Arhitekturalni dizajn aplikacije za vođenje evidencije o specijalističkom usavršavanju doktora medicine

Capek, Dino

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Dino Capek

ARHITEKTURALNI DIZAJN APLIKACIJE ZA VOĐENJE EVIDENCIJE O SPECIJALISTIČKOM USAVRŠAVANJU DOKTORA MEDICINE

ZAVRŠNI RAD

Varaždin, 2022.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Dino Capek

JMBAG: 0016142534

Studij: Informacijski sustavi

ARHITEKTURALNI DIZAJN APLIKACIJE ZA VOĐENJE EVIDENCIJE O SPECIJALISTIČKOM USAVRŠAVANJU DOKTORA MEDICINE

ZAVRŠNI RAD

Mentor/Mentorica:

Dr. sc. Marko Mijač

Varaždin, rujan 2022.

Dino Capek

Izjava o izvornosti

Izjavljujem da je moj završni rad izvorni rezultat mojeg rada te da se u izradi istoga nisam koristio drugim izvorima osim onima koji su u njemu navedeni. Za izradu rada su korištene etički prikladne i prihvatljive metode i tehnike rada.

Autor potvrdio prihvaćanjem odredbi u sustavu FOI-radovi

Sažetak

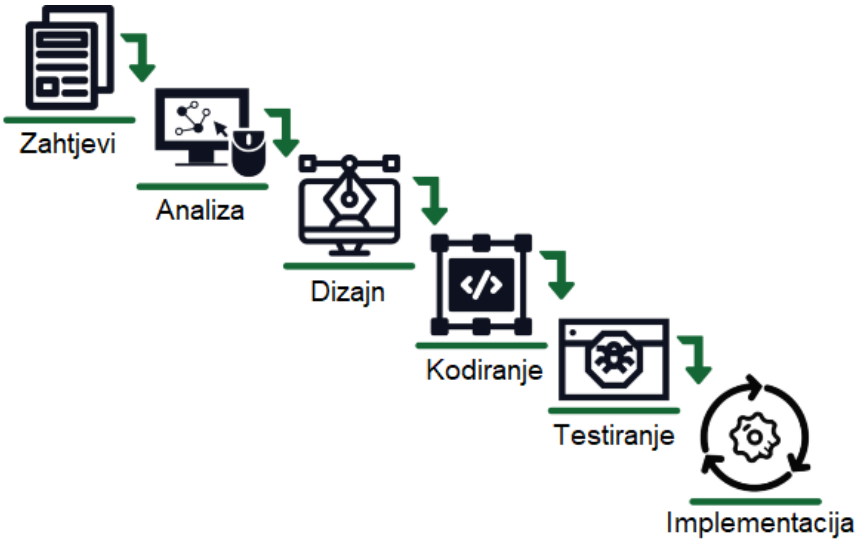
Proces razvoja aplikacija sastoji se od nekoliko faza, među kojima je i arhitekturalni dizajn. Arhitektura sustava, ili aplikacije, je vrlo važan dio u razvoju jer predstavlja strukturu aplikacije, definira kriterije uspješnosti, te pravila i preporuke po kojima se aplikacija gradi. Uz dobro definiranu strukturu, kriterije, pravila i preporuke, faza implementacije „teći“ će bez većih problema, što smanjuje troškove razvoja aplikacije, a olakšava i sam razvoj jer je probleme teže i skuplje uklanjati u kasnijim fazama. Iz tog razloga, cilj ovog završnog rada je pojasniti dijelove arhitekture sustava, te ući u detalje samih dijelova i na kraju naučeno primijeniti na dizajniranju arhitekture aplikacije za vođenje evidencije specijalizanata medicine.

Ključne riječi: arhitekturalni dizajn; arhitektura sustava; struktura; karakteristike; odluke; načela dizajna; aplikacija

Sadržaj

1. Uvod

Proces razvoja aplikacije sastoji se od nekoliko faza razvoja koje kada završe, kao rezultat daju funkcionalnu aplikaciju. Broj faza i način njihovih izvršavanja ovisi o odabranoj metodici razvoja aplikacije. Metodika razvoja aplikacije ima puno, a prilikom razvoja aplikacije biramo onu koja najbolje odgovara našim aplikacijskim zahtjevima. Neke od poznatijih metodika razvoja aplikacija su vodopadni model, RUP, XP, te Scrum. Za primjer, uzmimo sada vodopadni model kako bi proučili njegove faze. Vodopadni model sastoji se od 6 faza, te se svaka faza izvršava samo jednom, što je nedostatak s obzirom na agilne metode kao što je Scrum, gdje se svaka faza izvršava nekoliko puta. Kraj jedne faze označava početak druge faze. Dakle prva faza je faza planiranja. U ovoj fazi zadaju se ciljevi, određuju resursi, potrebno vrijeme, zadaje se strategija, taktika i određuju se operacije. Nakon što završi faza planiranja, kreće faza analize. U ovoj fazi se analiziraju i specificiraju performanse, sučelja, funkcionalnosti, tehnologije, podatkovni sadržaji, a kreira se i model podataka i procesa. Nakon ove faze slijedi faza dizajna. Faza dizajna će biti detaljno objašnjena i analizirana kroz nadolazeća poglavlja, a ono najvažnije što se u ovoj fazi radi je definiranje arhitekture sustava i kreiranja plana razvoja aplikacije. Arhitektura sustava je jedan od najvažnijih koraka u procesu razvoja aplikacije. Možemo ju usporediti s arhitekturom građevina. Prije nego započne građa, na primjer kuće, arhitekt prvo kreira planove za kuću, takozvane arhitektonske nacrte po kojima se kasnije gradi kuća. Važno je da je arhitektura kuće pravilno napravljena jer su izmjene u procesu gradnje kuće vrlo skupe, ili čak nemoguće, a također može doći i do loše izgrađene kuće koja se može urušiti. Sve to vrijedi i za arhitekturu sustava. Važno je imati dobru arhitekturu kako bi troškovi prilikom daljnjeg razvoja bili manji, te kako bi projekt uspješno završio. Završavanjem faze dizajna započinje faza kodiranja. U ovoj fazi se s obzirom na zahtjeve definirane u prijašnjim fazama piše kod, te se realiziraju funkcionalnosti i korisnička sučelja. Nakon faze kodiranja slijedi faza testiranja, gdje se početne verzije aplikacije daju testerima na testiranje, te se rješavaju pronađeni problemi. Završetkom faze testiranja dolazi zadnja faza, faza uvođenja. U ovoj fazi se aplikacija pušta u korištenje te se dalje radi na njenom održavanju (Zulqadar, 2019).



Slika 1: Vodopadni model *(Zulqadar, 2019)*

Ovo je ukratko objašnjen proces razvoja aplikacije uz pomoć vodopadne metode, a u radu će detaljno biti objašnjena faza koju imaju sve metodike razvoja, a to je faza dizajna. Specifičnije arhitekturalni dizajn sustava koji je dio faze dizajna, te sama arhitektura sustava koja je rezultat procesa arhitekturalnog dizajna. Teorijski dio o dizajnu aplikacije će zatim biti iskorišten i prikazan na praktičnom primjeru gdje će se detaljno provesti dizajn aplikacije za vođenje evidencije specijalizanata medicine.

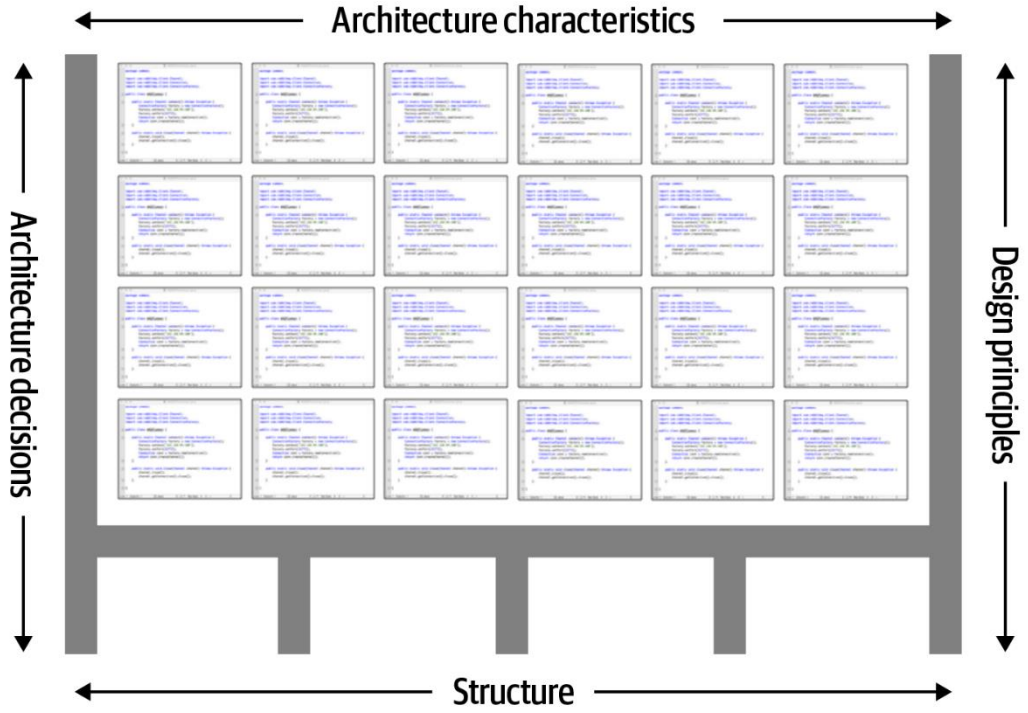
Ovaj rad pisan je u Microsoft Word aplikaciji po predlošku dostupnom na web stranicama Fakulteta organizacije i informatike. Za referenciranje je korišten alat unutar Word-a koji omogućuje dodavanje izvora i automatsko kreiranje tablice izvora na kraju rada. Od svih izvora najveći uzor kod pisanja ovog rada bila je knjiga Neal Ford-a i Mark Richards-a „Fundamentals of Software Architecture“ u kojoj autori daju svoj pogled na arhitekturu aplikacija. Praktični dio rada rađen je u Visual Studio-u, razvojnom okruženju ravzijenom od strane Microsoft-a koji se koristi za razvoj Windows aplikacija, kao u ovom radu za Windows Forms aplikaciju, web aplikacija i mobilnih aplikacija. Za izradu same aplikacije korištena su znanja i vještine stečene na kolegiju Programsko inženjerstvo, uz poneko provjeravanje online tutorijala i stranica kao što su Stack Overflow.

2. Arhitekturalni dizajn aplikacije

Arhitekturalni dizajn aplikacije dio je faze dizajna u procesu razvoja aplikacije, a kao rezultat arhitekturalnog dizajna dobivamo arhitekturu aplikacije. Stručnjaci u ovom području još nisu točno definirali što je arhitektura aplikacije. Neki kažu da je to nacrt budućeg sustava, a drugi da je to putokaz, tj. smjernice, koje će nas dovesti do krajnjeg sustava (Richards & Ford, Fundamentals of Software Architecture, 2020). No po (Sangwan, 2014) arhitektura sustava je skup elemenata sustava, povezanost tih elemenata i načini na koje ti elementi međusobno komuniciraju kako bi ostvarili tražene funkcionalnosti.

2.1. Arhitektura sustava

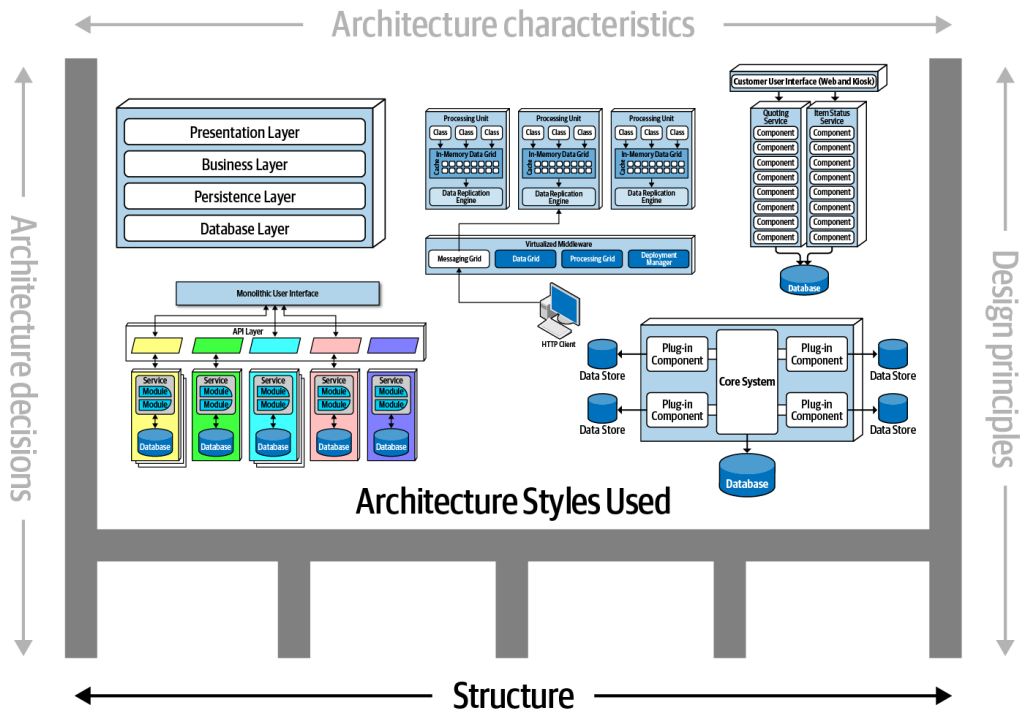
Prijašnje definicije nam u grubo govore što je arhitektura sustava, no ne i od čega se ona sastoji, tj. što sve spada u arhitekturu sustava. Richards i Ford u svojoj knjizi (Richards & Ford, Fundamentals of Software Architecture, 2020) daju nam jedan on načina na koji možemo shvaćati arhitekturu sustava. Dakle po njima arhitektura sustava sastoji se od četiri dijela: strukture sustava kao glavnog dijela, arhitekturalnih karakteristika koje sustav mora podržavati, arhitekturalnih odluka i načela dizajna (Richards & Ford, Fundamentals of Software Architecture, 2020).



Slika 2: Arhitektura sustava po Richards-u i Ford-u *(Richards & Ford, Fundamentals of Software Architecture, 2020)*

2.1.1. Struktura sustava

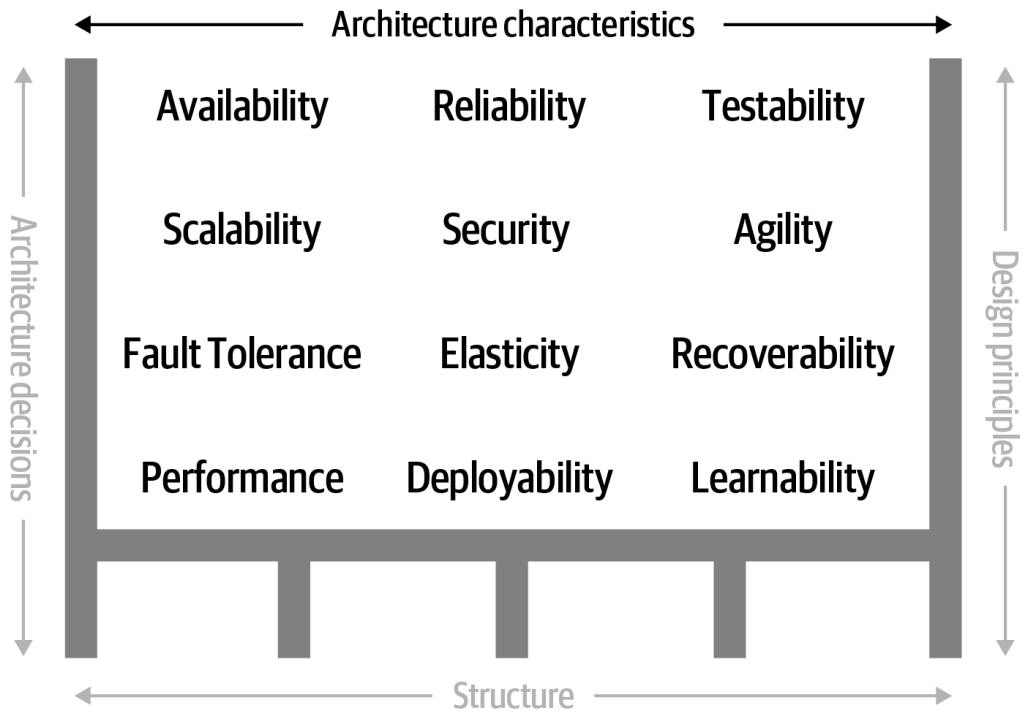
Struktura sustava, kao glavni dio arhitekture, odnosi se na vrstu, tj. arhitekturalni stil ili uzorak koji će biti implementiran u sustavu. „Arhitekturalni uzorak je dokazano rješenje na ponavljajuće probleme koji se javljaju kod specifičnih konteksta dizajna.“ (Sangwan, 2014). Dakle arhitekturalni uzorak je rješenje nekog softverskog problema kojeg uzimamo i oblikujemo naše rješenje po njemu, ako je naš problem sličan problemu za koje je to rješenje napravljeno. Uzmimo za primjer slojevit pogled („Layered View“) koji je jedan od mnogih uzoraka. Ovaj uzorak se koristi kod poznatog OSI modela koji se koristi u mrežnoj arhitekturi. OSI model se sastoji od sedam slojeva: fizički sloj, podatkovni sloj, mrežni sloj, transportni sloj, sloj sesije, prezentacijski sloj i aplikacijski sloj. Svaki sloj na višoj razini, aplikacijski sloj je na najvišoj razini, ovisi o sloju na nižoj razini, fizički sloj je na najnižoj razini, što je ukratko definicija slojevitog pogleda ili stila. Više o ovom sloju i drugim klasičnim slojevima bit će obrađeno kasnije u radu. Postavimo se sada u poziciju gdje razvijamo neku aplikaciju koja će se sastojati od slojeva. U tom slučaju da ne bismo morali sami iz „nule“ osmišljavati strukturu sustava, jednostavno uzimamo provjereno rješenje za taj dizajn aplikacije, tj. sustava i time smo uštedjeli na vremenu, a manje vrijeme razvoja znače manji troškovi.



Slika 3: Različiti arhitekturalni stilovi *(Richards & Ford, Fundamentals of Software Architecture, 2020)*

2.1.2. Arhitekturalne karakteristike

Arhitekturalne karakteristike su vrlo važan dio arhitekture sustava jer definiraju kriterije uspješnosti. Za njihovo definiranje nije nam nužno potrebno znanje o funkcionalnosti sustava kojeg razvijamo, ali su i dalje potrebne kako bi se osiguralo da razvijani sustav radi pravilno (Richards & Ford, Fundamentals of Software Architecture, 2020). Same karakteristike i njihov broj u arhitekturi sustava se razlikuju od autora do autora, no po (Richards & Ford, Fundamentals of Software Architecture, 2020) možemo ih svrstati u tri grupe, operativne arhitekturalne karakteristike, strukturne arhitekturalne karakteristike i unakrsne arhitekturalne karakteristike. Neke od važnijih arhitekturalnih karakteristika su dostupnost („Availability“), pouzdanost („Reliability“), provjerljivost („Testability“), skalabilnost („Scalability“), sigurnost („Security“), agilnost („Agility“), tolerancija na greške („Fault Tolerance“), elastičnost („Elasticity“), mogućnost oporavka („Recoverability“), performanse („Performance“), mogućnost uvođenja („Deployability“) i mogućnost učenja („Learnability“).



Slika 4: Arhitekturalne karakteristike *(Richards & Ford, Fundamentals of Software Architecture, 2020)*

Dostupnost – govori koliko će vremena sustav biti dostupan. Dogovara se s naručiteljem sustava te je potrebno predvidjeti i odrediti vrijeme kada sustav neće biti dostupan, tako zvani „downtime“ sustava, zbog održavanja ili greške u radu.

Pouzdanost – govori koliko je sustav pouzdan u kritičnim situacijama. Na primjer ako je sustav neizostavan dio poslovanja neke tvrtke, ako bez njega tvrtka ne može obavljati posao, sustav mora biti vrlo pouzdan te moraju postajati sigurnosne mjere koje će osigurati da sustav radi i u kriznim situacijama.

Provjerljivost – govori o mogućnosti arhitekture sustava i njezinih elemenata da budu provjerljivi (da ih se može testirati), tj. da se može provjeriti daju li elementi sustava očekivane rezultate. U visoko provjerljivim sustavima elementi se mogu testirati, te se ako test ne bude zadovoljavajuć, lako pronalazi greška nakon čega se ona ispravlja (Heusser, 2020).

Skalabilnost – govori o mogućnosti sustava da funkcionira dobro i kako je zamišljeno čak i kada se opterećenje sustava povećava (na primjer više korisnika, više zahtjeva, veći broj operacija, itd.) (Richards & Ford, Fundamentals of Software Architecture, 2020)

Sigurnost – je jedna od najvažnijih karakteristika. Govori o sigurnosti sustava od napada i neautoriziranog korištenja. Siguran sustav je siguran od vanjskih napada, a također sprječava rad ako osoba nije autorizirana za rad u njemu.

Agilnost – mogućnost arhitekture da jednostavno provede promjene u bilo kojem dijelu

Tolerancija na greške – govori o mogućnosti sustava da radi dobro i kako je zamišljeno usprkos greškama

Elastičnost – slično kao i skalabilnost, no za kraći vremenski period. Dakle to je sposobnost sustava da funkcionira dobro i kako je zamišljeno čak i kad se trenutno i brzo poveća opterećenje

Mogućnost oporavka – govori o vremenu potrebnom da se sustav oporavi nakon nekog incidenta. Dakle vrijeme potrebno da se sustav ponovo dovede u operativno stanje.

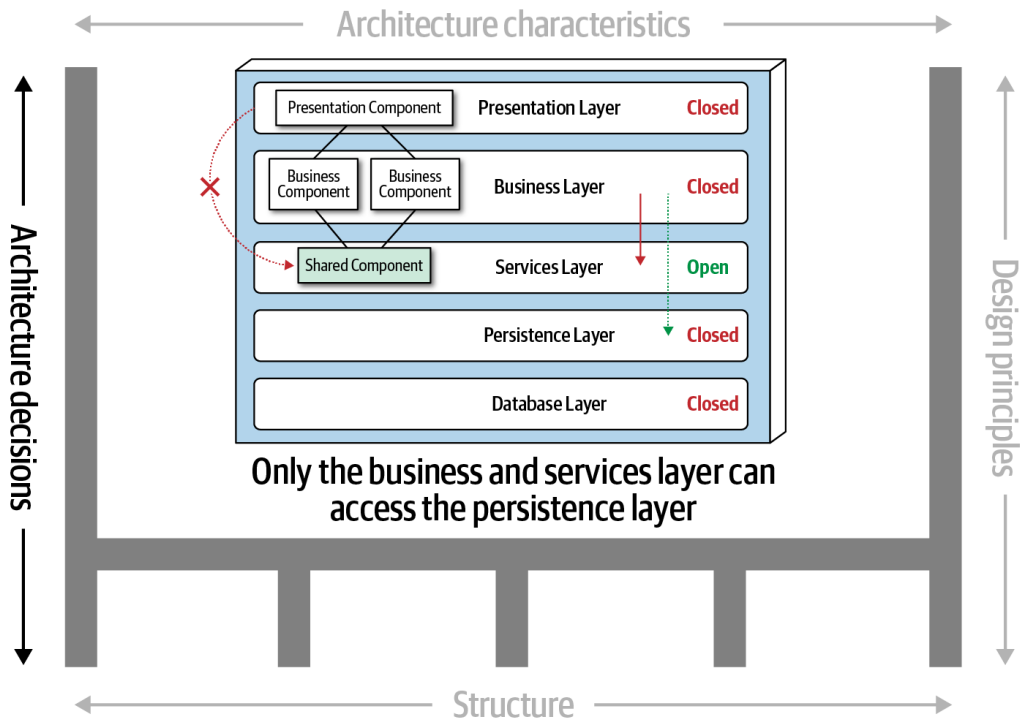
Performanse – performanse sustava se testiraju i analiziraju kako bi se potvrdilo da su zadovoljavajuće za pravilno funkcioniranje sustava

Mogućnost uvođenja – govori o tome koliko je lako uvesti sustav u rad iz razvoja

Mogućnost učenja – govori o tome koliko je vremena i truda potrebno uložiti od strane korisnika sustava da bi se naučili služiti njime

2.1.3. Arhitekturalne odluke

Arhitekturalne odluke su također važan dio arhitekture sustava. One definiraju pravila kako će sustav biti izgrađen, što će biti dozvoljeno i moguće, a što ne. Uzmimo za primjer da naš sustav ima standardnu i dobro poznatu troslojnu arhitekturu, dakle tri sloja. Prezentacijski sloj koji „prezentira“, tj. prikazuje rezultate obrade korisnicima sustava, te također prikuplja ulazne podatke i radnje korisnika koji se koriste sustavom. Sloj obrade koji obrađuje podatke, te nakon što su oni obrađeni prezentacijski sloj ih prikaže. Sloj upravljanja podacima, kao što samo ime govori, upravlja podacima, te „komunicira“ s bazom podataka. Ovaj sloj priprema podatke kako bi se obradili u sloju obrade. Arhitekturalna odluka bi na primjer u ovom slučaju bila da samo sloj upravljanja podacima može pristupiti bazi podataka, dok prezentacijski sloj i sloj obrade ne mogu. Tako se osigurava sigurnost sustava, a također je i olakšan razvoj sustava. Dakle prilikom izrade aplikacije, tj. sustava, programeri će tu arhitekturalnu odluku primijeniti, te će rezultat toga biti aplikacija s troslojnom arhitekturom gdje će samo sloj upravljanja podacima moći pristupiti bazi podataka.



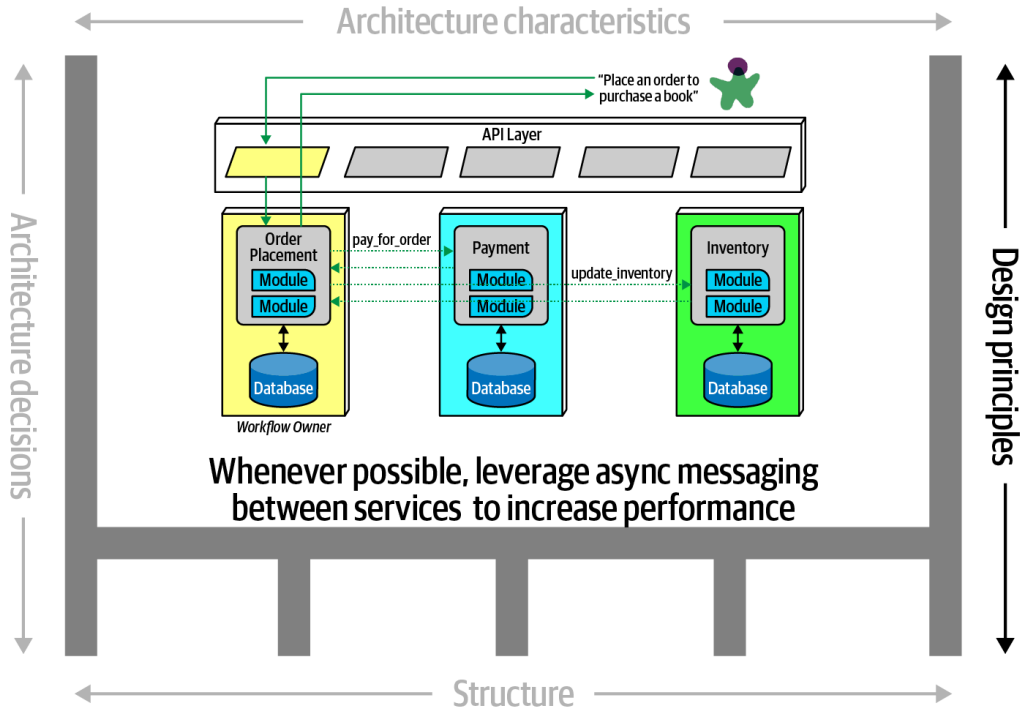
Slika 5: Arhitekturalna odluka *(Richards & Ford, Fundamentals of Software Architecture, 2020)*

Ovdje je arhitekturalna odluka bila prikazana na primjeru rada aplikacije, no arhitekturalne odluke donose se za cijeli sustav.

2.1.4. Načela dizajna

Zadnji dio arhitekture sustava je načelo dizajna. Načela dizajna slična su arhitekturalnim odlukama, no dok je neka arhitekturalna odluka strogo pravila kako se neki dio sustava treba realizirati, načelo dizajna je više kao putokaz u kojem smjeru bi se trebalo ići i razmišljati tijekom razvoja sustava.

Ford i Richards u svojoj knjizi (Richards & Ford, Fundamentals of Software Architecture, 2020) daju dobar primjer za načelo dizajna. Oni su za sustav, čija se arhitektura temelji na mikroservisnom uzorku, kao načelo dizajna odredili da se prilikom kreiranja rješenja preferiraju asinkrone poruke između servisa, a ne sinkrone, kako bi performanse sustava bile bolje. Razlog zašto to nije arhitekturalna odluka, kojom bi zasigurno osigurali da se u produkciji stavi korištenje asinkronih poruka, je da to strogo pravilo ne bi moglo pokriti sve uvjete i opcije za komunikaciju između servisa. Dakle određivanjem da su asinkrone poruke između servisa načelo dizajna, tj. kao neke smjernice programerima, osigurava da kada u produkciji treba odabrati način komunikacije između servisa, da to bude neki oblik asinkrone komunikacije, a to daje slobodu programerima da oni svojom stručnošću procijene što je najbolje.



Slika 6: Načela dizajna u sustavu sa mikroservisnom arhitekturom *(Richards & Ford, Fundamentals of Software Architecture, 2020)*

2.2. Arhitekturalni stilovi

Kao što je opisano u prijašnjem dijelu rada, arhitekturalni stil sustava je struktura sustava, tj. označava način na koji će se sustav izgraditi. Prilikom biranja stila vrlo je važno dobro analizirati funkcionalne i nefunkcionalne zahtjeve te izabrati odgovarajuć arhitekturalni stil. Da ponovimo, arhitekturalni stil je već gotovo, dokazano dobro rješenje za neki problem čije rješenje tražimo u softveru. Arhitekti softver koji nema nikakav arhitekturalni stil nazivaju velikom hrpom blata, „Big Ball of Mud“, i treba ga se izbjegavati jer je sustav s takvom arhitekturom teško, i vremenski zahtjevno, mijenjati i unaprjeđivati. Osim problema kod unaprjeđivanja i mijenjanja, također ga je teško i uvesti u rad, testirati, a i performanse mu nisu zadovoljavajuće (Richards & Ford, Fundamentals of Software Architecture, 2020).

Arhitekturalnih stilova ima mnogo, te je za optimalno funkcioniranje sustava vrlo važno odabrani ispravan stil. Stilove zapravo možemo svrstati u dvije grupe, „Monolithic“ i „Distributed“. „Monolithic“ je grupa stilova čiji sustav se cijeli uvede u rad odjednom, a „Distributed“ je grupa stilova čiji sustav se uvodi u više navrata, tj. u više jedinica koje su

međusobno povezane. Arhitekturalni stilovi koji spadaju u „Monolithic“ grupu su slojevit („Layered“), cjevovodni („Pipeline“) i mikrokernel („Microkernel“). S druge strane, arhitekturalni stilovi koji spadaju u „Distributed“ grupu su stil na temelju usluge („Service-based“), stil vođen događajima („Event-driven“), stil na temelju prostora („Space-based“), stil orijentiran na usluge („Service-oriented“) i mikroservisni stil („Microservices“).

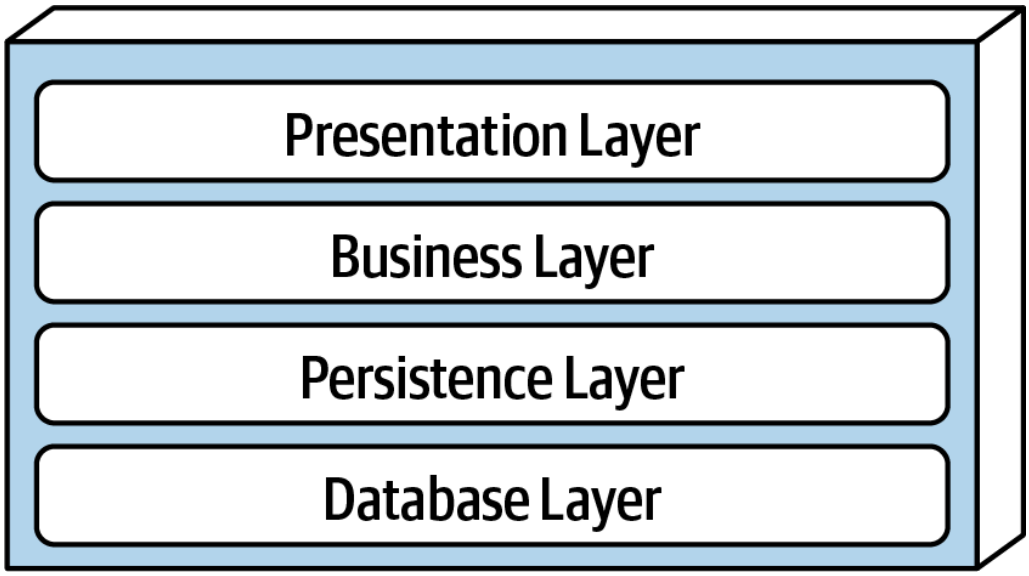
S obzirom na to da su arhitekturalni stilovi glavni dio arhitekture sustava, česta je praksa da se arhitektura sustava naziva po korištenom stilu, na primjer ako je u arhitekturi sustava korišten slojevit stil, možemo reći da sustav ima slojevitu arhitekturu („Layered architecture“).

U nastavku će biti objašnjeni svi prijašnje navedeni stilovi, no važno je napomenuti da to nisu svi stilovi koji postoje, ali su to temeljni na kojima nastaju ostali.

2.2.1. „Layered architecture“

„Layered architecture“, tj. slojevita arhitektura, kako samo ime govori je arhitektura sustava sa slojevitim stilom. Ova arhitektira se također naziva i n-slojna ili višeslojna arhitektura, te je zapravo najčešće korišten arhitekturalni stil kod razvoja aplikacija. Popularan je zbog svoje jednostavnosti implementiranja, a s tim i malim troškovima. Dakle iz imena već možemo predvidjeti da se taj stil sastoji od slojeva, točnije najčešće se sastoji od četiri sloja: „presentation layer“, „business layer“, „persistence layer“ i „database layer“. Osim četiri sloja, vrlo su popularni sustavi sa slojevitim stilom s tri sloja, u tom slučaju „business“ i „persistence“ slojevi spojeni su u jedan sloj koji zatim nazivamo sloj obrade ili aplikacijski sloj. Prezentacijski sloj je sloj najviše razine i njegov zadatak je da prikuplja naredbe i ulaze korisnika sustava te ih prosljeđuje sljedećem sloju, a to je „business layer“ ili sloj obrade podataka. Uz to prezentacijski sloj i prikazuje podatke koje sloj obrade obradi, a namijenjeni su prikazivanju korisniku. Sljedeći je „business layer“, tj. možemo ga nazvati slojem obrade jer je zadaća ovog sloja obrada podataka koje mu je prenio „persistence layer“ ili koje je dobio od prezentacijskog sloja. Dakle ovaj sloj obrađuje podatke tako da na njih primjenjuje zadanu poslovnu logiku, te nakon što ih obradi pošalje ih prezentacijskom sloju koji ih prikaže korisniku ili ih pošalje „persistence layer“ koji ih zatim spremi u bazu podataka. „Persistence layer“ možemo nazvati slojem za rad s podacima jer on kreira potrebne SQL upite koje zatim šalje podatkovnom sloju gdje se na kraju podaci pohrane. Dakle taj sloj dohvaća podatke iz baze podataka te ih prosljeđuje sloju za obradu, ili dobiva podatke od sloja za obradu te ih sprema u bazu. I na kraju imamo „database layer“ koji nazivamo slojem podataka ili sloj baze podataka, a to je ujedno i sloj na najnižoj razini. Taj sloj je zadužen za čuvanje podataka te komunicira sa slojem iznad, tj. sa slojem za rad s podacima. U ranije opisanom slučaju s tri sloja, srednji sloj, dakle

sloj obrade, ima funkcionalnosti sloja obrade i sloja rada s podacima iz četvero-slojnog primjera. Primjer iznad je samo jedan primjer kako slojevita arhitektura može biti organizirana, no primjera ima više te neki mogu imati čak i više od četiri sloja, a neki čak i manje od tri sloja. Ako sustav ima samo jedan sloj to više ne možemo nazivati slojevitom arhitekturom.



Slika 7: Standardna slojevita arhitektura *(Richards & Ford, Fundamentals of Software Architecture, 2020)*

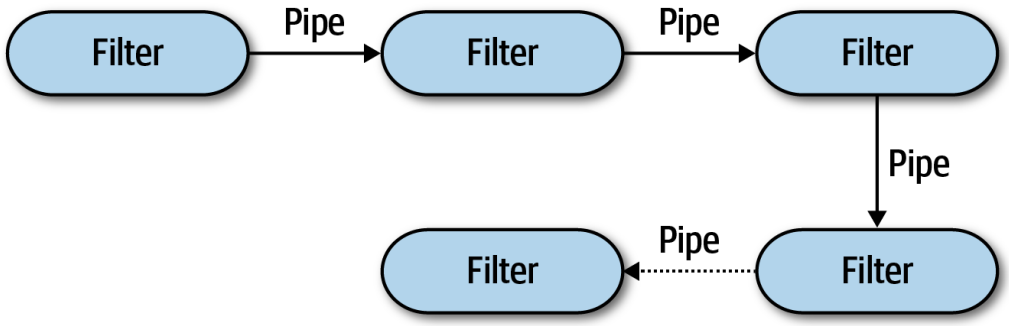
Ukratko ću spomenuti i otvorenost i zatvorenost slojeva. Dakle slojevi mogu biti otvoreni i zatvoreni. Što to znači? Dakle ako je sloj zatvoren, sloj iznad njega mora proći kroz zatvoreni sloj prilikom razmjene poruka s nižim slojevima, a taj zatvoreni sloj će zatim prenijeti poruku sloju ispod sebe. To može biti problematično ako na primjer obrada unesenih podataka nije potrebna, nego se odmah mogu podaci pohraniti u bazu. Podaci svejedno moraju ići prvo u sloj obrade jer je on zatvoren, a taj sloj će samo proslijediti podatke sljedećem sloju. U ovakvom scenariju se nepotrebno iskorištava memorija i smanjuju se performanse sustava, a može se riješiti tako da je sloj obrade otvoren. Ako je sloj otvoren, sloj iznad može preskočiti taj sloj i poslati podatke direktno sloju iza što eliminira navedene probleme. Naravno otvorenost sloja nosi svoje probleme. Ako je sloj otvoren, taj sloj postaje ovisan o slojevima neposredno prije i poslije njega, a to povećava kompleksnost sustava i otežava provođenje promjena (Richards & Ford, Fundamentals of Software Architecture, 2020).

Dakle slojevita arhitektura je dobar izbor ako se razvija jednostavnija i manja aplikacija, koja nije kompleksna, a također je vrlo dobar izbor za web aplikacije. Također je dobar izbor ako se radi na kompleksnijem sustavu s vrlo strogim budžetom i ograničenim vremenom, a još se ne zna koji bi bio idealan stil. U takvom slučaju vrlo je važno čim prije krenuti s razvojem pa se počinje sa slojevitim stilom, a kad se odluči za pravi stil zbog slojevitosti se lako prebaci na

njega (Richards & Ford, Fundamentals of Software Architecture, 2020). Dakle možemo zaključiti da zbog jednostavnosti ovog stila, a s tim i manji troškovi razvoja s obzirom na druge stilove, slojevit stil je idealan za manje aplikacije. No negativna strana ovog stila je da ako će aplikacija rasti, tj. razvijat će se i dalje i imat će sve više funkcionalnosti, postat će teža za održavanje i testiranje, a samo uvođenje novih verzija u rad je zahtjevno jer se svaki puta treba uvesti cijela jedinica, a ne samo dio koji je promijenjen.

2.2.2. „Pipeline architecture“

„Pipeline style“ ili cjevovodni stil pripada u grupu monolitnih arhitekturalnih stilova te se sastoji od filtera i cjevovoda. Iz tog razloga ovaj stil se još naziva i „pipes and filters architecture“, a način na koji radi je jednostavan. Dakle cijevi („pipes“) predstavljaju komunikacijske kanale između filtera. Ti komunikacijski kanali su jednosmjerni, dakle poruke se prenašaju samo u jednom smjeru, i svaki komunikacijski kanal povezuje samo dva filtera, jedan izvorišni i jedan odredišni. Poruke u komunikacijskim kanalom mogu biti bilo kakve, dakle bilo kakve veličine i formata, no poželjno je koristit poruke manjih veličina jer se time povećavaju performanse sustava. Filtri s druge strane su jedinice koje obično imaju samo jednu funkcionalnost, neovisni su o drugim filtrima i imaju barem jedan ulazni ili izlazni komunikacijski kanal, tj. cjevovod. Dakle s obzirom na vrstu filtra, oni obavljaju svoj zadatak te obrađene podatke ili šalju dalje kroz komunikacijski kanal sljedećem filtru, ili proces završava. Postoje četiri vrste filera, proizvođač („Producer“), pretvarač („Transformer“), tester („Tester“) i potrošač („Consumer“). Zadatak filtra proizvođač je da kreira, tj. prikuplja podatke, a također označava i početak procesa. Filter pretvarač primjenjuje zadanu logiku nad podacima te ih tako obrađuje i zatim ih šalje sljedećem filtru. Filter tester s obzirom na ulazne podatke određuje na koji će ih sljedeći komunikacijski kanal pustiti, tj. kojem će sljedećem filteru propustiti podatke. I na kraju filter potrošač označava kraj procesa. Ovaj filter na kraju pohranjuje dobivene podatke u bazu podataka ili ih prikazuje korisniku koji se koristi sustavom (Richards & Ford, Fundamentals of Software Architecture, 2020).



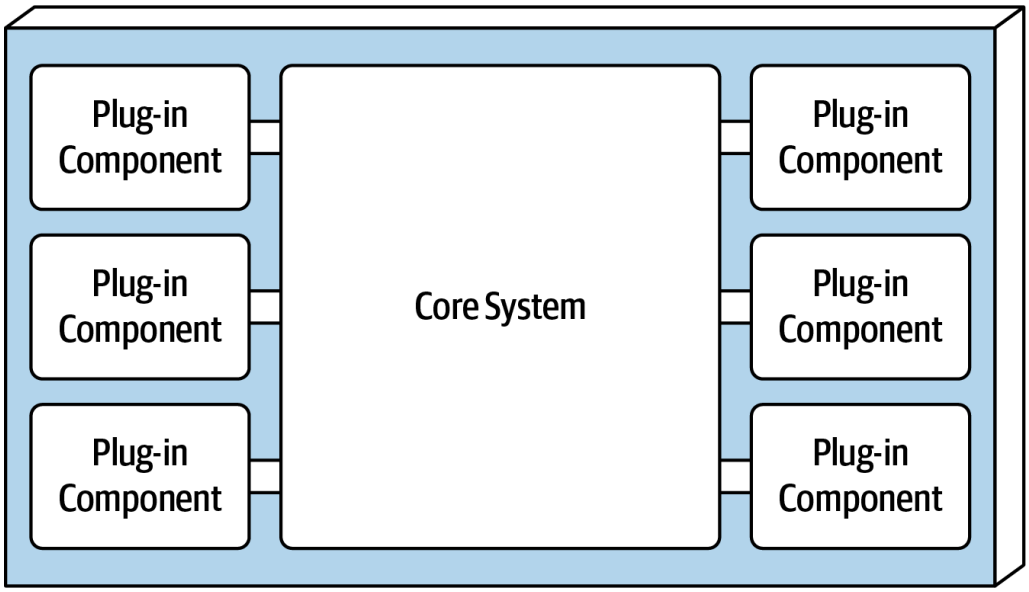
Slika 8: Cjevovodni arhitekturalni stil *(Richards & Ford, Fundamentals of Software Architecture, 2020)*

Jedan od primjera gdje se koristi cjevovodni stil jer u kompajlerima. Dakle kao što znamo zadaća kompajlera je da programski kod napisan od strane programera u programskom jeziku C#, na primjer, prevede u jezik koji je razumljiv računalu, tj. u jezik računala („machine language“). Način na koji to radi je da prolazi kroz filtre koji nakon što odrade svoj zadatak, obrađene podatke šalju sljedećem filtru, na primjer na početku izvorni kod prvo prolazi kroz filter za leksičku analizu, zatim kad se kod obradi ide u sljedeći filter za provjeru sintakse i tako sve dok ne prođe sve filtere i na kraju je rezultat procesa kompajliranja kod napisan u C# pretvoren u jezik računala.

Modularnost ovo arhitekturalnog stila je vidljiva na *Slici 8,* a to je ujedno i jača strana ovog stila. Dakle lako je dodati novi filter u sustav ili promijeniti postojeći, bez da se rade veće promjene u ostatku sustava. Uz modularnost, jače strane ovog stila su njegova jednostavnost i cijena razvijanja sustava. Nedostaci su slični kao kod slojevitog stila jer oboje spadaju pod monolite, a to znači da je uvođenje novih verzija zahtjevno jer se trebaju uvađati cijele jedinice što je sporo. No iako je to slabost ovog stila, bolji je od slojevitog zbog veće modularnosti.

2.2.3. „Microkernel architecture“

Mikrokernel stil, također znan kao i „plug-in style“, je monolitni stil koji se u osnovi sastoji od jezgre sustava („system core“) i dodataka („plug-in“). Jezgra sustava je vrlo jednostavna i najčešće sadrži „registry“ s popisom dodataka i referencama na iste. Dakle glavna svrha jezgre je da prepozna koji dodatak koristiti u određenoj situaciji. Dodaci, s druge strane, kao što im samo ime govori, su proširenja jezgre, te je svako proširenje jedinica za sebe i nije međusobno povezana s drugim dodacima. Oni sadrže zasebne funkcionalnosti za rješavanje nekog problema, te tako proširuju jezgru.



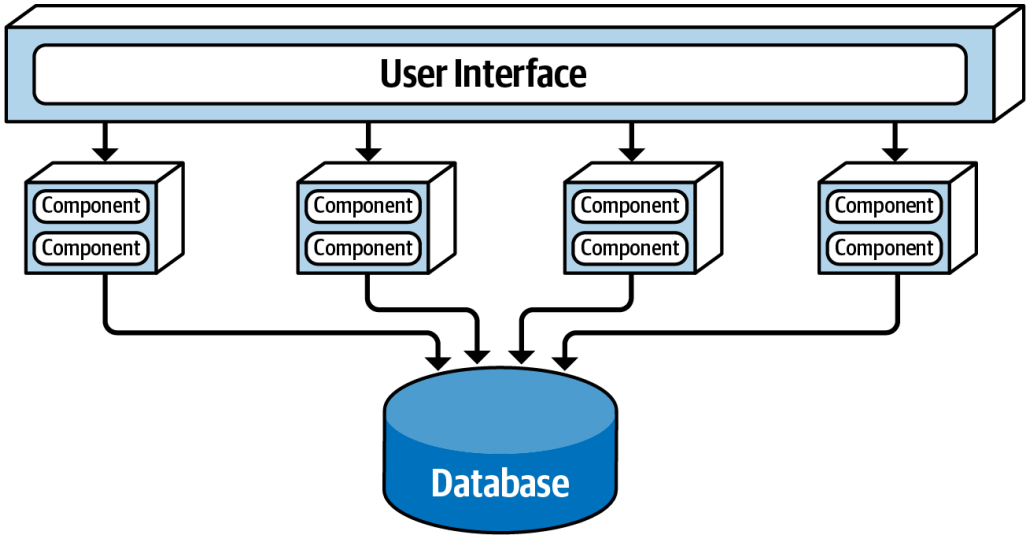
Slika 9: Dijelovi i njihova povezanost u mikrokernelskoj arhitekturi *(Richards & Ford, Fundamentals of Software Architecture, 2020)*

Za primjer možemo uzeti softver eclipse, tvrtke IBM. Eclipse je sama po sebi vrlo jednostavna aplikacija za otvaranje, ažuriranje i spremanje tekstualnih datoteka, no eclipse ima puno dodataka, tj. „plug-in“-ova, koji proširuju tu aplikaciju s puno dodatnih funkcionalnosti, koje zatim omogućuju Eclipse-u da bude „editor“ za više programskih jezika. U knjizi (Richards & Ford, Fundamentals of Software Architecture, 2020) autori također daju primjer za mikrokernelski stil s aplikacijom koja obavlja pregled elektroničkih uređaja. Dakle aplikacija mora moći prepoznati velik broj uređaja, u ovom slučaju pametnih mobilnih telefona, te s obzirom na model vrši zadane preglede. Ako bi taj problem pokušali riješiti aplikacijom bez nekog arhitekturalnog stila, ona bi bila vrlo kompleksna i imala bi loše performanse. No ako problem riješimo aplikacijom sa mikrokernelskim arhitekturalnim stilom, imali bi jezgru sustava koja bi identificirala uređaj te nakon što ga identificira pozvala bi proširenje za taj uređaj. Tako se održava jednostavnost aplikacije, performanse su bolje, a također je takvu aplikaciju i lakše proširiti s novim proširenjima.

Iz ovih primjera možemo vidjeti zašto je i nakon više desetaka godina ovaj arhitekturalni stil i dalje među popularnijima. Jednostavan je za implementirati, a samim tim i troškovi nisu veliki, aplikacije su modularne, a s tim se dobiva i lakše testiranje jer se svaki modul, tj. proširenje, testira zasebno. No naravno ni ovaj stil nije savršen, s obzirom na to da također spada u grupu monolitnih arhitekturalnih stilova sama njihova priroda sprječava da aplikacije budu velike i kompleksne.

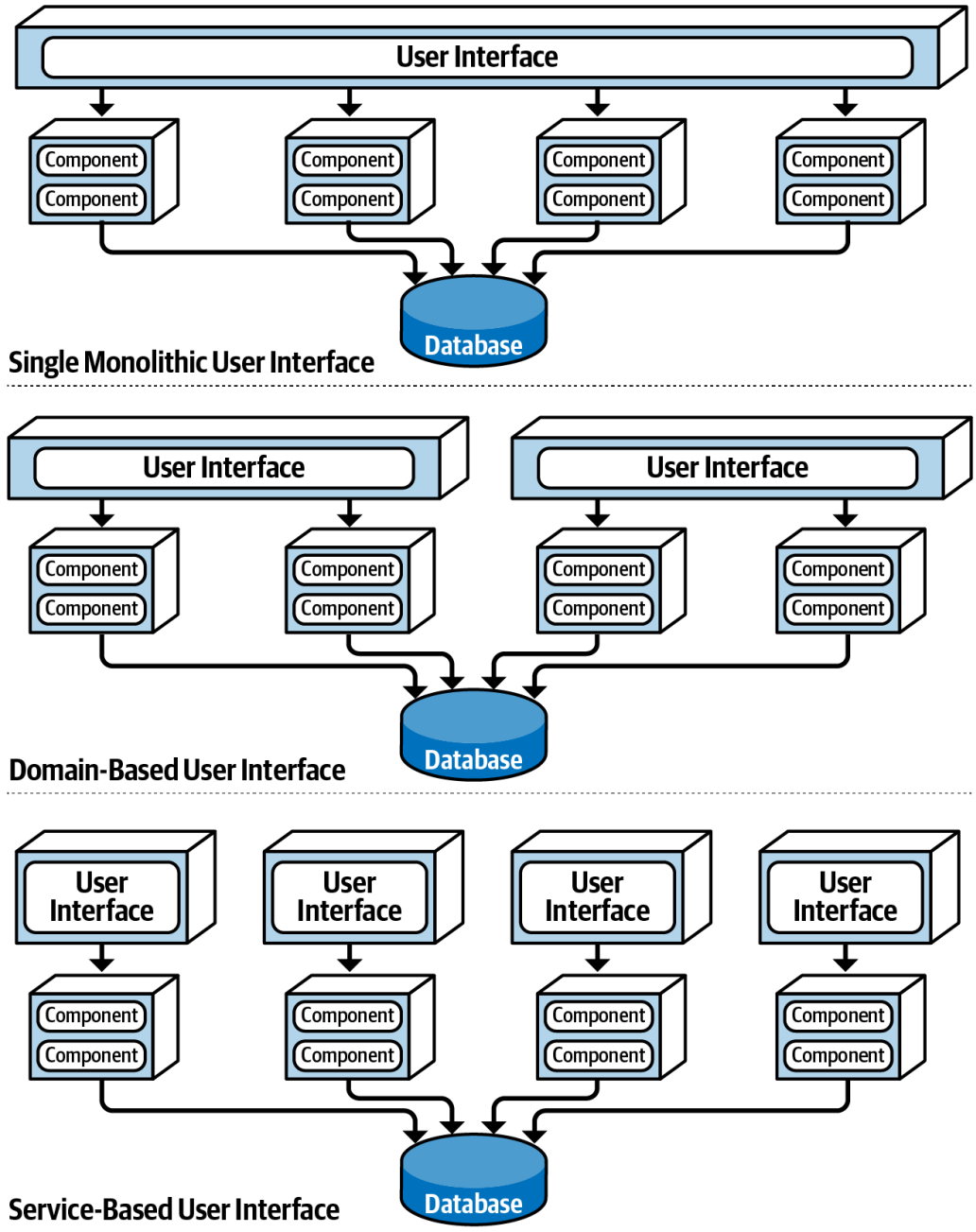
2.2.4. „Service-Based architecture“

Arhitekturalni stil na temelju usluge spada pod takozvane „distributed“ arhitekturalne stilove, te je zapravo hibrid mikrokernelskog stila. Ovaj stil je i jedan od najpraktičnijih stilova jer nudi veliku fleksibilnost što se tiče samog oblikovanja. Struktura ovog stila sastoji se od tri dijela: korisničkog sučelja, udaljenih usluga i baze podataka. Korisničko sučelje i usluge se uvode u rad kao zasebne jedinice, a još dodatno su usluge podijeljene u funkcionalne jedinice te je svaka jedinica odgovorna za određenu funkcionalnost. Baza podataka je centralna i sve jedinice usluga joj pristupaju.



Slika 10:Standardni oblik arhitekture temeljene na uslugama *(Richards & Ford, Fundamentals of Software Architecture, 2020)*

Fleksibilnost ovog stila se vidi u mnogim kombinacijama korisničkih sučelja, jedinica usluga, dizajna usluga, brojem centralnih baza podataka i više. Na primjer što se tiče korisničkog sučelja, ono je standardno organizirano kao jedna monolitna jedinica („Single Monolithic User Interface“), ali može također biti organizirano tako da ima više jedinica korisničkog sučelja i svaka pokriva određenu domenu usluga („Domain-Based User Interface“). Na kraju svaka funkcionalna jedinica usluge može imati svoje zasebno sučelje („Service-Based User Interface“) (Richards & Ford, Fundamentals of Software Architecture, 2020). No to je samo jedan primjer fleksibilnosti ovog stila. Na te načine na koje mogu biti organizirana korisnička sučelja, mogu biti organizirane i baze podataka, a također postoje i različite organizacije komponenata unutar funkcionalnih jedinica usluga.



Slika 11: Različite organizacije korisničkog sučelja *(Richards & Ford, Fundamentals of Software Architecture, 2020)*

Da bi bolje razumjeli ovaj stil iskoristit ću primjer iz knjige (Richards & Ford, Fundamentals of Software Architecture, 2020). Dakle u knjizi imamo situaciju gdje je potrebna aplikacija koja će se baviti prikupljanjem starih elektroničkih uređaja od kupaca za zamjenu za novac. Za rad aplikacije trebaju nam sljedeće usluge: procjena uređaja, prikupljanje uređaja, inspekcija uređaja, računovodstvo, praćenje stanja procesa, recikliranje i izvještavanje. Dakle svaka od tih usluga će u arhitekturi biti jedna funkcionalna jedinica koja će komunicirati s centralnom bazom i pripadajućim korisničkim sučeljem. Nadalje, da bi se povećale performanse i sigurnost, arhitektura će imati tri zasebna korisnička sučelja i dvije baze

podataka. Svako korisničko sučelje će pokrivati određenu domenu usluga, na primjer korisničko sučelje za prikupljanje uređaja će pokrivati usluge prikupljanja uređaja i inspekcije uređaja. Dvije baze podataka napravljene su za bolje performanse sustava, a i za sigurnost.

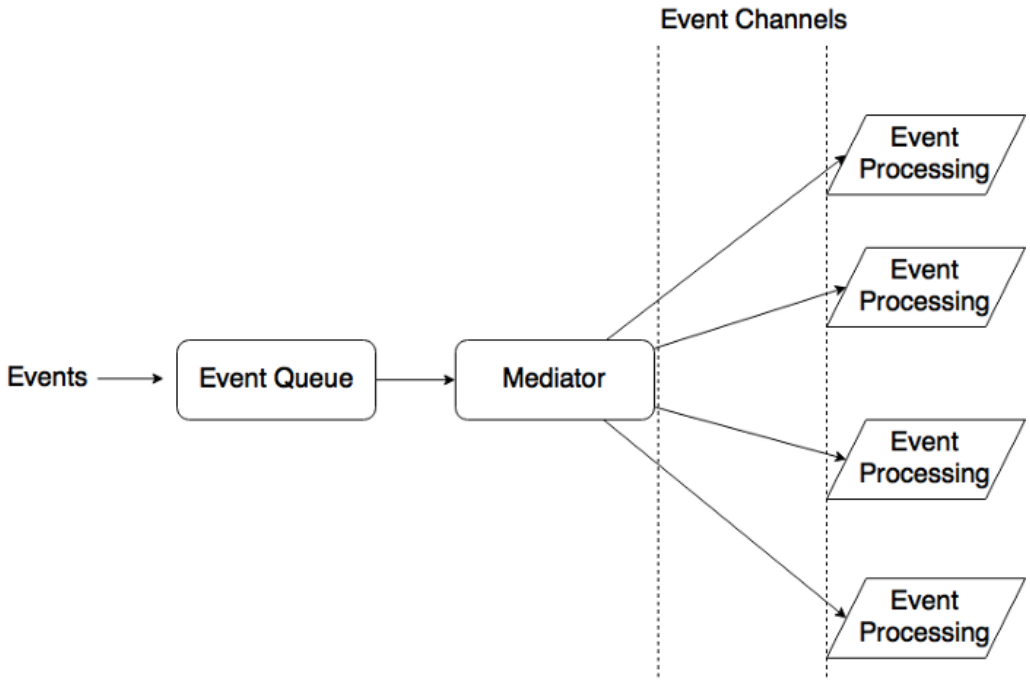
Iz primjera možemo zaključiti zapravo da je aplikacije, s arhitekturalnim stilom temeljenim na uslugama, moguće lakše nadograđivati u odnosu na aplikacije s monolitnim stilovima. Razlog tome je što ažuriranjem jedne usluge uvodimo u rad, tj. isporučujemo samo nju, dok kod monolitnih trebamo uvesti u rad ponovo cijelu logičku cjelinu. Osim toga, aplikacije s ovim stilom omogućavaju i brže ažuriranje dijelova sustava zbog modularnosti, lakše testiranje jer se testiraju zasebni dijelovi, te dijelovi koji su povezani s njima, te kao što sam već opisao uvađanje u rad je lakše jer se uvode samo promijenjeni dijelovi ili novi dijelovi aplikacije. Možemo zaključiti da ovaj stil donosi dobre rezultate za sustave koji imaju više funkcionalnosti, tj. imaju više usluga, a s druge strane nije toliko zahtjevan i skup za razviti, tj. možemo reći da je ovaj stil među najboljima po omjeru karakteristika koje nudi za kompleksnost i troškove razvoja.

2.2.5. „Event-Driven architecture“

Arhitekturalni stil vođen događajima je popularan „distributed“ arhitekturalni stil čije aplikacije su obično visokih performansi zbog korištenja asinkronih poruka. Vrlo je fleksibilan u smislu da se može koristiti i za velike, kompleksne aplikacije, ali i za manje. Sastoji se od više zasebnih komponenti za obradu događaja, a komponente primaju događaje i međusobno komuniciraju asinkrono (Richards & Ford, Fundamentals of Software Architecture, 2020). Ovaj stil ima više vrsta koje određuju organizaciju sustava, no dva glavna su „mediator topology“ i „broker topology“. „Mediator“ topologija se koristi u slučajevima kada je za određen događaj potrebno pokrenuti točno određene procesore događaja, a to se radi preko središnjeg posrednika. S druge strane „broker“ topologija nema posrednika, nego su događaji i reakcije na te događaje direktno povezani.

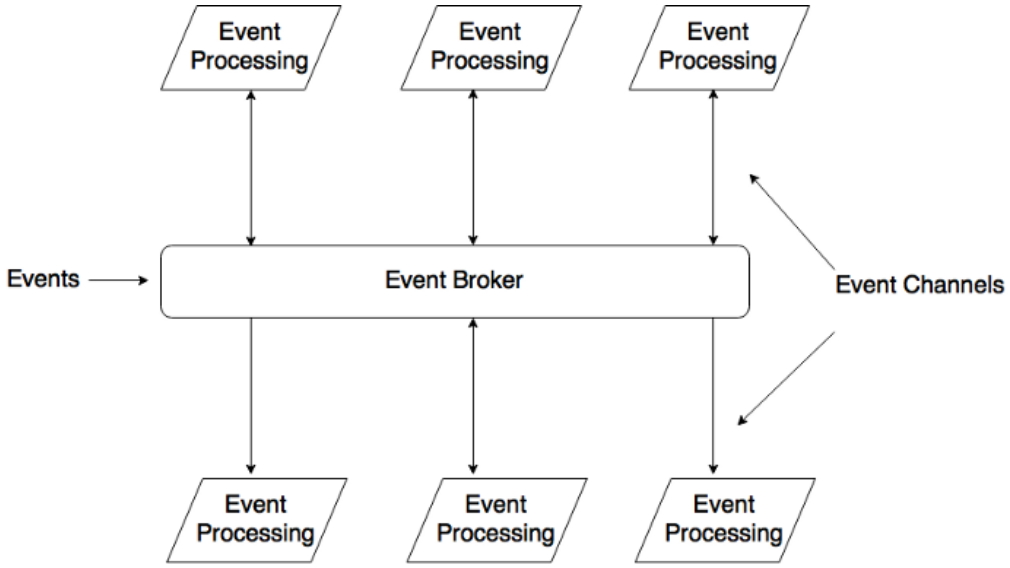
„Mediator topology“ ili topologija posrednika se koristi dakle za aplikacije koje obrađuju događaje sa kompleksnijim postupcima rješavanja tog događaja. Na primjer, ako imamo događaj za čije rješavanje trebamo obaviti nekoliko različitih procesa, te još dodatno procese nakon što se prva grupa procesa izvrši, to bismo rješavali aplikacijom s topologijom posrednika jer ona sadrži središnjeg posrednika koji pravovremeno aktivira potrebne procesore za rješavanje događaja. Osim centralnog posrednika i procesora događaja, u ovoj topologiji još imamo red u koji se spremaju događaji, te kanale kojima posrednik šalje događaje procesorima. Rješavanje nekog događaja bi se izvodilo sljedećim redom. Dakle nakon što se događaj desi on ide u red, te kad na njega dođe red ide u središnjeg posrednika. Posrednik

zatim modificira događaj kako bi bio razumljiv procesorima i zatim ih preko komunikacijskih kanala šalje procesorima koji obrađuju događaj.



Slika 12: EDA topologija posrednika *(Wickramarachchi, 2017)*

Broker topologija, za razliku od topologije posrednika, se više koristi za aplikacije gdje je obrada događaja jednostavnija i nije potreban posrednik koji šalje događaje procesorima. Sadrži samo brokera, procesore i komunikacijske kanale. Dakle kada se desi neki događaj on se šalje brokeru koji ga zatim prosljeđuje direktno svim procesorima kroz komunikacijski kanal. Nakon što procesori završe s obradom objavljuju to porukom te ako još neki procesor treba obraditi taj događaj on ga obradi. Proces rješavanja događaja završava kada ga više ni jedan procesor ne rješava.



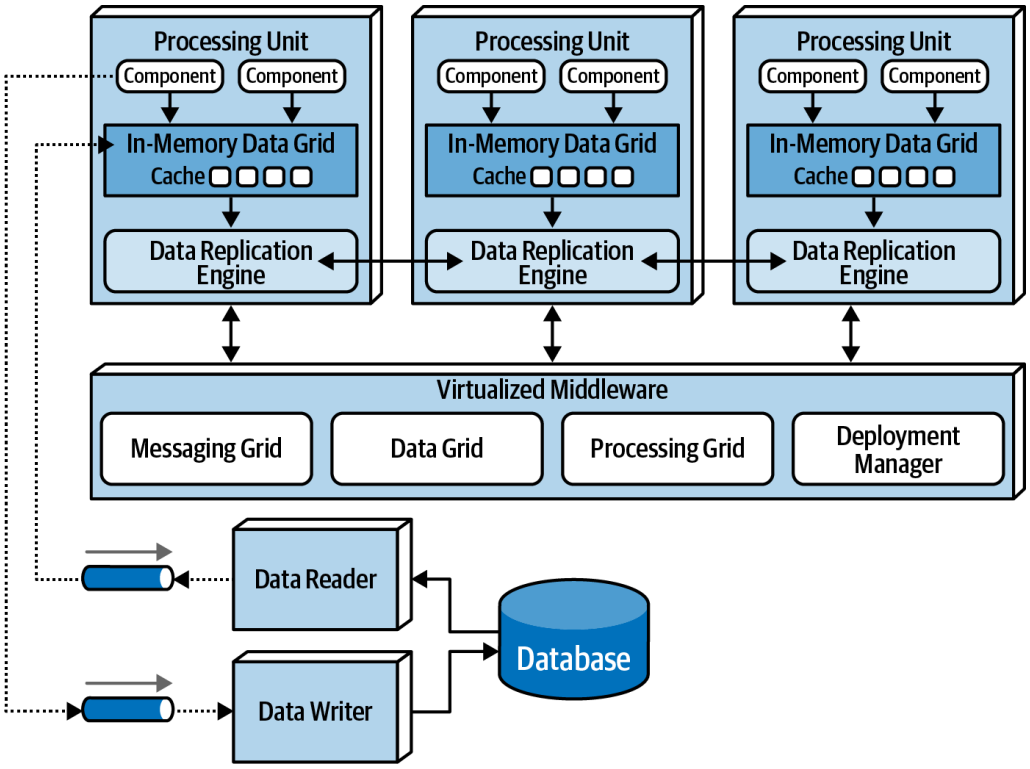
Slika 13: EDA broker topologija *(Wickramarachchi, 2017)*

Aplikacije s arhitekturalnim stilom vođenim događajima su zahtjevne i kompleksne za razviti, prvenstveno zbog dinamičnog toka događaja i asinkronih poruka, a s kompleksnošću su tu i veći troškovi s obzirom na prije opisane „distributed“ stilove. No naravno ovaj stil ima više pozitivnih karakteristika nego negativnih. Visoke performanse su mu jedna od jačih strana, a postižu se asinkronom komunikacijom i paralelnim obrađivanjem događaja na procesorima. Osim performansa, skalabilnost i tolerancija na greške su mu također jače strane, a također je i lako proširiti sustav novim funkcionalnostima, tj. novim procesorima za obradu događaja (Richards & Ford, Fundamentals of Software Architecture, 2020). Dakle kao što sam već na početku napisao, ovaj stil je vrlo dobar i za male i za velike aplikacije, pruža visoke performanse i laku proširivost, no zato je kompleksan i teže se shvaća.

2.2.6. „Space-Based architecture“

„Space-Based architecture“, ili arhitekturalni stil temeljen na prostoru spada u „distributed“ grupu arhitekturalnih stilova i najčešće se koristi za aplikacije s velikim trenutnim brojem korisnika. Drugim riječima aplikacije razvijene s ovim stilom su vrlo skalabilne i elastične, tj. mogu podnijeti velika opterećenja na duži period vremena ili na kraće skokove. Ovaj stil je kreiran kako bi se riješio problem opterećenja sustava u standardnim web topologijama. Naime povećanjem opterećenja, na primjer broja korisnika i zahtjeva, u takvim sustavima dolazi do uskih grla koja se rješavaju proširivanjem sustava. No proširivanje nakon nekog vremena postaje skupo i nepraktično, te se u takvim situacijama koriste sustavi sa stilom temeljenim na prostoru. Ovaj stil sastoji se od procesnih jedinica, virtualnog „middleware“-a,

jedinica koje šalju podatke, pisača podataka u bazu podataka i čitača podataka. Dakle procesnih jedinica ima više i svaka u sebi sadrži procesne komponente, memoriju i sustav koji replicira podatke. Ideja je da se podaci potrebni za rad čuvaju u memoriji procesnih komponenti, i da sve procesne komponente imaju iste podatke u memoriji, te da ako dođe do ažuriranja nekih podataka, ti podaci se asinkrono šalju jedinici za pisanje podataka u bazu podataka, te se tako podaci ažuriraju u bazi. Tako se postižu visoke performanse sustava, a dinamičkim paljenjem i gašenjem procesnih jedinica, ovisno o opterećenju sustava, osigurava se varijabilna skalabilnost.



Slika 14: Standrdni stil temeljen na prostoru *(Richards & Ford, Fundamentals of Software Architecture, 2020)*

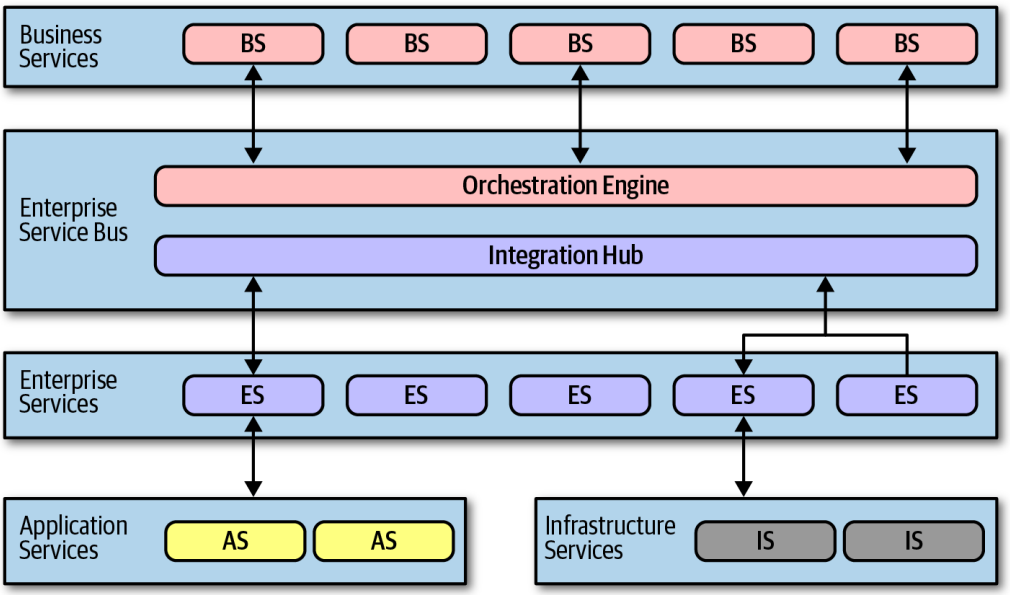
Aplikacije koje su razvijene s ovim stilom su dakle skalabilne, elastične i visokih performansi. U knjizi (Richards & Ford, Fundamentals of Software Architecture, 2020) su dana dva primjera aplikacija koje koriste ovakav stil, sustav za kupnju koncertnih karata i internetska aukcija. Sustav za kupnju karata na normalnoj dnevnoj bazi nije toliko opterećen, no kada u prodaju kreću karte za koncert neke popularne grupe, tisuće ljudi odjednom pokušava kupiti kartu. Sustav iz tog razloga mora biti vrlo stabilan, tj. elastičan, i mora biti visokih performansi kako bi što je brže mogao obrađivao zahtjeve. Također direktno pristupanje bazi u takvom sustavu gotovo i da nije moguće jer bi se bazi slalo tisuće zahtjeva što bi drastično smanjilo performanse sustava. Iz tih razloga je ovaj stil savršen za ovakav sustav. Dinamičan broj

procesnih jedinica znači da će sustav biti stabilan s tisuće kupaca, a kako svaka procesna jedinica sadrži memoriju s podacima iz baze podataka, pristupa se tim memorijama i time se ne stvara usko grlo na bazi, a baza se ažurira asinkronim porukama.

Visoke performanse, velika skalabilnost i elastičnost glavni su atributi ovog stila i omogućavaju razvijanje sustava koji su vrlo otporni na trenutna opterećenja. No ovakva struktura sustava je vrlo kompleksna i troškovi razvoja su veliki. Testiranje sustava je također otežano jer je sustav potrebno testirati pod velikim opterećenjima koje je teško kreirati u uredskom okruženju.

2.2.7. „Service-Oriented architecture“

Arhitekturalni stil orijentiran na usluge spada u grupu „distributed“ stilova i kao što mu samo ime govori, orijentiran je na usluge koje čine zasebne funkcijske jedinice, a usluge međusobno komuniciraju kako bi razmijenili podatke. No iako usluge međusobno komuniciraju, važno je napomenuti da one vrlo malo ovise jedna o drugoj, tj. komuniciraju putem takozvanog „loose coupling“ sustava. Sama struktura i vrste usluga se razlikuju od sustava do sustava, te postoji više vrsta ovog stila. U knjizi (Richards & Ford, Fundamentals of Software Architecture, 2020) je opisan stil orijentiran na usluge koji je vođen upravljačkom jedinicom. Taj sustav se sastoji od „Enterprise Service Bus“-a u kojem se nalazi upravljačka jedinica, funkcijske jedinice poslovnih usluga, usluge poduzeća, aplikacijske usluge i infrastrukturne usluge. Svaka od tih funkcijskih jedinica je zasebna aplikacija te one međusobno komuniciraju kako bi razmijenili podatke.

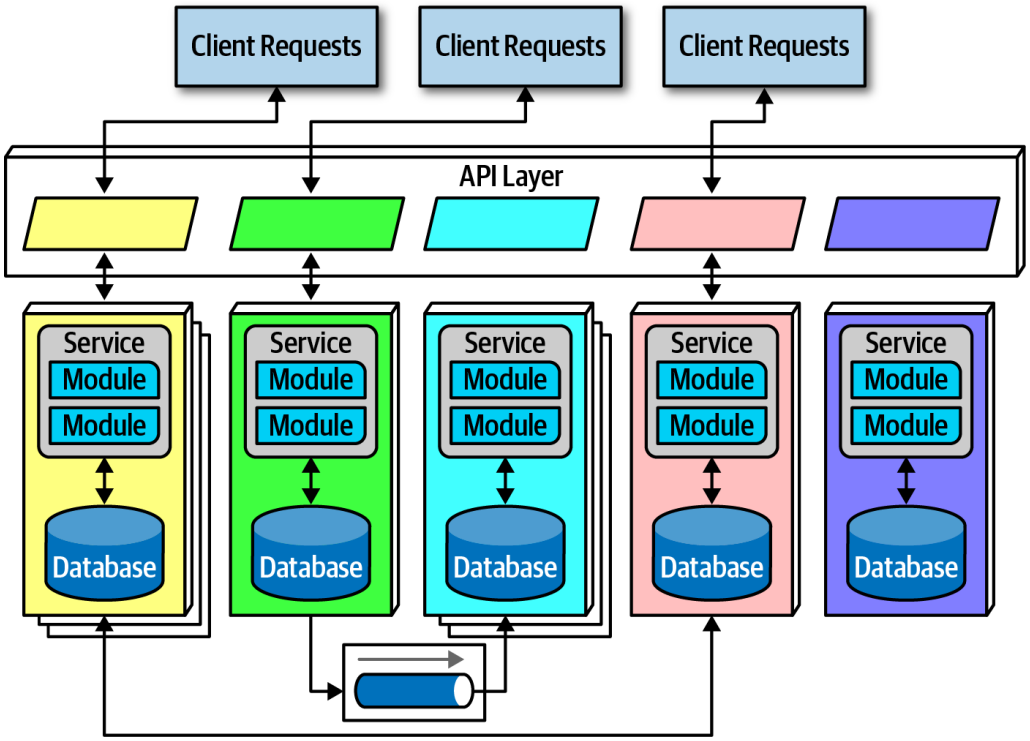


Slika 15: Arhitekturalni stil orijentiran na usluge *(Richards & Ford, Fundamentals of Software Architecture, 2020)*

Ovaj arhitekturalni stil nastao je 1990-tih godina, a arhitekturalne karakteristike koje su danas važne razlikuju se od onih koje su bile tad važne. Iz tog razloga karakteristike ovog stila prilično su loše, no s druge strane, problemi ovog stila doveli su do razvoja mikroservisnog arhitekturalnog stila koji će biti opisan u sljedećem poglavlju.

2.2.8. „Microservices architecture“

Mikroservisni arhitekturalni stil spada u grupu „distributed“ stilova, te kao što samo ime govori, sastoji se od više zasebnih, odvojenih funkcijskih jedinica od kojih je svaka zadužena za određenu uslugu. Svaka takva funkcijska jedinica sadrži svoj kod koji obavlja uslugu, sadrži svoju bazu podataka i svim jedinicama se pristupa preko API-ja. API zapravo u ovom sustavu nije bitan, no vrlo je poželjno imati ga jer odvaja usluge od klijenata, što pridonosi mogućnosti ažuriranja usluga bez da se ažuriraju klijenti. Dakle to su zapravo svi dijelovi sustava sa mikroservisnim stilom, usluge koje su zasebne funkcijske jedinice, API koji od klijenta prima naredbe te ih zatim proslijedi odgovarajućoj usluzi i komunikacijski kanali preko kojih usluge međusobno komuniciraju.



Slika 16: Mikroservisni arhitekturalni stil *(Richards & Ford, Fundamentals of Software Architecture, 2020)*

Aplikacije s ovom arhitekturom su dakle skalabilne, elastične, modularne, a s tim i lako proširive, lako se testiraju jer su usluge zasebne funkcijske jedinice i uvođenje u rad je također olakšano jer se uvode samo zasebne usluge, a ne cijeli sustav. No loša strana ovakvih aplikacija su lošije performanse s obzirom na ostale „distributed“ stilove. Zbog svih potrebnih komuniciranja između API-ja i usluga, i među uslugama, performanse su relativno slabe. Međusobna komunikacija između usluga je također teška za implementirati jer je cilj da usluge ne budu ovisne jedne o drugima, te to čini ovaj stil kompleksnim za razviti. Kako bi usluge ostale maksimalno nezavisne, ne preferira se ponovo korištenje koda prilikom razvoja usluga, a to povećava vrijeme razvoja, te time i troškove. Na kraju možemo zaključiti da je ovaj stil odličan za sustave s velikim opterećenjima i s velikim broj funkcionalnosti, no gdje performanse nisu najbitnija karakteristika.

2.3. Odabir odgovarajućeg stila

Arhitekturalnih stilova ima mnogo, a za biranje odgovarajućeg stila potrebno je prvo detaljno analizirati zahtjeve aplikacije. Biranje stila, dakle, zahtijeva analiziranje domene sustava, karakteristika, ciljeva i drugih aspekata budućeg sustava. Po (Richards & Ford,

Fundamentals of Software Architecture, 2020) prilikom biranja odgovarajućeg arhitekturalnog stila, arhitekt mora uzeti u obzir sljedeće faktore:

Domena sustava – arhitekt mora dobro poznavati glavne aspekte domene sustava i aspekte koji imaju utjecaj na karakteristike sustava

Karakteristike koje imaju utjecaj na strukturu – arhitekt mora pronaći arhitekturalne karakteristike koje će podržavati domenu

Arhitektura podataka – arhitekt mora dobro poznavati strukturu podataka u bazi podataka, te idealno sudjelovati u kreiranju baze jer struktura podataka može imati veliki utjecaj na dizajn

Organizacijski čimbenici – na arhitekturu mogu utjecati i vanjski čimbenici, te je iz tog razloga važno da arhitekt bude svjestan tih čimbenika kako bi mogao donijeti pravu odluku

Poznavanje procesa, timova i operativnih problema – arhitekt mora dobro poznavati faktore projekta i sposobnost razvojnih timova kako bi mogao donijeti dobru odluku o stilu, na primjer u knjizi (Richards & Ford, Fundamentals of Software Architecture, 2020) autori daju primjer gdje timovi nisu vrlo iskusni u agilnom razvoju, te iz tog razloga arhitekt neće odabrati stil koji se oslanja na takav način razvoja

Domenski/Arhitekturalni izomorfizam – označava kada domena odgovara topologiji arhitekture (Richards & Ford, Fundamentals of Software Architecture, 2020). Arhitekt mora prepoznati postoji li stil koji bi odgovarao domeni aplikacije, te ako postoji proučiti bi li taj stil odgovarao za druge, prije navedene faktore. No ovdje treba biti oprezan jer bi stil koji ne odgovara domeni donio probleme.

Imajući ove faktore na umu, arhitekt mora donijeti odluku hoće li stil biti monolitni ili distribuirani, gdje će se podaci spremati i kako će podaci teći kroz sustav, te hoće li razmjena podataka među servisima u sustavu teći sinkrono ili asinkrono (Richards & Ford, Fundamentals of Software Architecture, 2020). Tek nakon što arhitekt analizira, odredi i uzme u obzir sve ove zahtjeve, može odrediti koji stil će najbolje odgovarati sustavu.

U nastavku slijedi tablica u kojoj će biti opisane situacije u kojim slučajevima odabrati neki od prije navedenih stilova, a u kojim slučajevima ih ne odabrati.

Tablica 1: Kada koristiti/izbjegavati arhitekturalni stil

Naziv stila	Kada odabrati ovaj stil	Kada izbjegavati ovaj stil
Slojevit	U slučaju da razvijamo malu, jednostavnu ili web aplikaciju	Kada namjeravamo proširivati aplikaciju novim funkcionalnostima što bi rezultiralo rastom aplikacije

	(Richards & Ford, Fundamentals of Software Architecture, 2020).	(Richards & Ford, Fundamentals of Software Architecture, 2020).
Cjevovodni	U slučaju da radimo aplikaciju za migraciju podataka s jednog mjesta na drugo (Schalme, n.d.).	Kada radimo aplikaciju koja će često biti ažurirana (Schalme, n.d.).
Mikrokernel	U slučaju da aplikacija sadrži više zasebnih funkcionalnosti (Richards & Ford, Fundamentals of Software Architecture, 2020).	Kada razvijamo veliku i kompleksnu aplikaciju (Richards & Ford, Fundamentals of Software Architecture, 2020).
Na temelju usluga	U slučaju da razvijamo aplikaciju koja pruža više usluga (Richards & Ford, Fundamentals of Software Architecture, 2020).	Kada se očekuje čest nagli i kratkotrajni porast u broju korisnika (Richards & Ford, Fundamentals of Software Architecture, 2020).
Vođen događajima	U slučaju da razvijamo skalabilnu aplikaciju visokih performansi (Richards & Ford, Fundamentals of Software Architecture, 2020).	Kada razvojni timovi nisu spremni za razvoj vrlo kompleksne aplikacije (Richards, Software Architecture Patterns, 2015).
Na temelju prostora	U slučaju da razvijamo manju skalabilnu i elastičnu aplikaciju (Richards, Software Architecture Patterns, 2015).	Kada razvijamo veliku aplikaciju s relacijskom bazom podataka i velikim brojem razmjene podataka (Richards, Software Architecture Patterns, 2015).
Orijentiran na usluge	U slučaju da razvijamo aplikaciju koja pruža više usluga, no želimo da usluge bude samostalne funkcijske jedinice s minimalnom komunikacijom s drugim uslugama (Richards & Ford, Fundamentals of Software Architecture, 2020).	Kada su nam važne performanse i/ili mogućnost proširivanja aplikacije u budućnosti (Richards & Ford, Fundamentals of Software Architecture, 2020).
Mikroservisni	U slučaju kada želimo stabilnu, skalabilnu aplikaciju koju je moguće ažurirati bez da se ona ukloni iz uporabe (Richards, Software Architecture Patterns, 2015).	Kada su potrebne visoke performanse (Richards, Software Architecture Patterns, 2015).

Dakle odabir odgovarajućeg stila je poprilično važan i zahtjevan posao te iziskuje znanje i iskustvo arhitekta, no ako je dobro odrađen daljnji razvoj teći će bez većih problema i s minimalnim troškovima.

2.4. Dokumentiranje arhitekture sustava

Osim odabira odgovarajućeg stila, prepoznavanja važnih karakteristika za sustav, donošenja arhitekturalnih odluka i načela dizajna, važno je još arhitekturu sustava prikazati raznim dijagramima. Dijagrami nam pružaju mogućnost da vizualiziramo sustav, te tako prikazuju na koji način su komponente sustava međusobno povezane. Osim toga osiguravaju da svi ljudi koji su uključeni u razvoj, razumiju kako je sustav građen i kako funkcionira.

Arhitektura sustava može se prikazati raznim dijagramima, a za njihovo modeliranje najčešće se koristi UML. UML-om možemo prikazati dijagrame klasa, slučajeva korištenja, slijeda, tijeka, a također možemo prikazati i ERA dijagram za potrebe prikaza organizacije podataka. U knjizi (Richards & Ford, Fundamentals of Software Architecture, 2020) autori također spominju i C4. C4 je dijagramska tehnika koju je razvio Simon Brown kako bi dopunio UML. C4 se sastoji od konteksta („Context“), spremnika („Container“), komponenti („Component“) i klasa („Class“). Kontekst je na prvoj razini i predstavlja cijeli kontekst sustava, dakle prikazuje uloge koje će se koristiti sustavom i sam sustav. Sljedeća razina je spremnik i on detaljnije prikazuje sustav iz prijašnje razine tako da prikazuje aplikacije od kojih se sustav sastoji i spremišta podataka, tj. baze podataka. Sljedeća razina su komponente. Komponente su grupe povezanih funkcionalnosti od kojih se sastoje elementi spremnika. Sljedeća, i posljednja razina su klase ili kod. Ova razina nije nužna, no može se prikazati od kojih se sve klasa aplikacija sastoji pomoću dijagrama klasa, može se prikazati sam kod aplikacije ili čak ERA dijagram. Dakle ova dijagramska tehnika prikazuje arhitekturu sustava u četiri razine detaljnosti, od konteksta na prvoj razini kao najmanje detaljan prikaz, do klasa na posljednjoj razini kao najdetaljniji prikaz.

Sve ove dijagramske tehnike pomažu nam da bolje shvatimo organizaciju sustava i kako sustav treba funkcionirati, no treba napomenuti da nije stalno potrebno raditi sve navedene dijagrame, bitno je da se uz pomoć njih razumije sustav.

3. Arhitekturalni dizajn aplikacije za vođenje evidencije specijalizanata medicine

Specijalizanti medicine u procesu specijalizacije popunjavaju dvije knjižice u kojima se vodi evidencija o njihovim dnevnim aktivnostima. To su „Knjižica o specijalističkom usavršavanju doktora medicine“ i „Dnevnik rada doktora medicine na specijalističkom usavršavanju“ u kojima specijalizanti pišu obavljene zahvate, stečene kompetencije, dnevne

aktivnosti i drugo. Osim ručnog popunjavanja ovih knjižica, specijalizantima je u njima potreban pečat i potpis glavnog mentora i mentora na kojem odjelu su obavljali aktivnosti. Pretpostavlja se da se kroz period specijalizacije, koji traje pet godina, specijalizantima „udari“ i do tisuću petsto pečata, te mnogi smatraju da je to gubitak vremena. U današnje vrijeme s vrlo razvijenim IT sektorom i visokom digitalizacijom svih mogućih procesa, zapanjujuće je da je jedan od najvažnijih sektora u državi toliko zaostao. Dakle važna je informatizacija procesa popunjavanja ovih knjižica kako bi skratili vrijeme popunjavanja i provjeravanja od strane mentora, čime bi i specijalizanti i mentori dobili više vremena za rad na važnijim stvarima.

U ovom poglavlju rada primijenit će se teorija iz prijašnjeg poglavlja kako bi se dobila arhitektura aplikacije za vođenje evidencije specijalizanata, koja će zatim biti „putokaz“ u fazi implementacije kako bi se u konačnici kreirala aplikacija koja bi mogla poslužiti kao zamjena za ovu napornu i dugotrajnu formalnost.

Kao što je objašnjeno u prijašnjem poglavlju, važno je prvo kreirati dobru arhitekturu sustava kako bi se minimizirali troškovi razvoja aplikacije, jer je ispravljanje grešaka i propusta skuplje u fazi implementacije nego u fazi dizajna. Dakle važno je odrediti odgovarajuć arhitekturalni stil, arhitekturalne karakteristike i odluke te načela dizajna. Nakon toga će u fazi implementacije programeri uz minimalne troškove i zaostatke kreirati aplikaciju koja će uštedjeti vrijeme i specijalizantima i njihovim mentorima.

3.1. Odabir arhitekturalnog stila

Prije nego odaberemo odgovarajući stil moramo proučiti domenu aplikacije i potrebne karakteristike. Ove dvije knjižice su zapravo vrlo slične, sadržaj koji se u njih upisuje je drugačiji, ali su princip popunjavanja i provjere isti. Također je bitno spomenuti da se način na koji se knjižice popunjavaju nije mijenjao već deset godina, te po tome možemo zaključiti da se domena ne mijenja i da nema previše devijacija što se tiče podataka. Iz tog razloga možemo reći da bi neki monolitni stil bio odgovarajuć za ovu aplikaciju jer neće biti puno proširivanja i ažuriranja aplikacije. Aplikaciju će biti kompliciranije uvesti u rad, no kao što sam rekao, kako nema gotovo nikakvih promjena u procesu popunjavanja knjižica, to ne bi smio biti problem. Zatim možemo razmatrati elastičnost i skalabilnost sustava, tj. koliko će sustav dobro raditi kad se opterećenje poveća i kad se opterećenje poveća naglo na kraći period. Što se toga tiče sustav ne bi trebao biti previše elastičan i skalabilan jer u Hrvatskoj nema toliko specijalizanata medicine, niti će se broj drastično povećati u budućnosti, a također priroda procesa popunjavanja knjižice je takva da ju specijalizanti popunjavaju kada imaju vremena, što ne stvara velika trenutna opterećenja sustava. Također mentori prilikom provjera zapisa specijalizanata ne stvaraju veliko opterećenje iz istih razloga. Iz tog možemo zaključiti da je

dovoljna jedna centralna baza podataka, te kao što je prije napisano, elastičnost i skalabilnost sustava nisu od velike važnosti. Jedan od važnijih faktora bi bila cijena razvoja ove aplikacije. S obzirom na nedostatak novčanih sredstava u medicinskom sektoru, troškovi razvoja ove aplikacije ne bi trebali biti visoki. S obzirom na sve ove uvjete i zahtjeve možemo doći do zaključka da bi slojevit stil bio dobar izbor za strukturu ove aplikacije. Slojevita arhitektura je među jednostavnijima za implementirati te samim tim donosi male troškove razvoja, a s obzirom na to da nisu potrebne visoka elastičnost, skalabilnost i brzo uvođenje u rad, ovaj stil će biti odličan izbor. Također s obzirom na to da je aplikacija podijeljena na slojeve bit će ju relativno lako testirati i otkloniti greške, te će sigurnost podataka biti relativno dobra jer im je moguće pristupiti samo iz nekih slojeva što je bitno u slučaju da neki korisnici žele manipulirati podacima. Uz sve ovo još jedna prednost ovog sloja je da se svaki sloj može izvršavati na zasebnom računalu, što dodatno doprinosti sigurnosti, a i povećava performanse aplikacije.

Odabirom slojevitog stila možemo pričati o strukturi samog stila, tj. od koliko će se slojeva sastojati. Standardni oblik sastoji se od četiri sloja, no s obzirom na to da će aplikacija biti napravljena u „Visual Studio“-u pomoću .NET okvira u obliku „Windows Forms“ aplikacije, koristit će se slojevita arhitektura s tri sloja. Dakle to bi bili prezentacijski sloj, sloj obrade i sloj podataka. Prezentacijski sloj bi bilo korisničko sučelje prema specijalizantima, mentorima i administratorima koji bi prikazivao podatke i prikupljao ulazne naredbe. Sloj obrade bi obrađivao podatke za prikaz i za spremanje u bazu. Sloj podataka šalje SQL upite bazi podataka te tako dohvaća, ažurira, briše ili unaša podatke u nju.

Slika 17: Arhitekturalni stil aplikacije (samostalna izrada)

3.2. Karakteristike aplikacije

Neke arhitekturalne karakteristike već su spomenute prilikom razmatranja za odgovarajući stil aplikacije, no ovdje će biti razrađene sve. Karakteristike su nam zapravo važne kako bi definirali neke kriterije koje aplikacija mora ispunjavati kako bi ona bila uspješno razvijena i funkcionalna. Karakteristike koje treba odrediti su dostupnost, provjerljivost, skalabilnost, sigurnost, tolerancija na greške, elastičnost, mogućnost oporavka, performanse, mogućnost uvođenja i mogućnost učenja.

Dostupnost – s obzirom na to da specijalizanti rade oko 300 dana godišnje, aplikacija bi trebala biti stalno dostupna. S druge strane aplikacija nije ključna za rad specijalizanata, te ako dođe do pada sustava ili se radi na ažuriranju sustava, dokle god se to odradi u kraćem vremenskom roku ne bi smjelo biti problema. Naime ako je aplikacija nedostupna, iako je to nezgodno i za specijalizante i za mentore, oni uvijek mogu upisati svoje aktivnosti, tj. provjeriti aktivnosti, kasnije kada aplikacija postane ponovno dostupna.

Provjerljivost – sustav mora biti provjerljiv kako bi se nakon implementacije provjerio, tj. testirao, čime bi se pronašle potencijalne greške u radu i kako bi se one otklonile, a to bi doprinijelo kvalitetnijoj aplikaciji.

Skalabilnost – kao što je već napisano, ova aplikacija ne treba biti jako skalabilna zato što je broj specijalizanata i mentora koji će koristiti aplikaciju više manje isti uz male varijacije. U narednim godinama neće se desiti da će broj specijalizanata i mentora naglo porasti te iz tog razloga skalabilnost aplikacije ne mora biti visoka.

Sigurnost – ova aplikacija će sadržavati standardnu prijavu kojom će se korisnici autorizirati. Sadržavat će tri uloge: specijalizant, mentor i administrator. S obzirom na to da je to aplikacija za specifične ljude, tj. nije za svakoga, stvaranje računa vršit će administrator, dakle neće biti registracije. Specijalizanti se nakon dobivanja računa prijavljuju te popunjavaju knjižicu i dnevnik, a mentori provjeravaju aktivnosti specijalizanata i to potvrđuju digitalnim potpisima.

Elastičnost – kao što je već napisano, sama priroda popunjavanja knjižice i dnevnika ne bi stvarala velika trenutna opterećenja sustava zato što bi specijalizanti i mentori koristili aplikaciju kada bi imali slobodnog vremena, a gotovo je nemoguće da svi specijalizanti i mentori istovremeno koriste aplikaciju. Iz tog razloga elastičnost aplikacije ne mora biti visoka, a s druge strane ukupan broj specijalizanata i mentora nije toliko velik da bi bila potrebna velika elastičnost čak i u slučaju da ih većina krene raditi s aplikacijom istovremeno.

Performanse – performanse aplikacije ne moraju biti jako dobre, ali s druge strane ne mogu biti ni slabe. Bitno je da specijalizanti i mentori ne čekaju na duga učitavanja prilikom rada u aplikaciji, a to se može osigurati optimizacijom pregleda prijašnjih unosa, tj. da se ne učitavaju svi zapisi što bi moglo donijeti loše peformanse aplikacije na kratko vrijeme.

Mogućnost uvođenja – s obzirom na to da aplikacija neće biti često ažurirana, tj. ažurirat će se samo da se isprave greške u radu ako će ih biti, može se stalno isporučiti cijela nova aplikacija.

Mogućnost učenja – aplikacija bi bila zapravo samo digitalna verzija knjižice i dnevnika, te samim tim bi njezino popunjavanje bilo intuitivno i ne bi zahtijevalo veliku količinu truda za naučiti, jedino što bi specijalizanti i mentori trebali naučiti je koristiti se aplikacijom, u smislu gdje se nalaze koji dijelovi knjižice, tj. dnevnika. No ovaj problem će biti riješen intuitivnim dizajnom te također neće zahtijevati puno truda i vremena za naučiti.

3.3. Arhitekturalne odluke i načela dizajna aplikacije

Arhitekturalne odluke, kao što je već prije u radu objašnjeno, definiraju pravila po kojima će sustav biti izgrađen. Jedna odluka već je spomenuta, a to je da standardne registracije u aplikaciju neće biti, već će administrator sustava kreirati račune za specijalizante i mentore, a oni će se zatim normalno prijavljivati u sustav. Tom odlukom eliminira se mogućnost da se u aplikaciju registrira osoba koja nije specijalizant ili mentor. Sljedeća odluka je da će se trenutni pečat i potpis mentora zamijeniti samo digitalnim potpisom. Mogućnost da se u aplikaciji daju i digitalizirani pečati bilo bi kompleksno za izvesti, a digitalni potpis je i više nego dovoljan i pruža dovoljnu sigurnost jer ga je teže falsificirati od potpisa čovjeka. Sljedeća odluka je da će knjižica o specijalističkom usavršavanju doktora medicine i dnevnik rada biti implementirani u istoj aplikaciji. Razlog tome je što će to biti praktičnije i specijalizantima i mentorima, a i sam razvoj takve aplikacije u ovom slučaju neće biti puno kompleksniji od razvoja dvije aplikacije, jedne za knjižicu i jedne za dnevnik. Posljednja odluka je da se lozinke u bazu podataka spremaju u kriptiranom obliku pomoću SHA-256 algoritma. To će povećati sigurnost aplikacije ukoliko dođe do napada na njezinu bazu podataka.

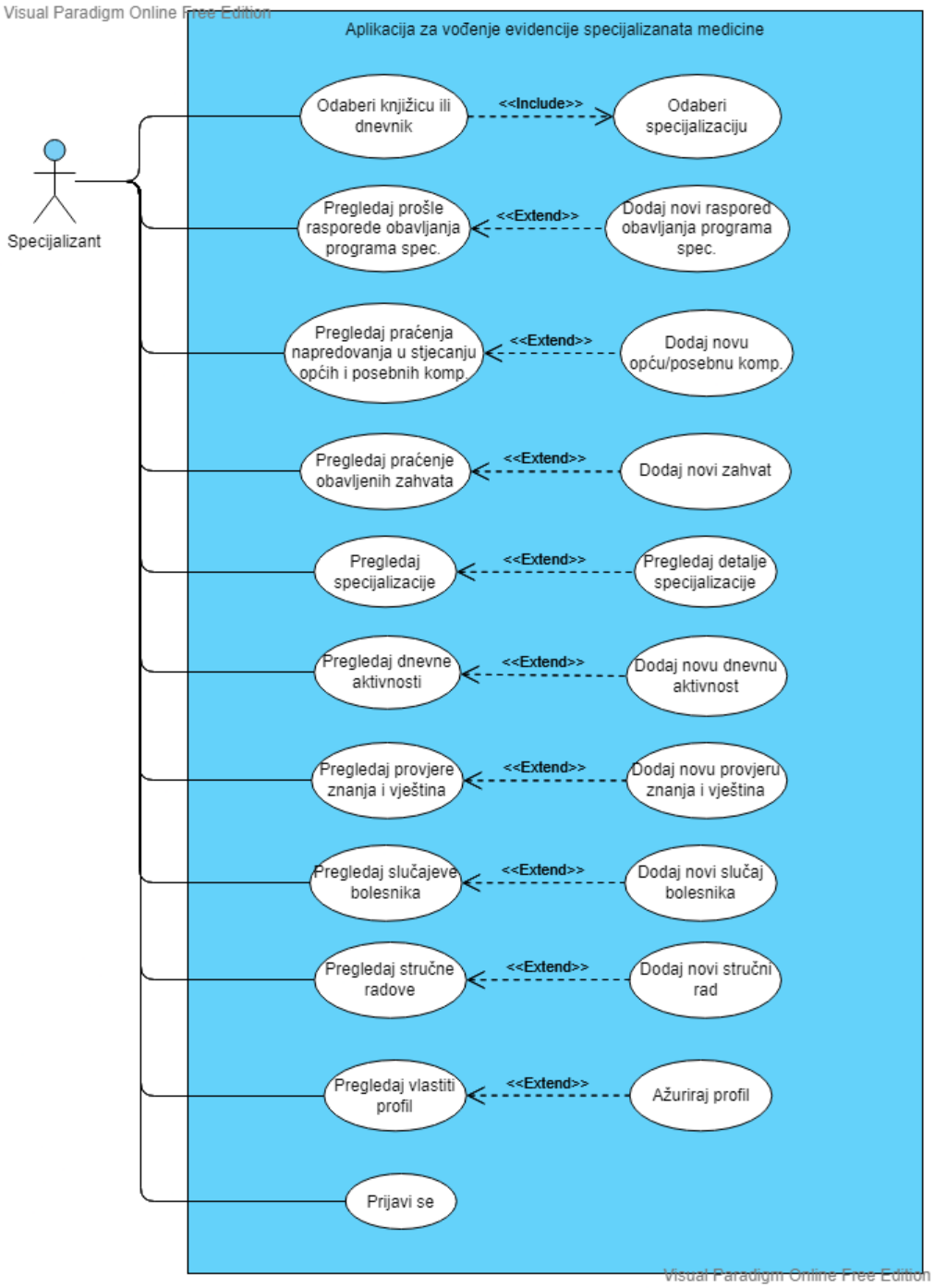
Načela dizajna, za razliku od odluka, samo su prijedlog što da se preferira tijekom implementacije aplikacije. Za ovu aplikaciju načelo dizajna je čak i standard, a to je da su varijable privatne gdje god je to moguće i da se koriste „get“ i „set“ za njihovo dohvaćanje i postavljanje. Nadalje, s obzirom na to da je arhitekturalni stil slojevit, sljedeće načelo je da slojevi budu zatvoreni. Kao što je prije u radu objašnjeno, zatvorenost slojeva znači da se slojevi ne mogu „preskakati“, tj. da prezentacijski sloj ne može direktno komunicirati sa slojem podataka, već prvo mora komunicirati sa slojem obrade, a tek onda sloj obrade komunicira sa slojem podataka.

S obzirom na to da ova aplikacije nije previše kompleksna, ovo su sve potrebne odluke i načela koja će ovu aplikaciju učiniti sigurnijom i praktičnijom za rad.

3.4. Slučajevi korištenja aplikacije

U ovom poglavlju će ukratko biti objašnjeni slučajevi korištenja aplikacije od strane specijalizanata, mentora i administratora. To će pobliže prikazati funkcionalnosti aplikacije koja će biti kreirana po arhitekturi kreiranoj u prijašnjim potpoglavljima.

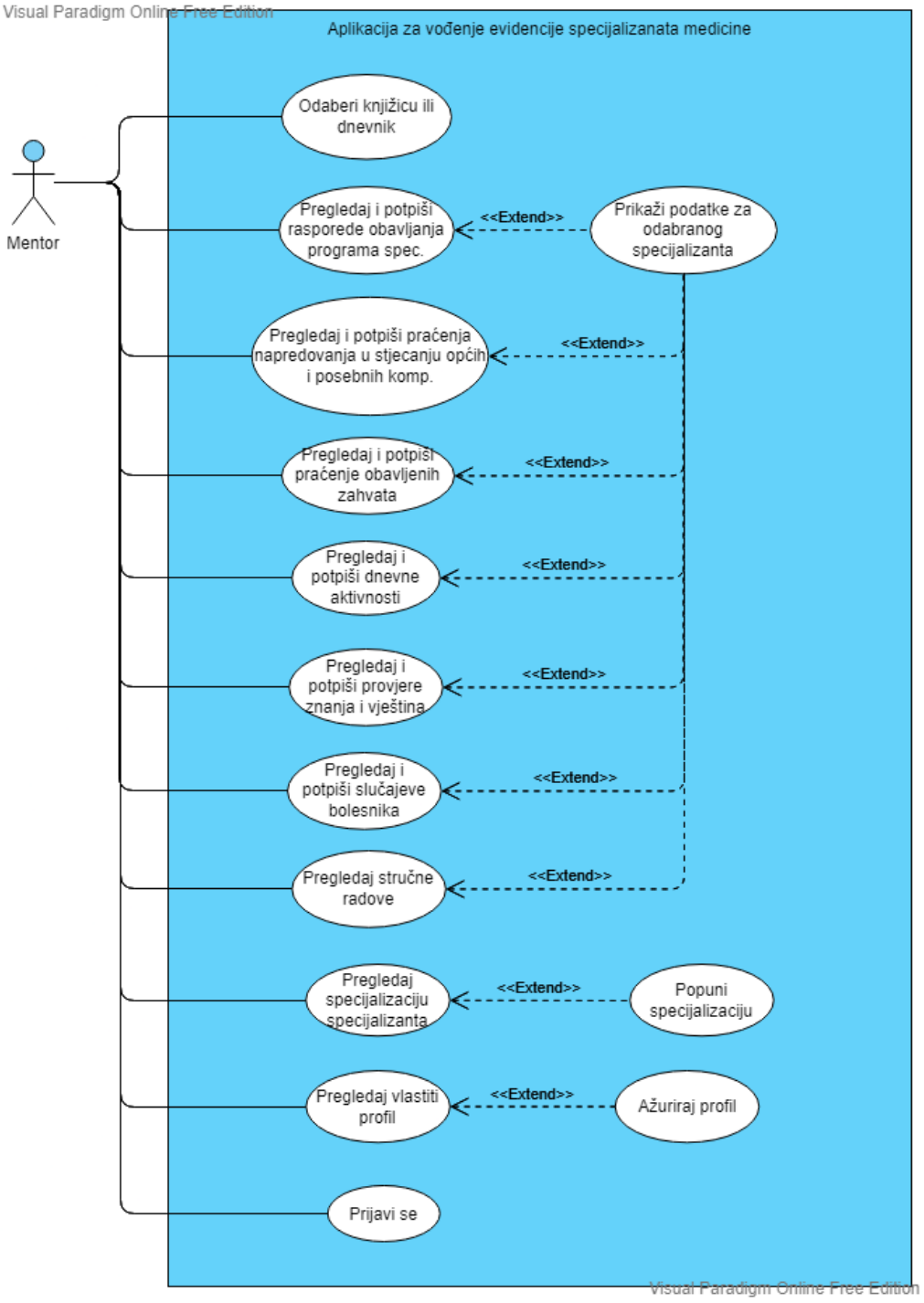
Specijalizanti korištenje aplikacije započinju prijavom u nju. Nakon što se uspješno prijave biraju hoće li pregledavati i popunjavati knjižicu o specijalističkom usavršavanju ili dnevnik rada. Također im je i omogućeno biranje specijalizacije. Specijalizanti naravno imaju više specijalizacija uz koje su vezane knjižica i dnevnik, te im se daje mogućnost da odaberu starije specijalizacije ili trenutnu. Ako odaberu knjižicu otvara se izbornik gdje biraju hoće li pregledavati dio knjižice o rasporedima obavljanja programa specijalizacije, praćenju napredovanja u stjecanju općih i posebnih kompetencija, praćenju obavljenih zahvata ili pregled samih općih podataka o trenutno odabranoj specijalizaciji. Osim pregleda, pruža im se mogućnost unašanja novih zapisa za svaku kategoriju osim za podatke o specijalizaciji. Ako s druge strane odaberu dnevnik, otvara se drugi izbornik s mogućnošću pregledavanja dnevnika aktivnosti, provjera znanja i vještina, slučajeva bolesnika ili stručnih radova. Također im je omogućen unos novih zapisa za svaku od kategorija. Osim ovih mogućnosti specijalizanti mogu vidjeti svoj vlastiti profil u aplikaciji, te ga ažurirati tako da na primjer promijene lozinku.



Slika 18: Slučajevi korištenja specijalizanata (samostalna izrada)

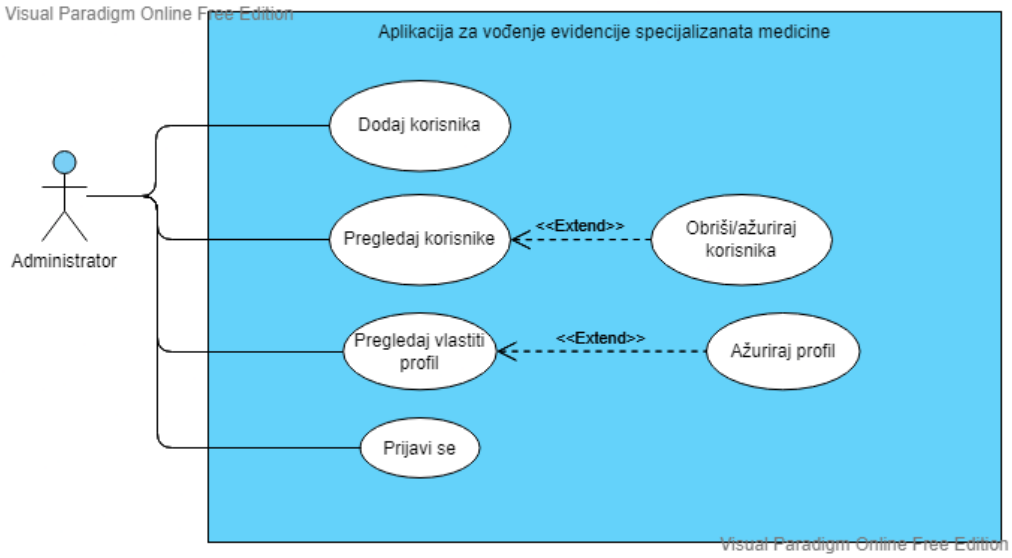
Mentori također korištenje aplikacije započinju prijavom u nju. Nakon toga biraju što žele pregledavati, knjižicu ili dnevnik. Odabirom jednog od ova dva izbora otvara im se izbornik bilo knjižice ili dnevnika. Iz izbornika knjižice mogu pregledavati zapise specijalizanata za rasporede obavljanja programa specijalizacije, praćenja napredovanja u stjecanju općih i posebnih kompetencija ili praćenja obavljenih zahvata. Osim pregledavanja zapise mogu i digitalno potpisati što odgovara potpisu i pečatu u originalnoj knjižici. Dodatno mogu filtrirati zapise po specijalizantima te mogu pregledavati detalje zapisa. Potrebno je također

napomenuti da se mentorima prikazuju samo zapisi koje još nisu digitalno potpisali i koji pripadaju specijalizantima kojima su oni mentori. Iz izbornika dnevnika mogu pregledavati i potpisivati dnevne aktivnosti, provjere znanja i vještina, slučajeve bolesnika, te samo pregledavati stručne radove specijalizanata. Osim toga, mentori mogu pregledavati specijalizacije specijalizanata kojima su glavni mentori, te ih popunjavati s potrebnim podacima. Na kraju kao i specijalizanti, mentori mogu pregledavati i ažurirati svoj profil.



Slika 19: Slučajevi korištenja mentora (samostalna izrada)

Administrator sustava rad također započinje prijavom u aplikaciju, no on ima samo mogućnosti kreiranja računa za specijalizante ili mentore, pregledavanje postojećih računa, te brisanje i ažuriranje postojećih računa. Osim ažuriranja i pregledavanja drugih računa, kao i ostale uloge ima mogućnost pregledavanja svoj profila i mogućnost ažuriranja istog.



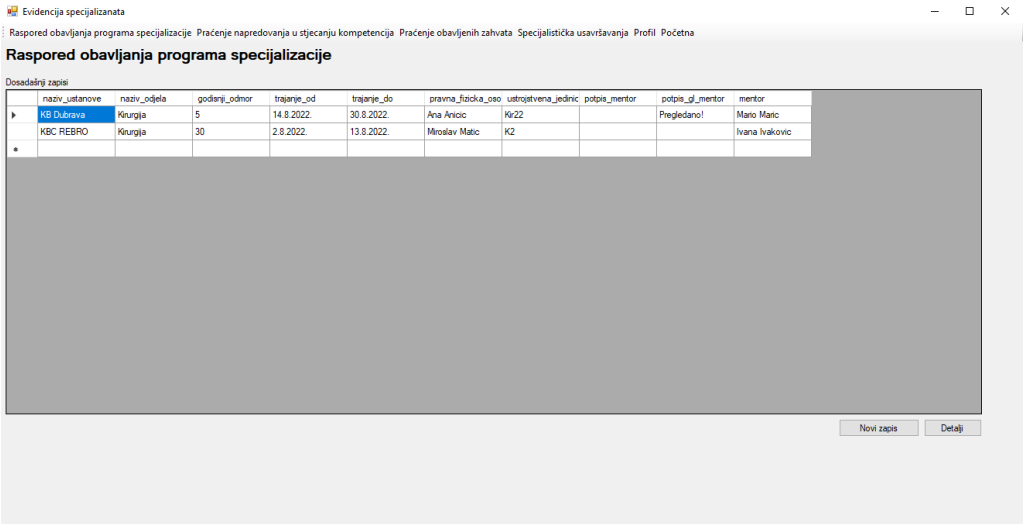
Slika 20: Slučajevi korištenja administratora (samostalna izrada)

3.5. Dokumentiranje arhitekture sustava

Nakon što smo odabrali odgovarajući arhitekturalni stil, postavili uz pomoć karakteristika kriterije uspješnosti aplikacije i nakon što smo donijeli arhitekturalne odluke i načela dizajna, važno je to sad razraditi, objasniti i prikazati pomoću raznih dijagrama.

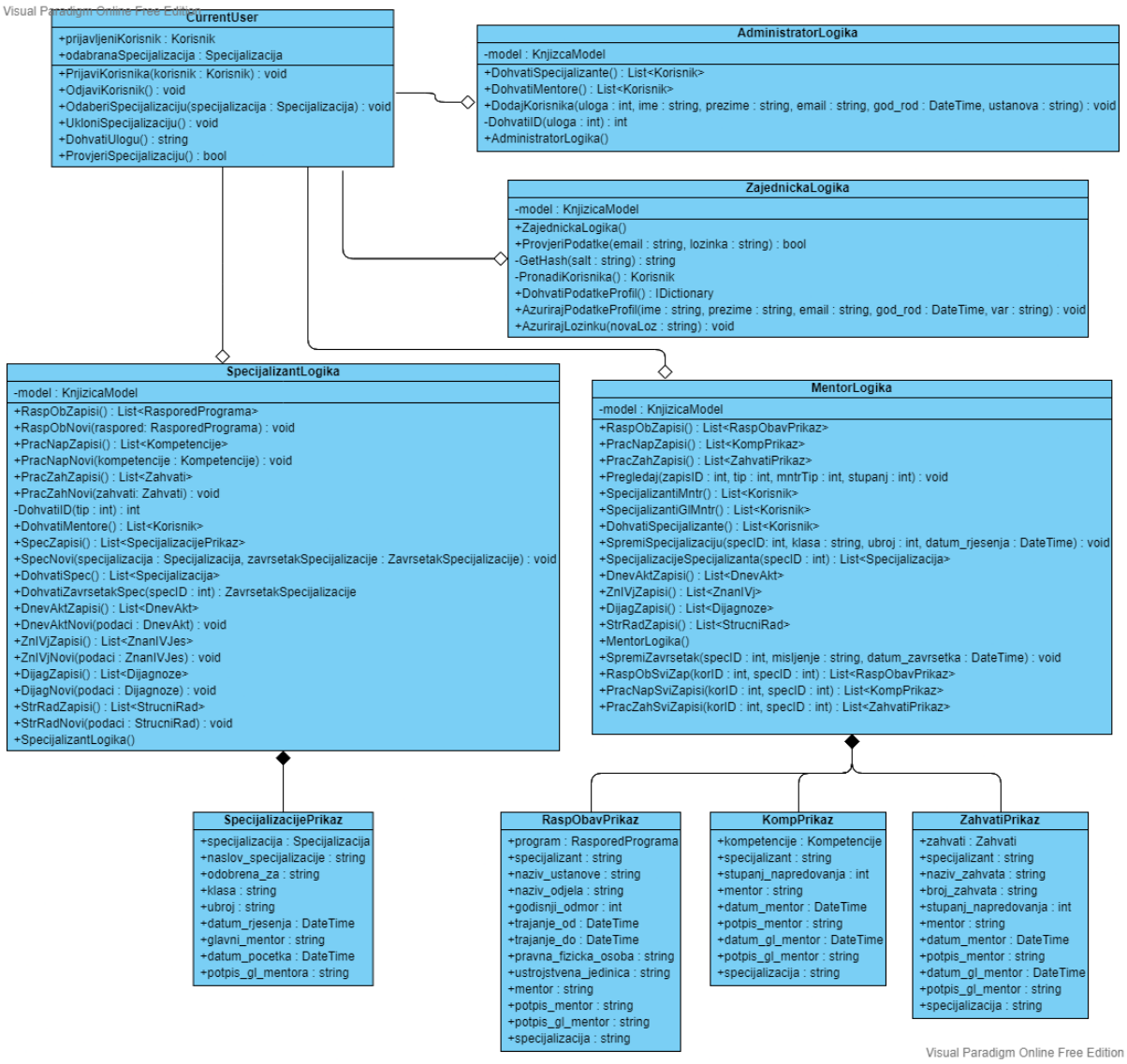
Dakle odabrali smo slojevit stil s tri sloja, a to su prezentacijski sloj, sloj obrade i sloj podataka. U prezentacijski sloj spadaju sva korisnička sučelja aplikacije. S obzirom na to da će ova aplikacija biti napravljena kao „Windows Forms“ aplikacija, u prezentacijski sloj svrstavamo sve parcijalne klase vezane uz forme i korisničke kontrole. Naime svaka forma ili korisnička kontrola sastoji se od dvije parcijalne klase, jedna za dizajn forme, tj. korisničke kontrole, a druga za logiku vezanu uz prezentaciju podataka na formama, tj. korisničkim kontrolama. Dakle u prezentacijskom sloju nalaze se sva korisnička sučelja realizirana pomoću forma ili korisničkih kontrola, od sučelja za prijavu u aplikaciju, sučelja za prikaz određenih zapisa, i druga. Podaci koji se prikazuju na korisničkim sučeljima dobavljaju se iz sljedećeg sloja, sloja obrade. Osim prikazivanja podataka, u prezentacijskom sloju se prikupljaju ulazni podaci i naredbe korisnika, koje se zatim prosljeđuju sloju obrade koji ih obrađuje. U nastavku

je za primjer prikazano korisničko sučelje s formom i korisničkom kontrolom. Forma služi kao izbornik specijalizantima, a korisnička kontrola, koja se otvara u elementu zvanom „Panel“ koji je dio forme, služi specijalizantima da pregledaju prijašnje zapise ili da dodaju nove.



Slika 21: Primjer korisničkog sučelja aplikacije (samostalna izrada)

Nadalje imamo sloj obrade koji je u aplikaciji realiziran kao biblioteka klasa. U ovom sloju nalaze se tri klase koje sadrže logiku za svaku od uloga, te nekoliko klasa koje nam pomažu prilikom rada s podacima. Ovaj sloj, dakle, pribavlja potrebne podatke od sloja podataka, obradi ih, te ih vraća prezentacijskom sloju koji ih zatim prikazuje. Osim toga, ovaj sloj prima podatke i naredbe od prezentacijskog sloja, obradi ih, te poduzima potrebne radnje kako bi vratio obrađene podatke prezentacijskom sloju ili poslao obrađene podatke sloju podataka koji ih zatim sprema u bazu podataka. Iz ovog opisa već možemo prepoznati da prezentacijski sloj nikad direktno ne pristupa sloju podataka, te se realizira zatvorenost slojeva kako je odlučeno u načelu dizajna jer razmjena poruka prvo ide kroz sloj obrade. Kako bi bolje prikazali sloj obrade, u nastavku se nalazi dijagram klasa koji prikazuje klase iz ovog sloja, te atribute i metode svake klase.



Slika 22: Dijagram klasa sloja obrade (samostalna izrada)

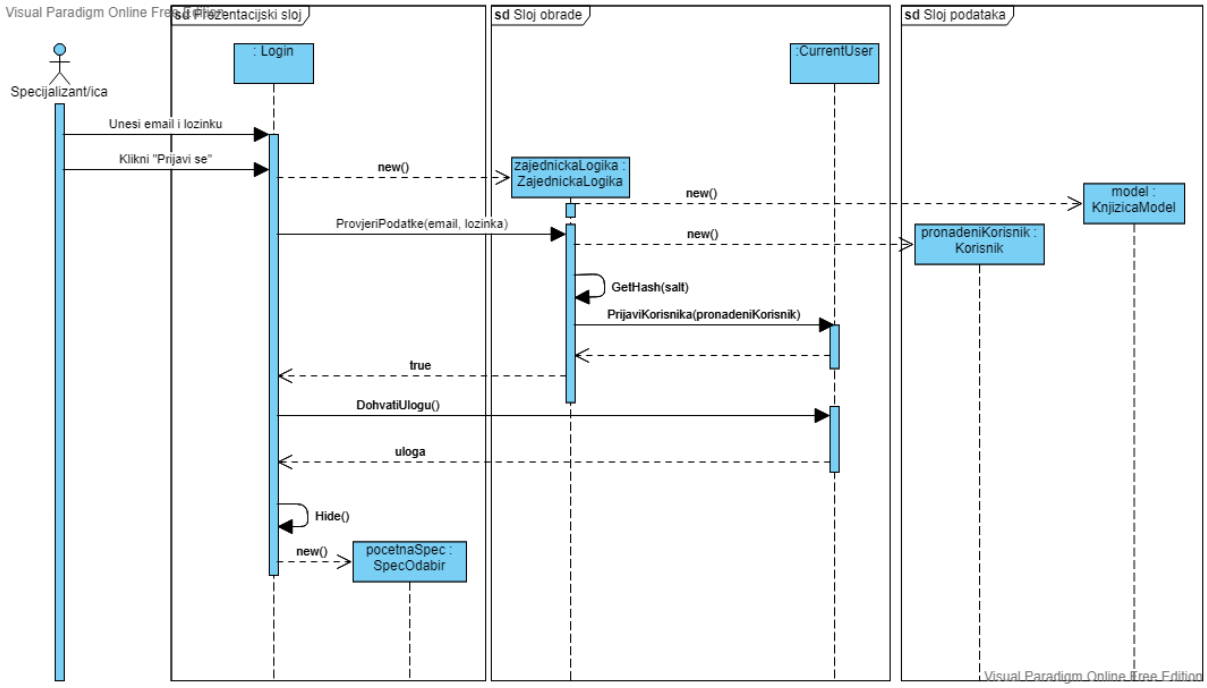
Sloj podataka sastoji se od klasa generiranih od strane „Entity Framework“-a. EF je okvir za rad sa ADO.NET-om, Microsoftovom tehnologijom koja omogućuje rad s relacijskim bazama podataka. Svaka klasa kreirana od strane EF predstavlja jednu relaciju u bazi podataka, te sadrži varijable koje su preslika atributa iz relacije. Osim što gotovo svaka relacija ima svoju klasu, relacije koje rješavaju vezu više-više ne preslikavaju se u klase već u varijable, EF kreira i klasu koja sadrži kolekcije instanci klasa kreiranih po relacijama u kojima su pohranjeni zapisi iz baze podataka. Dakle ove klase spadaju u sloj podataka jer čitanjem podataka iz tih lista, dodavanjem novih zapisa u te liste, ažuriranjem postojećih zapisa ili brisanjem istih i spremanjem promjena mi zapravo pristupamo i radimo s podacima iz baze podataka. Ovaj sloj je, dakle, zadužen za pribavljanje podataka iz baze podataka, te prosljeđivanje tih podataka sloju obrade, i pribavljanje podataka od sloja obrade, te dodavanje, ažuriranje ili brisanje tih podataka iz baze. Ove operacije postižu se tako da se u sloju obrade

deklariraju i inicijaliziraju objekti klasa iz sloja podataka, te da se ti objekti dodaju, ažuriraju ili brišu iz kolekcija koje se nalaze u zasebnoj klasi u sloju podataka, ili se zapisi iz tih kolekcija jednostavno čitaju. U nastavku će biti prikazan primjer klase kreirane od strane EF, a koja je kreirana po relaciji iz baze podataka.

```
[Table("Specijalizacija")]
public partial class Specijalizacija
{
    public Specijalizacija()
    {
        Kompetencije = new HashSet<Kompetencije>();
        RasporedPrograma = new HashSet<RasporedPrograma>();
        Zahvati = new HashSet<Zahvati>();
        ZavrsetakSpecijalizacije = new HashSet<ZavrsetakSpecijalizacije>();
    }

    [DatabaseGenerated(DatabaseGeneratedOption.None)]
    public int id { get; set; }

    public int specijalizant { get; set; }

    [Column("specijalizacija")]
    [Required]
    [StringLength(1000)]
    public string specijalizacija1 { get; set; }

    [Required]
    [StringLength(150)]
    public string odobrena_za { get; set; }

    [StringLength(100)]
    public string klasa { get; set; }

    [StringLength(100)]
    public string ubroj { get; set; }

    [Column(TypeName = "date")]
    public DateTime? datum_rjesenja { get; set; }

    public int glavni_mentor { get; set; }

    [Column(TypeName = "date")]
    public DateTime datum_pocetka { get; set; }

    [StringLength(1000)]
    public string potpis_gl_mentora { get; set; }

    public virtual ICollection<Kompetencije> Kompetencije { get; set; }

    public virtual Korisnik Korisnik { get; set; }

    public virtual Korisnik Korisnik1 { get; set; }

    public virtual ICollection<RasporedPrograma> RasporedPrograma { get;
set; }

    public virtual ICollection<Zahvati> Zahvati { get; set; }
```

```
    public virtual ICollection<ZavrsetakSpecijalizacije>
    ZavrsetakSpecijalizacije { get; set; }
}
```
Isječak koda 1: Klasa "Specijalizacija" kreirana pomoću Entity Framework-a (samostalna izrada)

Kao što je već nekoliko puta napisano, slojevi su zatvoreni, a to znači da komunicirati mogu samo prezentacijski sloj i sloj obrade, te sloj obrade i sloj podataka, prezentacijski sloj i sloj podataka ne mogu direktno komunicirati. Razlog tome je manja složenost i manja međusobna ovisnost slojeva. Dakle način na koji se poruke prenose kroz sustav je sljedeći. Krećemo od prezentacijskog sloja te u nekoj klasi u tom sloju kreiramo instancu klase iz sloja obrade koja sadrži potrebnu logiku za rad te forme, tj. klase. Ako želimo dohvatiti neke podatke za prikaz na formi, pozivamo metodu klase iz sloja obrade. Klasa iz sloja obrade koja sadrži tu metodu zatim obavlja svoju logiku, za koju koristi instance klase iz sloja podataka. Dakle klasa iz sloja obrade dohvaća pomoću instance klase iz sloja podataka potrebne podatke, te ih, ako je potrebno, restrukturira, tj. obradi, i vrati ih prezentacijskom sloju kao odgovor metode. Provođenje svih naredba i operacija za koje su potrebna sva tri sloja provodi se na vrlo sličan način. Sloj podataka podatke dobavlja iz baze podataka tako da kreira SQL upite koje šalje bazi gdje se oni izvršavaju i rezultat se vraća sloju podataka koji ih automatski prilagođava za rad aplikacije. Ovakvim načinom komunikacije dobivamo manju ovisnost među slojevima, te je moguće svaki sloj izvoditi na različitim računalima što nam pruža veću sigurnost i performanse. Za bolje razumijevanje komunikacije između slojeva u nastavku je kreiran dijagram slijeda koji prikazuje razmjene poruka između slojeva prilikom prijave specijalizanta/ice u aplikaciju. Dijagram slijeda prikazuje razmjenu poruka do trenutka kada se otvara nova forma za biranje specijalizacije.

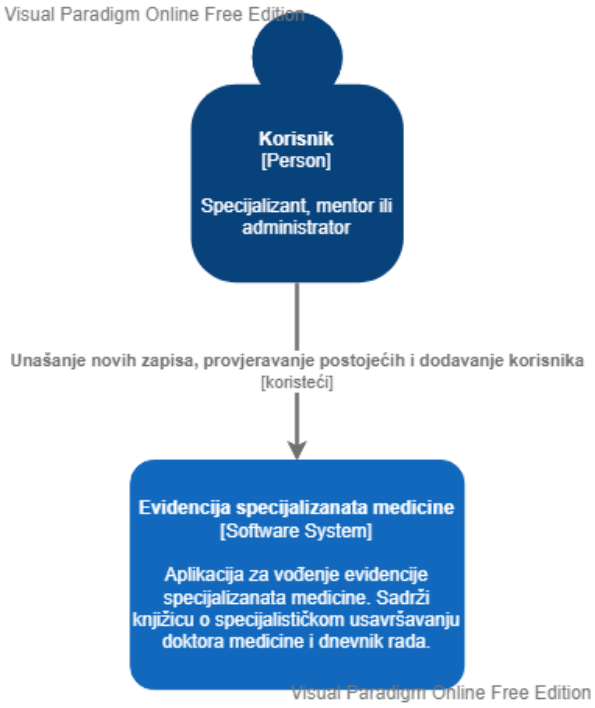


Slika 23: Dijagram slijeda prijave u aplikaciju (samostalna izrada)

3.5.1. C4 model

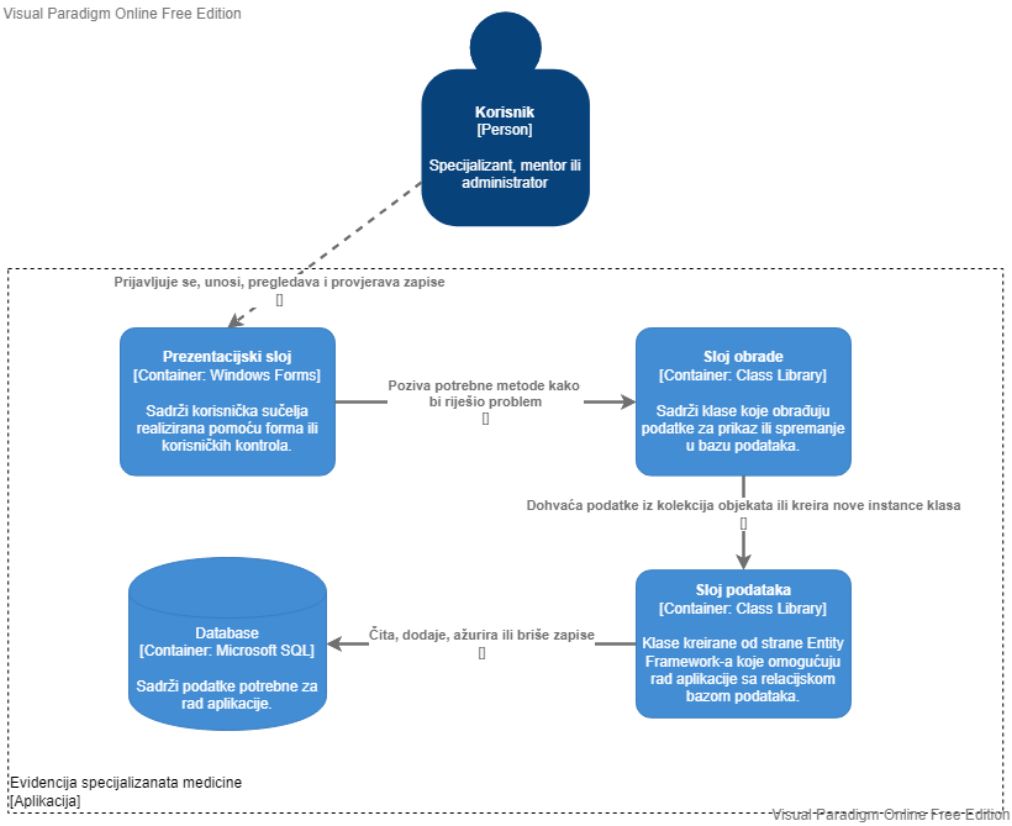
Kao što je prije u radu navedeno, C4 je dijagramska tehnika za prikazivanje arhitekture sustava. Sastoji se od četiri razine, svaka sa svojim dijagramom, s tim da je četvrti opcionalan, a to su razina konteksta, spremnika, komponenta i klasa. Svaka razina predstavlja određenu razinu detaljnosti prikaza sustava, počevši od prve razine, tj. konteksta.

Na razini konteksta gledamo „veliku sliku“ sustava, tj. sustav i njegove interakcije s okolinom. Okolinu čine korisnici sustava i drugi sustavi koje naš sustav koristi za rad. U nastavku se nalazi dijagram prve razine na kojem je prikazan naš sustav i korisnici koji se njime koriste.



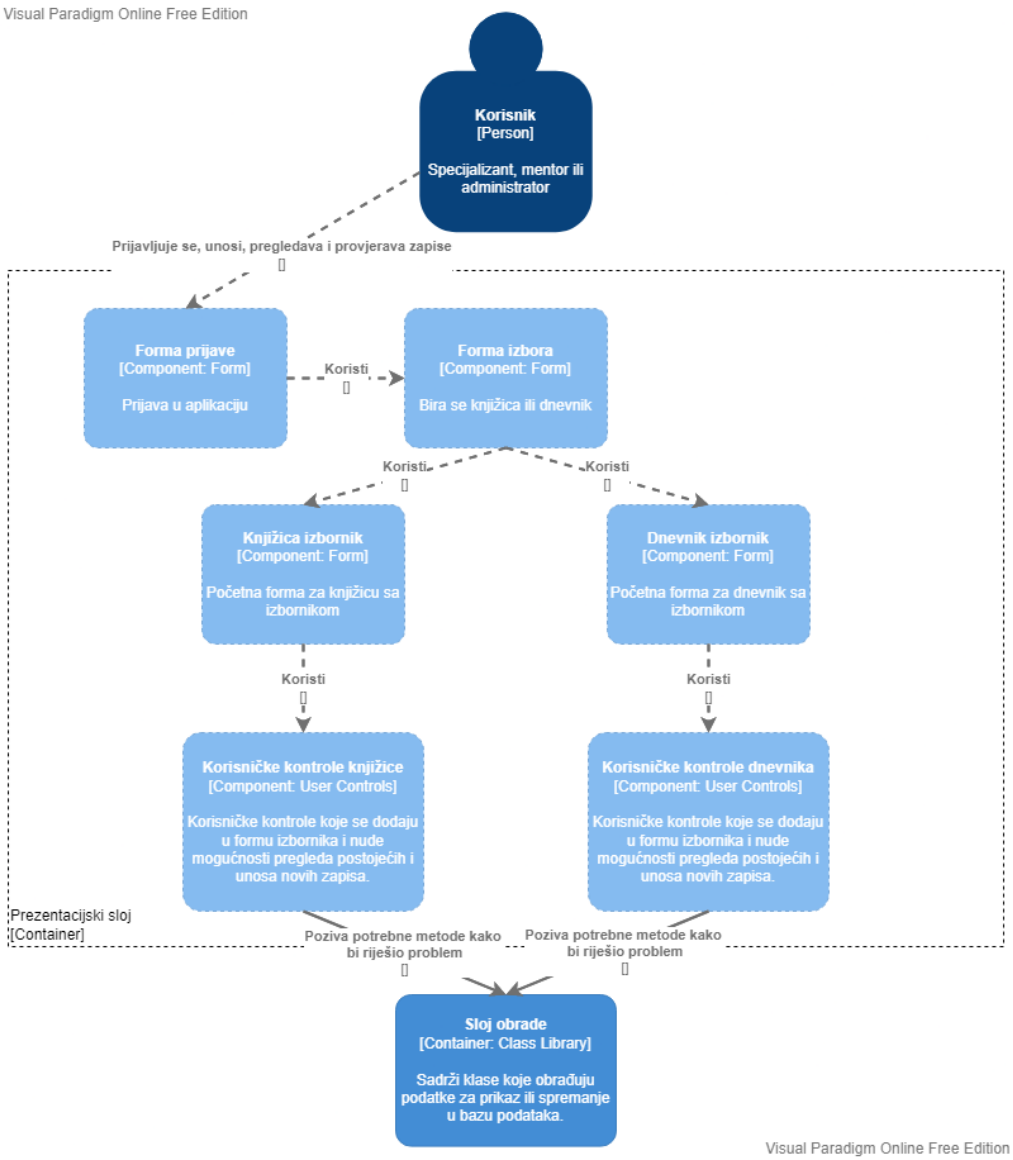
Slika 24: Prva razina, interakcija sustava i korisnika (samostalna izrada)

Sljedeća razina prikazuje sustav i spremnike koji se u njemu nalaze. Spremnici mogu biti mobilne aplikacije, računalne aplikacije, baze podataka, ili u ovom slučaju slojevi aplikacije. S obzirom na to da je stil slojevit i da se sastoji od tri sloja, uz doradu bi se svaki od slojeva mogao zasebno izvoditi na različitom računalu pa možemo zapravo svaki sloj shvaćati kao malu zasebnu aplikaciju. Iz tog razloga na drugoj razini u sustavu imamo četiri spremnika, prezentacijski sloj je prvi, sloj obrade je drugi, sloj podataka je treći i na kraju sama baza podataka je četvrti spremnik. Osim spremnika vidimo kako spremnici međusobno komuniciraju, te kako komuniciraju s okolinom. U nastavku slijedi dijagram druge razine gdje su navedeni spremnici prikazani.



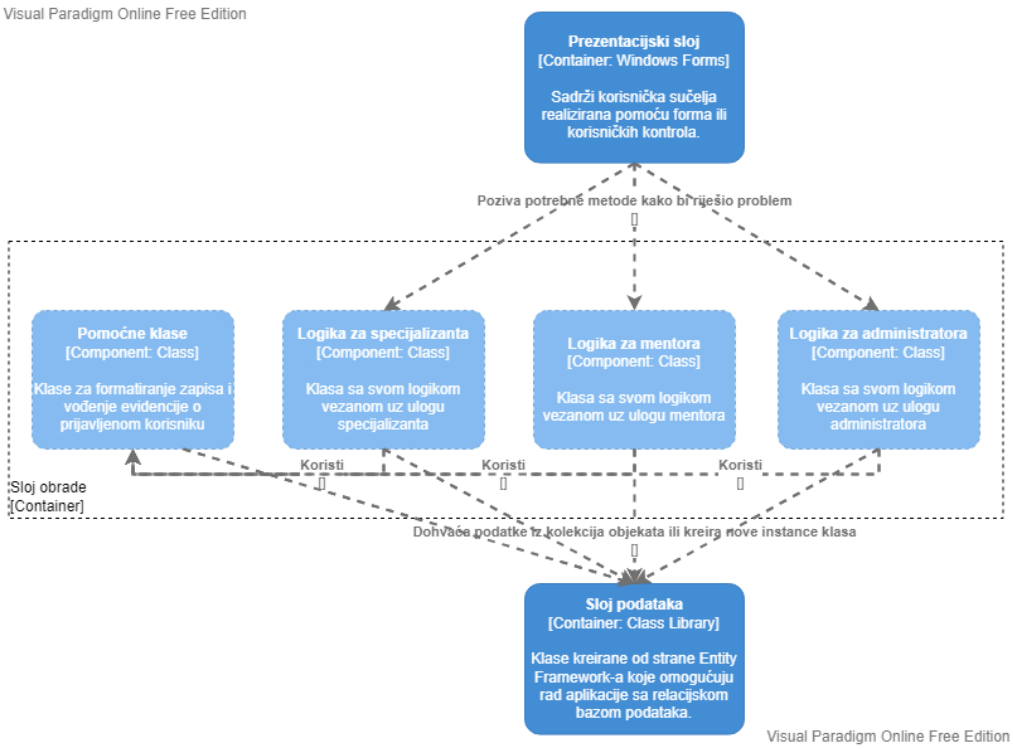
Slika 25: Druga razina, spremnici sustava (samostalna izrada)

Treća razina prikazuje detalje svakog spremnika. Dakle možemo zaključiti da kako idemo više po razinama, sustav je prikazan sve detaljnije. Svaki spremnik sastoji se od komponenata koje su važan element sustava i u nastavku će biti detaljnije prikazana sva tri sloja, tj. sva tri spremnika. Prezentacijski sloj sastoji se od forme prijave, forme izbora, izbornika za knjižicu i dnevnik, te korisničkih kontrola za knjižicu i dnevnik. Na dijagramu je prikazana komunikacija tih komponenti unutar spremnika i opis svake komponente.



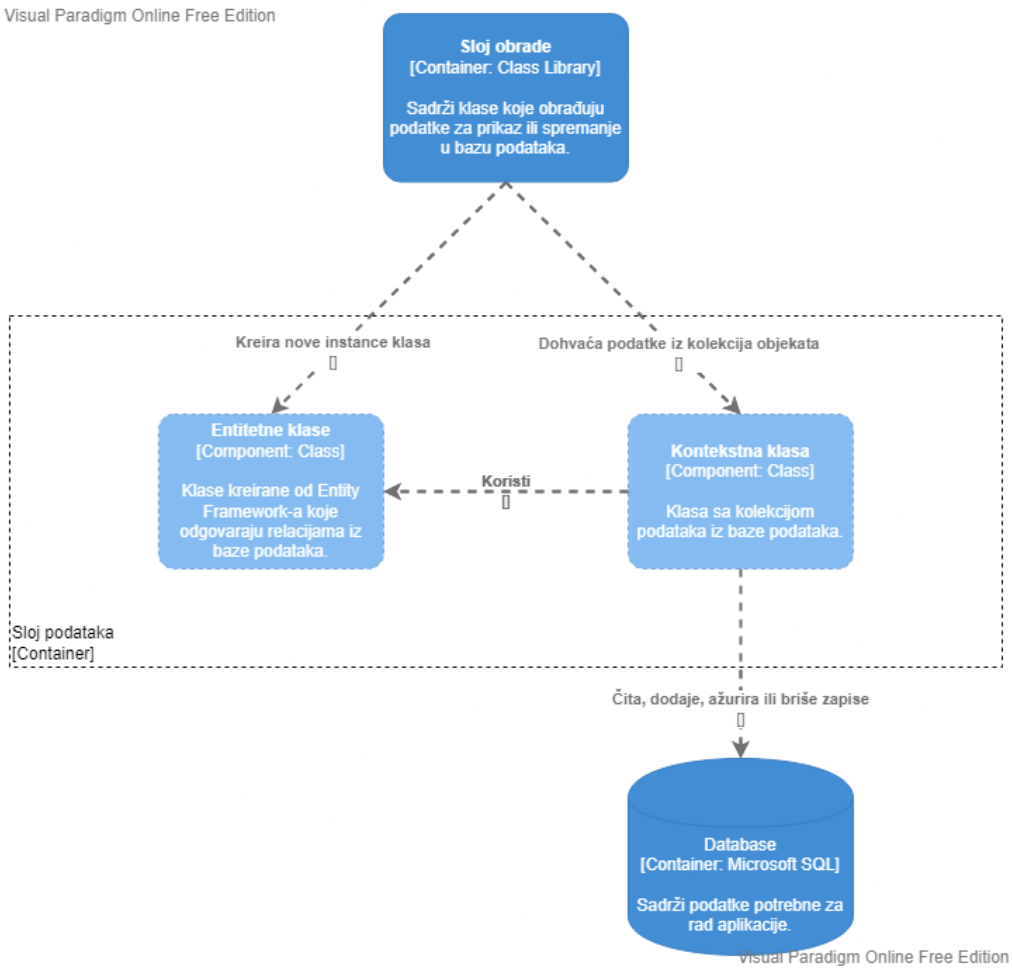
Slika 26: Treća razina, komponente prezentacijskog sloja (samostalna izrada)

Sljedeći spremnik je sloj obrade i sastoji se od pomoćnih klasa, logike za specijalizanta, mentora i administratora. Dakle to su važne komponente sloja obrade, te su prikazane na dijagramu zajedno sa njihovim opisom i interakcijama.



Slika 27: Treća razina, komponente sloja obrade (samostalna izrada)

Posljednji spremnik treće razine prikazuje komponente spremnika sloja podataka. Taj spremnik sastoji se od klasa relacija i klase modela. Komponenta ovog sloja na kraju razmjenjuje poruke s bazom podataka, a sama baza podata se ne prikazuje detaljnije na ovim dijagramima.



Slika 28: Treća razina, komponente sloja podataka (samostalna izrada)

Sljedeća razina, kako sam već prije naveo, nije nužna, a u njoj se obično sustav prikazuje dijagramima klasa, ERA dijagramom koji prikazuje strukturu baze podataka i dr. Prolazeći kroz ove tri razine možemo vidjeti kako je sustav dizajniran, a to nam pomaže pri njegovom shvaćanju.

Dijagrami korišteni za dokumentiranje arhitekture nam pomažu u vizualiziranju i shvaćanju sustava, a to je bitno kako bi u sljedećoj fazi razvoja aplikacije, fazi kodiranja, razvoj tekao uz minimalne probleme, s minimalnim zaostacima i s minimalnim greškama. Na kraju sve to dovodi do kvalitetnije aplikacije napravljene u najkraćem mogućem roku uz minimalne troškove.

4. Zaključak

Prilikom razvoja aplikacije vrlo je važno imati dobro definiranu arhitekturu kako bi njen razvoj „tekao“ bez većih problema. Imati dobro definiranu arhitekturu znači odabrati odgovarajući arhitekturalni stil, odrediti arhitekturalne karakteristike, te donijeti potrebne arhitekturalne odluke i načela dizajna. Arhitekturalnih stilova ima mnogo i ne prestaju se stvarati novi, iz tog razloga važno je biti upućen u trendove kako bi se mogla donijeti dobra odluka za korištenje stila. U ovom radu pojašnjeno je osam osnovnih stilova, a ostali su napravljeni po uzoru na njih. Arhitekturalne karakteristike zatim zadaju kriterije koje aplikacija mora ispunjavati kako bi funkcionirala pravilno i kako je zamišljeno. U ovom radu nabrojane su i objašnjene neke od važnijih karakteristika. Na kraju, arhitekturalne odluke i načela dizajna govore na koji način se gradi aplikacija u fazi implementacije.

U ovom radu je uz korištenje teorije o arhitekturi sustava napravljena arhitektura aplikacije za vođenje evidencije specijalizanata medicine. U toj arhitekturi odlučeno je korištenje slojevitog stila s tri sloja jer je odgovarao prirodi aplikacije. Osim ovog stila, također su donesene arhitekturalne karakteristike za ovu aplikaciju, te odluke i načela dizajna, a zatim je sama arhitektura dokumentirana uz pomoć raznih dijagramskih tehnika. Ova arhitektura je zatim iskorištena za izradu aplikacije kao „proof-of-concept“.

Aplikacija je izgrađena u Microsoft-ovom Visual Studio-u kao „Windows Forms“ aplikacija. Visual Studio je zapravo razvojno okruženje koje omogućuje razvoj u više programskih jezika, te također ima i mnogo dodataka koji ga proširuju, a u ovom radu je korišten za razvoj Windows aplikacija korištenjem C# programskog jezika. Sam razvoj aplikacije bio je zapravo vrlo jednostavan jer sam znao što raditi i na koji način to raditi. Dakle uz pomoć arhitekture aplikacije znao sam da aplikacija mora imati tri sloja, te sam ih odredio tako kako je zadano. U prezentacijskom sloju nalaze se forme i korisničke kontrole koje samo prikazuju i prikupljaju podatke. Sloj obrade zatim šalje prikupljene podatke sloju podataka ili dohvaća potrebne podatke od sloja podataka. Te na kraju sloj podataka sprema, čita, ažurira ili briše podatke iz baze podataka. Programiranje slojeva bilo je izravno jer sam znao što trebam napraviti. Što se tiče karakteristika, odluka i načela dizajna, uz pomoć njih znao sam kako oblikovati aplikaciju da ona radi kako je to zamišljeno. Arhitektura aplikacije mi je prilikom programiranja, tj. faze implementacije ili kodiranja, skratila vrijeme u smislu stajanja s radom i razmišljanja na koji način da ostvarim neku funkcionalnost ili na koji način da oblikujem aplikaciju, te mi je pomogla u smislu da sam znao što radim te sam tako imao manji broj grešaka. Nakon ovakvog načina razvoja aplikacije, dakle uz dobro kreiranu arhitekturu aplikacije, mogu zaključiti da je arhitektura aplikacije neophodna za razvoj aplikacije, pogotovo za razvoj kompleksnijih aplikacija, jer nam na ljudima razumljiv način prikazuje funkcionalnosti

aplikacije, te način na koji izgraditi i oblikovati aplikaciju, a sve to skraćuje vrijeme razvoja aplikacije, olakšava sam razvoj i dovodi do kvalitetnije aplikacije uz minimalne troškove razvoja.

Popis literature

bez autora (20.9.2021.). *EVENT-DRIVEN ARCHITECTURE TOPOLOGIES – BROKER AND MEDIATOR*. Dohvaćeno iz 3pillar global: https://www.3pillarglobal.com/insights/event-driven-architecture-topologies-broker-and-mediator/

bez autora (7.7.2022.). *Microservice architecture style.* Dohvaćeno iz Microsoft: https://docs.microsoft.com/en-us/azure/architecture/guide/architecture-styles/microservices

Brown, S. (n.d.). *The C4 model for visualising software architecture*. Dohvaćeno iz c4model: https://c4model.com/

Heusser, M. (30.3.2020.). *5 key software testability characteristics*. Dohvaćeno iz TechTarget: https://www.techtarget.com/searchapparchitecture/tip/5-key-software-testability-characteristics

Kayak, S. (18.01.2021.). *Understand 3-Tier Architecture in C#*. Dohvaćeno iz C#Corner: https://www.c-sharpcorner.com/UploadFile/dacca2/understand-3-tier-architecture-in-C-Sharp-net/

Krnić, L. G. (4.10.2018.). *Još jedan dokaz bolesne birokracije: bez tisuća pečata u dvjema knjižicama u Hrvatskoj ne možeš postati liječnik specijalist?!* Dohvaćeno iz Slobodna dalmacija: https://slobodnadalmacija.hr/vijesti/hrvatska/jos-jedan-dokaz-bolesne-birokracije-bez-tisuca-pecata-u-dvjema-knjizicama-u-hrvatskoj-ne-mozes-postati-lijecnik-specijalist-568089

Nolle, T. (1.2.2020.). *service-oriented architecture (SOA)*. Dohvaćeno iz TechTarget: https://www.techtarget.com/searchapparchitecture/definition/service-oriented-architecture-SOA

Polak, A. (27.5.2021.). *How to document software architecture?* Dohvaćeno iz The software house: https://tsh.io/blog/how-to-document-your-architecture/

Richards, M. (2015). *Software Architecture Patterns.* Sebastopol: O'Reilly Media.

Richards, M., & Ford, N. (2020). *Fundamentals of Software Architecture.* Sebastopol: O'Reilly.

Sangwan, R. (2014). *Software and System Architecture in Action.* Boca Raton: Auerbach Publications.

Schalme, B. (n.d.). *When to use the Pipeline Architecture Style*. Dohvaćeno iz Airspeed Consulting: https://airspeed.ca/when-to-use-the-pipeline-architecture-style/

Wickramarachchi, A. (15.9.2017.). *Event Driven Architecture Pattern*. Dohvaćeno iz Towards Data Science: https://towardsdatascience.com/event-driven-architecture-pattern-b54fc50276cd

Zulqadar, A. (12.2.2019.). *SDLC Waterfall Model: The 6 phases you need to know about*. Dohvaćeno iz Rezaid: https://rezaid.co.uk/sdlc-waterfall-model/

Popis slika

Popis tablica

Popis isječaka koda

Prilozi

Poveznica na GitHub repozitorij sa „proof-of-concept“ aplikcijom:

https://github.com/Capek66/Vodenje-evidencije-specijalizanata-medicine