, i.e. client class can remain unaware of the changes in class hierarchy. This makes LSP closely related to OCP. Indeed, violations of LSP are often just a symptom of disguised OCP violations. Some of the most frequent and obvious examples of LSP violations include: (1) forcing inheritance where it is not appropriate, (2) relying on downcasting and "type sniffing" instead of taking advantage of polymorphism, (3) degenerating methods - overriding methods in order to remove expected behavior implemented in a base class, (4) throwing new exceptions in derived classes, etc.

In order to check REFRAME for LSP compliance, the framework classes were re-examined for above mentioned symptoms of LSP violation. For example, existing hierarchies, such as node, graph, filter, exporter, GUI forms and other hierarchies have been critically assessed for its appropriateness. It has been concluded that all of them are sound hierarchies and represent real "is-a" class associations. The usage of derived classes from these hierarchies was analyzed in order to find and remove occurrences of downcasting or "type sniffing". Few occurrences that were found were resolved by applying "*replace type code with subclasses*" and "*replace conditional by polymorphism*" refactorings [56]. Derived class methods which override abstract or virtual methods from base class are examined for possible degeneration, but non was found. Finally, with regard to not throwing new exceptions in derived classes, this was secured by introducing REFRAME's own exception hierarchy. All the exceptions thrown in REFRAME's classes are inheriting *ReframeException* as a root exception class. Therefore, even if the more specific exception was thrown in the derived classes, the client class can handle it as a root *ReframeException*.

As with other principles, apart from keeping an eye on potential violations of principle, one should also be cautious not to over-design the system in the early stages, or be caught in design paralysis. Therefore, it is often reasonable to deal with the most obvious LSP violations early on, and defer dealing with other, not so obvious violations, until the issues requiring intervention appear.

Interface Segregation Principle (ISP)

As the name implies, Interface segregation principle (ISP) deals with design of the interfaces. It emphasizes the need to fight "fat interfaces" and interface pollution in software design. The obvious symptoms that suggest this might be needed is large number of public members and apparent lack of cohesion between them. Martin and Martin [101] formulated the principle as follows: "*Clients should not be forced to depend on methods they do not use*". The key notions from this formulation that is necessary to understand the principle is the notion of *client*. As is evident from the very formulation and also from the examples provided by the authors of the principle, a *client* is a class which *uses* methods of some other - *server* class and therefore depends on it. If the interface of the *server* class contains multiple methods with low cohesion between them (i.e. "fat interface"), it is probable that the client classes will only use subset of the server's interface methods. However, although the client classes are only interested in methods they explicitly use, they are dependent on changes of all methods of the server's interface. What ISP suggests in order to avoid this unnecessary dependence is that the server's interface should be partitioned in a way which allows client classes to only use and depend on methods they really need.

There is another very common interpretation of ISP that should be mentioned, because it is not only frequently found in various web sources, but also in books dealing with software design. According to this interpretation ISP aims to prevent having large interfaces which derived classes only partially implement, and leave number of methods unimplemented (e.g. by throwing *NotImplementedException*). However, while applying ISP would certainly contribute to prevention of such scenarios, it seems that this interpretation departs from original intent to tailor interfaces according to needs of *client* class. Instead it enters the area of assessing appropriateness of inheritance hierarchy which is intent of Liskov principle (LSP). In this section we will stick with the Martin's [101] original interpretation of ISP.

The mutual, strong association of the SOLID principles is especially evident when talk-

ing about ISP. If you, for example, tend to apply SRP in your design efforts and come close to classes with single and distinct responsibility, the chance of your design producing classes with "fat interfaces" significantly decreases. Similarly, applying LSP to have well-thought inheritance hierarchies also contributes in clean interface situation. However, design of the class which already has only one responsibility may still benefit from critically assessing its interface through the lens of ISP. This is because in the context of ISP we talk about *role interfaces* which are of smaller granulation than the *responsibility*. Also, role interfaces place a strong emphasis on the perspective and the needs of client classes.

As an example, lets look at the ISP compliance in the context of *ReframeCore*, although similar can be claimed also for *ReframeTools*. Figure 46 shows classes and interfaces at a conceptual level, and what is evident is that most classes implement their respective interfaces. Subset of these interfaces is shown in Figure 48 along with property, event and method members they specify. Even the cursory glance at the diagram reveals that *ISorter*, *IScheduler* and *INodeFactory* interfaces are very simple. Indeed, thanks to predominantly SRP, efforts in keeping the design clean allowed us to form almost trivial interfaces - exposing single (although overloaded) method. *INode*, *IUpdater* and *IDependencyGraph* interfaces are somewhat larger, so their members were examined to determine who their clients are, and also to determine if some of the members do not fit together with other members (low cohesion). This resulted in some new interfaces (see Figure 46), namely: *ITimeInfoProvider* and *ILoggable*, in which we moved certain utility members responsible for logging update progress. For remaining members we determined that they form fairly cohesive units. For example, *IDependencyGraph* interface prescribes cohesive set of methods required for managing the structure of dependency graph. Similarly, *IUpdater* interface requires implementation of members focused on performing, suspending and monitoring very update process. Finally, *INode* interface prescribes fundamental responsibilities of the reactive node, namely: reactive node identification, maintaining dependencies with other reactive nodes, and updating reactive node. Analysis of the client usage of interface members revealed that interface members are used by multiple clients, which raised the question of further interface segregation. There were several reasons we decided not to do that at this point. For example, additional interfaces would make framework design more complicated, and at this point it is not apparent if that is worth sacrificing. Indeed, even Martin [101] advises to take care to not overdo when exercising this principle. Also, the line separating

interface members according to their clients was not that straight and clear. Some members, for example, were used by multiple clients. Further segregation of interfaces in our case would require casting the same object from one interface type to another. Alternatively, explicit casting could be avoided by passing the same object multiple times as argument of different interface type, (Martin [101] calls this *polyadic* form of interface), however, by practicing this we would increase the size of method signatures.

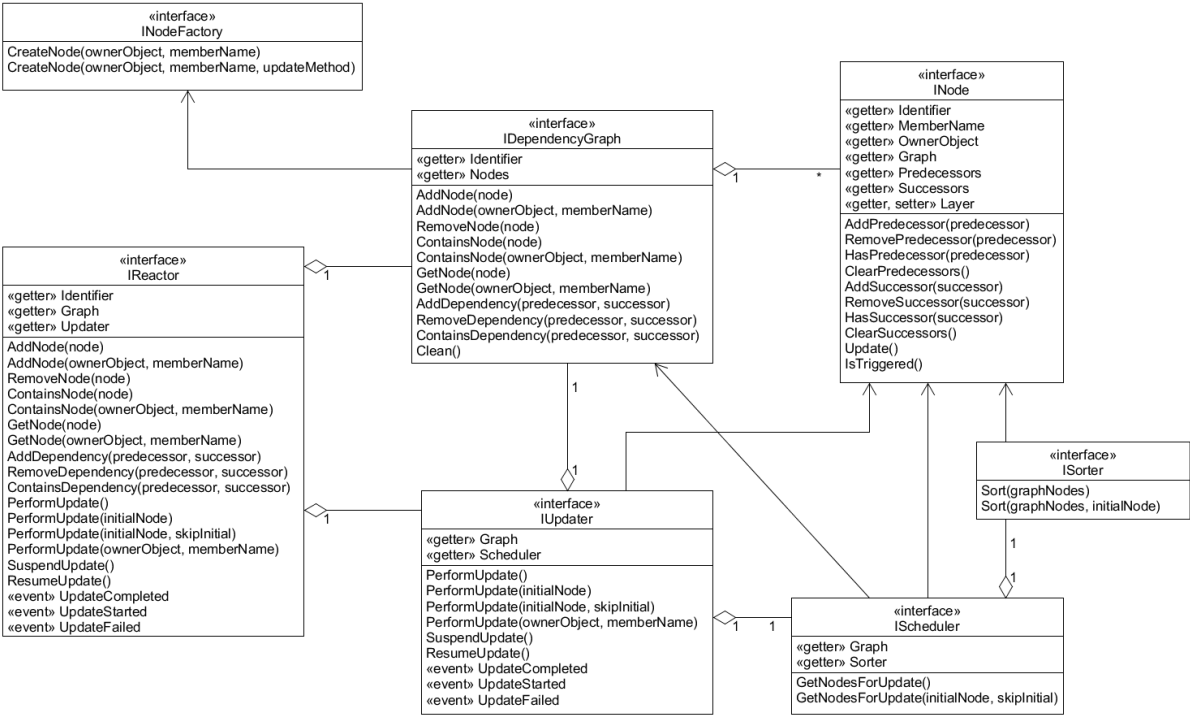


Figure 48: Core interfaces

The last interface we are going to discuss is *IReactor* interface. Since this is the biggest and arguably the least cohesive interface, it should raise the alarm from the perspective of ISP. However, the argument for *IReactor* interface remaining organized this way is the same as with SRP. *IReactor* interface simply plays the role of facade to a more complex underlying system, which makes it a deliberate "fat interface".

Dependency Inversion Principle (DIP)

Dependency inversion principle [99] aims at making software systems more flexible by organizing source code dependencies in a way that they point only to abstractions. The rationale behind this is that abstractions (e.g. abstract classes or interfaces) are consisted of mostly method specifications, which makes them more *stable* than concrete classes entirely consisted

of the often *volatile* implementation details. If the source code of a client class depends exclusively on abstractions, we can vary the details of the concrete implementations without affecting the client class. In such way we minimize and localize the effects that the changes will impose on the system, thus making the system easier to change. Of course, at some point we have to reference concrete classes. However, we should take care to minimize the number of such places in source code, and also make sure to minimize the volatility of these concrete classes. We already demonstrated exercising DIP in such scenarios when we were discussing reactive node hierarchy and the process of creating concrete reactive node objects (Abstract Factory pattern).

Another demonstration of the effects of DIP can be seen in Figure 49. In the example *a*), which shows the violation of the principle, one of the issues is that we have direct dependencies between concrete classes, namely *Updater*, *Scheduler* and *DFS_Sorter*. This surely has an adverse effect on the flexibility of the system, since any change affects the client classes. However, additional issue is the direction of dependencies. We can see that, for example, *Updater* class which handles higher-level update process, depends on classes *Scheduler* and *DFS_Sorter*, which are in charge of the details of that process. Since the details are more likely to change, such direction of dependencies is undesirable. The example *b*) shows how this is resolved by introducing abstraction (interfaces). The client class becomes dependent on the interface instead of on the concrete server class. On the other hand, server class implements the interface. By making both client and server concrete classes dependent on the abstraction, we removed original dependency between them. With regard to the notion of "inversion", in examples showing only classes it may not be apparent that the inversion indeed took place. However, if we would imagine packaging the client class and the introduced interface into one component, and the server class into another component, it would be obvious that the inversion was achieved.

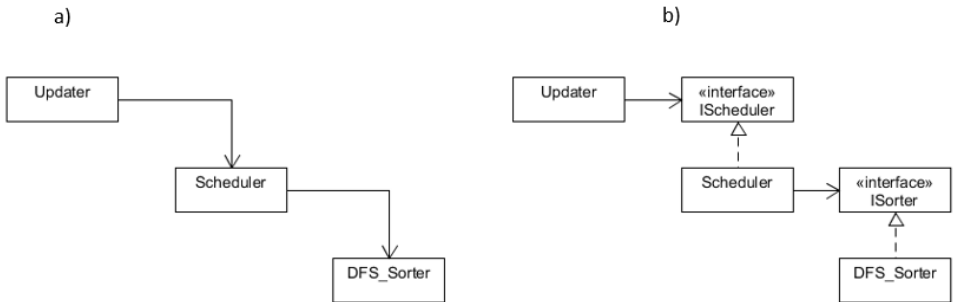


Figure 49: *a) violating Dependency inversion principle, b) conforming to Dependency inversion principle*

If we take a look at the Figure 46, we can see the broader picture of exercising Dependency inversion principle. None of the concrete classes depends on any other concrete class. Rather, the dependencies are specified at the level of abstractions. The dotted line which encircles part of the diagram represents the so-called *architectural boundary*. Martin [99] introduced it to separate the abstract part of the system (abstract classes and interfaces) which defines all the high-level rules and policies, from the concrete part containing the implementation details. In order to be in accordance to DIP, all dependencies which cross the architectural boundary have to point towards the abstract part of the system. As can be seen from the diagram, this is exactly the case. By depending only on the abstractions, we can easily change the implementation details without affecting high-level rules and policies. We can customize the system to fit our needs by introducing alternative concrete implementations of different classes, such as *Updater*, *DependencyGraph*, *Reactor* etc. As long as these new concrete classes conform to agreed interfaces, the rest of the system will not mind. This makes it a lot easier to reuse the system, which is why DIP is particularly important in the context of framework design. Additional benefit is that these different implementations can be used interchangeably at runtime.

6.4.2. Documenting design decisions

After discussing general underlying principles which steered our design efforts, in this subsection we will reflect on concrete design decisions which resulted from these efforts. During development of non-trivial piece of software, a developer may make hundreds of design decisions. These can refer to issues on a various levels of abstraction, such as: implementation level, micro-architecture level, or grand-scale architecture level. Each of the decisions are influenced by multiple factors, including available alternatives to chose from, prerequisites that have to be met, and constraints that narrow down our options. Hopefully, most of these decisions will prove to be correct, and end up in main development line. However, design decisions very often do not produce expected results, especially in novel domains where there is a lot of exploratory prototyping involved. In such cases design decisions and the resulting prototypes are abandoned and other candidate solutions take over.

Capturing each and every one of these right and wrong decisions is not feasible in a limited time frame of development project. This is why usually a subset containing most important design decisions is documented. Through the previous three sections of chapter 6 we tried to

describe the chronology of REFRAME development and elaborate important design decisions that had to be made along the way. In this section, however, we are going to briefly summarize these decisions.

Table 14: Summary of most important design decisions made in REFRAME development

Design issue	Decision elaboration
<p>1. What is going to be regarded as reactive node in OO application?</p>	<p>Decision: Reactive node is going to be regarded as a state (data) or behavior member, and the object on which that member is called. For a state member we chose .NET <i>property</i> language construct, which enforces data hiding and retains a sense of working with data members. This is consistent with requirements for the framework and similar approaches (e.g. design patterns such as Observer), and it also allows us sufficiently low level of granulation. For a behavior member we chose traditional methods.</p> <p>Considered alternatives: Reactive node could be regarded as only a state member or only a behavior member. From the implementation perspective, <i>fields</i> or explicit <i>getter/setter</i> methods could be used instead of <i>properties</i>. In terms of granulation, reactive node could have been modeled at the level of object, like in Observer pattern.</p> <p>Prerequisites: There needs to be a way to explicitly refer to members of an object (e.g. through reflection capabilities).</p> <p>Constraints: State members of complex types (e.g. arrays, collections) could require specific, more challenging implementation. Methods employed as update methods for state members, or as standalone reactive nodes should not return value, or accept parameters. Data exchange between methods needs to be handled some other way. Otherwise, the implementation of reactive node becomes cumbersome.</p>
<p>2. What language constructs shall be used to express reactive nodes?</p>	<p>Decision: Framework will introduce standalone abstractions, as it fits well to our needs, and is the most commonly used option of extending programming language with frameworks.</p> <p>Considered alternatives: Introducing a new keyword into programming language; utilizing meta-data to decorate class members; and Introducing 'reactive' data types.</p> <p>Prerequisites: Depending on the option for expressing reactive nodes, some extension capabilities of programming language could be required, e.g. access to compiler modifications, meta-programming techniques, etc.</p> <p>Constraints: Some of the options may not be possible within particular programming language. Extending the language with new keyword would require implementation of custom compiler, which is complex undertaking, more challenging to extend by framework user, and also results in diverging from official compiler. Utilizing meta-data is inadequate as the meta-attributes operate on a class-level and are too simplistic. Introducing 'reactive' data types would require inheriting large number of data types, and would seriously challenge the use of framework in existing end-user applications.</p>

Table Table 14 – Summary of most important design decisions made in REFRAME development

Design issue	Decision elaboration
<p>3. How are reactive nodes roles going to be represented?</p>	<p>Decision: Both roles are contained within one abstraction, which means reactive node can be at the same time both predecessor and successor.</p> <p>Considered alternatives: Instead of representing both roles with one abstraction, roles could be represented by separate abstractions (as in Observer pattern). Implementation of roles can vary by having various combinations of interfaces and abstract classes.</p> <p>Prerequisites: Reactive nodes have to be able to form acyclic dependency graph, therefore they have to support both roles at the same time.</p> <p>Constraints: In single-inheritance languages implementing roles as separate classes would prevent reactive node to play both predecessor and successor roles at the same time, making it impossible to form acyclic graph. Introducing roles as separate interfaces may prevent code reuse.</p>
<p>4. What will be the basis of determining reactive node's identity?</p>	<p>Decision: A unique identifier was generated based on the hash values of owner object and member.</p> <p>Considered alternatives: Skipping explicit identifier, and instead identifying reactive node directly by their object and member; Identifying reactive nodes by their reference.</p> <p>Prerequisites: No particular prerequisites were identified.</p> <p>Constraints: Since multiple instances of reactive nodes could refer to the same owner object-member pair, it is not possible to guarantee uniqueness in identify reactive nodes by their reference.</p>
<p>5. How will we make framework extensible in terms of adding new node types and customizing implementation of existing ones?</p>	<p>Decision: Reactive node hierarchy is introduced with top elements being <i>INode</i> interface and <i>Node</i> abstract class. Framework can be extended with new node types or alternative implementations of existing ones by providing concrete classes which derive from these top elements.</p> <p>Considered alternatives: Using composition instead of inheritance to handle variations in different node implementations.</p> <p>Prerequisites: No particular prerequisites were identified.</p> <p>Constraints: Inheritance hierarchies may become too large and rigid. On the other hand, with composition we have to re-implement all methods in derived type, even the ones which only forward the calls.</p>
<p>6. How are we going to decouple concrete implementations of different reactive node types from the rest of the framework?</p>	<p>Decision: Concrete implementations of different reactive node types have to conform to <i>INode</i> interface, through which all reactive node instances are accessed in the rest of the framework. The sole exception to this is the reactive node creation process, in which concrete classes have to be explicitly mentioned. However, in order to localize the effects of dependencies resulted from creation process, we introduced the role of <i>NodeFactory</i> (inspired by AbstractFactory design pattern).</p> <p>Considered alternatives: Using alternative design patterns (such as Builder or Factory Method) for localizing dependencies resulted from creation process.</p> <p>Prerequisites: No particular prerequisites were identified.</p> <p>Constraints: It can result in a complex design.</p>

Table Table 14 – Summary of most important design decisions made in REFRAME development

Design issue	Decision elaboration
7. How are reactive dependencies going to be represented?	<p>Decision: Reactive dependency is going to be represented implicitly (with no dedicated abstraction) as an association of two reactive nodes - predecessor and successor.</p> <p>Considered alternatives: Representing reactive dependency explicitly, by a dedicated abstraction.</p> <p>Prerequisites: No particular prerequisites were identified.</p> <p>Constraints: Representing reactive dependency implicitly, prevents additional data (e.g. weight, priority) to be assigned to reactive dependency. On the other hand, using explicit, dedicated abstraction may require adjusting dependency graph data structure and accompanying algorithms.</p>
8. What data structure will be used to hold reactive nodes interconnected with reactive dependencies?	<p>Decision: Directed-acyclic graph (DAG), implemented as an adjacency list, will be used as a proven data structure capable of expressing the nature of reactive nodes interconnected by reactive dependencies. When implementing dependency graph, we followed a centralized (middleware) approach, which means there is a dedicated abstraction representing dependency graph. At the same time we dispersed reactive dependencies across individual reactive nodes, so all nodes have information about their immediate predecessors and successors (due to better performance).</p> <p>Considered alternatives: Instead of adjacency list, one can use adjacency matrix or incidence matrix to implement dependency graph. Also, in case of adjacency list, nodes do not have to point at both their predecessors and successors.</p> <p>Prerequisites: Data structure should be able to efficiently hold a graph with a large number (thousands) of reactive nodes. It should allow dynamically adding and removing of reactive nodes and dependencies, as well as searching and sorting (topological) nodes.</p> <p>Constraints: Matrix structures are not so space efficient as adjacency lists. They also have performance problems in terms of dynamically adding or removing their elements. In adjacency list implementation, when reactive node points at both its predecessors and successors, it introduces additional memory footprint for the sake of performance.</p>
9. How is it going to be determined which reactive nodes have to be updated and in which order?	<p>Decision: Since reactive dependencies form an acyclic graph, exact nodes to be updated and the order in which this is done to avoid redundancy and inconsistency, is determined by performing topological sorting of the graph. A Tarjan's algorithm based on depth-first search (DFS) traversal algorithm is used.</p> <p>Considered alternatives: Instead of Tarjan's algorithm, other topological sorting algorithms can be used, such as Kahn's algorithm based on breadth-first search (BFS) algorithm</p> <p>Prerequisites: Reactive nodes have to be stored in a data-structure which allows topological sorting to be performed. Reactive nodes with their dependencies have to form an acyclic graph.</p> <p>Constraints: Original topological sorting algorithms are not capable of handling cyclic dependencies. Therefore, if there is a possibility for them to occur, algorithms have to be adapted to recognize them, and handle them appropriately.</p>

Table Table 14 – Summary of most important design decisions made in REFRAME development

Design issue	Decision elaboration
<p>10. How will we handle the case when there are multiple causes for update process?</p>	<p>Decision: In order to prevent multiple update processes in a short time frame as a result of multiple successive triggerings of change, two mechanisms are offered in REFRAME: (1) temporarily suspend update process, and then update entire graph, (2) temporarily suspend update process, and then introduce temporary source node to update only what is necessary.</p> <p>Considered alternatives: No alternatives have been considered.</p> <p>Prerequisites: No particular prerequisites are identified.</p> <p>Constraints: Performing update of entire graph may involve unnecessary updating of large number of nodes. Also, the user has to manually decide if and what prevention strategy he wants to employ.</p>
<p>11. How are we going to ensure interchangeability of sorting algorithms?</p>	<p>Decision: In order to ensure framework user can plug-in its own version of topological sorting algorithm without affecting other parts of framework, we used well-known <i>Strategy pattern</i>.</p> <p>Considered alternatives: Using Template method pattern instead of Strategy pattern. Using constructs offered directly by programming language (e.g. Func<> or Action<> delegate types in C#).</p> <p>Prerequisites: No particular prerequisites are identified.</p> <p>Constraints: Template method is based on inheritance, so it would not be possible to change sorting algorithms at runtime. Strategy can introduce additional complexity to software design.</p>
<p>12. How are we going to ensure responsiveness of end-user application during update process?</p>	<p>Decision: REFRAME will allow framework user to execute update process in a asynchronous manner - in a different thread than the GUI. The implementation will be based on .NET Task Parallel Library (TPL) and its core Task class.</p> <p>Considered alternatives: Using lower level threading mechanisms such as Thread class.</p> <p>Prerequisites: Programming environment needs to support multithreading.</p> <p>Constraints: Using lower level threading mechanisms may involve significant effort, and may be considered as obsolete. On the other hand, using higher-level frameworks or libraries such as TPL makes REFRAME dependent on more resources.</p>
<p>13. How are we going to enable parallel execution of update process?</p>	<p>Decision: Coffman-Graham's graph layering algorithm is used to form layers in which reactive nodes are independent and can be updated in parallel. The execution is based on .NET Task Parallel Library.</p> <p>Considered alternatives: No alternative algorithms were found.</p> <p>Prerequisites: Acyclic graph contains reactive nodes which are independent of each other, and can be separately updated.</p> <p>Constraints: Increase in performance by executing update process in parallel cannot be guaranteed. It depends on the graph structure, and how performance intensive is updating individual nodes. In some cases, overhead costs of dealing with threads could be higher than performance gains.</p>

Table Table 14 – Summary of most important design decisions made in REFRAME development

Design issue	Decision elaboration
14. How will we ensure that objects referenced in reactive nodes can be deallocated in application code?	<p>Decision: We use .NET language construct called weak reference, which provides access to object but does not prevent garbage collector from collecting the object if there are no strong references left.</p> <p>Considered alternatives: Rely solely on application developer to handle deallocation of reactive nodes when objects in application are removed.</p> <p>Prerequisites: Programming language has to support the concept of weak references.</p> <p>Constraints: Some programming environments may not support the concept of weak references.</p>
15. How will we simplify interaction with core features of the framework in common scenarios?	<p>Decision: We introduced higher level interface - Reactor, which is an implementation of Facade design pattern.</p> <p>Considered alternatives: No alternatives were considered.</p> <p>Prerequisites: No particular prerequisites were identified.</p> <p>Constraints: While Reactor abstracts some of the structural details of the system in order to simplify basic usage scenarios, it may discourage user from using advanced options. In addition, there is a risk for Reactor to become a "god" class.</p>
How will we keep track of <i>Reactor</i> instances and reuse them throughout the end-user application? 16.	<p>Decision: We introduced <i>ReactorRegistry</i> class (inspired by <i>Registry</i> design pattern) which is in charge of creating and fetching <i>Reactor</i> instances. Implementation of <i>ReactorRegistry</i> is also based on <i>Singleton</i> design pattern, so only one instance of it is globally available.</p> <p>Considered alternatives: Leave application developers to manually implement these features.</p> <p>Prerequisites: No particular prerequisites were identified.</p> <p>Constraints: No particular constraints were identified.</p>
17. What algorithms could be useful for analyzing dependency graphs?	<p>Decision: The graphs that we used and found useful were: <i>graph traversing</i> algorithms, <i>topological sorting</i> algorithms, <i>cycle detection</i> algorithms, <i>connectivity</i> algorithms, and algorithms for calculating <i>Centrality measures</i></p> <p>Considered alternatives: Depending on the application domain, any algorithm which operates on graph structure might be useful.</p> <p>Prerequisites: Algorithm must be able to perform on a acyclic directed graph and on a particular data structure representing that graph.</p> <p>Constraints: Algorithms operating, for example, on undirected graphs, or weighted graphs are not applicable to REFRAME.</p>
18. How to meaningfully reduce the number of nodes in dependency graph analysis?	<p>Decision: Through analyzer tool we offer two kinds of reductions: (1) <i>vertical</i> - allows viewing dependency graph on different levels of abstraction, and (2) <i>horizontal</i> - fetching subset of available nodes according to some criteria (filtering).</p> <p>Considered alternatives: No alternatives were considered.</p> <p>Prerequisites: Analyzer tool needs to gain access to runtime information about dependency graph in order to perform analysis.</p> <p>Constraints: Large number of different options for reducing the number of nodes may be confusing for user, and hard to interpret.</p>

Table Table 14 – Summary of most important design decisions made in REFRAME development

Design issue	Decision elaboration
19. At what levels of abstraction can dependency graph be viewed?	<p>Decision: We propose 6 levels of abstraction, namely: (1) <i>object-member level</i> (original, most detailed level), (2) <i>object level</i>, (3) <i>class-member level</i>, (4) <i>class level</i>, (5) <i>namespace level</i>, and (6) <i>assembly level</i>.</p> <p>Considered alternatives: No additional meaningful levels of abstraction were identified.</p> <p>Prerequisites: Analyzer tool needs to gain access to runtime information about dependency graph and its reactive nodes on an object-member level.</p> <p>Constraints: In large graphs, lower levels of abstraction (e.g. object-member level) may result in huge amount of information. This could be hard to process by tools or users if additional (e.g. filtering) mechanisms are not used to reduce the amount of information.</p>
20. What criteria can be used when fetching subset of dependency graph nodes?	<p>Decision: We proposed and implemented three criteria: (1) <i>filtering by affiliation</i> - filters nodes with regard to their affiliation with higher-level nodes, (2) <i>filtering by role</i> - filters nodes with regard to their role in dependency graph, and (3) <i>filtering by association</i> - filters nodes with regard to their association with other, same-level nodes.</p> <p>Considered alternatives: A feature to search reactive nodes by their identifier, owner, or some other criteria could be useful. A user could be allowed to save chosen filters for future use.</p> <p>Prerequisites: No particular prerequisites were identified.</p> <p>Constraints: Some combinations of different filters are not very useful and cannot be meaningfully interpreted.</p>
21. How can REFRAME tools be made an integral part of the framework?	<p>Decision: We implemented GUI and tool's logic in a traditional desktop application style. However, the tool is run as a Visual Studio extension. In this way we utilized flexibility of traditional desktop GUI, but also left possibilities for stronger association with Visual Studio IDE.</p> <p>Considered alternatives: Implementing tool as IDE extension (e.g. Visual Studio Extension), or IDE independent standalone tool.</p> <p>Prerequisites: Chosen IDE has to offer extension capabilities.</p> <p>Constraints: Due to limitations and issues in VS Extension technology, with pure IDE extension we were unable to access runtime state of end-user application, and also the GUI development was very crude and limited. With standalone tool we cut all ties with IDE and lose potentially useful mechanisms for IDE and REFRAME tools interaction.</p>

Table Table 14 – Summary of most important design decisions made in REFRAME development

Design issue	Decision elaboration
<p>22. How can Analyzer tool fetch dependency graph data from end-user application runtime?</p>	<p>Decision: In order to avoid executing both end-user application and REFRAME tools in the same process, we had to find a way to fetch runtime data from one process to another. We decided to use <i>Named Pipes</i> as it is a fast and reliable technology, and it offer everything we required without the need to introduce additional libraries. Named pipes allowed us to model exchange between Analyzer tool and end-user application in a form of client-server communication. Analyzer tool (client) can send a command requesting data, and the end-user application (server) would respond with data in XML form.</p> <p>Considered alternatives: Different technological solutions exists for implementing inter-process communication, ranging from simple file sharing to using dedicated frameworks such as WCF.</p> <p>Prerequisites: Used technology has to support some means of inter-process communications.</p> <p>Constraints: Inter-process communication may require significant amount of code on both client and server side. It can be hard to debug.</p>
<p>23. How to represent dependency graph data in Analyzer tool?</p>	<p>Decision: In order to decouple Analyzer tool from the core part of the framework, we implemented separate (although similar) classes to express graph and node structures. In this way core and Analyzer graph structures can evolve separately, without impeding each other. This allows Analyzer to be used even in completely different context.</p> <p>Considered alternatives: Reusing <i>Dependency graph</i> and <i>Node</i> hierarchy from core part of REFRAME.</p> <p>Prerequisites: No particular prerequisites were identified.</p> <p>Constraints: Reusing existing structures from core part of the framework are problematic due to differences in hierarchy and structure of dependency graph and nodes, as well as in domain logic. On the other hand, implementing separate structures may lead to code duplication.</p>
<p>24. How do we feed Visualizer tool with dependency graph data?</p>	<p>Decision: We used Analyzer tool as a provider of dependency graph data to Visualizer tool. This was justified because there was already an established IPC between Analyzer tool and end-user application, and also the natural input to Visualizer were different analyses provided by Analyzer tool.</p> <p>Considered alternatives: Establishing separate IPC directly between Visualizer and end-user application.</p> <p>Prerequisites: No particular prerequisites were identified.</p> <p>Constraints: Using Analyzer tool as a graph data provider for Visualizer tool makes these two tools coupled, and prevents Visualizer to be used without the Analyzer.</p>

Table Table 14 – Summary of most important design decisions made in REFRAME development

Design issue	Decision elaboration
<p>25. How to represent dependency graph data in Visualizer tool?</p>	<p>Decision: Here we applied the same reasoning as in Analyzer tool. Although the base graph-node representation is similar throughout the REFRAME, there are important differences in structure and methods, and also in purpose of classes. Therefore, Visualizer tool also got its own separate data structures for representing dependency graph and nodes for the purpose of visualizing them.</p> <p>Considered alternatives: Reusing <i>Dependency graph</i> and <i>Node</i> hierarchy from core part of REFRAME.</p> <p>Prerequisites: No particular prerequisites were identified.</p> <p>Constraints: Reusing existing structures from core part of the framework are problematic due to differences in hierarchy and structure of dependency graph and nodes, as well as in domain logic. On the other hand, implementing separate structures may lead to code duplication.</p>
<p>26. What visualization technologies can we use to draw dependency graphs?</p>	<p>Decision: We decided to use DGML - one of the available specialized graph drawing libraries. It was compatible in terms of technology, stable, mature, easy to use, had available viewer, and it was free.</p> <p>Alternatives: Two fundamental approaches were considered: (1) using general, low-level drawing libraries (such as .NET's System.Drawing namespace), or (2) using specialized, graph drawing libraries (such as yFiles.NET, DGML, Graph# etc.).</p> <p>Prerequisites: Used technology has to provide either low-level drawing capabilities or specialized libraries.</p> <p>Constraints:</p> <p>Using low-level drawing libraries involves dealing with very complex issues related to graph drawing, such as determining the layout of the graph. On the other hand, specialized graph drawing solutions may not be available in every programming language. In addition, existing solutions may have various disadvantages, such as being hard to use, lacking graph viewer, not being open-source or free, etc.</p>
<p>27. How can we make Visualizer flexible in terms of visualization technology?</p>	<p>Decision: In order to make visualization technology easily replaceable, client classes use Visualizer's classes through exposed interfaces. As long as these interfaces are respected, concrete implementation and technology may vary. In addition, interfaces and concrete classes are kept in separate components, therefore, in order to replace concrete implementation, we just have to reference new component and initialize instances.</p> <p>Alternatives: No alternatives were considered.</p> <p>Prerequisites: No particular prerequisites were identified.</p> <p>Constraints: Such flexibility may result in complex design.</p>

Table Table 14 – Summary of most important design decisions made in REFRAME development

Design issue	Decision elaboration
28. What parts of code should be subject of code generation in REFRAME?	<p>Decision: Code generation will receive the largest benefit from code statements which are frequently repeated. In the context of using REFRAME these are primarily statements in charge of (1) defining reactive nodes and (2) specifying reactive dependencies, but fair amount of repetition can also be expected when (3) triggering changes, and (4) fetching reactor instances.</p> <p>Alternatives: Introducing code generation also for other code statements involved with REFRAME.</p> <p>Prerequisites: There is a code repetition that cannot be avoided in some other way.</p> <p>Constraints: Only a fixed part of the code statements are generated, and the developer still has to manually insert variable part.</p>
29. How to increase code readability for most frequently used framework features?	<p>Decision: In addition to keeping interfaces clean, we offered alternative syntax for setting up reactive dependencies in a more declarative style, similar to natural language. In order to do that we used <i>fluent interface</i> syntax approach, which can be described as lightweight DSL.</p> <p>Alternatives: Offer clean imperative interfaces in combination with one or more Facade pattern implementations.</p> <p>Prerequisites: Techniques necessary to implement fluent interfaces should be available (e.g. extension methods).</p> <p>Constraints: Declarative style employed by fluent interface may not suit all developers. Also, fluent interface may not fit well with some features.</p>
30. What code generation techniques are suitable for use in the context of REFRAME?	<p>Decision: We decided to abandon heavyweight code generation approaches, and use something simple but still effective. After simplifying reactor’s interface and introducing fluent syntax, Visual Studio’s code snippets were used to generate parts of the repetitive code.</p> <p>Considered alternatives: A number of code generation techniques are available in .NET ecosystem, such as: T4 templates, CodeDOM, IL Rewriting, Roslyn API, Code snippets, Quick actions, and other.</p> <p>Prerequisites: Chosen code generation techniques should be able to alter existing code (code injection).</p> <p>Constraints: Although there are number of code generation techniques available in .NET ecosystem, most of them are not suitable for changing the existing code. Even those that make the <i>code injection</i> possible either at design or runtime (e.g. IL Rewriting or .NET Compiler API), still require from us to specify what exact statements should be generated. In the context of REFRAME, it was not particularly more convenient to specify this information rather than writing the target code statement itself.</p>

Table Table 14 – Summary of most important design decisions made in REFRAME development

Design issue	Decision elaboration
31. How to make REFRAME modular at the level of components?	<p>Decision: Through the series of refactorings, which included transferring responsibilities between classes and classes between components, framework components were molded into logically consistent units with low coupling. This allows application developers to use only components which are essential for framework functionalities they need, and also to introduce new or replace existing components with relative ease.</p> <p>Alternatives: No alternatives were considered.</p> <p>Prerequisites: No particular prerequisites were identified.</p> <p>Constraints: Requires a lot of experience, effort and experimenting to get things done right. The interaction between modules may be complex, and could require thorough documentation.</p>
32. How is the overall responsibility of the framework going to be divided among individual classes?	<p>Decision: Throughout the iterations of framework development <i>Single responsibility principle</i> was one of the guiding principles in assigning portions of framework's overall functionality to individual classes. This involved constant deliberation about coupling and cohesion of framework's constituent parts, and often resulted in performing different refactorings.</p> <p>Considered alternatives: Separation of concerns, High cohesion - Low coupling principle, Information Expert principle (GRASP), etc.</p> <p>Prerequisites: No particular prerequisites were identified.</p> <p>Constraints: Aforementioned principles can be fairly general, which makes them somewhat vague and open to various (mis)interpretations. Very often, fixating on particular principle can bring adverse effects to software design. Also, different principles can sometimes act as opposing forces and contradict each other.</p>
33. How to make sure the framework is able to evolve and be extended in a safe manner?	<p>Decision: One of the principles that we took as a guide for this goal was <i>Open-closed principle</i>. Being closed for modification for REFRAME meant that we have to provide fixed, invariant part of the framework in a form of interfaces and abstract classes. On the other hand, being open for extension meant these abstractions have to allow sufficient variability, such as framework user being able to provide new concrete implementations or configure existing ones. In addition to OCP, Liskov substitution principle was also important for this goal, especially for the purpose of evaluating appropriateness of class hierarchies.</p> <p>Considered alternatives: Polymorphism principle (GRASP).</p> <p>Prerequisites: No particular prerequisites were identified.</p> <p>Constraints: The same as in design issue number 32.</p>
34. How to provide clean design of interfaces?	<p>Decision: In order to prevent occurrence of "fat interfaces" and interface pollution we applied <i>Interface segregation principle</i>. This was done by continuously assessing cohesion of interface members, examining interfaces from the perspective of the roles they play towards client classes. A result of this was applying refactorings to further decompose the elements of the framework, which represented a natural continuation of the process started by applying Single responsibility principle.</p> <p>Considered alternatives: Single responsibility principle.</p> <p>Prerequisites: No particular prerequisites were identified.</p> <p>Constraints: The same as in design issue number 32.</p>

Table Table 14 – Summary of most important design decisions made in REFRAME development

Design issue	Decision elaboration
35. How to make framework more flexible?	<p>Decision: In order to make framework flexible. i.e. easy to change, we applied <i>Dependency inversion principle</i>. This meant allowing dependencies only towards most stable parts of the system - abstract classes and interfaces, and not the volatile concrete implementations. A result of this is client classes being unaffected by changes in implementation details. Visually, effects of DIP can be described by architectural boundary line.</p> <p>Considered alternatives: Hollywood principle, Inversion of control.</p> <p>Prerequisites: No particular prerequisites were identified.</p> <p>Constraints: The same as in design issue number 32.</p>

7. Artifact Evaluation

7.1. Episode I - Prototyping and testing

As described in chapter 3, the first evaluation episode was conducted in the formative period of the framework, intertwined with activities of design and development. The goal was to obtain feedback that would guide further development of the framework. Here we used two evaluation techniques: *prototyping* and *testing*. The main purpose of prototyping was to assess *technical feasibility* of various aspects of the framework, i.e. to try out and compare different technologies, design and implementation options. Testing, on the other hand, was more focused on demonstrating *efficacy* by showing that individual parts and a framework as a whole are working and can be used to solve a problem.

7.1.1. Prototyping

Prototyping was very tightly integrated into the framework development process, so it was extensively used in *Design and develop* phase of this dissertation. The ultimate results of prototyping were most directly visible in the very design and implementation of the framework. Since these were discussed and elaborated in detail in chapter 6, in this section we will only reflect on our general experiences in conducting prototyping.

Major prototyping sessions usually occurred when we faced significant uncertainty in decision making, especially if these decisions resulted in large-scale and long-lasting effects. Examples of such major prototyping sessions include: trying out different solutions for asynchronous and parallel update process; trying out different meta-programming techniques which could be used for automating dependency management and code generation; finding an adequate way to establish communication and data exchange between end-user application and REFRAME tools; comparing different graph visualization technologies; exploring extension capabilities of Visual Studio IDE, etc. Since all of these prototypes were exploratory, there was no point in writing production-quality code and wasting time integrating each one of them with the rest of the framework. Rather, prototypes were created as separate solutions/projects, which were easily discarded after the session was done (so-called *throwaway prototypes*). The most important things these prototyping sessions showed us was: (1) what design and implementation options are feasible, and (2) what are the general steps to realize these options. Direct reuse of

programming code was rarely the case.

We used prototyping also during *Sketch and build* activity when we wanted to try out different options for making smaller-scale design and implementation decisions. Due to their more limited scope and dependence on the rest of the framework, these prototypes were developed integrally with the framework. In these cases we used *git version control* capabilities to support throwing away or accepting the prototype. For example, each prototype had its own separate development branch (or more of them), which enabled us to experiment without the fear of breaking main development branch. In case the prototype proved to be unsuccessful, its branch was easily discarded and we could go back to main development branch. Otherwise, if the prototype proved to be successful, we could choose to *merge* the branch into main development branch. There was also a compromise solution, which enabled us to choose only subset of changes from prototype branch to be included, i.e. the so-called *cherry-picking*. In any case, if proved successful, these prototyping sessions resulted in direct reuse of design and implementation. The examples of such smaller-scale prototyping sessions include trying out different refactorings to increase the code quality.

7.1.2. Testing

Software testing is an activity in software process which, according to Sommerville [137] has two distinct goals: (1) demonstrating that software meets the requirements of its stakeholders, and (2) discovering incorrect or undesirable behavior. These two goals align well with the *efficacy* evaluation property. Indeed, we can argue that the software which meets its stakeholder's requirements with no major defects, can be considered as a working artifact with a potential to solve a problem pointed-out by stakeholders.

While testing is essential activity in general software development, it is particularly important in software framework development. This is because application developers expect to be able to rely on the framework and focus on testing application code. A bug in a framework would spread through-out all end-user applications which use the framework, thus potentially becoming much more devastating than the bug in application code. We discussed this in detail in section 2.2 of literature review where we covered the topic of software frameworks and framework-involved processes.

Traditionally, testing was often depicted as a separate activity within the software develop-

ment process, conducted *summatively* after the software was developed. However, the advent of agile software processes emphasized the need to introduce testing as early as possible, which shifted a lot of the testing efforts towards the code development activity. By interleaving it with implementation, testing was awarded a more active, *formative* role in shaping design and implementation of the software, which is especially relevant for this evaluation episode. One of the agile practices which enabled this to happen was *test-driven development* (TDD) [28]. While *intensive testing* during development activities was not particularly revolutionary, the idea of *testing-first* introduced by TDD was. It assumed software developers (not testers) first writing tests and only then writing programming code that satisfies these tests. As much as this seemed counter-intuitive, agile proponents boasted numerous benefits of this approach, including: introducing discipline in writing tests, providing greater test coverage, and increasing design and code quality.

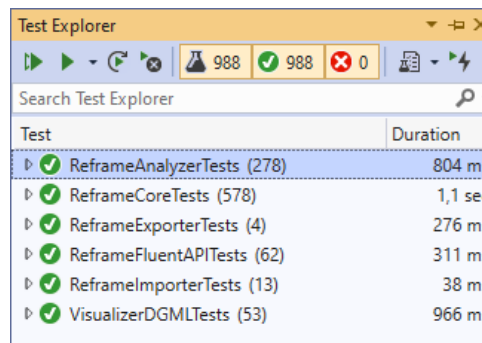
The primary automated testing technique which we were relying on in this evaluation episode is *unit testing*. This technique is a usual starting point in testing process and also the foundation of test-driven development. Osherove [112] in his popular and influencing book compiled the following definition: "*A unit test is an automated piece of code that invokes the method or class being tested, and then checks some assumptions about the logical behavior of that method or a class*". In addition to just stating what unit test is, Osherove also found it necessary to emphasize what *good* unit test is, so he extended his definition with the following: "*Unit test is almost always written using a unit-testing framework. It can be written easily and runs quickly. It is fully automated, trustworthy, readable and maintainable*".

In addition to its usual benefits, unit testing is very important in the context of this dissertation for two reasons. The first reason is related to the nature of how frameworks are used. We know that framework users, i.e. application developers, use software frameworks by writing code which, for example, instantiates framework's classes, inherits them, invokes their behavior and method members, etc. This means that, unlike in end-user applications, the framework's "user interface" is a set of available classes and their exposed members. The important implication here is that, besides testing for correctness of particular framework's feature, unit tests simulate the instances of framework use. Indeed, the code invoking some method from the framework's API, is the same, regardless of whether it is written by framework user as a part of application code, or it is written by framework developer as a part of unit test. This allowed us

to see framework through the eyes of potential users, which led to optimizations and fine-tuning of framework's interface with the goal of making it easier to use.

The second reason why we chose unit testing as a suitable testing technique in our context is related to evaluation process. If we take a look at how unit tests are conducted, we can see certain resemblance with laboratory experiments. Like experiments, unit tests also test assumptions we have about some system or phenomenon. By testing only small part of the system in isolation from other parts, unit tests create controlled environment. Finally, invoking method under test and asserting the outcome, can be seen as applying the treatment and observing the response. Therefore, we can see testing in this evaluation episode as conducting a set of large number of small-scale experiments on the framework.

During the course of REFRAME development around one thousand automated tests were written using *MSTest* unit testing framework. These tests were organized and run with the help of Test Explorer tool which is integrated with Visual Studio. As can be expected, due to their size and complexity, core part of REFRAME and Analyzer tool were the subject of most tests (see Figure 50). Depending on whether we intended to write production code from start or we just wanted to do some prototyping, we applied *test-first* or *test-after* approach respectively. Whatever we decided to do first, we tried to write both framework code and tests within the short time-frame.



Test	Duration
ReframeAnalyzerTests (278)	804 ms
ReframeCoreTests (578)	1,1 sec
ReframeExporterTests (4)	276 ms
ReframeFluentAPITests (62)	311 ms
ReframeImporterTests (13)	38 ms
VisualizerDGMLTests (53)	966 ms

Figure 50: Test Explorer panel in Visual Studio displaying written automated tests

Some of the reasons for such extensive testing we already elaborated when we discussed the perils of bugs in frameworks. However, there were some additional reasons for increasing the number of tests that are worth mentioning here. One of them is the fact that with frameworks it is harder to define the so-called *testing boundary*. For example, in end-user applications the programming code is usually under control of its authors, and is executed in a fairly predictable

manner by the user interacting with graphical user interface. In such cases, we can establish a boundary around parts of the code for which we can reasonably assume will work properly, and that any serious problems will be discovered by tests of larger granulation (e.g. end-to-end tests). On the other hand, with frameworks it is almost impossible to make such assumptions, as it is much harder to anticipate how application developers will use the framework. Thus, when testing REFRAME it was important to separately test each part of the code.

In Figure 51 we can see a simple example of unit test testing specific behavior manifested by individual method. One of the things that are visible from the very start are naming conventions, which play important role in organizing large number of unit tests. The very test class name, and test method name provide necessary information to quickly understand what the test is about, without the need to examine its code. The example unit test shown in Figure 51 is located in *PropertyNodeTests* class, which tells us that it tests some behavior of *PropertyNode* class. The very test method name follows the *MethodUnderTest_Scenario_ExpectedBehavior* [112] naming convention, which provides information on what method is tested, what exact scenario is tested, and what behavior we expect from the method under test. Again, in this example, we test the *HasSameIdentifier* method in scenario in which we have two nodes pointing at the same *object-member* pair. We can recall that in chapter 6 it was declared that we consider two reactive nodes to be identical if they point at the same *object-member* pair. Therefore, this unit test expects method under test to return *true*, i.e. that the two nodes have the same identifier.

```
[TestMethod]
0 references
public void HasSameIdentifier_NodesPointingAtTheSameObjectAndMember_ReturnsTrue()
{
    //Arrange
    Building00 building00 = new Building00();
    string memberName = "Width";

    var node1 = new PropertyNode(building00, memberName);
    var node2 = new PropertyNode(building00, memberName);

    //Act
    bool same = node1.HasSameIdentifier(node2);

    //Assert
    Assert.IsTrue(same);
}
```

Figure 51: Simple unit test example

If we take a look at the method body of unit test we can notice that it is structured according to well-known 3A (Arrange, Act, Assert) pattern [112]. The *arrange* part is in charge of creating a test scenario and preparing everything needed for a test to be performed. In our example, this involved creating two reactive nodes pointing at the same *object-member* pair. The second, *act* part performs the very test and captures the result, i.e. it executes *HasSameIdentifier* method. Finally, the *assert* part compares whether the outcome of the test is what we expected, i.e. whether the value returned from *HasSameIdentifier* method is true.

Unit test shown in Figure 51 is a simple one, as are many other tests we wrote for REFRAME. However, due to a complex domain, some test scenarios we wanted to cover also ended up being complex. Although all of our unit tests always aimed at testing only one method, in order to set-up a complex scenario multiple objects and method invocations were required in *arrange* part of the test. In extreme cases we even had to create a small infrastructure (utility methods and data structures) for defining test scenarios and asserting them. Such complex tests are, for example, written for *PerformUpdate* method of *Update* class. Testing this method involved constructing multiple differently structured dependency graphs, performing update process on them, and asserting whether correct nodes are updated in a correct order. In Figure 52 we can see an example of unit test for *PerformUpdate* method, where the test case involved constructing dependency graph with eight nodes, and performing update after a change in one of the nodes. In order to reuse test code and increase maintainability of unit tests, construction of dependency graph was done in separate method (*CreateTestCase1*). In this way, we were able to write other unit tests based on this dependency graph structure. Finally, we built a special log class (NodeLog) which enabled us to log reactive nodes in a specific order. The *actualLog* object was result of the update process and reflected the actual order in which reactive nodes were updated. On the other hand, *expectedLog* was manually constructed by us, and it reflected what we expected correct update process would look like. Asserting whether these two log objects are equal tells us if the test passes or fails.

Individual unit tests such as the ones shown in Figure 51 and Figure 52 only guarantee that a small piece of framework's code works in a specific scenario. However, when we ensure proper code coverage with hundreds of unit tests, we significantly increase the evidence that the framework works as a whole and that it can be used to solve a problem. In this way, testing contributes to the evaluation of framework's *efficacy*.

```

private Tuple<IDependencyGraph, Building00> CreateTestCase1()
{
    IDependencyGraph graph = new DependencyGraph("G1");
    Building00 building = new Building00();

    building.Width = 10;
    building.Length = 9;
    building.Height = 4;
    building.Area = 10 * 9 * 4;

    public void PerformUpdate1_Case1_GivenValidObjectAndMemberName_SchedulesCorrectUpdateOrder()
    {
        //Arrange
        Tuple<IDependencyGraph, Building00> caseParameters = CreateTestCase1();
        IDependencyGraph graph = caseParameters.Item1;
        Building00 building = caseParameters.Item2;
        Updater updater = CreateUpdater(graph);

        //Act
        updater.PerformUpdate(building, "Width");

        //Assert
        NodeLog actualLog = updater.NodeLog;
        NodeLog expectedLog = CreateExpectedLog_Case1_GivenWidthOrLengthAsInitialNode(graph, building);
        Assert.AreEqual(expectedLog, actualLog);
    }

private NodeLog CreateExpectedLog_Case1_GivenWidthOrLengthAsInitialNode(IDependencyGraph graph, Building00 building)
{
    NodeLog nodeLog = new NodeLog();
    nodeLog.Log(graph.GetNode(building00, "Area"));
    nodeLog.Log(graph.GetNode(building00, "TotalConsumption"));
    nodeLog.Log(graph.GetNode(building00, "Volume"));
    nodeLog.Log(graph.GetNode(building00, "TotalConsumptionPer_m3"));

    return nodeLog;
}

```

Figure 52: Complex unit test example

Although the code coverage metric is not perfect, and certainly not sufficient to characterize software as error-free, it is reasonable to assume that high code coverage will result in fewer errors. The exact percentage of code covered by tests that is considered sufficient is still the subject of ongoing debates. However, in literature and among practitioners most recommendations range from 80% up to 100%, which we also tried to achieve in the context of REFRAME. We calculated code coverage using Visual Studio's internal analyzer tool. The calculation is based on *block* measure, which official documentation for Visual Studio defines as a "piece of code with exactly one entry and one exit point" [3]. In Figure 53 we see that the overall code coverage for entire framework is 95,27%, while the coverage of individual components ranged between 91% and 100%. Since all of these numbers are at the upper half of recommended values, we can conclude that the framework is sufficiently covered with tests.

Code Coverage Results				
mijac_DESKTOP-OASO2L2 2020-10-16 14_41				
Hierarchy	Not Covered (Blocks)	Not Covered (% Bloc...	Covered (Blocks)	Covered (% Blocks)
└─ mijac_DESKTOP-OASO2L2 2020-1...	1118	4,73 %	22521	95,27 %
└─ reframeanalyzer.dll	95	3,83 %	2388	96,17 %
└─ reframeanalyzertests.dll	250	5,30 %	4464	94,70 %
└─ reframebaseexceptions.dll	0	0,00 %	2	100,00 %
└─ reframecore.dll	163	7,52 %	2005	92,48 %
└─ reframecoreexamples.dll	104	7,97 %	1201	92,03 %
└─ reframecoretests.dll	233	2,82 %	8034	97,18 %
└─ reframeexporter.dll	41	8,01 %	471	91,99 %
└─ reframeexportertests.dll	5	2,75 %	177	97,25 %
└─ reframefluentapi.dll	36	8,70 %	378	91,30 %
└─ reframefluentapitests.dll	44	3,18 %	1340	96,82 %
└─ reframeimporter.dll	0	0,00 %	62	100,00 %
└─ reframeimportertests.dll	5	6,41 %	73	93,59 %
└─ reframevisualizer.dll	0	0,00 %	23	100,00 %
└─ visualizerdgm.dll	78	6,22 %	1176	93,78 %
└─ visualizerdgmtests.dll	64	8,09 %	727	91,91 %

Figure 53: Code coverage results

7.2. Episode II - Demonstration

In the evaluation *Episode I* we evaluated *technical feasibility* and *efficacy* of the framework in a *formative* manner. This was done during the very REFRAME development, with evaluation's main intent being to provide feedback to influence further development, corrective activities and eventually completion of REFRAME. In the evaluation *Episode II* we stick to the evaluation of *technical feasibility* and *efficacy*, however, this time we do it *summatively*. This means that the *Episode II* is the first episode conducted during official evaluation activities, i.e. after REFRAME was developed. The intent here was an overall assessment of REFRAME in order to reach conclusions about its worth.

The evaluation method we used in this episode is *demonstration*, which is the most frequently used evaluation method in design science [119]. We will define several illustrative scenarios showing the use of REFRAME in the context of managing reactive dependencies in OO applications. This will demonstrate that we were able to build REFRAME through several iterations of design and implementation steps (*technical feasibility*), but also that REFRAME holds features and characteristics required to solve the problem of managing reactive dependencies (*efficacy*). By being the first evaluation episode after REFRAME is developed, demonstration is an early summative evaluation, and it will be conducted in artificial environment with fictional example of reactive dependencies. As such, it will represent the basis for conducting more naturalistic evaluation in third and fourth evaluation episodes.

7.2.1. Base demonstration example

In order to demonstrate different illustrative scenarios of REFRAME use, we first have to devise demonstration example on which these scenarios will be conducted. Since this is early summative evaluation, the example will be based on a simple fictional software application in need of managing reactive dependencies. We will simplify the application in terms of size, structure, business logic, and other aspects that are not directly related to handling reactive dependencies. However, the dependency graph resulted from created reactive dependencies will be complex enough to illustrate the advantages of using REFRAME to manage them.

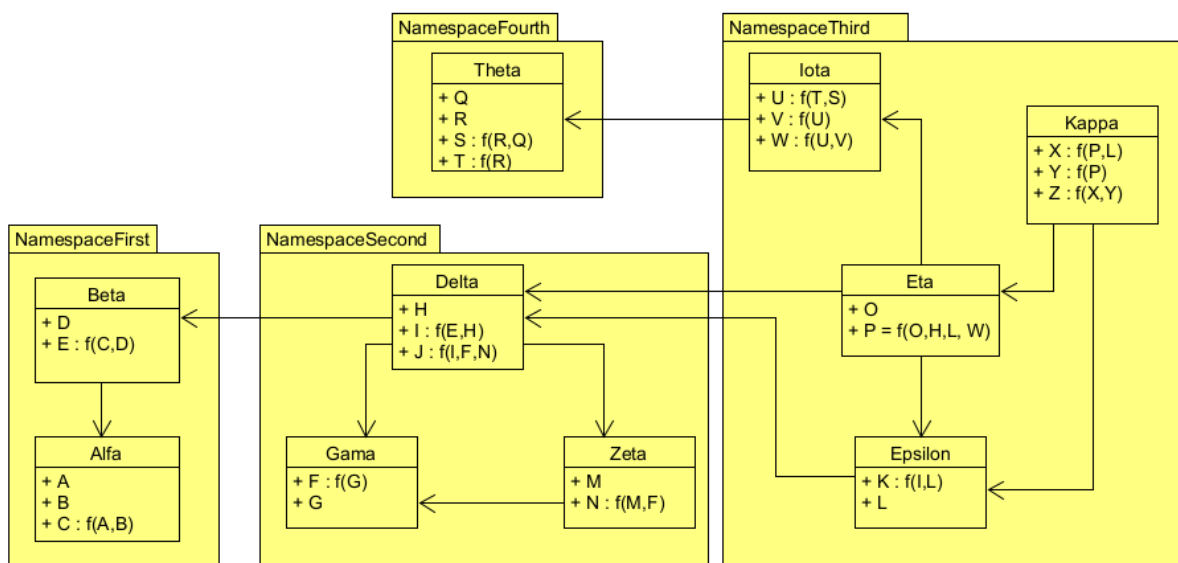


Figure 54: Class diagram of base demonstration example

As shown in Figure 54, demonstration example contains ten fictional classes arranged in four assemblies/namespaces. In addition to showing class members, we adjusted the class diagram to show that some members are a result of manipulating one or more other members, thus becoming dependent on them. For example, member P of class Eta is some kind of a function f of members O (class Eta), H (class $Delta$), L (class $Epsilon$), and W (class $Iota$). Each of these members, together with their respective owner objects (i.e. object-member pairs), will represent a reactive node in dependency graph (red nodes in Figure 55). These reactive nodes will also be mutually interrelated and form four reactive dependencies which can be described by the following ordered pairs: (O, P) , (W, P) , (H, P) , (L, P) .

For the sake of clarity, for each class shown in Figure 54 we created one instance, which resulted in total of 10 objects. Paired with their members, these objects formed total of 26

reactive nodes interconnected with 28 reactive dependencies. In Figure 55 we can see that even dependency graph of such limited size is complex enough to require significant effort to manage it manually using traditional solutions such as Observer pattern. Not only we would have hard time constructing dependency graph, but we would also need a lot of time to manually analyze and visualize the graph. As we will see, with REFRAME these tasks are going to be much easier to perform.

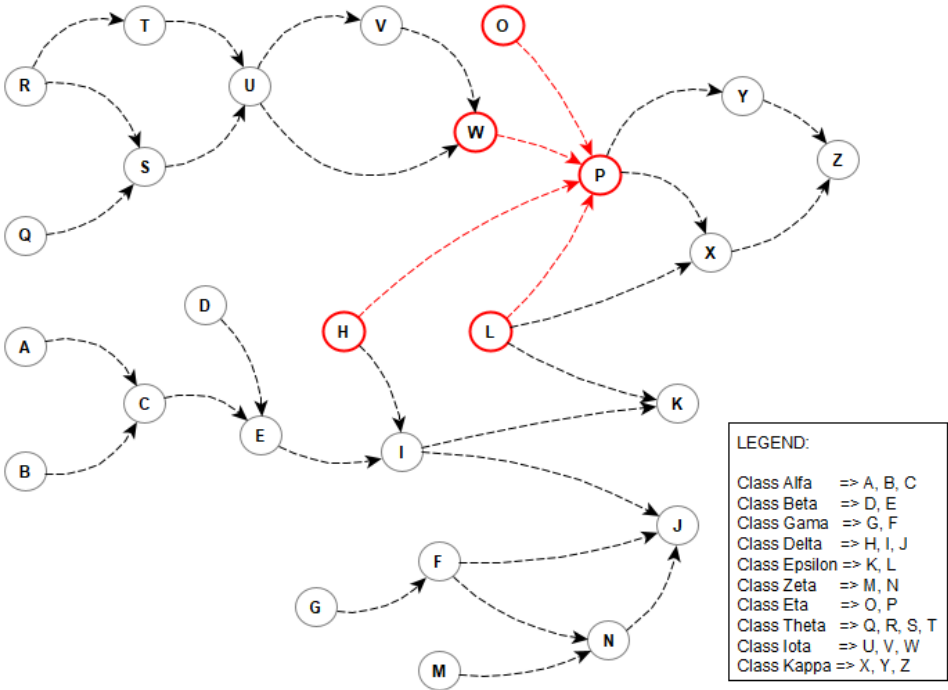


Figure 55: Dependency graph of base demonstration example

7.2.2. Illustrative scenarios

Scenario 1

- **Title:** Creating and registering new reactor object with identifier "default".
- **Prerequisites:**
 - Import following components in end-user application: *ReframeCore.dll* and *ReframeBaseExceptions.dll*.
- **Main scenario rollout:** New reactor object with identifier "default" is created and registered in *ReactorRegistry* in order to be available for further use.

```
var reactor = ReactorRegistry.Instance.CreateReactor("default");
```

- **Alternative scenarios:** Failing to provide reactor identifier, or attempting to create reactor with existing identifier will result in *ReactorException* exception. Also, other overloads of *CreateReactor* method can be used to inject dependencies.

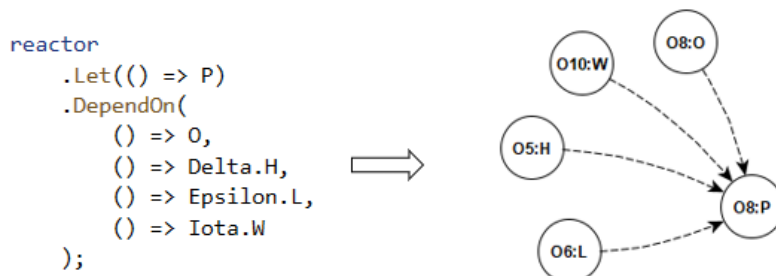
Scenario 2

- **Title:** Getting existing reactor object with identifier "default" from registry.
- **Prerequisites:**
 - Import following components in end-user application: *ReframeCore.dll* and *ReframeBaseExceptions.dll*.
 - Reactor with specified identifier exists in *ReactorRegistry*.
- **Main scenario rollout:** Existing reactor object with identifier "default" is fetched from registry.


```
var reactor = ReactorRegistry.Instance.GetReactor("default");
```
- **Alternative scenarios:** Given no reactor with specified identifier exists in registry, a *ReactorException* is thrown. Also, a *GetOrCreate* method can be used if we want to create reactor in case it is not found in registry.

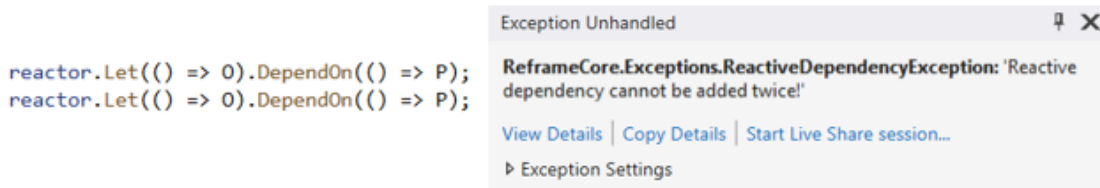
Scenario 3

- **Title:** Specifying reactive dependencies.
- **Prerequisites:**
 - Import following components in end-user application: *ReframeCore.dll*, *ReframeBaseExceptions.dll* and *ReframeFluentAPI.dll*.
 - Existing reactor is fetched from *ReactorRegistry*.
 - *Eta* has instances of *Delta*, *Epsilon* and *Iota* classes in scope.
- **Main scenario rollout:** After fetching reactor, in *Eta* class *Let- > DependOn* fluent syntax is used to add reactive nodes and specify four reactive dependencies. In all four dependencies, *P* is a successor which depends on four of its predecessors, i.e. *O*, *W*, *H*, *L*.

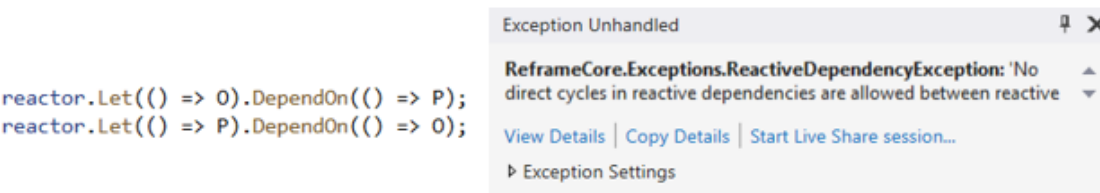


■ **Alternative scenarios:**

Given an attempt to specify already existing reactive dependency, the framework will throw *ReactiveDependencyException*.



Given an attempt to create direct cycle between two nodes, i.e. make them both a predecessor and successor of each other, the framework will throw *ReactiveDependencyException*.



Scenario 4

■ **Title:** Performing update process of entire dependency graph.

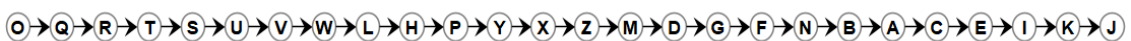
■ **Prerequisites:**

- Import following components in end-user application: *ReframeCore.dll*, *ReframeBaseExceptions.dll* and *ReframeFluentAPI.dll*.
- Existing reactor containing dependency graph shown in Figure 55 is fetched from *ReactorRegistry*.

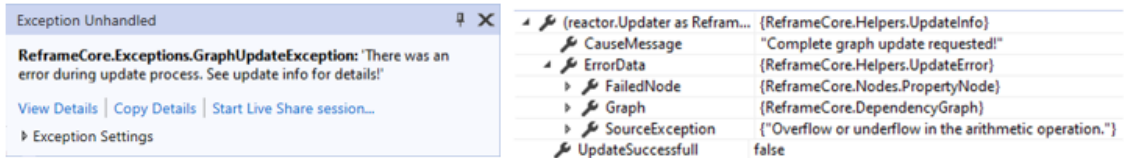
■ **Main scenario rollout:** *PerformUpdate* method is called on fetched reactor, and all of the 26 reactive nodes contained in dependency graph are updated synchronously.

```
var reactor = ReactorRegistry.Instance.GetReactor("default");
reactor.PerformUpdate();
```

The order in which reactive nodes are updated is following:



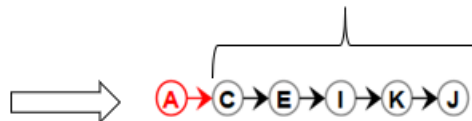
■ **Alternative scenarios:** Given a problem occurs during update process (e.g. exception in application code), framework will throw a *GraphUpdateException*. Additional update process and error information is available through *UpdateInfo* and *UpdateError* objects.



Scenario 5

- **Title:** Performing update process as a result of triggering change in object's state.
- **Prerequisites:**
 - Import following components in end-user application: *ReframeCore.dll*, *Reframe-BaseExceptions.dll* and *ReframeFluentAPI.dll*.
 - Existing reactor containing dependency graph shown in Figure 55 is fetched from *ReactorRegistry*.
- **Main scenario rollout:** Using fetched reactor, a change is triggered in a setter of property *A* (class *Alfa*) using *reactor.Update* method. This resulted in only few of the graph's nodes being updated.

```
public int A
{
    get { return _a; }
    set
    {
        _a = value;
        reactor.Update(this);
    }
}
```



- **Alternative scenarios:** In case of *MethodNode*, update process can be started as a result of invoking a method. Following example shows how we a reactive dependency between method node representing *DetermineCalculationProcedure* method (predecessor) and property node representing *P* property is created. Update process is started at the end of the method.

```
reactor
    .Let(() => P)
    .DependOn(
        () => PerformCalculation()
    );

public void PerformCalculation()
{
    //Method logic...
    reactor.Update(this);
}
```

Scenario 6

- **Title:** Performing update process asynchronously.
- **Prerequisites:**

- Import following components in end-user application: *ReframeCore.dll*, *Reframe-BaseExceptions.dll* and *ReframeFluentAPI.dll*.
 - Existing reactor is fetched from *ReactorRegistry*.
- **Main scenario rollout:** In the reactor's updater object, the *UpdateStrategy* is set to *Asynchronous*. Now, calling the *PerformUpdate* method results in update process being executed in separate thread, thus preventing GUI thread to be blocked.


```
var reactor = ReactorRegistry.Instance.GetReactor("default");
(reactor.Updater as Updater).Strategy = UpdateStrategy.Asynchronous;
await reactor.PerformUpdate();
```
 - **Alternative scenarios:** As in Scenario 4, in case of a problem during update process, framework will throw a *GraphUpdateException*. Additional update process and error information is available through *UpdateInfo* and *UpdateError* objects.

Scenario 7

- **Title:** Performing update process in parallel.
- **Prerequisites:**
 - Import following components in end-user application: *ReframeCore.dll*, *Reframe-BaseExceptions.dll* and *ReframeFluentAPI.dll*.
 - Existing reactor is fetched from *ReactorRegistry*.
- **Main scenario rollout:** In the reactor's updater object, the *UpdateStrategy* is set to *Parallel*. Now, calling the *PerformUpdate* method results in update process which forms layers of independent reactive nodes. Layers are updated one at a time, while reactive nodes within the layer are updated in parallel.


```
var reactor = ReactorRegistry.Instance.GetReactor("default");
(reactor.Updater as Updater).Strategy = UpdateStrategy.Parallel;
await reactor.PerformUpdate();
```
- **Alternative scenarios:** As in Scenario 4, in case of a problem during update process, framework will throw a *GraphUpdateException*. Additional update process and error information is available through *UpdateInfo* and *UpdateError* objects.

Scenario 8

- **Title:** Reporting the progress and status of the ongoing update process.
- **Prerequisites:**

- Import following components in end-user application: *ReframeCore.dll*, *Reframe-BaseExceptions.dll* and *ReframeFluentAPI.dll*.
 - Existing reactor is fetched from *ReactorRegistry*.
- **Main scenario rollout:** End-application code subscribes to events *UpdateStarted*, *UpdateCompleted*, and *UpdateFailed*.

```
var reactor = ReactorRegistry.Instance.CreateReactor("default");
reactor.UpdateStarted += Reactor_UpdateStarted;
reactor.UpdateCompleted += Reactor_UpdateCompleted;
reactor.UpdateFailed += Reactor_UpdateFailed;
```

Event handler methods are defined which contain arbitrary code handling the events (e.g. informing user, refreshing GUI, etc.)

```
private void Reactor_UpdateStarted(object sender, EventArgs e)
{
    var graph = (sender as IUpdater).Graph;
    MessageBox.Show($"Update process for graph {graph.Identifier} has started!");
}

private void Reactor_UpdateFailed(object sender, EventArgs e)
{
    var error = sender as UpdateError;
    var graph = error.Graph;
    var failedNode = error.FailedNode;
    MessageBox.Show($"There was an error in node {failedNode.Identifier} " +
        $"during update process of graph {graph.Identifier}");
}

private void Reactor_UpdateCompleted(object sender, EventArgs e)
{
    RefreshGUI();
}
```

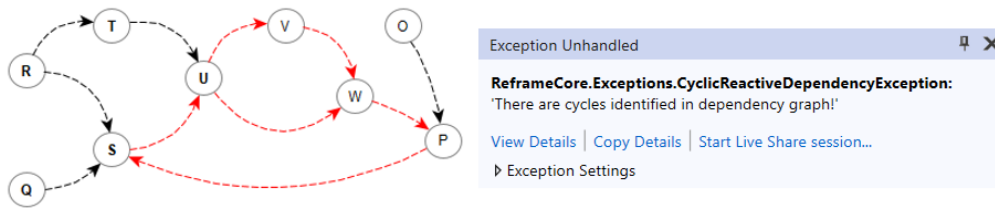
- **Alternative scenarios:** In case of asynchronous or parallel update process, cross-thread calls need to be prevented when, for example, refreshing GUI.

```
private void Reactor_UpdateCompleted(object sender, EventArgs e)
{
    this.Invoke(new Action(() => RefreshGUI()));
}
```

Scenario 9

- **Title:** Informing about cycle formed in dependency graph.
- **Prerequisites:**
 - Import following components in end-user application: *ReframeCore.dll*, *Reframe-BaseExceptions.dll* and *ReframeFluentAPI.dll*.
 - Existing reactor is fetched from *ReactorRegistry*.
- **Main scenario rollout:** Update process was started on a graph which contains cycles.

This resulted in *CyclicReactiveDependencyException*.



■ **Alternative scenarios:** -

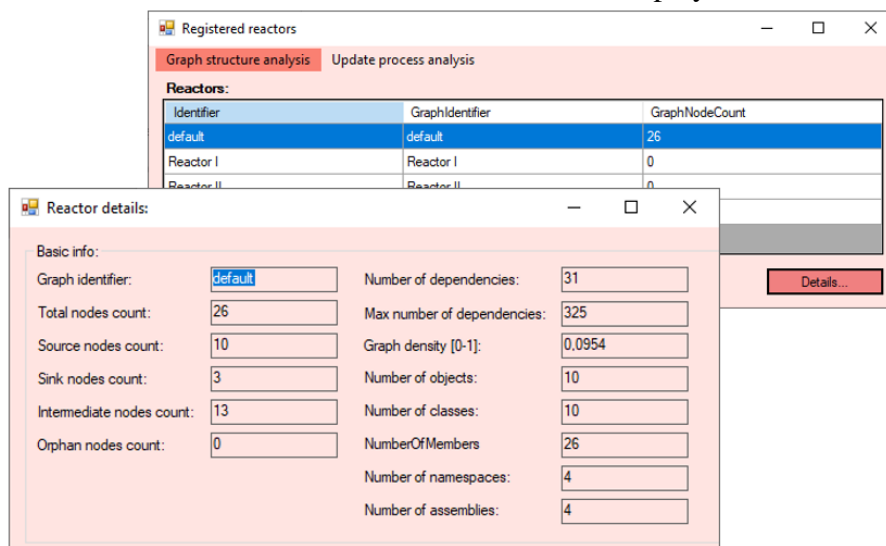
Scenario 10

■ **Title:** Displaying the list of registered reactors.

■ **Prerequisites:**

- Import following components in end-user application: *ReframeCore.dll*, *Reframe-BaseExceptions.dll*, *ReframeFluentAPI.dll*, *ReframeExporter.dll*, *ReframeServer.dll*, and *IPCServer.dll*.
- End-user application with reactors registered in *ReactorRegistry* is up and running.
- *ReframePipeServer* is started in end-user application.

■ **Main scenario rollout:**REFRAME Tools are invoked, and a graphical user interface appeared showing the list of registered reactors in end-user application. For each registered reactor a detailed data about its structure can be displayed.



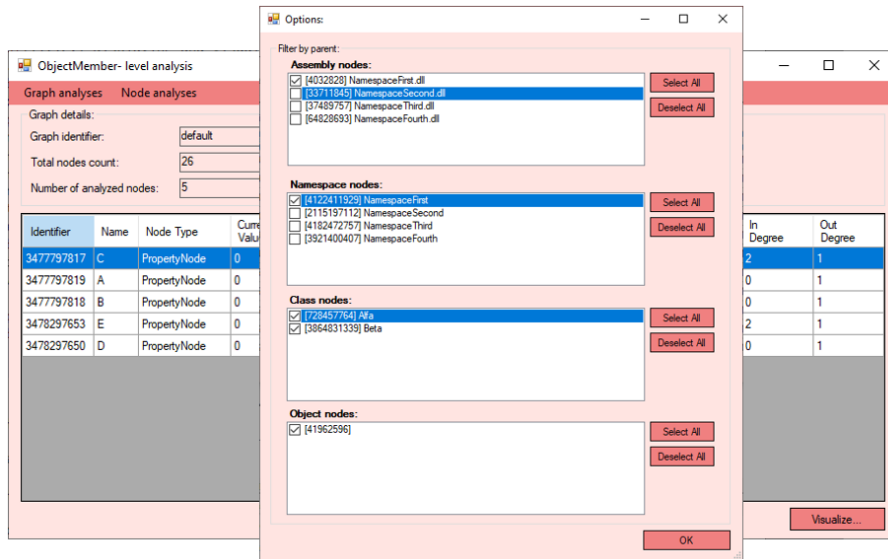
■ **Alternative scenarios:** If no end-user application is running with *ReframePipeServer* started, REFRAME tools inform user they are unable to fetch reactors from end-user application.

Scenario 11

- **Title:** Displaying the list of reactive nodes at different levels of abstraction.
- **Prerequisites:**
 - Import following components in end-user application: *ReframeCore.dll*, *Reframe-BaseExceptions.dll*, *ReframeFluentAPI.dll*, *ReframeExporter.dll*, *ReframeServer.dll*, and *IPCServer.dll*.
 - End-user application with reactors registered in *ReactorRegistry* is up and running.
 - *ReframePipeServer* is started in End-user application.
- **Main scenario rollout:** After selecting registered reactor, available menu offers six levels at which reactive nodes can be shown, e.g. *object-member* level. List of nodes are then displayed in a table with data available at chosen level of abstraction.

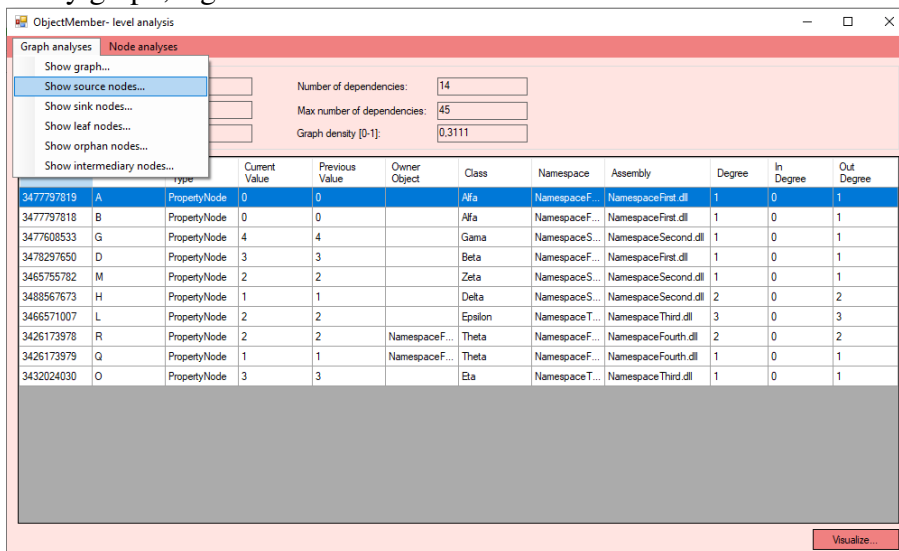
Identifier	Name	Node Type	Current Value	Previous Value	Owner Object	Class	Namespace	Assembly	Degree	In Degree	Out Degree
3477797817	C	PropertyNode	0	0		Afa	Namespace...	Namespace...	3	2	1
3477797819	A	PropertyNode	0	0		Afa	Namespace...	Namespace...	1	0	1
3477797818	B	PropertyNode	0	0		Afa	Namespace...	Namespace...	1	0	1
3477608534	F	PropertyNode	1	0		Gama	NamespaceSecond	Namespace...	3	1	2
3477608533	G	PropertyNode	0	0		Gama	Namespace...	Namespace...	1	0	1
3478297653	E	PropertyNode	0	0		Beta	Namespace...	Namespace...	3	2	1
3478297650	D	PropertyNode	0	0		Beta	Namespace...	Namespace...	1	0	1
3465755783	N	PropertyNode	1	0		Zeta	Namespace...	Namespace...	3	2	1
3465755782	M	PropertyNode	0	0		Zeta	Namespace...	Namespace...	1	0	1
3488567670	I	PropertyNode	0	0		Delta	Namespace...	Namespace...	4	2	2
3488567673	H	PropertyNode	0	0		Delta	Namespace...	Namespace...	2	0	2
3488567671	L	PropertyNode	2	0		Delta	Namespace...	Namespace...	3	3	0

- **Alternative scenarios:** Prior to displaying list of reactive nodes, it is possible to filter them with regard to their affiliation with higher-level nodes (*filtering by affiliation*). For example, we can decide to display only nodes coming from certain assembly.



Scenario 12

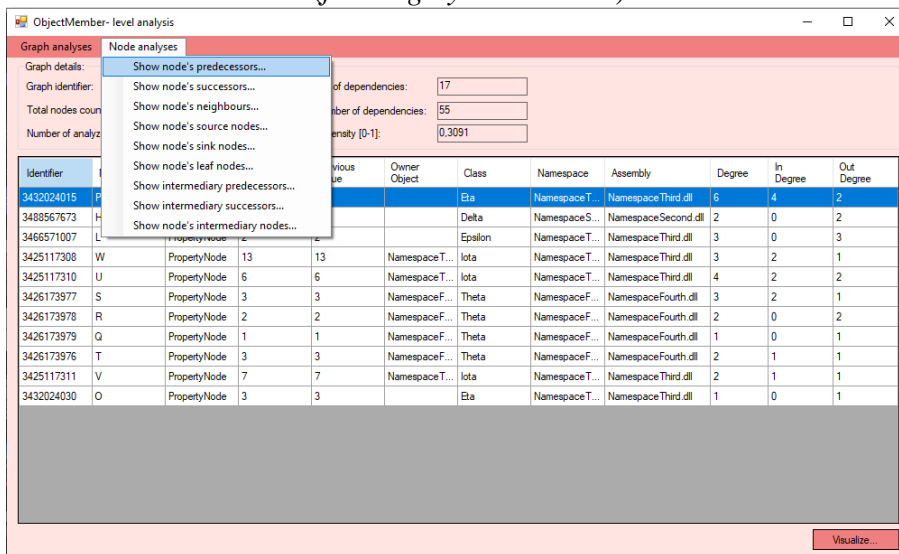
- **Title:** Displaying reactive nodes with different roles in dependency graph.
- **Prerequisites:**
 - Import following components in end-user application: *ReframeCore.dll*, *Reframe-BaseExceptions.dll*, *ReframeFluentAPI.dll*, *ReframeExporter.dll*, *ReframeServer.dll*, and *IPCServer.dll*.
 - End-user application with reactors registered in *ReactorRegistry* is up and running.
 - *ReframePipeServer* is started in End-user application.
- **Main scenario rollout:** After selecting registered reactor and a level of abstraction, it is possible to show only reactive nodes with particular role (*filtering by role*) in a dependency graph, e.g. *source nodes*.



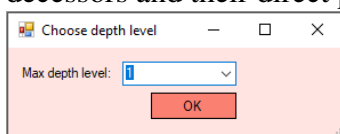
- **Alternative scenarios:** Prior to displaying list of reactive nodes, it is possible to filter them with regard to their affiliation with higher-level nodes (*filter by affiliation*).

Scenario 13

- **Title:** Displaying reactive nodes associated with selected node.
- **Prerequisites:**
 - Import following components in end-user application: *ReframeCore.dll*, *Reframe-BaseExceptions.dll*, *ReframeFluentAPI.dll*, *ReframeExporter.dll*, *ReframeServer.dll*, and *IPCServer.dll*.
 - End-user application with reactors registered in *ReactorRegistry* is up and running.
 - *ReframePipeServer* is started in End-user application.
- **Main scenario rollout:** After any list of reactive nodes is displayed, and any reactive node is selected, it is possible to display reactive nodes which are in various ways associated with selected node (*filtering by association*).



- **Alternative scenarios:** Prior to displaying list of reactive nodes, it is possible to filter them with regard to their affiliation with higher-level nodes (*filter by affiliation*). Also, when displaying selected node's predecessors, successors and neighbors, it is possible to choose the depth level. For example, when showing predecessors, level 1 would show only selected node's direct predecessors; level 2 would show selected node's direct predecessors and their direct predecessors, and so forth.



Scenario 14

- **Title:** Displaying information about last performed update process.
- **Prerequisites:**
 - Import following components in end-user application: *ReframeCore.dll*, *Reframe-BaseExceptions.dll*, *ReframeFluentAPI.dll*, *ReframeExporter.dll*, *ReframeServer.dll*, and *IPCServer.dll*.
 - End-user application with reactors registered in *ReactorRegistry* is up and running.
 - *ReframePipeServer* is started in End-user application.
- **Main scenario rollout:** After selecting registered reactor we can show information about the latest update process. Here we can see information about the overall update process, such as update status, cause, duration, error, etc. Also, we can see the list of all reactive nodes which participated in the update process. In addition to usual information about reactive nodes, for each node we can see the duration of update, as well as whether the update resulted in a value change.

The screenshot shows the 'Registered reactors' application with the 'Update process information' dialog box open. The dialog displays the following information:

Basic update process information

Graph identifier: default Started at: 11:24:36.282 Update cause: Update caused by triggering individual
 Total node count: 25 Ended at: 11:24:36.283 Initial node identifier: 3477797818
 Updated nodes count: 6 Duration: 00:00:00.0009994 Initial node member: B
 Displayed nodes count: 6 Initial node owner: Alfa
 Update successful: True Current value: 3
 Previous value: 0

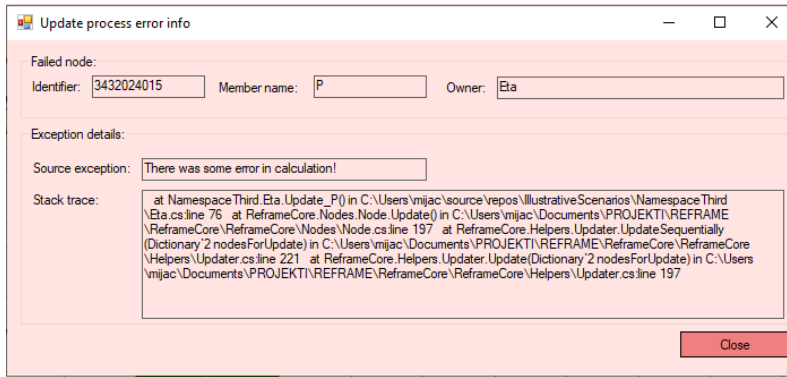
Filter:
 Show all nodes Show only nodes with differences Show only nodes with NO differences

Updated nodes:

Identifier	Name	Node Type	Current Value	Previous Value	Update Order	Update Layer	Started At	Completed At	Duration	Degree	In Degree	Out Degree
3477797818	B	PropertyNo...	3	0	0	4	10:26:19.862	10:26:19.862	00:00:00.0...	1	0	1
3477797817	C	PropertyNo...	4	1	1	3	11:24:36.283	11:24:36.283	00:00:00.0...	2	1	1
3478297653	E	PropertyNo...	7	4	2	2	11:24:36.283	11:24:36.283	00:00:00.0...	2	1	1
3488567670	I	PropertyNo...	8	5	3	1	11:24:36.283	11:24:36.283	00:00:00.0...	3	1	2
3466571006	K	PropertyNo...	10	7	4	0	11:24:36.283	11:24:36.283	00:00:00.0...	1	1	0
3488567671	J	PropertyNo...	20	17	5	0	11:24:36.283	11:24:36.283	00:00:00.0...	1	1	0

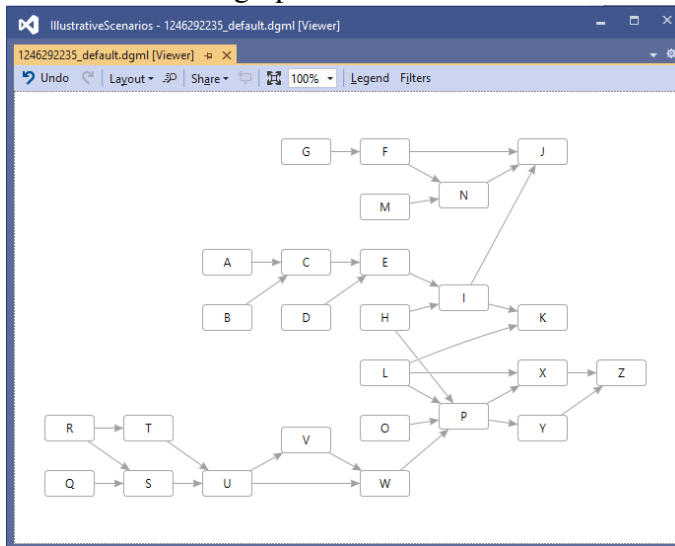
Visualize...

- **Alternative scenarios:** In case of failed update process, user can obtain error information.

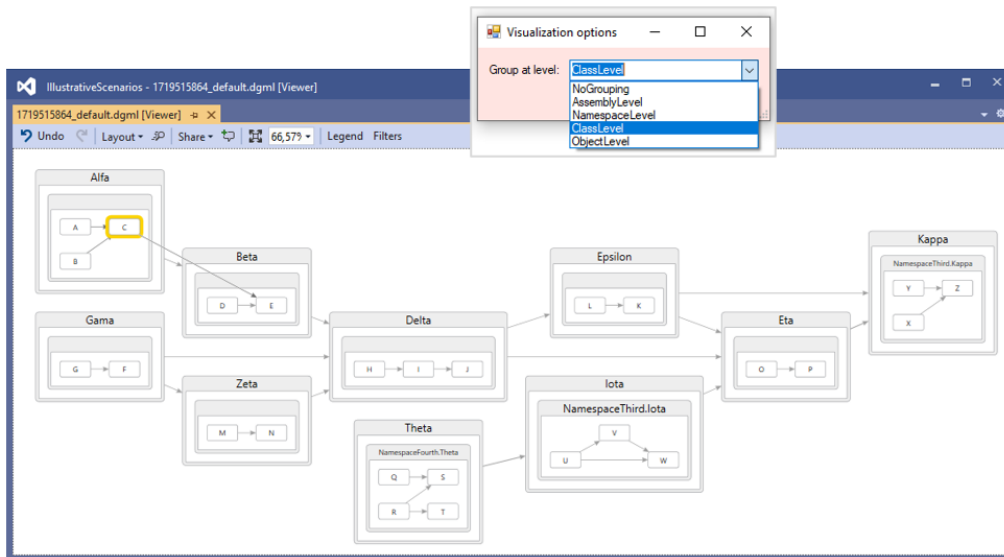


Scenario 15

- **Title:** Visualizing the list of reactive nodes.
- **Prerequisites:**
 - Import following components in end-user application: *ReframeCore.dll*, *Reframe-BaseExceptions.dll*, *ReframeFluentAPI.dll*, *ReframeExporter.dll*, *ReframeServer.dll*, and *IPCServer.dll*.
 - End-user application with reactors registered in *ReactorRegistry* is up and running.
 - *ReframePipeServer* is started in End-user application.
- **Main scenario rollout:** After displaying any list of reactive nodes we can visualize it in a form of DGML graph shown in Visual Studio's DGML Viewer.



- **Alternative scenarios:** Prior to displaying the graph, we can choose whether we want to group displayed nodes at certain level or not.



7.3. Episode III - Focus group

Focus group is one of the most popular qualitative methods, which tries to cover particular topic in even greater depth than individual interviews by allowing multiple participants to interact and discuss. Within design science, focus group is frequently used in various stages of research, including problem explication, requirement specification, and artifact evaluation [78]. Similarly, the value of focus group in requirements elicitation and evaluation in the field of software engineering is also seen by Kontio et al. [89]. Tremblay et al. [150] adapted the traditional focus group method for the use in design science projects. According to them, this method can be used to achieve two goals: (1) artifact improvement - in this case we talk about *explorative focus group*, and (2) artifact evaluation - a *confirmatory focus group*.

By being part of evaluation activity within design science project, i.e. one of the summative evaluation episodes, our focus group can be characterized as confirmatory. Our aim is to perform evaluation of the framework's usefulness by first time stepping outside of the laboratory setting, and showing the framework to outside users. This introduces small, albeit important step towards evaluation in realistic conditions in practice. In addition, focus group is expected to result in valuable ideas for future improvements and extensions.

In the remainder of this section, we describe the process of conducting confirmatory focus group by following the guidelines offered in [150] and [89].

7.3.1. Research problem

REFRAME software framework (new design science artifact) has been designed, developed, tested and demonstrated on illustrative scenarios as a potential solution for management of reactive dependencies. However, there is still no proof that REFRAME will be perceived as useful by developers. Therefore, by conducting focus group we aim to evaluate framework's usefulness and in this way answer research question **RQ6**: *How does the use of REFRAME affect the management of reactive dependencies in development of object-oriented applications?* This aligns with the general goal of confirmatory focus groups, i.e. demonstrating the utility of artifact design in the application field.

7.3.2. Sample frame

According to Tremblay et al. [150], in this step it is necessary to define (1) how many focus group sessions to conduct, (2) how many participants will focus group have, and (3) what type of participants are to be recruited. With regard to the number of focus group sessions, as a part of the third evaluation episode we will conduct one focus group session. It is important to point out that Tremblay et al. [150] do suggest conducting at least two focus groups, as only one may be insufficient as a sole argument in favor of artifact's value. However, since this is only one out of four evaluation episodes, and the usefulness is also assessed in the next evaluation episode, we argue one focus group to be sufficient in this particular case.

As per the number of participants in focus group session, Kontio et al. [89] report this to vary between 3 to 12 participants. Tremblay et al. [150] emphasize that having more than 6 participants in focus group conducted within design science research may be tricky, as the topic is often more complex than in traditional focus groups. We will follow that advice and keep the number of participants at 6 maximum. Thus, total of 10 potential participants were invited to participate in focus group, out of which 6 initially accepted and 4 declined the invitation. This was satisfactory response, as we did not need to reject or put on hold any of the applicants to be within the limit of 6 participants. Also, we felt relatively safe in the event of one or two applicants drop out at the last moment. Unfortunately, this is exactly what happened, as at the last moment two of the applicants had to drop out due to sudden but urgent family and work-related matters. This left us with still satisfactory number of 4 participants.

The end users of REFRAME framework are software developers, so potential focus group participants included individuals from both academic community and practice which are actively involved in the processes of software applications development. The general prerequisites for participation included relevant knowledge and experience in the field of software engineering and object-oriented programming, and at least some experience in .NET or Java technology. Each of the 4 applicants satisfy these requirements by having a formal education (master degrees) in the field of informatics, where they successfully completed several courses related to software development. After their formal education, participants also gained between 3 and 8 years of relevant experience. Another, more specific requirement, was for participants to be acquainted with the problem of reactive dependencies, which is addressed by REFRAME. With regard to this, two participants had large experience as they were involved in development of

KI Expert Plus software for a significant time (more than 6 months). The remaining two participants had no experience with such extreme examples of managing reactive dependencies, but were acquainted with problem in terms of related design patterns (e.g. Observer pattern), or their language specific implementations (e.g .NET and Java event mechanisms).

Table 15: Focus group participants

Participant	Qualifications
Participant 1	He holds a master degree in Information and Software Engineering at Faculty of Organization and Informatics. During his master degree studies he was involved in different software development projects, including the project of developing KI Expert Plus software application. He currently holds the position of Hybris Java Developer in ecx.io Croatia d.o.o. where he works for the past five years.
Participant 2	He holds a master degree in Information and Software Engineering at Faculty of Organization and Informatics. For the past eight years he was working as a software developer for IN2 d.o.o. on the project of developing software for magistrates court. As of recently he is working for Omega Software d.o.o. as a backend developer and BizTalk specialist. He was also involved in development of KI Expert Plus application.
Participant 3	He holds a master degree in Information and Software Engineering at Faculty of Organization and Informatics. For two years he worked as a teaching assistant at Faculty of Organization and Informatics (Department of Information Systems Development). During that time he worked as a project associate on IRI Hyper - User Experience of the Future project, where he was developing both frontend and backend part of the chatbot system. Currently he holds a position of middle Java developer at Ingemark d.o.o.
Participant 4	He holds a master degree at Business Systems Organization at Faculty of Organization and Informatics. For the past three years he worked in King ICT d.o.o. and Ars Futura d.o.o. software companies on the jobs related to automated testing and development using wide range of technologies. He currently works at Ars Futura d.o.o., and holds the nominal position of QA/DevOps Engineer, but his usual activities also involve development using Ruby on Rails.

7.3.3. Moderator

Focus group moderator should necessarily have a deep understanding of the discussed topic, including both the research problem, as well as proposed solution. REFRAME as a framework,

is full of technical details related to chosen technologies, design and implementation decisions. Unfortunately, at this stage, other than the researcher himself there is no other person who is sufficiently acquainted with the framework. This makes the researcher the best candidate for moderating the focus group, despite the threat of possibly introducing personal bias. According to Tremblay et al. [150], the moderator should also have a wide range of interpersonal and communication skills, ranging from ability to communicate clearly and include others into discussion, to having sense of humor. While it is hard for anyone to claim having such traits, the researcher's usual activities of researching, collaborating and teaching, also demand a lot of these traits to be demonstrated. That being said, the researcher will take the role of the moderator.

7.3.4. Questioning route

Planning of the focus group involved creating ordered checklist of tasks to be performed. The very focus group session will start with some administrative and technical details related to focus group. This includes reminding participants about the focus group goal, their voluntary participation, the fact that the session will be recorded for the purpose of data analysis, and finally the fact that their names will not be contained anywhere in the report. Also, at this point the participants will briefly introduce themselves.

The first major task in conducting the focus group is to ensure participants are adequately familiarized with the research problem and the proposed solution. While, as we already pointed out, some participants do have practical knowledge of the underlying problem, with other participants this is more of a theoretical knowledge. In addition, because of REFRAME being a novel artifact, none of the participants are familiarized with its features and design. This is why it is necessary to educate the participants prior to starting the discussion part of the focus group. The education will be conducted in two parts, and it involves presenting the original research problem of managing reactive dependencies, as well as the user features and design aspects of the REFRAME as proposed solution. Firstly, in a week before focus group is held, participants are going to receive 20-minute video lecture which should introduce them to the research problem of managing reactive dependencies. The second part of education will be done after initial introduction within the focus group session, in which the first 45-minutes will be reserved for a presentation on REFRAME as a proposed solution.

After the presentation, a discussion will take place moderated by the researcher. Since the focus group has a clear and pragmatic goal, i.e. assessment of REFRAME usefulness, the discussion will be guided with the help of previously defined set of questions (see Table 16). These questions reflect the problems and goals of the design artifact seen through the lenses of the research goal and research question addressed in this evaluation episode. However, they primarily serve as conversation starters and motivation mechanisms, and not to prevent an open discussion. In terms of steering the discussion, the researcher will intervene only in case of prolonged digression from discussion goal, or to motivate inactive participants. This, however, will be done in a discrete and subtle manner in order to not offend or discourage the participants.

Table 16: Focus group questions and topics

#	Question (topic)
1.	How do you perceive presented problem related to managing reactive dependencies? Do you see it as a relevant for practice and worth addressing?
2.	Can you think of any particular software systems (or types of software systems) which manifest this problem (e.g., spreadsheets, GUI, animations, games, reactive systems, event-driven systems)?
3.	At what point managing reactive dependencies becomes a problem in OO? What contributes most to the problem?
4.	What are the most serious symptoms/effects of this problem?
5.	Can you imagine how would the ideal solution for managing reactive dependencies in OO look like? What features and characteristics would it have?
6.	Do you find frameworks to be a suitable artifact type for this kind of a problem? Is there a better type of software artifact for this (specific programming language, domain specific language, design pattern, architecture, libraries, components, tool)?
7.	Are concepts from graph theory (node, edges, acyclic directed graph) suitable for expressing interrelated reactive dependencies? Can you think of any suitable alternative representations?
8.	What do you think about the way how reactive dependencies are specified in REFRAME? Is the Fluent API syntax in a form of Let->Depend straightforward enough?
9.	Are code snippets appropriate tool for generating parts of boilerplate code? Can you see any benefits? Can you think of any other way to support specifying reactive dependencies, either through the code or visually?
10.	Do you think visual representation in a form of graph can increase the ability to comprehend and understand the relationships that reactive dependencies form? Can you think of an example where application developer or an end-user would benefit from this kind of visualization? Can you think of any alternative which would increase comprehension and understandability of reactive dependencies?

Table Table 16 – Focus group questions and topics

#	Question (topic)
11.	What about the issue of graph size in terms of e.g. visualization? At the original level of abstraction dependency graph could potentially hold thousands of nodes. Tools usually struggle to show such number of elements. But even if we ignore this fact, how does a human user benefit from being presented with such large structure? How to solve this problem?
12.	Do you think REFRAME Analysis tool provides enough capabilities for a user to reduce the number of displayed nodes and dependencies, and focus on the things of interest at particular moment (horizontal/vertical reduction).
13.	Is there any existing feature/characteristic of the core part of the framework, or its accompanied tools (Visualizer, Analyzer, Generator) that you find (1) particularly useful; (2) is missing and should be introduced; or (3) should be removed or significantly altered.
14.	How extensible and customizable do you find the framework? What do you think are the most important and probable points of extension and customization in REFRAME?
15.	Do you think underlying framework model can be utilized for alternative implementation in some other programming language?

After the official time for discussion runs out, the researcher will wait for appropriate moment and inform the participants. The participants will then be asked to directly assess the usefulness of REFRAME using a 7-point Likert's measurement scale (see Figure 56) for *perceived usefulness* provided by Davis in his popular Technology acceptance model (TAM) [41]. Since the scale itself contains concepts such as *performance*, *productivity*, *effectiveness* and *useful*, which may be differently understood by the participants, explanations for each of these concepts were also provided. Finally, upon finishing the questionnaire, the researcher will kindly thank participants for their time and effort, and inform them they are free to go.

7.3.5. Focus group session

Originally, the focus group was intended to be held in person in one of the faculty's laboratories. However, with regard to the situation related to Coronavirus (SARS-CoV-2) outbreak and the introduced epidemiological measures, it was decided to conduct focus group as an online meeting using Skype platform. All of the participants are frequently using such platforms for business meetings and collaboration, so this was not a new thing for them.

The focus group started at the appointed time, and all of the four applicants showed up. At

1. Using REFRAME would enable me to manage reactive dependencies more quickly .						
Strongly agree	Agree	Somewhat agree	Neither agree nor disagree	Somewhat disagree	Disagree	Strongly disagree
2. Using REFRAME would improve my performance in managing reactive dependencies.						
Strongly agree	Agree	Somewhat agree	Neither agree nor disagree	Somewhat disagree	Disagree	Strongly disagree
3. Using REFRAME would increase my productivity in managing reactive dependencies.						
Strongly agree	Agree	Somewhat agree	Neither agree nor disagree	Somewhat disagree	Disagree	Strongly disagree
4. Using REFRAME would enhance my effectiveness in managing reactive dependencies.						
Strongly agree	Agree	Somewhat agree	Neither agree nor disagree	Somewhat disagree	Disagree	Strongly disagree
5. Using REFRAME would make it easier to manage reactive dependencies.						
Strongly agree	Agree	Somewhat agree	Neither agree nor disagree	Somewhat disagree	Disagree	Strongly disagree
6. I find REFRAME would be useful for managing reactive dependencies.						
Strongly agree	Agree	Somewhat agree	Neither agree nor disagree	Somewhat disagree	Disagree	Strongly disagree

Figure 56: Measurement scale for perceived usefulness (adapted from [41])

the beginning the moderator (researcher) greeted the participants, ensured that everyone’s audio and video equipment is working, and reminded participants that their participation is voluntary and that the session will be recorded for the sole purpose of transcribing the discussion. After that, the participants were introduced to focus group goal, and were briefly presented with the overall agenda.

This was followed by 45 minute presentation on REFRAME as a proposed solution, in which the moderator covered essential aspects of both use and design/implementation of the framework. Participants had no questions either during or after presentation, and they explicitly confirmed that the presentation was clear and understandable.

After that, the moderator opened another presentation with the questions intended to drive the discussion. During this part, the moderator first called out the individual participants as a signal for him speak. However, soon this was not needed anymore, as it became evident that participants have experience in online meetings and their technical limitations, and they spontaneously discussed the topic one at the time. The moderator only called out the participant in a discrete and subtle way, if the participant did not provide his opinion by himself. The overall atmosphere was relaxed from the beginning, and there was no sign of hesitation from the participants’ part. The discussion lasted total of 90 minutes, after which the moderator thanked the participants, and kindly asked them to fill-out a simple questionnaire. The entire

focus group session went smoothly, with no communication or technical issues.

7.3.6. Data analysis

In order to perform data analysis, we used *template analysis* method, which was proposed by King [87] for thematically analyzing qualitative data collected through interviews and focus groups. The idea is to develop a coding template which represents a summary of meaningfully organized themes that were identified as important by the researcher. This template is then used to interpret qualitative data obtained by the focus group. Template analysis method prescribes following steps: (1) Define *a priori* themes and codes, (2) transcribe focus group, (3) carry out initial coding, (4) produce initial template, (5) develop your template, and (6) interpret data set according to produced template.

As prescribed by template analysis method, we started by defining a small set of *a priori* themes (**step 1**), which in our case represented high level categories of possible perceptions that could be related to REFRAME *usefulness*. This initial set consisted of following themes: (1) *problem relevance*, (2) *positive impression*, (3) *negative impression* and (4) *framework evolution*. The *problem relevance* intends to show perception of participants related to whether the problem addressed by REFRAME is relevant or not. Surely, even the most technically advanced and perfected software systems cannot boast about their usefulness if they address trivial, or non-existent problem. The *positive impression* is high level theme which intends to encompass all perceptions which view the REFRAME in a positive light, and thus possibly characterize it as useful. In a similar way *negative impression* does this for all negative perceptions which can possibly affect framework usefulness in an adverse manner. *Negative impression* theme may represent also suggestions for performing *corrective* and *perfective* activities on framework, such as improving the ease of use of existing features. *Framework evolution* theme, however, represents those comments which contain *adaptive* suggestions for improvements which were outside of the scope of initially proposed set of requirements, or features which were simply too extensive to be covered in this phase of framework development. Not only these themes do not necessarily bear negative connotation with regard to framework usefulness, but they often indicate the capability of framework to be extended and be useful in additional ways. Finally, both *negative impression* and *framework evolution* themes hold valuable ideas for overall improvement of the framework.

After defining *a priori* themes, we proceed to transcribe the audio/video recording of the discussion held in focus group (**step 2**). At first we considered using transcription tools, however, in the end we performed transcription process manually. This was primarily because the discussion was not especially long (90 minutes), and even the automatic transcription would require re-watching entire discussion and proof-checking the generated content. It was our estimation that the entire process would last roughly the same amount of time. A result of transcription was 10-pages of textual content ready for analysis.

Table 17: Final version of template

#	Theme
1.	Problem relevance
1.1.	Problem presence
1.2.	Problem complexity
2.	Positive impression
2.1.	General positive remark
2.2.	Possible use case
2.3.	Useful characteristic/feature
3.	Negative impression
3.1.	Feature needs improvement
3.2.	Feature needs to be added
4.	Framework evolution

The coding process (**steps 3 to 5**) involved several iterations of assigning theme codes, re-reading transcription, and modifying codes. During initial coding, high-level *a priori* themes were confirmed as appropriate, and entered initial template. After that, *problem relevance*, *positive impression* and *negative impression* themes from initial template became theme categories, with their own lower level themes. It required several iterations of inserting, removing and changing these lower level themes to finally have the final version of the template. This template is then used for data interpretation and writing up the findings (**step 6**). These can be found in the next subsection.

7.3.7. Results

Problem relevance

In order to infer about the problem relevance, we first searched for parts of conversation in which participants showed whether they perceive the problem of managing reactive dependencies as a real problem present in practice (**problem presence**). The two participants which had an experience with working on KI Expert Plus application development immediately recognized this software application as a clear example of the problem. *"KI Expert Plus is quite specific because, you know, like 90% of things revolve around: you enter some data - some other data changes."* Besides that, participants did not experience the problem of managing reactive dependencies at that scale. They tried, however, to think of possible domains in which that problem could be present. For example, one participant made following observation: *"Yes, we had KI Expert Plus in which we had a lot of calculation procedures, however there is whole lot of other domains rich with calculation procedures which could be implemented."* In this context, *bookkeeping* and *student record keeping* often done in spreadsheet software were mentioned as possible domains with reactive dependencies. Other examples given by participants described dependencies between business processes, microservices, data import, but we did not qualify them as reactive dependencies.

Participants in general perceived the problem of managing reactive dependencies as quite complex problem (**problem complexity**). They were also sure they never encountered as complex example as the one presented in KI Expert Plus application. *"I'm trying to remember whether I needed something like that... But they were all simpler dependencies, which we managed to handle using events, listeners..."; "I'm thinking, but, I too did not have a problem with such extensive dependency graph"*. The participants also discussed underlying reasons why handling reactive dependencies might be hard, and listed the very size and complexity of dependency graph to be decisive factors. *"I think if you accidentally fail to specify some dependency in large graph it would be nearly impossible to find an error. You can debug application for a three days until you finally find you are missing one dependency and this is why you are getting wrong results."; "I think it depends on graph complexity. To certain degree you can handle it manually, however after some point it becomes too large and then... you struggle."*

Positive impression

Throughout discussion, participants frequently expressed positive impressions about the REFRAME framework in a various way. Some of these impressions were general positive statements in which participants pointed out that the framework would be useful in software systems such as KI Expert Plus, or simply that they find some aspect of the framework as fine and have no objections (**general positive remark**). *"From the perspective of KI Expert Plus I see it as a real lifesaver."; "... well, for KI Expert Plus it would really mean a lot."*

Participants also acknowledged a number of specific use cases in which the framework could be useful (**possible use cases**). For example, because it is a framework, and not a special-purpose programming language, REFRAME was perceived as good for use in both new and legacy systems. *"I think a framework is a good way to go... it is easier to fit it into existing project."; "... you can use it to refactor legacy solutions, but also to create new solutions from scratch."* Since the underlying model used in REFRAME is based on directed acyclic graph (DAG), participants recognized potential use cases in systems which naturally represent their dependencies in such way. They perceived REFRAME as useful in such cases, especially for overall understanding of dependencies in graph, performing analyses and visualizations, and debugging. *"In reality, when you only have class diagram, it often is not enough, a lot of details is left in developer's mind. Here (in REFRAME) graph tells you everything that happens under to hood. It is also readable by non-developers... even the "layman" can read it."* Another interesting idea that came up while discussing possible use cases was to utilize meta-data about reactive dependencies in terms of software testing, especially unit testing. Perhaps graph data can be used to generate parts of testing code, generate list of test cases, or analyze testing coverage. *"Well, that could be possible (use REFRAME with the purpose of testing). You know what are your inputs and outputs, and what to expect."* In addition to REFRAME as an instantiation artifact (framework implementation), the model artifact (framework design) was also perceived as useful for framework re-implementation in alternative programming languages. However, discussion also brought up the question whether some design decisions and mechanisms used in .NET implementation can be easily replicated in other languages. *"These diagrams (REFRAME model) are always useful. I for sure would not be trying to figure out all from scratch."; "Yes the diagrams would make things easier, however particular programming language (its characteristics and options) might make things harder."*

Finally, with regard to positive impressions, participants identified several specific characteristics and features of the framework as being useful (**useful characteristic/features**). The most praised feature of the framework, repeatedly mentioned in different parts of the discussion, was graph visualization. This does not come as a surprise, because as we previously showed, it was the very size and complexity of graph that participants perceived as decisive factor in making this problem hard to deal with. And visualizing complex structures is one of the best way to aid their understanding. *"In general, I find the graph suitable for visualization, its very clear.";* *"The graph is self-explanatory, you don't need to be an expert.";* *"In one of our company projects we visualized processes and steps of one of our entities in a similar way. It was the easiest way for both developers and the users.";* *"In my opinion this is one of the features that is very clear and necessary. If you want to see detailed dependencies, it is much more useful than associations between classes in UML class diagram."* Of course, graph visualization offered by REFRAME would not be possible if not for other, underlying features of the framework. For example, without core part of the framework, which allows developers to specify reactive dependencies, we would not even have anything to visualize. With regard to specifying reactive dependencies, one of the features participants acknowledged as useful was fluent syntax. *"Yes, it is simple and easy to understand.";* *"I like how you can put dependencies in one method, and have them in one place... I can quickly see how complex is that class in terms of reactive dependencies."* The participants also found code snippets as an appropriate way for generating the code in charge of specifying reactive dependencies. *"Well, doing stuff in GUI is slow. Whenever you need to use mouse and click on things, its really slow. And code snippets is a common thing in Visual Studio, it speeds things up."* While core part of the framework makes visualization possible, it is the filtering options offered by graph analysis features that increase the usefulness of graph visualization. From the discussion, it seemed that participants were very aware of that. *"To me it seems that (using graph analysis tool) we can reach all desired graph nodes and visualizations. At this moment I wouldn't say that any option is missing.";* *"Yes, there is a bunch of options, everything is covered.";* *"At every moment you can see graph state, how much nodes you have, and their values. It is easier to understand the results... before that, debugging was harder. In any case I did not expect values were going to be displayed in real time."* Due to being a framework, participants perceived REFRAME as an extensible and customizable software artifact, which is useful when you need to make adjustments to cover specific needs of

application domain. At the same time, the participants saw the framework as a more concrete and immediately applicable software artifact, unlike some more abstract techniques (such as design patterns). *"I see design patterns as something that is fairly abstract, there is a lot of implementation involved before you have something concrete from that... DSLs are good if you create the solution from scratch, and it is questionable how much customization you can do. But in this way, with framework, you are flexible."*; *"I think it is good that the framework has a lot of extension points, because you never what extensions are going to be needed."*

Negative impression

Besides positive comments about the framework usefulness, participants raised several concerns about certain existing features, as well as about features that are lacking. We argue, however, that these "negative impressions" do not diminish the usefulness of the framework in a significant way. In most cases these are minor issues which when resolved would improve the ease of use of the framework. Also, most of the features that participants initially requested to be added, were actually found to already be possible as a combination of existing features. This, in fact, stands as a testament of framework's extensibility and completeness.

With the respect to features that were identified as the ones needing improvement (**feature needs improvement**), we start with the core part of the framework. During the discussion, some participants were rethinking the way reactive dependencies are specified and update process triggered. For example, one participant raised the question whether it would be better to keep reactive dependency specifications separately from the code (e.g. in json files). *"... maybe it would be good to use some kind of configurations for specifying reactive dependencies, and keep them somewhere outside of the code in a form of annotations, json..."* While this proposal aims at making code generation easier, making this possible would be very hard as the reactive dependencies are formed at runtime, and are bound to particular runtime objects and values. Another participant made similar proposal by wondering whether the code for triggering updates (`reactor.Update(this)`) can be replaced by annotations. The problem again, is that annotations are static (bound to a class) features, and the annotation specification would not contain any less amount of code than the current way of triggering updates. The syntax simplification of `Let->Depend` command was also suggested. *"I don't know if there is a way to avoid using these brackets... with that we could perhaps avoid the need for code snippets."* This suggestion is being seriously considered and different ways of simplifying the syntax are being looked for,

however limitations of the host programming language currently do not allow this particular improvement. Finally, the suggestion came for better support for circular dependencies. *"With the framework you only aim at implementing dependencies such that there is an input and a final output. However, sometimes these go into circles... inputs are also outputs, e.g. in animations."* This is a valid point, as currently, circular dependencies are recognized by the framework, and the developer is only warned about them. One of the future, priority improvements will definitely go in direction of enabling framework to perform update even if circular dependencies exist.

In addition to suggested improvements in the context of core framework features, there were also some remarks with regard to framework tools. For example, some participants found the listing of reactive nodes in a table form (within Analyzer tool) as somewhat cumbersome and not very clear. *"Tables in Analyzer which show the list of nodes are not clear. They are good when you know exactly what you are looking for, but examining dependencies in this way is very hard."* Although the Visualizer tool is aimed at solving this problem, this is still a valid point, and improvements in this regard are planned in the future. Also, suggestion came to better integrate the Analyzer and Visualizer tool. *"One thing for improvement of user experience is that everything becomes the part of the same interactive user interface."* This is another valid point, however, as we previously mentioned, we still have not been able to find a good-quality open-source .NET component for graph display.

All of the features that were suggested to be added to a framework referred to options in Analyzer and Visualizer tools (**feature needs to be added**). For example, participants found following features necessary: automatic graph layouts, interactive graph with zoom in/out options, showing graph update paths, graph node coloring, and graph node grouping. As previously indicated, during discussion it turned out that all of these features (with the exception of graph coloring) were in fact already present, and the user simply needs to chose right combination of options available in Analyzer and Visualizer tools. However, further discussion revealed that while there is huge number of options and their combinations, users will usually be using only a small number of them. Therefore, a suggestion was placed to make these commonly used features more accessible by creating an options shortcut system in REFRAME tools. This system would offer: (1) predefined list of useful option combinations, (2) list of option combinations frequently used by the user, and (3) list of custom option combinations created by the

user. Along with graph coloring feature, options shortcut system was also added to a priority list of future developments.

Framework evolution

Framework evolution theme encompass suggestions for extending and upgrading the framework in a way which was outside of the planned scope of the dissertation. However, these suggestions hold valuable ideas for future developments. Most of the suggestions went in the direction of supporting multiple platforms and multiple programming languages. *"One of the limitations is support for only .NET platform."*; *"Core part of the framework, as well as Visualizer and Analyzer tools should be implemented in more programming languages and support more platforms."* While supporting other programming languages is currently not planned in a near future, there is a plan to implement a .NET Core version of the framework, which would make the framework cross-platform (Windows, Linux, MacOS).

One of the suggestions was also about support for debugging of reactive dependencies. *"I think it would be easier if we could analyze nodes also in terms of debugging..."*. This is an interesting suggestion, as some of the scientific efforts in the field of reactive programming are directed towards developing better debugging support. Also, if we recall from the previous chapter, one of the solutions for accessing runtime state of end-user application that we considered was debugger. Therefore, this suggestions is definitely the one to seriously consider dealing with in a near future, as it holds potential for both practical and scientific contributions. The next suggestion that also has a scientific potential is using UML diagrams as an alternative way to visualize reactive dependencies, or some aspect of them. In addition to analyzing existing UML diagrams and seeing which of them could be useful for representing characteristics of reactive dependencies, one possible research direction would be to develop UML profile for reactive dependencies.

The last two suggestions were related to utilizing framework capabilities in terms of generating tests and documentation for end-user application. *"... because you know what your inputs are, and also what you expect. So it would be good to have covered certain scenarios with unit tests."*; *"Perhaps i would not be bad if we could generate some part of documentation, because the framework knows methods in which the processing takes place, as well as attributes used."* Both of these suggestions are very interesting, and are taken into consideration for future development.

7.4. Episode IV - Technical action research

Observational research methods demand from researcher to refrain from interaction with the object under study, thus making the researcher a passive observer. As opposed to that, action research necessarily requires intervention in order to not only gain knowledge, but also to improve the state of the "world". This makes action research compatible with the problem-solving nature of the design science itself. Traditional action research is *problem-driven*, which means that the researcher teams with the client to solve a practical problem relevant to that client, as well as to devise new knowledge during that process. Technical action research (TAR), on the other hand is *artifact-driven*, i.e. it uses artifact to help a client with some particular problem, and during that process evaluates the artifact itself and learns about its effects in practice. Indeed, Wieringa and Morali [157] present TAR as one of the methods for evaluation of artifacts created through design science.

TAR can be seen as one of the last steps in the process of transferring the artifact from idealized conditions in laboratory to realistic conditions in practice [158]. This view is directly reflected in our evaluation strategy (see Figure 9), which starts with pure artificial evaluation episodes (I and II), continues with episode III (focus group) with a touch of reality in a form of external participants, and finally scales up to more realistic conditions (real users and real problems) in episode IV (TAR).

According to Wieringa and Morali [158], during TAR we can differentiate three cycles, in all of which the researcher plays distinct and separate role. In the first cycle, called *design cycle*, the researcher takes part in designing and developing an artifact aimed at improving the *class of problems*. In addition to building artifact, evaluation of artifact in laboratory setting is also done here. In this dissertation, design cycle would correspond to the overall 5-activity design science process. In the second cycle, called *empirical research cycle*, researcher aims at answering knowledge questions about the artifact's interaction with the problem context. In this dissertation, empirical research cycle corresponds to designing technical action research in evaluation episode IV. Finally, in third cycle, called *client cycle*, the researcher uses artifact to improve a *particular problem* for *particular client*. In this dissertation, this corresponds to the very execution of TAR designed in episode IV.

In the remainder of this section, we describe the process of conducting TAR research (em-

pirical and client cycle) guided by the checklist from [158], and present the obtained results.

7.4.1. Research context

The first thing we want to define when designing TAR is the **knowledge goal** to be achieved by conducting TAR. Knowledge goal defines what we want to know about the treatment, i.e. what exactly about the artifact we want to validate by placing it into a problem context. In this particular TAR we specify the following knowledge goal: *To investigate whether the REFRAME application framework, as a newly designed design science artifact, is perceived as useful in practice by potential users.* Aside from knowledge goal, TAR also requires stating the **improvement goal**, which is none other than the goal of the treatment, i.e. designed artifact. Therefore, we reiterate the goal of REFRAME as improvement goal: *Improve and facilitate the management of reactive dependencies in object-oriented applications.*

Finally, describing TAR's context necessarily includes stating the **current knowledge** about the artifact and its use. Since this TAR is an inner engineering cycle within higher-level cycle (design science project), current knowledge is already stated in previous chapters. This includes thorough literature review, structural and behavioral characteristics of designed and developed artifact, and feedback obtained from the first three evaluation episodes. In first evaluation episode individual aspects of artifact are tested, in the second one the basic features are demonstrated on illustrative scenarios, and finally feedback on potential usefulness of artifact is obtained in third episode. Here we also take account of researcher's own experience and knowledge obtained while being involved in development of real object-oriented software application with large and complex reactive dependency graph.

7.4.2. Research problem

When defining research problem TAR demands **conceptual framework** of the artifact to be stated. Conceptual framework can be described as a set of definitions of concepts [158]. These concepts correspond to notion of constructs in design science, which are used to specify and communicate artifact's structure, surrounding environment and interactions, as well as related research questions. Constructs from artifact's conceptual framework are formally defined and mutually associated in chapter 5, and include: reactive node, reactive dependency, dependency graph, update process, etc.

Knowledge questions are validation questions which ask about effects of transferring artifact into a real world. Since TAR is a lower-level cycle conducted within design science project, its knowledge question corresponds to evaluation of **usefulness** evaluation property as defined in design science project. Therefore, the knowledge question can be reiterated as design science research question **RQ6**: *How does the use of REFRAME affect the management of reactive dependencies in development of object-oriented applications?*

Population of interest in TAR can be described as a combination of artifact (or artifact variants) and the contexts in which the artifact is used. We define our population as any individual programmer, team or a company which would use REFRAME (as is or customized) to handle complex dependency graphs when developing object-oriented applications.

7.4.3. Research design and validation

Objects of study

After we defined the population of interest, we need to acquire particular elements of the population to be our **objects of study**. An object of study is a particular context in which the artifact will be used, i.e. the client to which we apply REFRAME during the client cycle of TAR. REFRAME software framework will be applied within the project of developing KI Expert Plus, which is a real software application continuously developed and maintained for 15 years by the team from Faculty of Organization and Informatics, Varaždin in cooperation with the company Knauf Insulation Ltd. and other experts from the field of construction and mechanical engineering. KI Expert Plus is one of the official software applications which are used by hundreds of professionals in Croatia and Bosnia and Herzegovina for designing energy characteristics of buildings and their technical systems.

In order to acquire the client and perform successful TAR, there has to be a mutual trust between the researcher and a client, which often needs years to develop. In our case, the researcher was involved in the development of aforementioned software applications for many years, so there was mutual trust between all TAR participants. Also, the researcher did not only have the knowledge of the project's development process and software artifacts, but he also had access to code repositories and other required resources.

REFRAME framework had not been in any way customized to be used in KI Expert Plus application. This is because REFRAME is in its early versions, and besides evaluation of use-

fulness, it is intention of this TAR to provide feedback on both general and KI Expert Plus specific requirements for improvements and adjustments.

One of the potential threats to generalization of TAR's results happens when the researcher who developed the artifact is also the one who is applying the artifact in the client cycle. This is because the researcher, due to its knowledge and expertise, may be using the artifact in a way no other person could. Since we did not want to lose researcher's own observations on using REFRAME, but we also wanted to address this threat, in addition to researcher two more KI Expert+ members were chosen to participate in TAR. The second important threat is with regard to other TAR participants giving feedback in a socially desirable way, in order to please or support the researcher. In an attempt to mitigate this threat, instead of choosing currently active KI Expert Plus team members, we chose the former project members to participate. This still provided us with participants acquainted with the project, but at the same time decreased the chance of participants being obligated to provide desirable opinions. That being said, there are threats to generalization that are still valid, such as the fact that TAR was applied on only one client/project, and also that the researcher itself was the one that interpreted obtained feedback.

Sampling

During TAR, two client cycles were performed on KI Expert Plus application. In the first client cycle, the researcher himself replaced existing mechanism for handling reactive dependencies with REFRAME in one of the KI Expert Plus modules. In addition to previously stated knowledge goal, this cycle also aimed at: (1) gaining working knowledge and experience with REFRAME, (2) refining resources and teaching materials for the second client cycle, and (3) discovering critical shortcomings or bugs and resolving them before second client cycle. In the second client cycle, two of the former KI Expert Plus team members were each assigned with one existing software module, in which they replaced old mechanism for handling reactive dependencies with REFRAME.

TAR client cycles were conducted with the help of former KI Expert Plus members, who performed their assigned tasks on a separate development branch. Therefore, there was no real risk to the usual ongoing activities of the KI Expert Plus project, regardless of the outcome of TAR. However, in case REFRAME proved to be useful artifact, there would be a potential to improve both the development process as well as the KI Expert Plus product. That being said, from the perspective of KI Expert Plus project, taking part in TAR could only be beneficial.

From the perspective of TAR, the participation of KI Expert Plus was important because this was a good and a real case of object-oriented software with numerous reactive dependencies forming complex graphs. Not only was the KI Expert Plus application relevant in this aspect, but it strongly emphasized the characteristics of the population of software applications we wanted to generalize over. However, we could argue that this application represent an upper borderline example from the perspective of dependency graph complexity. For more complex applications in this sense, instead of REFRAME, recommended solutions would probably be in the sphere of specialized programming languages outside of OO paradigm. On the other extreme, in software application with very simple dependency graphs, using REFRAME could be considered as overkill (although the framework itself is fairly unobtrusive). In addition to KI Expert Plus being a representative **sample** for target population of software application, all client cycle participants (see Table 18) are software developers which allows us to make required generalizations.

Table 18: TAR client cycle participants

Participant	Qualifications
Researcher	He graduated at Faculty of Organization and Informatics, and is currently pursuing a PhD. He works as a teaching assistant at the same faculty on courses related to software engineering. His research interests are also directed towards software development in general, especially software reuse techniques such as frameworks and design patterns. For past more than ten years, the researcher was continuously and actively involved in development of various software systems, including KI Expert Plus.
Developer 1	He holds a master degree in Databases and Knowledge Bases at Faculty of Organization and Informatics. For the past seven years he worked as software developer in multiple companies and organizations, such as: Evolva d.o.o., Faculty of Organization and Informatics, and Mobilisis d.o.o. He was also involved in development of KI Expert Plus application. Currently, he holds the position of Senior .NET Engineer at Mobilisis d.o.o.
Developer 2	He holds a master degree in Information and Software Engineering at Faculty of Organization and Informatics. During his master degree studies he was involved in different software development projects, including the project of developing KI Expert Plus software application. He currently holds the position of Hybris Java Developer in ecx.io Croatia d.o.o. where he works for the past five years.

Treatment design

The problem of KI Expert Plus software application development includes complex and dynamic domains of construction, engineering, thermodynamics and energy efficiency, which results in great number of domain concepts and mutually dependent calculations. These mutual dependencies are implemented in a form of event-delegate mechanism based on Observer design pattern. Managing these dependencies in such way proved to be extremely demanding, error-prone and hard to comprehend.

Given the problems in development of KI Expert Plus software provided initial motivation for this research, REFRAME's characteristics largely coincide with the needs of KI Expert Plus. Thus, the artifact itself was not customized in any way for the purpose of performing this TAR. However, some alterations were made to KI Expert Plus application. For example, application was stripped from several modules which were not necessary for the module TAR participants were assigned with. This was done in order to avoid distributing entire application code to external parties, but also to avoid unnecessary compilation and loading large number of modules. In addition, since the modules chosen for TAR were legacy modules, some code refactoring was done in order to equate design and coding style with the more recent modules. This was one of the additional benefits KI Expert+ project received from participation in TAR.

Prior to performing client cycle (applying REFRAME), the researcher prepared necessary resources and teaching materials. These included: (1) GitHub repository for each TAR participant, which contained KI Expert Plus application code (including assigned module); (2) archive (.zip) file containing all REFRAME elements (.dll components, Reframe Tool, code snippets, and brief instructions for use); (3) introductory presentation; (4) 30-minute video instructions on how to start using REFRAME with KI Expert Plus. Participants were required to have Visual Studio Community installed (free version) and GitHub account made. Other than their personal computers, no hardware equipment was required.

Since the artifact was experimental, not yet tested in realistic setting, a plan was made for researcher to use REFRAME in one KI Expert Plus module as a part of first client cycle. After that, any critical shortcomings that appeared would be addressed before moving on to the second client cycle. Duration of the first cycle (not including corrective activities on REFRAME and refinement of materials) was estimated at one week.

In the second cycle, with two remaining TAR participants it was agreed to try-out RE-

FRAME by each participant using it in one KI Expert Plus module. Duration of the second cycle was (together with participants) estimated at 3 weeks, taking care not to overload the participants and make their usual responsibilities suffer. The second cycle begins with 60-minute presentation in which participants will be familiarized with the underlying research problem, REFRAME as an offered solution, their tasks, and other general information about participation in TAR. After presentation, participants will be given tasks, as well as necessary resources and materials. Also, when requested by participants, multiple meetings and possibly pair programming sessions were planned to be held in order to help them learn the framework and resolve potential issues.

Measurement design

In the first client cycle, during which the researcher himself applies the treatment, the researcher will keep a log about his own experience on using REFRAME. Together with code repositories of KI Expert Plus modules implemented using REFRAME, researcher's log will provide a valuable source of information for further discussion of framework's usefulness. Since the researcher himself is in charge of it, this cycle will offer only discussion points, without making definitive judgments about the usefulness of the framework.

In the second client cycle, researcher will continuously be in touch with other two participants, either through online meetings or peer programming sessions, and will also keep a log of notable topics that arise in these interactions. At the end of the second client cycle, the researcher will conduct a 30-minute open-ended interview with each of the two other participants, in which he will gather their experiences and impressions on using REFRAME. This will provide us valuable information not only about the usefulness of the framework, but also about possible improvements and future developments.

The interview will be guided by a set of predefined questions which are aimed at motivating participant to express his thoughts and experiences with REFRAME, as well as ideas for future development. However, the interview will not be constrained by these questions, and both researcher and participant will be allowed to address other concerns that may arise during the interview.

Table 19: TAR interview questions

#	Question
1.	How hard was for you to initially set-up the framework for use in end-user application? Is the required effort appropriate?
2.	How useful to you was the core part of the framework in terms of specifying individual reactive dependencies, forming dependency graph and keeping graph updated?
3.	Were APIs of the core part of the framework easy to understand and use?
4.	Did you find any aspect of core part of the framework as inadequate or missing?
5.	What would you suggest for improving or extending the core part of the framework?
6.	How useful for you was the Analyzer tool in terms of helping you understand dependency graphs?
7.	Was Analyzer tool easy to use?
8.	Did you find any aspect of the Analyzer tool as inadequate or missing?
9.	What would you suggest for improving or extending the Analyzer tool?
10.	How useful for you was the Visualizer tool in terms of helping you understand dependency graphs?
11.	Was Visualizer easy to use?
12.	Did you find any aspect of Visualizer tool as inadequate or missing?
13.	What would you suggest for improving or extending the Visualizer tool?
14.	How useful for you were code generation capabilities offered by the framework (code snippets)?
15.	Was code generation easy to use?
16.	Did you find any aspect of code generation as inadequate or missing?
17.	What would you suggest for improving or extending code generation capabilities of the framework?

Inference design

Inference in TAR will be conducted according to descriptive inference method, similar as it has been done in episode III. It starts with *data preparation* step, which includes transcription of audio/video recording of interview with two developers after they implement requested modules. After that, in *data interpretation* step, transcribed text is going to be interpreted using template analysis, in the same way it has been done in episode III. In addition to transcribed text analyzed using template analysis, we will also examine other data source for potentially useful information, such as: personal notes of the researcher, log from source code repository, and the source code itself.

7.4.4. Research execution

After the extensive testing of the framework which is conducted in evaluation episode II, we started with the first TAR client cycle from episode IV in parallel with episode III. As planned and explained in TAR research design, this cycle was performed by the researcher itself with the goal of testing REFRAME on a real life application and preparing resources for the second client cycle. In order to do this, in one of the modules of KI Expert Plus, the researcher replaced the event-delegate mechanisms used for managing reactive dependencies with the mechanisms offered by REFRAME. This involved performing several technical tasks, including: preparation of stripped down KI Expert Plus application, matching REFRAME and KI Expert Plus versions, setting up version control repositories, building a deployable REFRAME, etc. During the implementation itself, the researcher kept notes about required steps and other important insights that would be valuable for preparing the second client cycle. The first cycle also revealed a few bugs and necessary adjustments, which were not obvious in previously limited use, or were simply result of switching to production environment.

After resolving identified shortcomings, and refining the documentation and teaching materials, the second client cycle was able to begin. First, an 60 minute online meeting using Skype platform was organized, and the two developers were briefed about the problem of managing reactive dependencies, proposed solution, and the nature of their involvement in the context of TAR. The meeting was recorded to serve as a part of overall documentation. After the meeting, developers were sent the remaining of the required resources: (1) link to their own GitHub repository containing the code of KI Expert Plus application, (2) the sample KI Expert Plus project, (3) 30-minute video tutorial for using REFRAME, and (4) documentation for the KI Expert Plus module that developers need to implement.

Participants were initially given 3 weeks to accomplish the assignment, however, due to their personal and professional obligations, this was prolonged to 5 weeks. During this time, participants and the researcher had multiple online interactions, sometimes initiated by the participants and sometimes by the researcher. In these interactions, the participants usually requested clarifications or help with some particular issue, and briefed the researcher about their progress. Since two participants had different modules to implement, and their dynamic of working on this task was different, they did not finish at the same time. Therefore, an interview with the participant who finished first was conducted, and the collected data from this interview was

used to draft preliminary themes for the template analysis. After the second participant finished his assignment, the interview with him was conducted, and we were able to fully proceed to the next step - data analysis.

The interviews themselves were conducted in an online environment (using Skype platform), as per recommendations related to behaving during Coronavirus (SARS-CoV-2) outbreak. However, this posed no problem as all involved participants were frequent users of online meeting and collaboration tools. Before officially starting the interview, participants were notified that the session will be recorded for the sole purpose of transcribing the interview. While being focused on its goal, interviews were conducted in an informal and relaxed atmosphere. Both participants looked comfortable answering the motivational questions, and often, on their own initiative, jumped back and forth between the questions when they remembered something important. The questions were deliberately formulated and asked in a way to allow them to express their honest opinion, without feeling that there are things they are allowed to say, and some other things they are not. Both participants showed no hesitation in their answers, even the ones containing critique.

7.4.5. Data analysis

In previous evaluation episode (III) participants were asked to evaluate the framework based on researcher's demonstration of the problem and REFRAME as a solution. Although their opinions were informed ones, they were for the most part theoretical, as they did not have practical experience with the framework. Contrary, in this evaluation episode (IV) participants were given a task to use REFRAME, therefore, their evaluation was focused on a solution and it was based on practical experience. Aside from these differences, evaluation episode III and IV have a lot of in common, especially in terms of data analysis. Both evaluation episodes have the same goal, and they use similar methods (group discussion and interview) to collect opinions from participants. The consequence of this is that both evaluation episodes result in qualitative data of similar structure and meaning. This allowed us to again perform data analysis using *template analysis* method [87] by following 6 steps: (1) Define *a priori* themes and codes, (2) transcribe focus group, (3) carry out initial coding, (4) produce initial template, (5) develop your template, and (6) interpret data set according to produced template.

During the **step 1**, instead of inventing *a priori* themes from scratch, we reused themes from

focus group. We left out, however, problem relevance theme, because interviews in this evaluation episode were entirely focused on a solution. Therefore, set of *a priori* themes consisted of: (1) *positive impression*, (2) *negative impression*, and (3) *framework evolution*. Chosen themes retained their original meaning set out in the previous episode. *Positive impression* encompasses all perceptions which directly or indirectly characterize REFRAME as useful. In a similar way *negative impression* classifies perceptions which could oppose REFRAME’s usefulness. Finally, *framework evolution* represented those statements which suggest useful improvements which were not planned for this phase.

After we defined *a priori* themes, we proceed to **step 2** and transcribed TAR interviews. The duration of each of the two interviews was approximately 40 minutes, therefore total of 80 minutes were manually transcribed. This resulted in 7 pages of relevant textual content ready for further analysis.

Table 20: Final version of template

#	Theme
1.	Positive impression
1.1.	General
1.2.	Initial setup
1.3.	REFRAME Core
1.4.	Analyzer
1.5.	Visualizer
1.6.	Generator
2.	Negative impression
2.1.	REFRAME Core
2.2.	Analyzer
2.3.	Visualizer
3.	Framework evolution
3.1.	General
3.2.	REFRAME Core
3.3.	Analyzer
3.4.	Generator

Initial coding (**step 3**) confirmed *a priori* themes as suitable and they entered initial template as theme categories (**step 4**). The comments within these categories were further assorted (**step 5**) with regard to the part of the framework they referred to, i.e. *REFRAME Core*, *Analyzer*, *Visualizer*, and *Generator*. The underlying idea was to be able to discern contribution (as perceived

by users) of each major framework part in the overall usefulness of the framework. However, since not all comments fit these themes, a new, *General* theme was introduced. Finally, in the next iteration, few of these General comments were assigned with *Initial setup* theme, as they explained how participants viewed the process of initially setting up and configuring the framework. The final template was then used for data interpretation and writing up the findings (**step 6**).

Positive impression

Although we restrain ourselves from making quantitative claims while analyzing qualitative data, it is necessary to say that both interviews resonated in a positive way with regard to REFRAME. This was not only reflected in the amount of positive comments, and the points these comments were trying to make, but also with general attitude and nonverbal communication.

When asked about usefulness of the framework in **general**, with taking into consideration their previous experience in managing reactive dependencies using events in KI Expert Plus application, both participants resolutely expressed satisfaction with REFRAME in this regard. *"Developer 1: Using REFRAME on such large systems definitively makes sense. With events it is easy to get lost... you have to go through a lot of code to find what you need."*; *"Developer 2: Using it in our case, I think the improvements would be manifold: it would be easier to write a code, one would produce less bugs, GUI refresh happens only once, it is easier to debug..."*

With respect to more specific topics, the interview started of with discussing how hard was to install and make **initial setup** of the framework. This was important to address, as the framework itself may be useful, but very complex installation and setup can dissuade potential users from giving it a chance. However, participants were again clear in stating they had no problems whatsoever in making REFRAME and its accompanying tools work. This can be attributed partly to fairly simple setup procedure, but also to clear video instructions participants received. *"Developer 2: Well it was quite simple to put this together in terms of framework... just had to add dlls to project, and that was it. Also running the tools worked the same way, without any additional configurations... I just needed to run the tool."*; *"Developer 1: It was as simple as it can be, fetch dlls, reference them... I don't think it goes simpler than that. Without instructions I would not be able to run tools, but with instructions I did it all in my first try."*

After discussing initial setup of the framework, we proceeded to address the essential and the most important part of the framework - the features of **REFRAME Core**. The participants

recognized a lot of benefits these features bring to a table. For example, the fact that REFRAME handles what has to be updated and in which order, was very appreciated. *"Developer 1: So, that core part is definitively an improvement over what we were using so far (.NET events). It is evident that the core part takes care of updating the right thing at the right time."*; Furthermore, the option to keep the code in charge of specifying reactive dependencies at one place in class was also well received. This makes sense as it prevents code scattering, and also makes reactive dependencies more readable and understandable from the code itself. *"Developer 2: To me, the core part was useful because I could add most of the code related to managing reactive dependencies in one method... I could easily examine the class and see if I specified reactive dependencies correctly... There is no scrolling and searching, one method contains 90% of important information for this class."* The core part of the framework also helped participants to reduce the time required for managing reactive dependencies, and also to increase the quality of code. *"Developer 1: With regard to the time required to make code react to changes, I think we spent more time with our old way of doing that... this includes also more time for testing and debugging. With REFRAME, however, I specified reactive dependencies, and made the changes reflect in GUI, and that was all, it worked! All in all, it was easy to use, it was intuitive, and it resulted in more stable code."* The final thing participants explicitly acknowledged related to REFRAME Core, was the straightforward way in which e.g. GUI can track changes resulted from update process. *"Developer 1: I think that this UpdateCompleted event is definitely worth to have, because you want to be notified when that happens. Because, one of the things that was bothering us before, in old system, was that events were triggered multiple times, and every time the GUI was refreshing, and that often caused application to stop responding."*

While **Analyzer** tool is extremely important, as it allow us to reduce the number of nodes and focus only on particular part of the graph, it seems that participants saw it as a sort of pre-step to Visualizer tool, rather than a separate, standalone tool. This might be the case because Visualizer always takes input from the Analyzer and is in fact run from the Analyzer. Also, it was evident that participants favored more the visual representation in a form of graph, rather than in a form of nodes listed in a table. However, in general, the participants acknowledged the role of Analyzer in the toolset. *"Developer 1: The Visualizer in a combination with Analyzer and its filtering capabilities... it is powerful stuff. Especially when you have such a complex software."*; *"Developer 2: I managed to find my way around Analyzer... the tool pretty much*

delivers what it needs to deliver. I don't see that anything is missing." Aside from these general remarks, participants also mentioned some specific options that were useful to them, for example node sorting. "**Developer 2:** *It was nice to be able to sort graph nodes in a table, for example alphabetically, to quickly find my way around a bunch of nodes.*"; Also, during a more in-depth discussion about mechanics and filtering capabilities of Analyzer tool, developers expressed their awe in what actually Analyzer can do. "**Developer 2:** *If it is, as you say, possible to list even the instances of some particular class, and I can choose what exact instance I want to focus on,... then, that is really awesome."*

As already mentioned, **Visualizer** was the tool which was most attractive to participants. When asked about it, participants were very clear that Visualizer was very useful, and they immediately offered a concrete example. "**Developer 1:** *Well, it was quite useful. After I specified reactive dependencies, I couldn't wait to run Analyzer and Visualizer, and see what actually happened behind the scenes. I mean, it provides additional insight into a situation... that is the best thing for me. And also, it is nicely presented in a visual sense, you can zoom in, zoom out, reposition nodes...*"; "**Developer 2:** *When I was trying out the Analyzer, I wanted to see what are inputs and what are outputs. And not until I went to visualize it did I understand that three classes from my example have the same final outputs... This was not evident before because the classes had the same parent class."* One of the participants raised a question whether the visualized graphs can be saved for future use, and was pleased to see that they can be saved, and also manually altered. Finally, the participants expressed no major problems in using Visualizer, and they found its current features to be adequate. "**Developer 2:** *Yeah, with Visual Studio the graph was visualized with no problems...*"; "**Developer 1:** *I don't see that there should be anything different here."*

Finally, participants also found a code *Generator* to be useful and easy to use tool. "**Developer 1:** *Code snippets were very useful. I mean, the clipboard is always an alternative, but I would rather use code snippets. And I use them on a daily basis. The code snippet Let->Depend is definitively useful, especially because you can use TAB key to walk from one property to another. Without further ado, that is very useful.*"; "**Developer 2:** *Well, they definitively make code writing faster, they are useful. I have a habit of copying line of code and then changing parts of it... but, this is faster. And they also reduce the possibility of making type errors. They are not hard to add, so if there is a line of code which is frequently written, then it makes sense to use*

them."

Negative impression

During interviews participants raised several concerns related to framework use or its features, and made comments which represent a direct or indirect critique. We have taken these concerns and critiques very seriously, and some are going to be addressed in the framework's next release. However, with some of them we are unfortunately constrained with technological limitations. Nonetheless, we argue that raised concerns are not severe to the point that they would annul the usefulness of the framework. Even the participants themselves often emphasized that.

With regard to **REFRAME Core** features, the thing that participants found the most problematic is the fact that despite many improvements framework offered, they still had to write certain amount of code to specify reactive dependencies and trigger update process. "**Developer 2:** *I was a bit disappointed with C# syntax, as it did not offer a more readable solution for specifying reactive dependencies, so there would be no brackets in Let->Depend statements. But I understand this is more the limitation of programming language, and less of the framework.*" As the participant correctly noticed, this was the only syntax in host programming language which allowed us to pass both owner object and the property as one code element. Considering how much is accomplished by Let->Depend statement, we find this to be acceptable trade-off. "**Developer 1:** *It is true that there is some manual work to be done until you write these lines of code, but there is no silver bullet, you have to define reactive dependencies somehow.*" Again, we did consider alternative ways for specifying reactive dependencies, including doing that through GUI. However, this did not turn out to be any less time consuming, and it was definitely less flexible. "**Developer 1:** *You know that part when you have to manually invoke update process in input properties... well it would be great if this would not be necessary, but its not too much of a problem.*" While this critique is perfectly valid, we were unable to avoid these lines of code. During framework development we tried several approaches, however, concerns arose regarding framework performance. That being said, we still are actively searching for a solution to this. "**Developer 1:** *Also, it would be great if we would not have to manually invoke refreshing of graphical interface when the update process is completed. I don't know if that is possible?*" This can in part be done using the *binding* capabilities of .NET UI controls. However, we still need to bind certain control with e.g. property it represents. Doing that either through a

graphical user interface or programmatically is not particularly easier than just responding to REFRAME's *UpdateCompleted* event. Also, not all UI controls are *bindable*, so there has to be a convenient way to manually signal GUI to refresh itself. Another critique that could be read from participants' responses was the fact that prior to using framework one needs to consult instructions and documentation. On one hand, this critique is understandable because, it is known that developers are not fond of studying the documentation, and like to jump right in and start using the framework. "**Developer 2:** *Well, if I went and started using the framework, as I usually do, without reading the documentation, I would probably need some time to think things through.*" However, as much as we try to hide complexity of the framework and make its use straightforward, there is always a learning curve. In our case, watching 30 minute demonstration video provided enough information for participants to accomplish their task. We argue this to be reasonable amount of time required to be acquainted with some framework.

A dominant issue related to **Analyzer** which was revealed during interviews, was the fact that participants only in part acknowledged the importance and capabilities of Analyzer. There are several reasons for this. First of all, the Analyzer indeed has large number of available options and combinations of these options, which makes it hard to understand. Participants recognized that, and suggested these options to be better organized and documented in order to make them easier to understand. "**Developer 1:** *I suggest Analyzer's GUI to be improved, to reduce the number of windows popping up. Also, it would be useful to have a help documentation, because it is not easy to understand what each of the numerous options represent.*" This is certainly a valid critique, therefore GUI usability improvements, as well as additional user documentation is planned for the next release. One of the things that was also clearly evident was the fact that participants favored the visual graph representation, rather than tabular view of graph nodes offered by Analyzer, and thus were more focused on Visualizer. "**Developer 1:** *I didn't use Analyzer too much, I was more interested in graph visualization. That was enough for me.*"; "**Developer 2:** *When I was trying out the Analyzer, I wanted to see what are inputs and what are outputs. And not until I went to visualize it did I understand that three classes from my example have the same final outputs... This was not evident before because the classes had the same parent class.*" Although the exact purpose of Visualizer was to provide visual experience which Analyzer simply could not, aforementioned GUI improvements and better documentation should make Analyzer more attractive to users. Finally, one of the participants reported

minor bug related to filtering options not being automatically refreshed in certain cases. This was also put to a list of issues to be solved for the next release.

Finally, with regard to **Visualizer**, participants had no particular concerns other than reporting a minor bug which caused wrong name of a node group to be displayed. This issue is also set to be resolved for the next release.

Framework evolution

As opposed to *negative impression*, this theme describes suggestions for improvement which were not motivated by raised concerns or identified deficiencies. Rather, these suggestions were result of brainstorming process intended to generate ideas which would improve and extend the framework outside of its current boundaries. In order to do this, participants were encouraged to be open to new ideas and views, and even throw wild guesses. As a consequence, not all of the suggestions are applicable. For example, in the context of **REFRAME Core**, one of the participants suggested reactive dependencies to be specified between class-member pairs, rather than between object-member pairs. *"Developer 2: We are currently specifying reactive dependency by pairing the property of one object with the property of another object. Maybe these dependencies could be defined in a way to not depend on the instance, but on the class."* The participant and the researcher engaged in discussion in which it became apparent that this would not produce desirable result. The researcher gave a traditional example of Order and OrderDetails class. Although these two classes are associated with each other, that does not mean all instances of Order class are associated with all instances of OrderDetail class. Specifying reactive dependencies at a class-member level would mean that, in this particular example, all orders would depend on all order details, which is incorrect.

One of the suggestions that was applicable, was the **general** suggestion to offer framework implementations for more programming languages (also suggested in focus group). As we mentioned in focus group, in the near future we only plan to realize .NET Core version of the framework. However, since this is an open-source framework, support for other programming languages could come from the community.

With regard to **Analyzer**, there was a suggestion to mimic the *Watch* feature of Visual Studio debugger. *"Developer 2: This reminds me of debugger in Visual Studio, where you can watch the values of some variables over time. Maybe in Analyzer we can mark nodes in some way, which would make them at the fixed position while we analyze some calculations..."* This is an

interesting suggestion made in line of supporting debugging process of system with reactive dependencies. It is something we are going to take into consideration for future development of the framework.

Finally, suggestion related to both Visualizer and Generator capabilities came in a form of enriching graph visualizations, so that changes to visualized graph are reflected in source code.

"Developer 2: Maybe one idea, that... by changing visualized graph we can modify code."

This model-driven style of working with reactive dependencies is certainly a worthwhile idea, however, because of various conceptual and technical challenges it will require further research and prototyping to determine its feasibility.

8. Discussion

8.1. Reflection on the research

In this dissertation, we have been addressing the problem of managing reactive dependencies in OO applications. Since OO paradigm lacks native support for expressing and managing reactive dependencies, developers are left to handle this manually, e.g. by implementing Observer or similar design patterns. However, in even slightly more complex scenarios such approach requires a huge effort to avoid errors, redundancies, and to understand and maintain dependency graphs. The practical relevance of this problem is witnessed by the researcher himself from his own experience in developing software systems, but it is also evident in professional literature, developer forums, and other relevant information sources used and authored by software developers. In addition, the literature review conducted within this dissertation also showed that the stated problem has scientific relevance, and that it has been frequently investigated by researchers within multiple paradigms, including object-oriented and reactive paradigm.

Trying to solve or mitigate a problem that is both practically and scientifically relevant also requires contributions to both practice and science. Practice can be supported in dealing with the problem, for example, by creating an artifact that removes the problem or some part of it, lessens its effects, or enhances our capabilities to cope with it. On the other hand, science is contributed by generating new knowledge in the process of analyzing the problem, creating an artifact (solution), and using the artifact in a problem domain. Such highly pragmatic research efforts, directed to problem-solving and artifact-building, required equally pragmatic research paradigm to guide them. We found this guidance in a form of design science paradigm.

Guided by design science, we set-off to mitigate the stated problem by building an artifact that would offer abstractions and mechanisms used to support various tasks related to managing reactive dependencies. Through research and careful consideration, we identified software framework as a type of software artifact suitable for this purpose. Further analysis of the problem and existing solutions gave us an insight into what characteristics and features such software framework should exhibit, and allowed us to form a requirement specification document. In accordance to set requirements, a REFRAME software framework was designed (model artifact), and implemented in C# programming language (instantiation artifact). By building and testing REFRAME software framework we demonstrated that an artifact aimed at improving

the management of reactive dependencies is technically feasible, that it works under certain assumptions, and that it has potential to solve or mitigate the stated problem. Evaluation efforts that followed provided further evidence for this, and, in addition, also showed that potential users perceive REFRAME as useful artifact in the context of managing reactive dependencies.

8.2. Answering research questions

While in previous section we briefly reflected on the overall design science research process and its key results, in this section we describe how these results contribute to answering each of the six research questions.

RQ1 What makes the management of reactive dependencies in development of object-oriented applications a challenging task?

The first research question (RQ1) is addressed in detail within the Explicate problem activity. This activity is formally described in chapter 4, however, in a broader sense, this activity also encompasses efforts in reviewing existing literature, analyzing problem domain, and utilizing researcher's own experiences - all presented in chapters 1 and 2.

The main reason why managing reactive dependencies is a challenging task in general, is the mere fact that relationships in applications modeled by reactive dependencies may be numerous and interwoven. These reactive dependencies may form inherently complex graph-like structures called dependency graphs. The sheer size and complexity of these dependency graphs challenge the human brain capacity and capability to understand and reason about their structure and dynamics. Unfortunately, OO paradigm in particular lacks means to overcome these limitations. There are no built-in abstractions to express reactive dependencies, nor are there mechanisms to manage structure of dependency graph and their updating process. In addition, there are also no integrated tools which would support and automate basic operations, or provide visualization, analysis and reasoning aids in the context of reactive dependencies. Instead, in OO context, developers are usually left with manually handling reactive dependencies, often in an ad-hoc manner, by implementing Observer or similar design pattern. This, however, requires writing large amount of boilerplate code which tangles with application's core functionality, and often distorts natural inheritance hierarchies. Even more, since no proper abstractions and mechanisms are available, such code represents fairly low-level take on reactive dependencies.

This means that developer spends a lot of time and effort handling reactive dependencies instead of focusing on core functionality. After some time, an application becomes very hard to maintain, because any change in code that could possibly require structural changes in dependency graph, requires detailed analysis of the current state, and the impact that the change would or would not have. Since analyzing dependency graph is all done manually, it becomes increasingly time-consuming to do it. Even if we are to invest required time, analyzing dependency graph would, due to its size and complexity, still be error-prone, and we would frequently miss important information.

There are several issues that frequently occur in this context. For example, one can due to mistake or defensive coding specify the same reactive dependency multiple times, which will result in performing update process redundantly. Redundant updates, or even updates that are unnecessary in the first place, also happen when there is a lack of mechanism which determines what elements of dependency graph should be updated and in what particular order. Other than performance issues, updating redundantly and in a wrong order may result in temporary (so called glitches) or even permanent inconsistencies and invalid results. One other issue that can happen while building dependency graph is the existence of circular dependencies, which can cause infinite loops and result in throwing exceptions. Aggravating circumstance is that circular dependency may manifest itself only if certain, very specific conditions occur. If dependency graph is not built correctly, application may also fail at performing updates which are indeed necessary, and again cause invalid results.

RQ2 What are the means we can use to support development of object-oriented applications in order to improve the management of reactive dependencies?

The second research question (RQ2) is also covered in detail within the Explicate problem activity, which as we already stated encompasses chapter 4, but also chapters 1 and 2.

We certainly cannot take away inherent complexity that characterizes structure and dynamics of graphs formed by reactive dependencies. However, we can amplify our capability to tackle this complexity by offering different means to aid the process of managing reactive dependencies in OO applications. Essentially, we want to design a solution in which we would localize as much of the complexities related to managing reactive dependencies as possible and support different operations with reactive dependencies. Then, application developer could reuse that solution when developing individual applications, and to a large extent be free from

the details of managing reactive dependencies.

Such solution should at first try to increase the level of abstraction in working with reactive dependencies, by offering dedicated abstractions for key concepts from the problem domain. It should provide support for specifying individual reactive dependencies, forming dependency graphs from them, and performing update process. Parts of these operations may also be automated through built-in tools.

Besides supporting basic operations with reactive dependencies, a solution should provide additional tools that would help application developers in comprehending and better understanding structure and dynamics of dependency graphs. For example, visual tools and techniques have been frequently used to help developers better understand complex systems and complex interactions between elements of the systems. Specifically, graph-like structures have been often represented visually in mathematics and computing. Also, tools for analysis and reasoning about dependency graph can help discover patterns and in-depth characteristics of the graph.

As extensively covered in section 2.2, software frameworks are one of the most popular reuse techniques that offer both the reuse of design and implementation. They traditionally extract and capture common domain abstractions, and allow them to be reused and extended in developing multiple different applications. In order to further enrich software frameworks and facilitate their use, they are often accompanied by built-in tools. Therefore, in the context of managing reactive dependencies, development of OO applications can be supported by providing software framework which contains dedicated abstractions for reactive dependencies, enables basic operations with these abstractions, and offers useful built-in tools.

RQ3 What functional and non-functional requirements should REFRAME software framework meet in order to manage reactive dependencies?

While within Explicate problem activity we analyzed the problem domain and peeked into solution space, within Define requirements activity covered in chapter 5 we take a firm step toward specifying future solution, and in this way address the research question number three (RQ3).

Firstly, the base concepts and characteristics of the artifact are outlined. This involved confirming software framework as a suitable artifact, and positioning the framework within software framework's ecosystem. Also, from the perspective of design science research, the future software framework was set to be built in a form of both *model* and *instantiation* artifacts.

Finally set of five high-level requirements were specified to serve as a base for elaborating more detailed requirements.

These detailed requirements, in their entirety formalized as *Software Requirements Specification document - SRS* (section 5.2), are offering direct answer to RQ3. The SRS document starts answering the RQ3 by describing fundamental aspects of the future framework, such as its purpose, scope, environment, and users. It also serves as a place to define and elaborate fundamental domain constructs (e.g. reactive node, reactive dependency, dependency graph, update process, etc.) which form a vocabulary necessary to discuss both problem and solution. Finally, and most importantly, SRS prescribes total of *34 functional requirements* assorted into *6 features*, and *4 non-functional requirements*.

RQ4 What prerequisites, constraints and other factors have to be met in order for REFRAME software framework to be designed and working?

Research question number four (RQ4) is primarily addressed by efforts in Design and develop activity of design science process, described in detail in chapter 6. During this activity, design and implementation ideas were gathered, described, tried out, assessed, and implemented. This resulted in a large number of design decisions being documented and elaborated. Evaluation Episode I - Prototyping and testing (section 7.1), due to its formative nature, overlaps with Design and develop activity, and therefore also contributed to answering RQ4. Through prototyping, this evaluation episode had major role in trying out alternative ideas and in this way has heavily influenced design decisions. Similarly, testing not only provided evidence that individual parts of the framework are functioning correctly, but it also (by writing testable code) significantly influenced framework design. Finally, Episode II (section 7.2) with its summative demonstration can also be considered to have contributed to answering RQ4, as it provided evidence (in a form of scenarios) that the framework works as a whole.

In order to design working piece of software, developers face numerous challenges that they have to respond to. Prerequisites have to be met, constraints have to be circumvented or adjusted to, alternative options have to be compared and the best fitting one chosen, available resources and stakeholder preferences have to be considered, etc. All of this places huge number of design and implementation decisions in front of developers. Some of these decisions are known a priori, or they can be anticipated with a reasonable certainty, which gives us a chance to better prepare. However, some of these decisions emerge only during the design and development

activities, and can impair the overall process or have adverse effects on the artifact itself. This stands true also for REFRAME software framework. During its design and develop activity, numerous design and implementation decisions were made, influenced by set of prerequisites and constraints, available alternatives and other factors. While these decisions are discussed and documented throughout entire chapter 6, the 35 most important and impactful ones are summarized in Table 14. By doing this, we provided valuable experience for future similar endeavors, and in this way answered the research question RQ4.

RQ5 What are the typical scenarios of managing reactive dependencies that can be used to evaluate REFRAME software framework?

In addition to contribution in answering RQ4, the main purpose of evaluation Episode II was to answer research question RQ5. This was done by identifying and describing 15 illustrative scenarios of REFRAME use (see section 7.2), which demonstrated most important features of the framework. Although these 15 scenarios do not cover all the details of what the framework can offer, they were sufficient to demonstrate that REFRAME supports the entire process of managing reactive dependencies, ranging from specifying reactive dependencies and ensuring the dependency graph is properly updated, to performing various graph analyses and visualizations.

RQ6 How does the use of REFRAME software framework affect the management of reactive dependencies in development of object-oriented applications?

Evaluation episodes I and II, previously mentioned regarding their role in answering research questions RQ4 and RQ5 showed that the framework for managing reactive dependencies, such as REFRAME, is technically feasible and can work for a range of scenarios. However, one thing that was still missing, was the proof that such framework would be useful to its users (application developers) when applied to realistic scenarios. This issue is formally stated in research question RQ6, and is addressed by joint contributions of evaluation episodes III and IV.

Evaluation Episode III (section 7.3), in which we conducted confirmatory focus group, represented the first time the framework stepped outside of the laboratory setting. The aim was to present focus group participants (4 software developers) with the problem of managing reactive dependencies, and the REFRAME framework as a solution to that problem. Through a discussion motivated and steered by the predefined questioning route, a data is gathered on

how these participants perceived REFRAME framework in terms of its usefulness. Template analysis of the focus group discussion transcript showed that participants perceived the problem of managing reactive dependencies as a relevant problem, and REFRAME as a useful solution to that problem. Most of the raised concerns were related to minor issues, which did not diminish the overall positive impression. In addition, focus group resulted in several suggestions for useful extensions of the framework, which indicates that the framework has a good potential. These interpretations were also backed up by results from small questionnaire based on Technology acceptance model. On a 7-point Likert's scale, all participants either *strongly agreed* or *agreed* that using REFRAME would enable them to manage reactive dependencies: (Q1) **more quickly**, (Q2) with **improved performance**, (Q3) with **increased productivity**, (Q4) with **enhanced effectiveness**, and (Q5) **easier**. Lastly, all participants *strongly agreed* that REFRAME is **useful** for managing reactive dependencies (Q6).

Evaluation Episode IV (section 7.4) had the same goal as the Episode III. However, this time, instead of just being presented with the problem and REFRAME as a solution, researcher himself and two external software developers actively used the framework as a part of technical action research (TAR). During a few weeks, each of the three TAR participants familiarized themselves with REFRAME framework and used it to manage reactive dependencies in one of the modules of real software application KI Expert Plus. The fact that participants were able to successfully accomplish that, provides a first evidence of REFRAME's usefulness in real applications. Further evidence about framework's usefulness was obtained by conducting a post-implementation interview with each of the two external developers. During interviews a range of predefined topics were raised, with some of them directly questioning usefulness of individual parts of framework, and others doing that more subtly. As with focus group, template analysis was performed on transcripts of conducted interviews. The analysis showed that positive impressions directed in favor of framework usefulness prevailed during interviews. When asked about usefulness of the framework, participants expressed both general satisfaction, as well as satisfaction with a range of specific REFRAME features that were useful to them during development. The reported benefits of using the framework included easier and faster development, fewer errors in code, easier debugging, and increased understandability of dependencies. On the other hand, reported concerns were not particularly severe, especially not to the point of annulling the usefulness of the framework. Even participants themselves explic-

itly acknowledged that for some of these concerns. In addition, some concerns were the result of technological limitations, or they were the part of usual experience related to framework use in general. The two minor bugs which were reported do not prevent normal functioning of the framework. Nonetheless, they are going to be resolved in the next release. Finally, suggestions for framework evolution provide evidence that the potential of REFRAME is not fully exhausted, and that both research and development activities are possible in the future.

All this being said, evidence gathered in evaluation episodes III (focus group) and IV (TAR) allows us to answer research question **RQ6** and state that the use of REFRAME **positively** affects the management of reactive dependencies in development of object-oriented applications.

8.3. Contributions

Design science is a pragmatic, problem-solving paradigm. Researchers involved in design science projects (such as this dissertation), are not only theorizing about the existing phenomena in the world, but are actively and purposefully changing the world. They do this by creating an artifact which solves (or at least mitigates) practical problem, and also by generating new knowledge related to that artifact and its use. Therefore, by being a representative of design science research, this dissertation contributes to both practice and scientific community.

Practical contributions

In order to claim practical contributions of the dissertation and its outputs, we first need to discuss the practical relevance of the problem, and then offer the evidence on whether and how that problem can be solved or mitigated. With regard to relevance, the problem of managing reactive dependencies was at first encountered by the researcher himself while participating in development of software application KI Expert Plus. After examining professional literature, developer forums, and other available information resources, it became evident that the problem was, to a greater or lesser extent, present also in other software applications. Indeed, several design patterns were found, including one of the most well-known patterns - Observer, that were intended to deal with this problem. Since design patterns by their definition aim at addressing common problems, this was a clear indication of the problem prevalence in OO applications. This is additionally supported by the fact that mainstream OO programming languages such as C# and Java, even include dedicated language constructs that mimic the intent of these design

patterns. More details supporting the problem's practical relevance can be found in Introduction (chapter 1), Literature review (chapter 2), and Explicate problem (chapter 4) chapters. Finally, the problem's relevance to practice was confirmed during evaluation episodes III and IV, in which participants of focus group and technical action research confirmed both the presence of the problem as well as significant undesirable effects it produces during software development.

Most design science contributions fall into category of *improvements*, i.e. new or substantially enhanced artifacts which provide improvement of some undesirable state of affair. This is because the problems addressed by design science have characteristics of so-called *wicked problems*, which Johannesson and Perjons [78] describe as problems difficult to solve due to incomplete knowledge and complexity in general. Therefore, efforts addressing such problems usually come in a form of incremental improvements and problem mitigation, rather than definitive solutions. Such is the case with this dissertation. The practitioners involved in software development can use several dissertation outputs to mitigate the problem of managing reactive dependencies. At the highest level of abstraction, SRS document presented in chapter 5 can be used as a basis for creating new software requirement specification. Alternatively, SRS document can be used in its original form to support design and implementation of new software framework for managing reactive dependencies. Furthermore, the model of REFRAME software framework, extensively elaborated in chapter 6, can be customized or used as-is, to offer a completely new implementation of the framework. Finally, and this is the most likely case, software developers will take the source code or binary version of REFRAME implementation, and with or without customization use it in application development.

The evidence that REFRAME is a functioning solution and that it is capable of mitigating the problem, is provided by evaluation episodes I and II. In these episodes, the correctness of individual fragments of framework functionality is demonstrated using unit testing, while most important features as a whole are demonstrated by implementing and documenting illustrative scenarios. Furthermore, evidence that REFRAME is mitigating the problem in real conditions, was provided by evaluation episodes III and IV, in which representatives of target user population (software developers) perceived the framework as useful.

Knowledge contributions

In order to systematically describe concrete knowledge contributions of this dissertation, it is useful to assort contributions with regard to what type and form of knowledge they represent. According to Johannesson and Perjons [78], *knowledge type* describes the purpose for which knowledge can be used, and they roughly distinguish between: (1) definitional, (2) descriptive, (3) explanatory, (4) predictive, and (5) prescriptive knowledge. While this dissertation contains *definitional knowledge* (e.g. concepts and definitions needed to communicate about problem and solution space), *descriptive knowledge* (e.g. statements describing the characteristics of the problem and solution), and *explanatory knowledge* (e.g. root-cause analysis), the most important contribution comes in a form of *prescriptive knowledge*. Firstly, SRS document, presented in chapter 5, prescribes what kind of solution we need and what features should it possess in order to be able to manage reactive dependencies. Secondly, framework design and implementation decisions, presented in chapter 6, prescribe what behavioral and structural characteristics should this solution have in order to be realized and be able to address the problem. Finally, as we will see in the next paragraph, implemented and working software systems, such as REFRAME software framework, also contains prescriptive knowledge.

According to Johannesson and Perjons [78], the *knowledge form* describes where and in which shape can knowledge be found. From this perspective, we can regard knowledge as either: (1) explicit, (2) embodied, or (3) embedded knowledge. Aforementioned prescriptive knowledge related to documented framework requirements, and design and implementation aspects, can be characterized as an *explicit* knowledge, as it is explicitly presented (chapters 5 and 6) in a form of text, code and diagrams. However, not all knowledge can be explicated in this way, and even for the one that can be explicated we can expect some level of information loss. This is what makes design science research very important, as one of its distinguishing features are contributions in a form of implicit knowledge, embedded in the built artifact itself. For example, design aspects described in chapter 6 represent an explicated prescriptive model of the framework. However, model in any form is an abstract, simplified representation of the original. REFRAME implementation is a working artifact that necessarily embeds this model, extends it and adds numerous details on top of it, and is therefore a vital contribution of design science research. By studying it we can reveal knowledge about both problem and solution, we can extract architectures, patterns, best practices, rules of thumb, algorithms, etc. Finally,

the very use of REFRAME in the problem context, such as one described in dissertation's evaluation activities, leads to new insights.

As can be seen, one of the distinguishing characteristics of design science is creation of artifacts, which in addition to their practical purpose also carry knowledge contributions. Design science artifacts are usually classified as either: constructs, models, methods or instantiations [73]. **Constructs** are small artifacts, used to express concepts and definitions that are necessary to communicate about the problem and solution. They are typical representatives of definitional knowledge in design science research. In this dissertation, constructs that are necessary to speak about the problem and solution are listed during Explicate problem (chapter 4) and Define requirements activities (chapter 5). **Models** can be seen as abstracted representations of structural and behavioral characteristics of possible solutions to a problem. While, in accordance with their purpose, they are often categorized as descriptive, explanatory, predictive or prescriptive, Johannesson and Perjons [78] emphasize that only prescriptive models qualify to serve as a design science artifact. In this dissertation, a prescriptive model of REFRAME software framework is documented and elaborated in chapter 6. It contains both structural and behavioral aspects of the framework, as well as design rationale describing possible alternatives and argumentation for numerous design decisions. Like models, **methods** also offer prescriptive knowledge by defining procedures for accomplishing some task. They range from more formalized types such as algorithms, to informal ones such as guidelines and best practices. In this dissertation we did not offer such artifacts. Finally, **instantiations** are functioning systems, which can be used in practice and are representatives of embedded knowledge. In this dissertation we offered an instantiation artifact in a form of REFRAME - a working software framework. REFRAME embeds the knowledge about the problem and solution related to managing reactive dependencies, and can be used in practice to mitigate the problem.

Artifact types that can be found in this dissertation, i.e. constructs, model and instantiation of REFRAME software framework, are not unrelated artifacts. On the contrary, they are stacked on top of each other. Constructs that define concepts such as reactive node, reactive dependency, update process, etc., form a foundation of building blocks for defining REFRAME model. That model then serves as a blueprint for implementing REFRAME instantiation artifact. However, this association does not go only one way, because instantiation also affects artifacts on a lower level of artifact stack. Indeed, as March et al. [96] report, by building instantiation artifacts, we

operationalize constructs, models and methods they contain, thus demonstrating their feasibility and effectiveness. This should be considered in the context of our evaluation activities, because, by evaluating REFRAME instantiation, we also provided a confirmation for its underlying artifacts.

Although it should involve practical contributions, the backbone of design science research has to be scientific one, i.e. knowledge contributions. Thus, as opposed to pure design, design science research required the fulfillment of three additional requirements [78]: (1) rigorous research methods should be used to generate new knowledge, (2) this new knowledge should be well-founded and original, i.e. it should be put into a relation with already existing knowledge, and (3) research results should be explicitly communicated to practitioners and researchers.

The first requirement was fulfilled by choosing and following rigorous methodological framework for design science research [78]. Within individual steps prescribed by this methodological framework, various research methods were applied, including literature review, prototyping, modeling, and analysis. However, as befits design science research, scientifically most rigorous step was evaluation. Using FEDS framework [154] as a guide, we first designed evaluation strategy suitable for the type of artifact we were building. Each evaluation episode within this strategy was assigned with proven, and for that particular purpose, frequently used research methods. Therefore, a testing, demonstration, focus group, technical action research, and template analysis joined previously mentioned research methods.

In order to fulfill the second requirement, we have to relate efforts done in this dissertation with efforts described in existing literature. Conducted literature review shows the problem of managing reactive dependencies being approached and addressed by several different paradigms. Although heavily criticized, Implicit invocation and Event-driven programming based on Observer pattern is still a dominant way of managing reactive dependencies in OO paradigm. Some improvements over the plain Observer pattern can be achieved using more advanced design patterns (such as Propagator [53]), or some aspectized versions of Observer (e.g. [149]). Indeed some of REFRAME's design characteristics are inspired by these design patterns. However, as was reported by Mijač et al. [107], no single pattern was capable enough to address the problem of managing reactive dependencies in more complex cases. Our vision of solution to that problem, surpassed these design patterns in both reuse scale and abstraction level. Indeed, design patterns were simply too small to capture design aspects of entire

solution we wanted to offer. Also, design patterns only reuse design, which leaves a lot of implementation effort for developers if they want to have a working solution. This precisely was the reason why we opted for software frameworks - a popular large-scale software artifact that allows the reuse of both design and implementation. As opposed to OO paradigm, solutions from reactive programming paradigm can offer already built-in support for reactive dependencies. However, due to functional/declarative background the paradigm shift can be a challenge to OO developers. Even if developers would decide to sacrifice benefits of OO paradigm for the sake of easier management of reactive dependencies, there is still a question of already existing OO applications and the effort required to rewrite them in a new programming language and a new paradigm. On the other hand, one of the very goals of REFRAME framework was to provide a non-intrusive way to manage reactive dependencies in existing applications, which was demonstrated during TAR on KI Expert Plus application.

As Boix et al [65] reported, in an attempt to get the best from reactive and OO paradigm, some start with reactive programming as a base point and proceed to introduce OO features (e.g. [129]), while others start with OO programming and introduce reactive features (e.g. [65]). REFRAME follows the second approach, as we started from traditional OO programming language, and extended its capabilities for managing reactive dependencies by implementing software framework. Other proposed solutions that attempt at reconciling these two paradigms usually come in a form of special reactive programming languages, or frameworks with custom compilers. As opposed to REFRAME, such solutions due to their own, special compilers do offer tighter and more natural integration of reactive dependencies into a host programming language, which may result in higher-level automation and better performance. However, these solutions are not very flexible in terms of possibility to adapt and extend them to suit specific needs. Conversely, as a part of this dissertation's contribution, we not only offer flexible and open-source implementation of the framework, but we also offer a detailed framework model and requirement specification which can be used to implement custom frameworks. In addition, while other available solutions offer only the core engine for managing reactive dependencies, REFRAME's significant contributions are its accompanying tools for analysis and visualization of dependency graphs. Due to a clean internal design and clean APIs, new tools that utilize existing REFRAME features can be made.

As per third requirement for design science, communication towards practitioners will take

place through official GitHub repository of the framework. Through this repository, developers will foremost be able to fetch source code and binary versions of REFRAME, and access framework documentation. However, they will also be able to actively participate in framework development, e.g. by reporting issues and offering their improvements. With regard to communication with researchers, aside from this dissertation being available at Faculty's repository, dissertation content is also planned to be published in a form of several conference and journal papers.

8.4. Research limitations

Design science paradigm and the chosen methodological framework fit well with the problem addressed by this dissertation, as well as with the proposed solution. However, while conducting individual activities, several limitations can be recognized with regard to research design and the execution. In this section we systematically report these limitations with respect to activity of the chosen methodological framework.

Literature review conducted within this dissertation was thorough and extensive, however, it was not done in a form of systematic literature review. Therefore, there is a chance that some important aspects were not taken into consideration during **explicate problem** activity. Also, this activity is influenced by researcher's own rich experience in practice, which might introduce a bias in terms of understanding and interpreting the problem.

Define requirements activity mirrors the limitations of explicate problem activity. Requirements were gathered and chosen from relevant literature and researcher's personal experience, which again might cause omission of potentially important requirements, and result in incomplete requirements specification. Although requirements were discussed in informal conversations with co-workers, no external experts were formally used in either gathering or evaluating them.

With regard to **design and develop artifact** activity, one of the limitations is the choice of technology (C# .NET) used to implement REFRAME framework. While reasonable effort was made to make implementational aspects of the framework applicable in OO setting in general, some parts of implementation might require significantly different approach in other technologies. Also, since design and development was done iteratively, these technology-specific implementational aspects could have penetrated design as well, and thus made us move away from

abstract and technology-agnostic design. Finally, several practical limitations of the framework were hinted during evaluation activities, including: cumbersome syntax for specifying reactive dependencies, still a lot of boilerplate code is needed, Analyzer tool GUI not being well organized, scarce documentation, etc. As announced, some of these practical limitations are going to be addressed by upcoming framework releases.

Evaluation activities were extensive, and were conducted in accordance with specifically developed research strategy containing four evaluation episodes. Each evaluation episode has its own strengths which complement and enforce the strengths of other episodes. However, they also have limitations with regard to used methods or the way these methods were executed. In **episode I**, we used unit testing to evaluate efficacy of the framework. While with almost a thousand tests we achieved high code coverage, this metric alone cannot guarantee that the framework performs well as a whole. These limitations are partly mitigated in **episode II**, which used 15 illustrative scenarios to demonstrate framework on a more coarse-grained level. However, due to framework complexity and number of features, these 15 scenarios were able to cover only most essential framework features. In **episode III** we conducted focus group to evaluate usefulness of the framework. One limitation related to focus group was the fact that participants only had theoretical knowledge about the framework (i.e. through video materials and presentation). This may have resulted in limited understanding of the framework, and less reliable evaluation. Also there was a possibility for bias because the researcher himself moderated focus group, and interpreted results. Finally, through technical action research, **episode IV** addressed some of the limitation concerns stated for episode III. It enforced the evaluation of usefulness by having participants actively use the framework in a development of real software application. However, since the framework was applied only on one software application, this might pose as a limiting factor in terms of generalizability.

8.5. Future research

While this dissertation describes the process and state of development for initial version of REFRAME software framework, developing software frameworks is an ongoing process. We can classify future efforts with regard to whether they result in practical or knowledge contributions. In this way we can distinguish between practical, combined, and research efforts. Practical efforts involve routine development activities, which do not require research, nor they

result in knowledge contributions. The examples of planned *practical efforts* include: (1) correcting already reported and yet to be discovered bugs, (2) improving graphical user experience in Analyzer tool, (3) increase the number of available analyses, (4) better integration between Analyzer and Visualizer, (5) making framework cross-platform, (6) implementing alternative visualizing technology, etc.

The *combined efforts* involve development activities which significantly increase the practical worth of the framework, but due to overall complexity and novelty of the improvement, also offer knowledge contributions. Such efforts would be typical instances of design science research. These may include researching and developing: (1) custom debugger for REFRAME, (2) syntax enhancement and code generation capabilities using advanced meta-programming capabilities (e.g. .NET compiler API), (3) unit test and documentation generator on the basis of dependency graph, (4) Visualizer based on UML, (5) Visualizer able to generate code, etc.

Finally, purely research efforts refer to those activities, which do not alter REFRAME itself, but rather investigate its quality attributes and effects of its use. For example, some of possible future research includes: (1) performance evaluation, (2) investigation of its influence on code quality (e.g. testability), (3) demonstrations of its potential use in different applications, (4) comparisons with alternative solutions, (5) design patterns and good practices, (6) interaction with other common frameworks, etc.

9. Conclusion

This dissertation addresses the problem of managing reactive dependencies in OO applications. The initial motivation for this topic came from practice, i.e. researcher's personal involvement in development of real software applications. One of these applications represented a problem domain characterized by numerous calculation procedures and mutually dependent calculation parameters. Dependencies between those parameters were *reactive*, i.e. changing the value of one parameter had to result in updating the values of all (directly or indirectly) dependent parameters. Large number of these mutually dependent calculation parameters had a potential to form large graph-like structures, difficult to understand and maintain.

With all the benefits that OO technology offers, it lacks native support for managing reactive dependencies. Most well-known solution applied in such cases is manual implementation of Observer pattern, which is no match to any scenario of managing reactive dependencies, other than the most basic ones. Therefore, the need arose to build more advanced solution for OO setting. Design science, as a problem-solving research paradigm, was suitable option to systematically guide our efforts in achieving both practical and scientific relevance. REFRAME software framework for managing reactive dependencies, as well as other, more abstract development artifacts (e.g. framework model and framework requirements), were direct practical contributions. Scientific contributions, as customary in design science research, came in a form of explicated or embedded prescriptive knowledge. The explicated form was represented by *model* design science artifact, which contained REFRAME's requirements, elaboration of design and implementation decisions, and behavioral and structural characteristics of the framework. On the other hand, embedded knowledge is represented by *instantiation* design science artifact, i.e. implementation of REFRAME software framework. More formally, scientific contributions were in dissertation addressed by stating and answering one main research question (MRQ), six sub-questions (RQ1 - RQ6), and one high-level hypothesis (H1).

In accordance to chosen methodological framework for conducting design science research, the first activity conducted within design science research was *problem explication*. Although the efforts of this activity were summarized in chapter 4, Literature review (chapter 2) is also implicit part of this activity. The first part of literature review (section 2.1) was focused on problem space, i.e. topic of managing reactive dependencies. It provided us with the evidence

of stated problem being relevant for both practice and scientific community. Indeed managing reactive dependencies was in focus of multiple different practical solutions, ranging from design patterns to specialized programming languages. Similarly, numerous research papers in various fields (e.g. reactive programming, event-driven programming, OO design patterns, etc.) were found dealing with this topic. Analyzing literature and personal experience allowed us to broaden our understanding of the problem and the challenges it brings, as well as to reason about its causes and effects. In this way we answered research question **RQ1**. Following that (also in chapter 4), we were able to transition from problem space to solution space, supported by extensive literature review on software frameworks (section 2.2). This helped us legitimize software frameworks as a type of artifact nominally suitable for addressing the stated problem, and in this way answered research question **RQ2**. Overall, activity *Explicate problem* contributed to knowledge related to both managing reactive dependencies and software frameworks.

Having analyzed the causes of the stated problem, and the ways the prospective solution could attend to them, we laid the foundations for the next design science activity - *Define requirements* for the solution. At the beginning of chapter 5 we outlined the basic characteristics of REFRAME as a framework and as a design science artifact, and we also devised a set of five high-level requirements. We then proceeded to specify detailed requirements, which resulted in *Software Requirements Specification document - SRS* (section 5.2) containing 34 functional requirements and 4 non-functional requirements. This provided the direct answer to research question **RQ3**.

In accordance with requirements specified in SRS document, aforementioned REFRAME *model* and *instantiation* artifacts were designed and implemented during *Design and develop artifact* activity. Chapter 6 contains elaborate description of this iterative activity, including the steps of gathering ideas, trying and assessing them, and finally selecting and implementing the most suitable ones. While the model artifact is documented within chapter 6, the instantiation artifact is available in a form of binary files or source code files at REFRAME's GitHub repository (<https://github.com/MarkoMijac/REFRAME.git>). By documenting both the process of building the solution and the solution itself, we provided valuable experience and in this way contributed to answering research question **RQ4**.

Finally, *evaluation* activity was conducted in a form of 4 mutually complementary evaluation episodes. In episode I, through prototyping and exhaustive unit testing (almost 1000 tests)

we provided evidence that REFRAME software framework is *technically feasible*, and that it represents a functioning solution for managing reactive dependencies (*efficacy*). *Technical feasibility* and *efficacy* are further supported in episode II, by implementing and demonstrating 15 common scenarios of REFRAME use. In addition, both episodes I and II contribute to answering research question **RQ4**, while episode II fully answers research question **RQ5**. In order to provide evidence of REFRAME's usefulness in real context, during episodes III and IV we conducted focus group and technical action research respectively. Participants of both focus group and technical action research were resolute in declaring REFRAME as useful for managing reactive dependencies. In this way we answered research question **RQ6**. In addition, discussion with participants also yielded critiques and suggestions which are important for further improvement of REFRAME.

Given that we already addressed six research sub-questions (in Discussion chapter in detail, and also briefly in this chapter), we can now restate and answer the main research question (**MRQ**): "*How can we improve the management of reactive dependencies in the development of object-oriented applications?*". The management of reactive dependencies in development of object-oriented applications can be improved by designing and implementing software framework, which provides dedicated abstractions and mechanisms for specifying individual reactive dependencies, constructing dependency graphs, and performing graph update process. Such framework can be further enriched by accompanying tools, which would provide means to analyze and visualize dependency graphs, and in this way support understanding of the underlying dependency graph as well as its individual parts. Finally, accompanying tool for code generation can improve and speed-up the writing of boilerplate code.

In this dissertation we were able to design, implement and evaluate such solution in a form of REFRAME software framework. We also elaborately documented both the process and the solution. Therefore, we can determine that dissertation's main goal "*Improve and facilitate the management of reactive dependencies in object-oriented applications by designing and evaluating REFRAME software framework, which will allow specification, propagation, visualization, and analysis of reactive dependencies*" is accomplished.

In addition to already confirmed practical and scientific relevance of the problem and proposed solution, the very process of design science research described in this dissertation was a rigorous one. Indeed, designing, developing, and evaluating REFRAME software framework

was guided by 5-activity methodological framework for conducting design science research. Within each activity appropriate and proven methods were used to achieve its set outcomes. This was especially true for evaluation activity, which has decisive role in making design science research a scientific research. With this in mind, a specialized framework for evaluation in design science was used to form rigorous evaluation strategy. This strategy involved conducting 4 complementary evaluation episodes which helped gradually transform REFRAME software framework into a functioning and useful solution. This allows us to confirm hypothesis H1: *"Designed and implemented software application framework (REFRAME) for management of reactive dependencies in development of object-oriented applications will fulfill both relevance and rigor requirements of design science."*

Bibliography

- [1] Asynchronous programming in C#.

<https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/async/>.

- [2] Boo programming language.

<https://boo-language.github.io/>.

- [3] Code coverage testing - Visual Studio.

<https://docs.microsoft.com/en-us/visualstudio/test/using-code-coverage-to-determine-how-much-code-is-being-tested>.

- [4] Covariance and Contravariance (C#).

<https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/covariance-contravariance/>.

- [5] Definition of FRAMEWORK.

<https://www.merriam-webster.com/dictionary/framework>.

- [6] Definition of FRAMEWORK.

<https://en.oxforddictionaries.com/definition/framework>.

- [7] Extension Methods - C# Programming Guide.

<https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/extension-methods>.

- [8] Handling and Raising Events.

<https://docs.microsoft.com/en-us/dotnet/standard/events/>.

- [9] How to: Use Named Pipes for Network Interprocess Communication.

<https://docs.microsoft.com/en-us/dotnet/standard/io/how-to-use-named-pipes-for-network-interprocess-communication>.

- [10] Introduction to Event Listeners (The Java Tutorials > Creating a GUI With JFC/Swing > Writing Event Listeners).
<https://docs.oracle.com/javase/tutorial/uiswing/events/intro.html>.
- [11] ISO/IEC 25010:2011.
<https://www.iso.org/cms/render/live/en/sites/isoorg/contents/data/standard/03/57/35733.html>.
- [12] The .NET Compiler Platform SDK (Roslyn APIs).
<https://docs.microsoft.com/en-us/dotnet/csharp/roslyn-sdk/>.
- [13] The Reactive Manifesto.
<http://www.reactivemanifesto.org/>.
- [14] Reactive Streams.
<http://www.reactive-streams.org/>.
- [15] ReactiveX.
<http://reactivex.io/>.
- [16] Starting to Develop Visual Studio Extensions - Visual Studio.
<https://docs.microsoft.com/en-us/visualstudio/extensibility/starting-to-develop-visual-studio-extensions>.
- [17] 830-1998 - IEEE Recommended Practice for Software Requirements Specifications, 1998.
<http://ieeexplore.ieee.org/servlet/opac?punumber=5841>.
- [18] *Unified Modeling Language*. Object Management Group (OMG), 2017.
- [19] Asynchronous programming patterns, 2018.
<https://docs.microsoft.com/en-us/dotnet/standard/asynchronous-programming-patterns/>.
- [20] Parallel Programming in .NET, 2018.
<https://docs.microsoft.com/en-us/dotnet/standard/parallel-programming/>.

- [21] Topological sorting, 2019.
https://en.wikipedia.org/w/index.php?title=Topological_sorting&oldid=917759838.
- [22] Abdennadher, S. and Fruhwirth, T. *Essentials of Constraint Programming*. Springer, Berlin ; New York, ISBN 978-3-540-67623-2.
- [23] Aguiar, A. and David, G. Patterns for Effectively Documenting Frameworks. In Noble, J., Johnson, R., Avgeriou, P., Harrison, N. B., and Zdun, U., editors, *Transactions on Pattern Languages of Programming II*, volume 6510. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011. ISBN 978-3-642-19431-3 978-3-642-19432-0.
- [24] Avgeriou, P. and Zdun, U., Avgeriou, Paris and Zdun, Uwe, 2005, Architectural Patterns Revisited - A Pattern Language. In *Proceedings of 10th European Conference on Pattern Languages of Programs (EuroPlop 2005)*.
- [25] Axelsen, E. W., Sorensen, F., and Krogdahl, S., Axelsen, Eyvind W. and Sorensen, Fredrik and Krogdahl, Stein, 2009, A reusable observer pattern implementation using package templates, organization.
- [26] Backes, M., Pfitzmann, B., and Waidner, M. A General Composition Theorem for Secure Reactive Systems. In Goos, G., Hartmanis, J., van Leeuwen, J., and Naor, M., editors, *Theory of Cryptography*, volume 2951. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004. ISBN 978-3-540-21000-9 978-3-540-24638-1.
- [27] Bainomugisha, E., Carreton, A. L., Cutsem, T. v., Mostinckx, S., and Meuter, W. d. A survey on reactive programming. *ACM Computing Surveys*, ISSN 03600300.
- [28] Beck, K. *Test-driven development: by example*. Addison-Wesley Professional, 2003.
- [29] Borella, J., Borella, J., 2003, The observer design pattern using aspect oriented programming. In *Proc. of the 2nd Nordic Conference on Pattern Languages of Programs*.
- [30] Borning, A., Borning, Alan, 2016, Wallingford: toward a constraint reactive programming language, organization.

- [31] Bosch Jan, Molin Peter, Mattson Michael, and Bengtsson PerOlof. Object-Oriented Frameworks - Problems & Experiences. Research 9/97, University of Karlskrona/Ronneby, Karlskrona, 1997.
- [32] Boussinot, F. Reactive C: An extension of C to program reactive systems. *Software: Practice and Experience*, ISSN 00380644, 1097024X.
- [33] Chaturvedi, A. and T.V., P. Ontology - Driven Observer Pattern. In *New Trends in Databases and Information Systems*, volume 241. Springer International Publishing, Cham, 2014. ISBN 978-3-319-01862-1 978-3-319-01863-8.
- [34] Cleary, S. *Concurrency in C# Cookbook*. O'Reilly Media, Beijing ; Sebastopol, CA, ISBN 978-1-4493-6756-5.
- [35] Cleven, A., Gubler, P., and Huner, K. M., Cleven, Anne and Gubler, Philipp and Huner, Kai M., 2009, Design Alternatives for the Evaluation of Design Science Research Artifacts. In *Proceedings of the 4th International Conference on Design Science Research in Information Systems and Technology*, DESRIST '09, organization.
- [36] Constant, C., Jeron, T., Marchand, H., and Rusu, V. Integrating formal verification and conformance testing for reactive systems. *IEEE Transactions on Software Engineering*, ISSN 0098-5589.
- [37] Cooper, G. H. and Krishnamurthi, S. Embedding Dynamic Dataflow in a Call-by-Value Language. In *Programming Languages and Systems*, volume 3924. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006. ISBN 978-3-540-33095-0 978-3-540-33096-7.
- [38] Coplien, J., Hoffman, D., and Weiss, D. Commonality and variability in software engineering. *IEEE software*, 15(6):37–45, 1998.
- [39] Courtney, A. Frappe: Functional Reactive Programming in Java. In *Practical Aspects of Declarative Languages*, volume 1990. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001. ISBN 978-3-540-41768-2 978-3-540-45241-6.
- [40] Dams, D., Gerth, R., and Grumberg, O. Abstract interpretation of reactive systems. *ACM Transactions on Programming Languages and Systems*, ISSN 01640925.

- [41] Davis, F. D. Perceived usefulness, perceived ease of use, and user acceptance of information technology. *MIS quarterly*, 1989.
- [42] Della, L. and Clark, D., Della, Lew and Clark, David, 1998, From interface to persistence: a framework for business oriented applications. In *Technology of Object-Oriented Languages, 1998. TOOLS 28. Proceedings*.
- [43] Demetrescu, C., Finocchi, I., and Ribichini, A., Demetrescu, Camil and Finocchi, Irene and Ribichini, Andrea, 2011, Reactive imperative programming with dataflow constraints. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, organization.
- [44] di Battista, G., Eades, P., and Tamassia, R. Layered drawings of digraphs. In *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice Hall, 1999.
- [45] Drechsler, J., Salvaneschi, G., Mogk, R., and Mezini, M., Drechsler, Joscha and Salvaneschi, Guido and Mogk, Ragnar and Mezini, Mira, 2014, Distributed REScala: An Update Algorithm for Distributed Reactive Programming. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA '14*, organization.
- [46] Eales, A., Eales, Andrew, 2005, The Observer Pattern Revisited.
- [47] Elliott, C. and Hudak, P., Elliott, Conal and Hudak, Paul, 1997, Functional Reactive Animation. In *Proceedings of the Second ACM SIGPLAN International Conference on Functional Programming, ICFP '97*, organization.
- [48] Eugster, P. T., Felber, P. A., Guerraoui, R., and Kermarrec, A.-M. The Many Faces of Publish/Subscribe. *ACM Computing Surveys*,
- [49] Evain, J. Cecil.net, 2019.
<https://github.com/jbevain/cecil>.
- [50] Faison, T. *Event-Based Programming*. Apress, 2006. ISBN 978-1-59059-643-2. DOI: 10.1007/978-1-4302-0156-4.
- [51] Fayad, M. and Schmidt, D. C. Object-oriented application frameworks. *Communications of the ACM*, ISSN 00010782.

- [52] Fayad, M., Schmidt, D. C., and Johnson, R. E., editors. *Building application frameworks: object-oriented foundations of framework design*. Wiley, New York, 1999. ISBN 978-0-471-24875-0.
- [53] Feiler, P. and Tichy, W., Feiler, Peter and Tichy, Walter, 1997, Propagator: A family of patterns. In *Technology of Object-Oriented Languages and Systems*, organization.
- [54] Fowler, M. *Patterns of enterprise application architecture*. Addison-Wesley Longman Publishing Co., Inc., 2002.
- [55] Fowler, M. Fluent interface, 2005.
<https://martinfowler.com/bliki/FluentInterface.html>.
- [56] Fowler, M. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 2018.
- [57] Freeman-Benson, B. N. and Borning, A. Integrating constraints with an object-oriented language. In *ECOOP '92 European Conference on Object-Oriented Programming*, number 615 in Lecture Notes in Computer Science. ISBN 978-3-540-55668-8 978-3-540-47268-1.
- [58] Froehlich, G., Hoover, H. J., Liu, L., and Sorenson, P. Designing object-oriented frameworks. *S. Zamir, editor, Handbook of Object-Oriented Technology*, 1997.
- [59] Froehlich, G., Hoover, H. J., Liu, L., and Sorenson, P. Using object-oriented frameworks. *Handbook of Object-Oriented Technology*, 1998.
- [60] Froehlich, G., Hoover, H. J., Liu, L., and Sorenson, P., Froehlich, Gary and Hoover, H James and Liu, Ling and Sorenson, Paul, 1997, Hooking into object-oriented application frameworks. In *Proceedings of the 19th international conference on Software engineering*.
- [61] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1 edition, 1995.
- [62] Garlan, D. and Notkin, D. Formalizing design spaces: Implicit invocation mechanisms. In Goos, G., Hartmanis, J., Prehn, S., and Toetenel, W. J., editors, *VDM'91 Formal*

Software Development Methods, volume 551. Springer Berlin Heidelberg, Berlin, Heidelberg, 1991. ISBN 978-3-540-54834-8 978-3-540-46449-5.

- [63] Garlan, D. and Shaw, M. An introduction to software architecture. In *Advances in software engineering and knowledge engineering*. World Scientific, 1993.
- [64] Gasiunas, V., Satabin, L., Mezini, M., Nunez, A., and Noye, J., Gasiunas, Vaidas and Satabin, Lucas and Mezini, Mira and Nunez, Angel and Noye, Jacques, 2011, EScala: Modular Event-driven Object Interactions in Scala. In *Proceedings of the Tenth International Conference on Aspect-oriented Software Development, AOSD '11*, organization.
- [65] Gonzalez Boix, E., Pinte, K., Van de Water, S., and De Meuter, W., Gonzalez Boix, Elisa and Pinte, Kevin and Van de Water, Simon and De Meuter, Wolfgang, 2013, Object-oriented Reactive Programming is Not Reactive Object-oriented Programming. In *Proceedings of the 1st Workshop on Reactivity, Events, and Modularity*.
- [66] Goodrich, M. T. and Tamassia, R. *Algorithm design and applications*. Wiley Publishing, 2014.
- [67] Hall, G. M. *Adaptive Code: Agile coding with design patterns and SOLID principles*. Microsoft Press, 2017.
- [68] Hannemann, J. and Kiczales, G. Design pattern implementation in Java and aspectJ. *ACM SIGPLAN Notices*, ISSN 03621340.
- [69] Harel, D., Lachover, H., Naamad, A., Pnueli, A., Politi, M., Sherman, R., Shtull-Trauring, A., and Trakhtenbrot, M. STATEMATE: a working environment for the development of complex reactive systems. *IEEE Transactions on Software Engineering*, ISSN 00985589.
- [70] Harel, D. and Pnueli, A. On the Development of Reactive Systems. In *Logics and Models of Concurrent Systems*. Springer Berlin Heidelberg, 1985.
- [71] Hazzard, K. and Bock, J. *Metaprogramming in .NET*. Manning, New York, 2013.
- [72] Heron, T. *Programming with Dependency*. Master Degree, Univesity of Warwick, 2002.

- [73] Hevner, A. R., March, S. T., Park, J., and Ram, S. Design science in information systems research. *MIS Quarterly*, 28(1):75–105, 2004.
- [74] Hinze, A., Sachs, K., and Buchmann, A., Hinze, Annika and Sachs, Kai and Buchmann, Alejandro, 2009, Event-based Applications and Enabling Technologies. In *Proceedings of the Third ACM International Conference on Distributed Event-Based Systems*, DEBS '09, organization.
- [75] Hudak, P., Courtney, A., Nilsson, H., and Peterson, J. Arrows, Robots, and Functional Reactive Programming. In Goos, G., Hartmanis, J., van Leeuwen, J., Juring, J., and Jones, S. L. P., editors, *Advanced Functional Programming*, volume 2638. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003. ISBN 978-3-540-40132-2 978-3-540-44833-4.
- [76] Jarvinen, H., Kurki-Suonio, R., Sakkinen, M., and Systs, K., Jarvinen, H. and Kurki-Suonio, R. and Sakkinen, M. and Systs, K., 1990, Object-oriented specification of reactive systems, organization.
- [77] Jicheng, L., Hui, Y., and Yabo, W., Jicheng, Liu and Hui, Yin and Yabo, Wang, 2010, A novel implementation of observer pattern by aspect based on Java annotation. In *Computer Science and Information Technology (ICCSIT), 2010 3rd IEEE International Conference on*, volume 1, organization.
- [78] Johannesson, P. and Perjons, E. *An introduction to design science*. 2014. ISBN 978-3-319-10632-8 3-319-10632-5 3-319-10631-7 978-3-319-10631-1.
- [79] Johnson, R. E., Johnson, Ralph E, 1992, Documenting frameworks using patterns. In *ACM Sigplan Notices*, volume 27.
- [80] Johnson, R. E. Frameworks=(components+ patterns). *Communications of the ACM*, 40 (10):39–42, 1997.
- [81] Johnson, R. E. and Foote, B. Designing reusable classes. *Journal of object-oriented programming*, 1(2):22–35, 1988.
- [82] Johnson, R. J2ee development frameworks. *Computer*, 38(1):107–110, 2005.

- [83] Kahn, A. B. Topological Sorting of Large Networks. *Commun. ACM*, ISSN 0001-0782.
- [84] Kelleher, C. and Levkowitz, H., Reactive data visualizations.
- [85] Khamis, A. and Abdelmonem, A. The unified software development process and framework development. *Doğuş Üniversitesi Dergisi*, 3(1):109–122, 2011.
- [86] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., and Irwin, J. Aspect-oriented programming. In *ECOOP'97 - Object-Oriented Programming*, number 1241 in Lecture Notes in Computer Science. ISBN 978-3-540-63089-0 978-3-540-69127-3.
- [87] King, N. Using templates in the thematic analysis of text. *Essential guide to qualitative methods in organizational research*, 2004.
- [88] Kobryn, C. Modeling components and frameworks with UML. *Communications of the ACM*, ISSN 00010782.
- [89] Kontio, J., Lehtola, L., and Bragge, J., Kontio, J. and Lehtola, L. and Bragge, J., 2004, Using the focus group method in software engineering: obtaining practitioner and user experiences, organization.
- [90] Krajnc, A. and Hericko, M. *Classification of object-oriented frameworks*, volume 2. IEEE, 2003.
- [91] Landin, N., Niklasson, A., Bosson, G., and Regnell, B. Development of object-oriented frameworks. *Department of Communication System. Lund Institute of Technology, Lund University. Lund, Sweden*, 1995.
- [92] Lopes, S., Tavares, A., Monteiro, J., and Silva, C., Sergio Lopes and Adriano Tavares and Joao Monteiro and Carlos Silva, 2006, Describing framework static structure: promoting interfaces with uml annotations. In *In Proc. of the 11 th International Workshop on Component-Oriented Programming of the 20 th ECOOP*.
- [93] Lopes, S. F., Afonso, F., Tavares, A., and Monteiro, J., Lopes, Sérgio F and Afonso, Francisco and Tavares, Adriano and Monteiro, João, 2009, Framework characteristics-a starting point for addressing reuse difficulties. In *2009 Fourth International Conference on Software Engineering Advances*.

- [94] Lopes, S. F., Tavares, A., Silva, C., and Monteiro, J. L., Lopes, Sérgio F and Tavares, AC and Silva, CA and Monteiro, João L, 2005, Application development by reusing object-oriented frameworks. In *Computer as a Tool, 2005. EUROCON 2005. The International Conference on*, volume 1.
- [95] Maier, I., Rompf, T., and Odersky, M. Deprecating the observer pattern. Technical report EPFL-REPORT-148043, Ecole Polytechnique Federale de Lausanne, Lausanne, Switzerland, 2010.
- [96] March, S. T. and Smith, G. F. Design and natural science research on information technology. *Decision support systems*, 15(4):251–266, 1995.
- [97] Margara, A. and Salvaneschi, G., Margara, A. and Salvaneschi, G., 2013, Ways to react: Comparing reactive languages and complex event processing. In *Proceedings of the 1st Workshop on Reactivity, Events, and Modularity*.
- [98] Markiewicz, M. E. and de Lucena, C. J. P. Object oriented framework development. *Crossroads*, ISSN 15284972.
- [99] Martin, R. *Clean architecture*. Prentice Hall, 2017.
- [100] Martin, R. C. *Clean code: a handbook of agile software craftsmanship*. Pearson Education, 2009.
- [101] Martin, R. C. and Martin, M. *Agile principles, patterns, and practices in C# (Robert C. Martin)*. Prentice Hall PTR, 2006.
- [102] Mattsson, M. Object-oriented frameworks. *Licentiate thesis*, 1996.
- [103] Mattsson, M., Bosch, J., and Fayad, M. E. Framework integration problems, causes, solutions. *Communications of the ACM*, 42(10):80–87, 1999.
- [104] Meyer, B. *Object-oriented software construction*, volume 2. Prentice hall New York, 1988.
- [105] Meyerovich, L. A., Guha, A., Baskin, J., Cooper, G. H., Greenberg, M., Bromfield, A., and Krishnamurthi, S., Meyerovich, Leo A. and Guha, Arjun and Baskin, Jacob and Cooper, Gregory H. and Greenberg, Michael and Bromfield, Aleks and Krishnamurthi,

- Shriram, 2009, Flapjax: A Programming Language for Ajax Applications. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA '09*, organization.
- [106] Mijač, M., Mijač, Marko, 2019, Evaluation of design science instantiation artifacts in software engineering research. In *Central European Conference on Information and Intelligent Systems*.
- [107] Mijač, M., Kermek, D., and Stapić, Z., Complex propagation of events - Design pattern comparison. In *Proceedings of 23rd International Conference on Information System Development*.
- [108] Moore, E. F., Moore, Edward F, 1959, The shortest path through a maze. In *Proc. Int. Symp. Switching Theory, 1959*.
- [109] Murray, L., Carrington, D., and Strooper, P., Murray, Leesa and Carrington, David and Strooper, Paul, 2004, An approach to specifying software frameworks. In *Proceedings of the 27th Australasian Conference on Computer Science - Volume 26, ACSC '04*, organization.
- [110] Noda, N. and Kishi, T., Noda, Natsuko and Kishi, Tomoji, 2001, Implementing Design Patterns Using Advanced Separation of Concerns. In *In: OOPSLA 2001 Workshop on AsoC in OOS (2001)*.
- [111] Notkin, D., Garland, D., Griswold, W. G., and Sullivan, K. Adding implicit invocation to languages: Three approaches. In Nishio, S. and Yonezawa, A., editors, *Object Technologies for Advanced Software*, number 742 in Lecture Notes in Computer Science. ISBN 978-3-540-57342-5 978-3-540-48075-4.
- [112] Osherove, R. *The Art of Unit Testing: With Examples in. Net*. Manning Publications Co., 2009.
- [113] Parent, S. *A Possible Future of Software Development*, 2007.
- [114] Parsons, D., Rashid, A., Speck, A., and Telea, A., Parsons, D. and Rashid, A. and Speck, A. and Telea, A., 1999, A "framework" for object oriented frameworks design, organization.

- [115] Peffers, K., Rothenberger, M., Tuunanen, T., and Vaezi, R. Design Science Research Evaluation. In Peffers, K., Rothenberger, M., and Kuechler, B., editors, *Design Science Research in Information Systems. Advances in Theory and Practice*, Lecture Notes in Computer Science.
- [116] Peffers, K., Tuunanen, T., Rothenberger, M. A., and Chatterjee, S. A Design Science Research Methodology for Information Systems Research. *Journal of Management Information Systems*, ISSN 0742-1222.
- [117] Piveta Kessler, E. and Zancanella, L. C. Observer pattern using aspect-oriented programming. *Scientific Literature Digital Library*, 2003.
- [118] Polančič, G., Horvat, R. V., and Rozman, I. Improving object-oriented frameworks by considering the characteristics of constituent elements. *Journal of Information Science and Engineering*, 25:1067–1085, 2009.
- [119] Prat, N., Comyn-Wattiau, I., and Akoka, J. A Taxonomy of Evaluation Methods for Information Systems Artifacts. *Journal of Management Information Systems*, ISSN 0742-1222, 1557-928X.
- [120] Pree, W. and Koskimies, K. Framelets - small and loosely coupled frameworks. *ACM Computing Surveys (CSUR)*, 32(1es):6, 2000.
- [121] Raymond, P., Nicollin, X., Halbwachs, N., and Weber, D., Raymond, P. and Nicollin, X. and Halbwachs, N. and Weber, D., 1998, Automatic testing of reactive systems, organization.
- [122] Riehle, D., Riehle, Dirk, 1996, The Event Notification Pattern - Integrating Implicit Invocation with Object-Orientation.
- [123] Riehle, D. and Gross, T. Role model based framework design and integration. *ACM SIGPLAN Notices*, ISSN 03621340.
- [124] Roberts, D. and Johnson, R., Don Roberts and Ralph Johnson, 1996, Evolving frameworks: A pattern language for developing object-oriented frameworks. In *Proceedings of the Third Conference on Pattern Languages and Programming*, organization.

- [125] Salvaneschi, G., Margara, A., and Tamburrelli, G., Salvaneschi, G. and Margara, A. and Tamburrelli, G., 2015, Reactive Programming: A Walkthrough. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering (ICSE)*, volume 2.
- [126] Salvaneschi, G., Amann, S., Proksch, S., and Mezini, M., Salvaneschi, Guido and Amann, Sven and Proksch, Sebastian and Mezini, Mira, 2014, An empirical study on program comprehension with reactive programming. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, organization.
- [127] Salvaneschi, G., Drechsler, J., and Mezini, M. Towards Distributed Reactive Programming. In Nicola, R. D. and Julien, C., editors, *Coordination Models and Languages*, number 7890 in Lecture Notes in Computer Science. ISBN 978-3-642-38492-9 978-3-642-38493-6.
- [128] Salvaneschi, G., Eugster, P., and Mezini, M. Programming with Implicit Flows. *IEEE Software*, ISSN 0740-7459.
- [129] Salvaneschi, G. and Mezini, M., Salvaneschi, Guido and Mezini, Mira, 2013, Reactive Behavior in Object-oriented Applications: An Analysis and a Research Roadmap, organization.
- [130] Salvaneschi, G. and Mezini, M., Salvaneschi, Guido and Mezini, Mira, 2016, Debugging for reactive programming. In *Proceedings of the 38th International Conference on Software Engineering*, organization.
- [131] Samuel-Ojo, O., Olfman, L., Reinen, L. A., Flenner, A., Oglesby, D. D., and Funning, G. J. Design Methodology for Construction of Mapping Applications. In *Design Science at the Intersection of Physical and Virtual Design*, volume 7939. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.
- [132] Sanada, Y. and Adams, R. Representing design patterns and frameworks in uml - towards a comprehensive approach. *Journal of Object Technology*, 1(2):143–154, 2002.

- [133] Santos, A. L., Lopes, A., and Koskimies, K., Santos, André L and Lopes, Antónia and Koskimies, Kai, 2007, Framework specialization aspects. In *Proceedings of the 6th international conference on Aspect-oriented software development*.
- [134] Schmidt, D. C. and Buschmann, F., Schmidt, Douglas C. and Buschmann, Frank, 2003, Patterns, frameworks, and middleware: Their synergistic relationships. In *Proceedings of the 25th International Conference on Software Engineering, ICSE '03*, organization.
- [135] Schmidt, D. C., Gokhale, A., and Natarajan, B. Leveraging application frameworks. *Queue*, 2(5):66, 2004.
- [136] Schuster, C. and Flanagan, C., Schuster, Christopher and Flanagan, Cormac, 2016, Reactive programming with reactive variables, organization.
- [137] Sommerville, I. *Software engineering*. Pearson, Boston, 9th ed edition, 2011. ISBN 978-0-13-703515-1 978-0-13-705346-9.
- [138] Sonnenberg, C. and vom Brocke, J. Evaluation patterns for design science research artefacts. In *Practical Aspects of Design Science*. Springer.
- [139] Sparks, S., Benner, K., and Faris, C. Managing object oriented framework reuse. *Computer*, 29(9):52–61, 1996.
- [140] Srinivasan, S. Design patterns in object-oriented frameworks. *Computer*, 32(2):24–32, 1999.
- [141] Srinivasan, S. and Vergo, J., Srinivasan, Savitha and Vergo, John, 1998, Object oriented reuse: experience in developing a framework for speech recognition applications. In *Proceedings of the 20th international conference on Software engineering*.
- [142] Stanojević, V., Vlajić, S., Milić, M., and Ognjanović, M., Stanojević, Vojislav and Vlajić, Siniša and Milić, Miloš and Ognjanović, Marina, 2011, Guidelines for framework development process. In *Software Engineering Conference in Russia (CEE-SECR), 2011 7th Central and Eastern European*.
- [143] Steimann, F., Pawlitzki, T., Apel, S., and Kastner, C. Types and modularity for implicit invocation with implicit announcement. *ACM Transactions on Software Engineering and Methodology*, ISSN 1049331X.

- [144] Štuikys, V. and Damaševičius, R. Taxonomy of fundamental concepts of meta-programming. In *Meta-Programming and Model-Driven Meta-Program Development*. Springer, 2013.
- [145] Syromiatnikov, A. and Weyns, D., A Journey through the Land of Model-View-Design Patterns. In *2014 IEEE/IFIP Conference on Software Architecture (WICSA)*.
- [146] Szyperski, C., Gruntz, D., and Murer, S. *Component software: beyond object-oriented programming*. Addison-Wesley Component software series. Addison-Wesley, London, 2nd ed edition, 1998. ISBN 978-0-321-75302-1. OCLC: 838151413.
- [147] Taligent. Building object-oriented frameworks. Technical report, IBM, 1994.
- [148] Tarjan, R. E. Edge-disjoint spanning trees and depth-first search. *Acta Informatica*, 6(2): 171–185, 1976. ISSN 0001-5903, 1432-0525.
- [149] Tennyson, M. F., A study of the data synchronization concern in the Observer design pattern, organization.
- [150] Tremblay, M., Hevner, A., and Berndt, D. Focus Groups for Artifact Refinement and Evaluation in Design Research. *Communications of the Association for Information Systems*, ISSN 1529-3181.
- [151] Tsvetinov, N. *Learning Reactive Programming with Java 8*. Packt Publishing Ltd, 2015.
- [152] van den Broecke, J. A. and Coplien, J. O. Using design patterns to build a framework for multimedia networking. *Bell Labs Technical Journal*, 2(1):166–187, 1997.
- [153] van Gorp, J. and Bosch, J. Design, implementation and evolution of object oriented frameworks: concepts and guidelines. *Software: Practice and Experience*, ISSN 0038-0644, 1097-024X.
- [154] Venable, J., Pries-Heje, J., and Baskerville, R. FEDS: a Framework for Evaluation in Design Science Research. *European Journal of Information Systems*, 2014. ISSN 0960-085X.
- [155] W.B. Frakes and Kyo Kang. Software reuse research: status and future. *IEEE Transactions on Software Engineering*, ISSN 0098-5589. SCHOLAR.

- [156] Wieringa, R. *Design methods for reactive systems: Yourdan, Statemate, and the UML*. Morgan Kaufmann Publishers, Boston, 2003. ISBN 978-1-55860-755-2.
- [157] Wieringa, R. and Morali, A. Technical Action Research as a Validation Method in Information Systems Design Science. In Peffers, K., Rothenberger, M., and Kuechler, B., editors, *Design Science Research in Information Systems. Advances in Theory and Practice*, number 7286 in Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2012. ISBN 978-3-642-29862-2 978-3-642-29863-9.
- [158] Wieringa, R. J. Technical Action Research. In *Design Science Methodology for Information Systems and Software Engineering*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014. ISBN 978-3-662-43838-1 978-3-662-43839-8. DOI: 10.1007/978-3-662-43839-8_19.
- [159] Yixing, X. and Yaowu, C., Yixing, Xia and Yaowu, Chen, 2007, A component-based framework for embedded digital instrumentation software with design patterns. In *Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing, 2007. SNPD 2007. Eighth ACIS International Conference on*, volume 2.
- [160] Zanarini, D., Jaskelioff, M., and Russo, A., Precise Enforcement of Confidentiality for Reactive Systems, organization.
- [161] Zhang, W. and Kim, M., Zhang, Wusheng and Kim, Mik, 2005, Application frameworks technology in theory and practice'. In *Proceedings of the Fifth International Conference on Electronic Business*.
- [162] Zhuang, Y. Y. and Chiba, S. Expanding Event Systems to Support Signals by Enabling the Automation of Handler Bindings. *Journal of Information Processing*, 24(4):620–634, 2016. ISSN 1882-6652.
- [163] Zhuang, Y. and Chiba, S., Zhuang, YungYu and Chiba, Shigeru, 2012, Supporting methods and events by an integrated abstraction, organization.
- [164] Zhuang, Y. and Chiba, S. Method Slots: Supporting Methods, Events, and Advices by a Single Language Construct. In *Transactions on Aspect-Oriented Software Development*

XI, volume 8400. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014. ISBN 978-3-642-55098-0 978-3-642-55099-7.

- [165] Zhuang, Y. and Chiba, S., Enabling the Automation of Handler Bindings in Event-Driven Programming. In *Computer Software and Applications Conference*, organization.

Curriculum vitae

Marko Mijač was born on the 30th of October 1985. He graduated at Faculty of Organization and Informatics, University of Zagreb, where he currently works as a teaching assistant. He is the member of Department for Information Systems Development, and is involved in teaching of Software engineering, Business Information Systems, and Geographical Information Systems courses. Before joining the faculty, he worked as a software developer on developing production information system in Boxmark Leather d.o.o., as well as on other smaller projects. Also, during his work at the faculty, he participated in numerous projects, such as: Development and maintenance of KI Expert Plus software; MEDINFO - Curriculum Development for Interdisciplinary Postgraduate Specialist Study in Medical Informatics; IRI Hyper, User Experience of the Future - Smart Specialization and Contemporary Communication and Collaboration Technology, and others.

His area of interest is a broad field of software engineering and information systems development. In particular, he is interested in software frameworks, software architectures, design patterns, and other forms of code and design reuse. He has authored numerous scientific and professional papers.

LIST OF PUBLICATIONS

- [1] Mijač, M., García-Cabot, A., Strahonja, V. (2021). Reactor design pattern. TEM Journal, 10(1) (Accepted for publication)
- [2] Stapić, Z., Lechner, N. H., & Mijač, M. (2020). Software engineering knowledge transfer channels between university and medical device industry: a gap analysis. Fachtagung des GI-Fachbereichs Softwaretechnik 24.-28. Februar 2020 Innsbruck, Austria, 183.
- [3] Mijač, M. (2019). Evaluation of Design Science instantiation artifacts in Software engineering research. In Central European Conference on Information and Intelligent Systems (pp. 313-321). Faculty of Organization and Informatics Varazdin.
- [4] Mijač, M., Picek, R., & Andročec, D. (2019). Determinants of ERP Systems as a Large-Scale Reuse Approach. In MATEC Web of Conferences (Vol. 292, p. 03007). EDP Sciences.

- [5] Andročec, D., Picek, R., & Mijač, M. (2018). The ontologically based model for the integration of the IoT and Cloud ERP services. In Proceedings of the 8th International Conference on Cloud Computing and Services Science (pp. 481-488).
- [6] Mijač, M., Andročec, D., & Picek, R. (2017). Smart city services driven by IoT: A systematic review. *Journal of Economic and Social Development*, 4(2), 40-50.
- [7] Picek, R., Mijač, M., & Andročec, D. (2017). Acceptance of Cloud ERP systems in Croatian companies: Analysis of key drivers and barriers. *Economic and Social Development: Book of Proceedings*, 513-522.
- [8] Stapić, Z., Mijač, M., & Strahonja, V. (2016, May). Methodologies for development of mobile applications. In 2016 39th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO) (pp. 688-692). IEEE.
- [9] Picek, R., Mijač, M., & Andročec, D. (2015). Use of business information systems in Croatian companies. *Economic and Social Development: Book of Proceedings*, 154.
- [10] Mijač, M., & Stapić, Z. (2015, September). Reusability metrics of software components: survey. In Proceedings of the 26th Central European Conference on Information and Intelligent Systems (pp. 221-231).
- [11] Tepavac I., Valjevac K., Kliba S., and Mijač M. (2015). Version Control Systems, Tools and Best Practices: Case Git CASE 27, Zagreb
- [12] Šimić D., Deželić G., Vondra P., Jovanović M., Mijač M., Hercigonja-Szekeres M., Božikov J., Kern J. (2015). "Medicinska informatika - Kvalifikacije i zanimanja", Faculty of Organization and Informatics, Varaždin (Book chapter)
- [13] Mijač, M., Kermek, D., & Stapić, Z. (2014). Complex Propagation of Events: Design Patterns Comparison. In Proceedings of the 23rd International Conference on Information Systems Development (ISD2014 Croatia) (p. 306).
- [14] Alen, H., Macan, D. A., Antolović, Z., Tomaš, B., & Mijač, M. (2014). Image pattern recognition using mobile devices. *Razvoj poslovnih i informacijskih sustava CASE 26*, 107.

- [15] Mijač, M., Picek, R., & Stapić, Z. (2013). Cloud ERP System Customization Challenges. In Central European Conference on Information and Intelligent Systems.
- [16] Stapić, Z., Mijač, M., & Tomaš, B. (2013). Monetizing Mobile Applications. Razvoj poslovnih i informacijskih sustava CASE 25, 61.
- [17] Šaško, Z., Mijač, M., Stapić, Z., Domínguez Díaz, A., & Saenz de Navarrete Royo, J. (2012). Windows Phone 7 Applications development using Windows Azure Cloud. Razvoj poslovnih i informacijskih sustava CASE 24.
- [18] Mijač M. (2009). Razvoj aplikacije za veleprodajno poslovanje, koja se oslanja na objektno-relacijsku bazu podataka, Faculty of Organization and Informatics (Graduate thesis)