Modeliranje programske potpore UML dijagramima

Frid, Nikolina; Jović, Alan

UDŽBENICI SVEUČILIŠTA U ZAGREBU

MANUALIA UNIVERSITATIS STUDIORUM ZAGRABIENSIS

Modeliranje programske potpore UML dijagramima

Sveučilišni priručnik

Nikolina Frid, Alan Jović

UDŽBENICI SVEUČILIŠTA U ZAGREBU

MANUALIA UNIVERSITATIS STUDIORUM ZAGRABIENSIS

NIKOLINA FRID, ALAN JOVIĆ

MODELIRANJE PROGRAMSKE POTPORE UML DIJAGRAMIMA

SVEUČILIŠNI PRIRUČNIK

Uporabu ovog sveučilišnog priručnika odobrio je Senat Sveučilišta u Zagrebu.

(Klasa: 032-01/24-02/26, Urbroj: 251-25-07-01/2-24-5 od 19. studenog 2024.)

Elektroničko izdanje

Zagreb, 2024.

Sadržaj

Popis slika

Popis tablica

Predgovor

Programsko inženjerstvo (engl. *software engineering*) suvremena je i široka tehnička disciplina u području računarstva koja obuhvaća niz procesa kojima se nastoji razviti funkcionalni programski proizvod sa značajkama koje ispunjavaju zahtjeve naručitelja. U okviru programskog inženjerstva se od devedesetih godina prošlog stoljeća razvija modelno-usmjereni pristup (engl. *model-based design*) razvoju programske potpore. Nastao je kao posljedica ubrzanog rasta veličine i složenosti programa koji su u velikoj mjeri nadišli mogućnosti neorganiziranog i nestrukturiranog razvoja. Modelno-usmjereni pristup primarno karakterizira razvoj modela arhitekture programske potpore i njihovo dokumentiranje, a u praksi se ostvaruje nizom alata i vizualnih jezika za izradu dijagrama. Jedan od najpopularnijih jezika, koji se s vremenom etablirao kao industrijska norma u modelno-usmjerenom razvoju unificirani je jezik za modeliranje – UML. Zbog tog je grafičkog jezika, i njegove sintakse i semantike, na temelju razumnog broja raznovrsnih dijagrama na relativno jednostavan i inženjerima pristupačan način omogućeno modeliranje razvoja programske potpore.

U ovom se priručniku, koji se sastoji od dva dijela, opisuje modeliranje programske potpore jezikom UML. U prvom se dijelu razmatra sintaksa i semantika UML-a, najnovije inačice 2.5.1. iz 2017., s naglaskom na sedam u praksi dokazano korisnih UML dijagrama. Pritom se kroz brojne primjere za svaku vrstu dijagrama pokazuje kako ispravno modelirati pojedine aspekte projekta razvoja programske potpore, s naglaskom na arhitekturi i implementaciji. Drugi se dio sastoji od zadataka za vježbu za svaku obrađenu vrstu dijagrama (s rješenjima). Čitatelju savjetujemo da najprije samostalno prouči prvi dio priručnika i potom proba riješiti zadatke za vježbu, a da tek onda provjeri rješenja.

Ovaj je priručnik u prvom redu namijenjen studentima prijediplomskog studija računarstva Fakulteta elektrotehnike i računarstva, Sveučilišta u Zagrebu u okviru predmeta „Programsko inženjerstvo”, ali i studentima srodnih studija te široj zainteresiranoj publici. Želja je autora da on pomogne svim razvojnim inženjerima kojima manjka formalno obrazovanje o jeziku UML, a žele ga naučiti da bi ga uspješno primijenili u svojoj karijeri.

Zagreb, studeni 2024.

Autori

Sintaksa i semantika

1. UML – jezik i norma

UML (engl. *Unified Modeling Language*) normirani je vizualni jezik koji se koristi za modeliranje programske potpore. Pomoću jezika UML mogu se izraditi UML dijagrami – grafički prikazi koji se koriste za modeliranje programske potpore. Oni pružaju normirani i vizualni način za prikazivanje arhitekture, oblikovanja i ponašanja složenih programskih sustava, što olakšava komunikaciju, suradnju i razumijevanje među razvojnim inženjerima. UML dijagrami pomažu u održavanju dosljednosti i jasnoće u dokumentaciji projekata te u uočavanju problematičnih područja prije implementacije.

UML dijagrame je u devedesetim godinama prošlog stoljeća predložila grupa razvojnih inženjera predvođenih Gradyjem Boochom, Jamesom Rumbaughom i Ivarom Jacobsonom, koji su u jedinstvenu normu objedinili različite načine bilježenja modela programske potpore koji su se dotad koristili pri razvoju, a inače potječu iz ranije razvijenog objektno orijentiranog pristupa analizi i dizajnu (engl. *Object-Oriented Analysis and Design*, OOAD), tehnike za objektno modeliranje (engl. *Object Modeling Technique*, OMT) i objektno orijentiranog programskog inženjerstva (engl. *Object-Oriented Software Engineering*, OOSE). Glavni je cilj bio pružiti normirani način bilježenja (notaciju) za modeliranje programskih sustava, koji se temelji na najboljim praksama i tehnikama koje koriste iskusni razvojni inženjeri.

Od tada se UML kontinuirano razvija te se danas smatra najčešće korištenom grafičkom notacijom za modeliranje programske potpore. Razlog njegove popularnosti leži u tome što ga podržava velik broj programskih alata i radnih okvira, što olakšava razvojnim inženjerima izradu, izmjenu i dijeljenje UML dijagrama. Zbog svoje opće prihvaćenosti, UML dijagrami postali su nezaobilazan alat u razvoju programske potpore, a njihova primjena omogućuje brže i učinkovitije kreiranje i razvijanje složenih programskih sustava.

1.1 Norma UML

Norma UML (engl. *UML standard*) službeni je dokument[1] kojim se utvrđuje i objašnjava uporaba jezika UML. Normom UML definira se sintaksa i semantika jezika UML i pružaju se upute o tome kako se on treba koristiti u različitim situacijama. Normu UML razvio je i održava Object

[1] https://www.omg.org/spec/UML/2.5.1/About-UML

Management Group (OMG)[2], organizacija koja se bavi normiranjem programskih tehnologija.

Prva inačica UML-a, nazvana UML 0.8, objavljena je 1996. Uslijedilo je nekoliko revizija i proširenja, a UML 1.0 objavljen je 1997. Najnovija je inačica UML 2.5.1, objavljena 2017. U nastavku je popis glavnih inačica UML-a s godinama i detaljima promjena:

- **UML 0.8 (1996.)** – prva inačica UML-a, koju su stvorili Booch, Rumbaugh i Jacobson, uključivala je osnovnu notaciju i koncepte UML-a, kao što su razredi, pridruživanja i obrasci uporabe.
- **UML 1.0 (1997.)** – uvedena su značajna poboljšanja i proširenja, kao što su sekvencijski dijagrami, dijagrami suradnje i dijagrami stanja. Također, u ovoj je inačici uveden koncept stereotipa i označenih vrijednosti.
- **UML 1.1 (1998.)** – dodano je više značajki, poput dijagrama aktivnosti, dijagrama komponenti i dijagrama razmještaja. Isto tako, u ovoj je inačici uveden koncept sučelja.
- **UML 1.3 (1999.)** – donosi manja poboljšanja i pojašnjenja prethodne inačice, poput ponovnog definiranja semantike krajeva pridruživanja i uvođenja koncepta paketa.
- **UML 1.4 (2001.)** – ova inačica UML-a uključivala je daljnja poboljšanja i proširenja, no nije u potpunosti kompatibilna s UML 1.3. Promjene su uglavnom vezane za značajke vidljivosti, elemente dijagrama interakcije te dijagrame komponenti. Uveden je pojam artefakta kao fizičke realizacije komponente. Podinačica 1.4.2 postala je 2005. dio ISO norme – ISO/IEC 19501.
- **UML 2.0 (2003.)** – prva značajna izmjena UML-a kojom su uvedene mnoge nove značajke i poboljšanja, poput novog metamodela za UML, normiranog formata XMI za razmjenu UML modela te koncepta profila za prilagođavanje UML-a za specifične domene ili platforme. Uvode se novi dijagrami, kao npr. dijagrami objekata, paketa, kompozitne strukture, pregleda interakcije, vremenski dijagrami i dijagrami profila. Dijagrami suradnje postaju komunikacijski dijagrami. Dodatna poboljšanja uvedena su u dijagrame aktivnosti, razreda, sekvencijske dijagrame itd.
- **UML 2.1 (2005.)** – donosi manja poboljšanja i ispravke u odnosu na inačicu 2.0, kao što su razjašnjavanje semantike agregacije i kompozicije te poboljšanje specifikacije dijagrama stanja.
- **UML 2.2 (2009.)** – uključuje dodatna poboljšanja i proširenja vezana za dijagrame aktivnosti i vremenske dijagrame. Uveden je koncept pridruženih razreda.
- **UML 2.3 (2010.)** – manja poboljšanja i pojašnjenja vezana za pridružene razrede, dijagrame komponenti i dijagrame aktivnosti.
- **UML 2.4.1 (2011.)** – uvedena su dodatna poboljšanja u dijagrame razreda i paketa. Također dolazi do značajnih promjena vezanih za akcije i događaje.
- **UML 2.5 (2015.)** – druga velika izmjena UML-a kojom su uvedene mnoge nove značajke i poboljšanja, posebno u pogledu strukture i organizacije dokumenta specifikacije UML-a. Napravljen je velik pomak u razjašnjavanju nejasnoća i dvosmislenih tumačenja iz prijašnjih verzija te su popravljene neke prethodno uočene pogreške. Uvedena je podrška za ugniježđene klasifikatore, a poboljšana je i podrška za modeliranje operacija i mogućnost definiranja alternativnih sintaksi za UML.

[2] http://www.omg.org/

- **UML 2.5.1 (2017.)** – uključuje manje ispravke i pojašnjenja, kao što su ispravljanje pogrešaka u specifikaciji i poboljšanje definicije određenih koncepata UML-a.

Ukratko, UML je s godinama prošao mnoge izmjene i proširenja, pri čemu su svakom inačicom dodavane nove značajke te se poboljšavala specifikacija postojećih. Unatoč razvoju, ostao je vjeran svojem izvornom cilju pružanja normiranog i vizualnog jezika za modeliranje programske potpore, što pomaže u osiguravanju dosljednosti, kompatibilnosti i interoperabilnosti između različitih sustava i alata. Njegova posljednja inačica, UML 2.5.1, najopsežnija je i najnovija specifikacija UML-a, a danas se široko koristi u razvoju programske potpore.

1.2 UML dijagrami

Postoji mnogo vrsta UML dijagrama. Neki od najčešće korištenih su dijagrami obrazaca uporabe, dijagrami razreda, sekvencijski dijagrami, dijagrami aktivnosti, dijagrami komponenti itd. Svaka vrsta UML dijagrama služi određenoj svrsi i prikazuje različite aspekte dizajna ili ponašanja sustava. Na primjer, dijagrami obrazaca uporabe koriste se za modeliranje interakcije između sustava i korisnika, dok se dijagrami razreda koriste za modeliranje strukture sustava u pogledu njegovih razreda i odnosa. Ovisno o specifičnostima programskog sustava koji se razvija, može se koristiti jedna vrsta dijagrama ili više njih da bi se što preciznije prikazala funkcionalnost i arhitektura sustava.

UML dijagrami ključni su za razvoj programske potpore zbog mnogo razloga. Prije svega, pružaju normirani vizualni jezik koji razvojnim inženjerima omogućuje pretvaranje složenih ideja u jednoznačan i jezgrovit vizualni model koji će lakše komunicirati s članovima tima, dionicima i klijentima. U usporedbi s korištenjem prirodnog jezika ili obične normirane grafičke notacije, izrada modela korištenjem UML dijagrama osigurava konzistentnost i unificiranost tako da svi uključeni u projekt imaju zajedničko razumijevanje arhitekture i ponašanja sustava. To rezultira i mogućnošću integracije UML modela u same projekte i automatizaciju provjere modela i generiranja programskog koda na temelju modela.

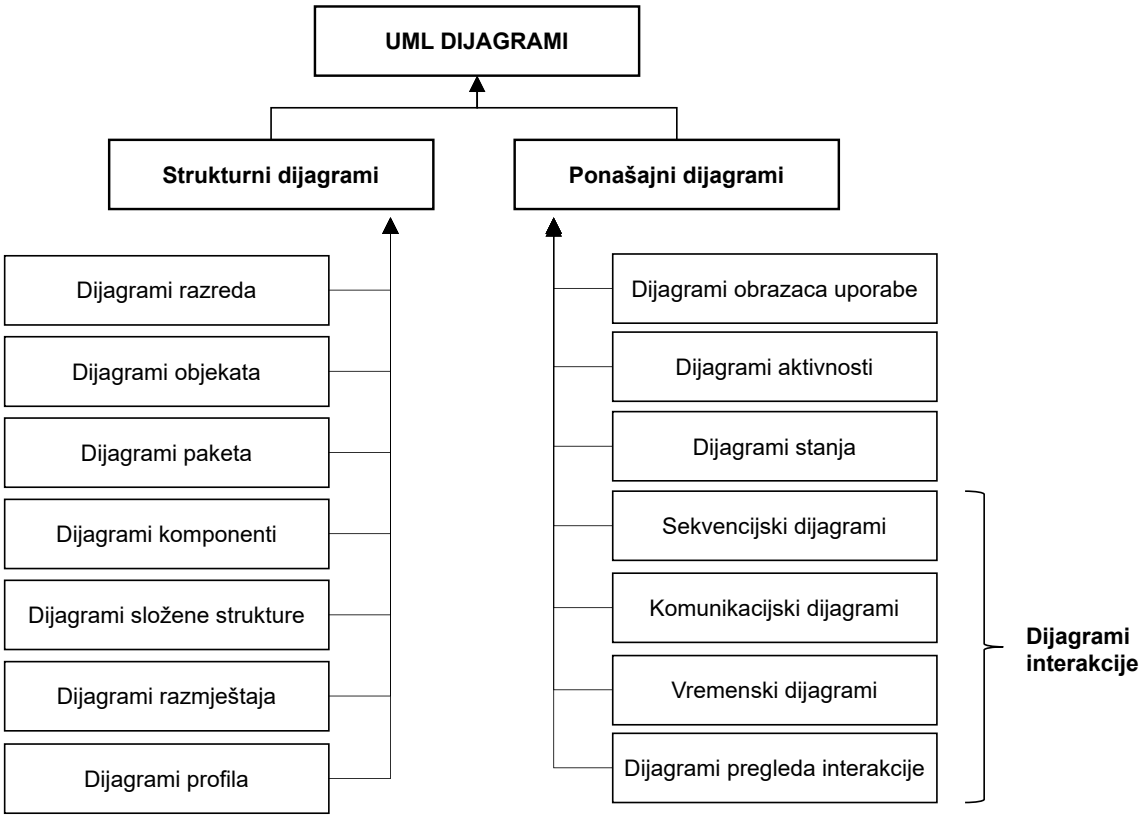
Nadalje, UML dijagrami mogu se koristiti u cijelom procesu razvoja programske potpore, počevši od specifikacije, preko oblikovanja i implementacije, pa sve do održavanja sustava. Na primjer, pri analizi zahtjeva sustava UML dijagrami olakšavaju prepoznavanje ključnih zahtjeva, sudionika te potencijalnih problema i ovisnosti. Isto tako, pružaju način za izražavanje odluka o oblikovanju na normiran i organiziran način, što omogućuje uzimanje u obzir svih važnih aspekata oblikovanja sustava, poput njegove strukture, ponašanja i interakcija. Modeliranjem programskog sustava korištenjem UML-a, razvojni inženjeri mogu simulirati različite scenarije i ispitati ponašanje sustava pod različitim uvjetima.

Konačno, UML dijagrami koriste se i za dokumentaciju, održavanje i nadogradnju programske potpore. Dijagrami pružaju vizualnu referencu za razvojne inženjere, voditelje projekata i druge dionike tako što im pomažu da razumiju kako je sustav strukturiran, kako funkcionira te da razumiju interakcije s drugim sustavima. Pri tome je važno istaknuti potrebu za kontinuiranim ažuriranjem dijagrama za vrijeme razvoja i nadogradnje sustava da bi dokumentacija i stvarni sustav bili u skladu.

1.2.1 Vrste dijagrama

Prema najnovijoj normi UML-a, UML 2.5.1, postoji 14 vrsta UML dijagrama, a oni se mogu svrstati u dvije glavne kategorije: strukturni dijagrami i ponašajni dijagrami. Unutar kategorije ponašajnih dijagrama, zasebnu podskupinu čine dijagrami interakcije, koji su usmjereni na interakciju i

suradnju dijelova sustava te interakciju sustava s okolinom. Na slici 1.1 prikazana je klasifikacija UML dijagrama prema normi UML 2.5.



Slika 1.1: Klasifikacija UML dijagrama prema normi UML 2.5.

Strukturni dijagrami (engl. *structure diagram*) koriste se za modeliranje statičkih aspekata sustava, uključujući njegovu strukturu, komponente i odnose među njima. Postoji ukupno sedam vrsta strukturnih dijagrama:

- **Dijagram razreda** (engl. *class diagram)* – prikazuje razrede, atribute, operacije i odnose među razredima u programskom sustavu. Riječ je o najčešće korištenom UML dijagramu, koji se upotrebljava za modeliranje strukture programskih sustava koji se temelje na objektno orijentiranoj programskoj paradigmi.
- **Dijagram objekata** (engl. *object diagram*) – usko je vezan za dijagram razreda i prikazuje sliku sustava (engl. *snapshot*) u određenom trenutku tako što prikazuje njegove objekte i njihove odnose. Koristi se za modeliranje specifičnih instanci sustava i prikazivanje ponašanja sustava u različitim situacijama.
- **Dijagram paketa** (engl. *package diagram*) – prikazuje ovisnosti između paketa i njihov sadržaj (razredi i komponente). Upotrebljava se za modeliranje organizacije sustava u logičke grupe i za prikazivanje odnosa među različitim dijelovima sustava.
- **Dijagram komponenti** (engl. *component diagram*) – vrsta strukturnog dijagrama koji prikazuje logičke komponente sustava (veće cjeline od razreda) i njihove odnose. Služi za modeliranje arhitekture sustava i prikazivanje toga kako se komponente međusobno povezuju.
- **Dijagram složene strukture** (engl. *composite structure diagram*) – prikazuje unutarnju strukturu razreda ili komponente - podrazrede ili podkomponente, sučelja itd. Također

omogućuje navođenje atributa i višestrukost na vezama između elemenata. Dijagram složene strukture sadržava mješavinu sintakse i semantike dijagrama razreda i dijagrama komponenti te nije ograničen na objektno orijentiranu paradigmu. Riječ je o jednoj od novijih vrsta dijagrama čije sintaksa i semantika još uvijek nisu dobro razrađene te se stoga umjesto njega i dalje znatno češće upotrebljavaju dijagrami razreda i komponenti.

- **Dijagram razmještaja** (engl. *deployment diagram*) – prikazuje fizički razmještaj programskih artefakata na fizičkoj ili virtualnoj infrastrukturi.

- **Dijagram profila** (engl. *profile diagram*) – omogućuje prilagodbu normiranih elemenata UML-a za specifične industrije ili domene. Dopušta programskim arhitektima da dodaju nove oznake, pravila i proširenja da bi precizno opisali koncepte koji nisu pokriveni osnovnom UML notacijom. Dijagrami profila omogućuju dodatne informacije i prilagodbe da bi se bolje prilagodili različitim projektima i potrebama.

Ponašajni dijagrami (engl. *behavior diagram*) prikazuju dinamičke aspekte sustava, uključujući njegovo ponašanje, interakcije i promjene tijekom vremena. Osnovni ponašajni dijagrami u UML 2.5.1 su dijagrami obrazaca uporabe, dijagrami aktivnosti, dijagrami stanja i dijagrami interakcije. Podskupinu dijagrama interakcije čine sekvencijski dijagrami, komunikacijski dijagrami, vremenski dijagrami i dijagrami pregleda interakcije. U nastavku se nalazi kratak opis svake vrste.

- **Dijagram obrazaca uporabe** (engl. *use case diagram*) – prikazuje odnose između aktora (korisnika ili vanjskih sustava) i sustava radi izvršavanja neke funkcionalnosti. Koristi se za modeliranje funkcionalnosti sustava iz perspektive korisnika te prepoznaje različite obrasce uporabe (ili scenarije) koje korisnik može izvesti i aktore koji komuniciraju sa sustavom.

- **Dijagram aktivnosti** (engl. *activity diagram*) – prikazuje tijek radnje ili procesa sustava, tako što prikazuje njegove aktivnosti, odluke i tijekove kontrole. Upotrebljava se za modeliranje dinamičkog ponašanja sustava, poput poslovnih procesa ili tijeka izvođenja obrazaca uporabe te za prikazivanje toga kako su različite aktivnosti poredane i međusobno povezane.

- **Dijagram stanja** (engl. *statemachine diagram*) – prikazuje životni ciklus objekta ili komponente tako što prikazuje njegova stanja, događaje i prijelaze. Služi za modeliranje ponašanja objekta ili komponente u odgovoru na različite događaje ili podražaje te za prikazivanje toga kako se prelazi između različitih stanja tijekom vremena.

- **Sekvencijski dijagram** (engl. *sequence diagram*) – prikazuje interakcije između objekata ili komponenata u sustavu razmjenom poruka na vremenskoj osi. Upotrebljava se za modeliranje dinamičkog ponašanja sustava tako što prikazuje kako se različite komponente međusobno povezuju i reagiraju na različite događaje ili podražaje. Podržava modeliranje uvjetnog izvođenja, ponavljanja, paralelnog izvođenja, prekida itd.

- **Dijagram komunikacije** (engl. *communication diagram*) – prethodno poznat i kao dijagram suradnje (engl. *collaboration diagram*) vrsta je dijagrama interakcija koja prikazuje interakcije između objekata ili komponenata u sustavu putem njihovih poruka i asocijacija. Primjenjuje se za modeliranje uzoraka komunikacije između različitih komponenata sustava i za prikazivanje toga kako surađuju da bi postigli određeni cilj ili izvršili zadatak. Ovaj je dijagram sličan sekvencijskom dijagramu, ali ima znatno jednostavniju sintaksu.

- **Dijagram pregleda interakcije** (engl. *interaction overview diagram*) – prikazuje interakcije između objekata ili komponenata u sustavu na visokoj razini apstrakcije, tako što pokazuje njihove aktivnosti i tijekove kontrole.

- **Vremenski dijagram** (engl. *timing diagram*) – prikazuje interakcije između objekata u sustavu tijekom vremena, tako što pokazuje njihove vremenske osi i poruke kao vremenske intervale. Upotrebljava se za modeliranje ograničenja vremena i sinkronizacije između različitih dijelova sustava te za prikazivanje toga kako koordiniraju svoje radnje tijekom vremena.

U ovom će se priručniku detaljno obraditi sedam UML dijagrama koji se u praksi najčešće koriste: dijagram obrazaca uporabe, sekvencijski dijagram, dijagram razreda, dijagram stanja, dijagram aktivnosti, dijagram komponenti i dijagram razmještaja.

1.3 Literatura

Jedna je od najznačajnijih knjiga o UML-u „The Unified Modeling Language User Guide" autora G. Boocha, J. Rumbaugha i I. Jacobsona, koji su zajednički i osmislili jezik UML. Ona pruža detaljan uvod u UML i sve njegove koncepte, pri čemu u prvoj inačici iz 1999. pokriva UML do verzije 1.3, a u kasnijoj istoimenoj inačici iz 2005. opisuje normu UML 2.0. Za razliku od referentnih priručnika o UML-u u kojima se opisuje norma, knjiga se usmjerava na praktičnog korisnika UML-a i opisuje uvodne i napredne koncepte strukturalnog i ponašajnog modeliranja programske potpore UML-om.

Možda je najpopularnija knjiga o UML-u ona od M. Fowlera, „UML Distilled", koja je izašla u tri izdanja, 1997., 1999. i 2003., pri čemu u onom posljednjem autor pokriva normu UML inačice 2.0. Ta je knjiga zadobila veliku popularnost zbog vrlo praktičnog i preciznog opisa ključnih UML dijagrama i njihovih elemenata, što je bilo vrlo značajno za šire prihvaćanje te industrijske norme. Fowler piše vrlo jasno, ali i raspravlja o tome kako najbolje iskoristiti pojedine elemente dijagrama ovisno o tome što se želi modelirati.

Osim tih naslova, neke od značajnijih knjiga koje opisuju UML i njegove elemente su:

- „Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development" autora C. Larmana iz 2004.
- „UML 2.0 in a Nutshell" autora D. Pilonea i N. Pitmana iz 2005. i
- „UML and Data Modeling: A Reconciliation" autora D. C. Haya iz 2011.

Osim navedenih naslova autori ove knjige preporučuju i nekoliko internetskih izvora koji kvalitetno i detaljno pokrivaju normu UML:

- Unified Modeling Language, dostupno na https://www.omg.org/spec/UML/2.5.1/About-UML
- The Unified Modeling Language, dostupno na https://www.uml-diagrams.org/
- What is Unified Modeling Language (UML)?, dostupno na https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-uml/

Te knjige i *web*-izvori daju niz perspektiva i pristupa učinkovitom korištenju UML-a u razvoju programske potpore i vrijedan su izvor za svakoga tko želi naučiti više o teoriji UML-a i njegovoj primjeni u praksi.

Na hrvatskom je jeziku, po saznanju autora, dosad izašla samo jedna knjiga u kojoj se detaljno razmatra norma UML u inženjerskom modeliranju, a to je sveučilišni priručnik „UML dijagrami:

zbirka primjera i riješenih zadataka", autora A. Jovića, M. Horvata i I. Grudenića iz 2014., izdavača GRAPHIS d.o.o. Za razliku od aktualne knjige, navedeni je sveučilišni priručnik više usmjeren na tri vrste UML dijagrama: dijagrame obrazaca uporabe, sekvencijske dijagrame i dijagrame razreda, koje obrađuje većinom u skladu s normom UML 2.0, ali na više mjesta koristi i stariju sintaksu iz norme UML 1.X. Ostali dijagrami obrađeni su manje detaljno, s naglaskom na primjerima i riješenim zadacima, od kojih velik dio pokriva primjere nevezane za programsku potporu. Kao i kada je riječ o sličnoj stranoj literaturi, ponešto zastarjela sintaksa na mnogim mjestima otežava čitateljima snalaženje u suvremenim alatima za modeliranje UML-om. Uz ranije spomenuti priručnik, na hrvatskom jeziku je objavljen i sveučilišni udžbenik "Softversko inženjerstvo", autora R. Mangera iz 2016., izdavača Element d.o.o., koji sadrži predavanja iz kolegija Softversko inženjerstvo na diplomskom studiju Računarstvo i matematika na Matematičkom odsjeku PMF-a Sveučilišta u Zagrebu. Udžbenik se fokusira na osnovne pojmove, modela i metode programskog inženjerstva, dok se samo u manjoj mjeri govori o UML standardu.

2. Primjena u praksi

UML dijagrami danas su široko prihvaćen i neizostavan alat u razvoju programske potpore. Oni pružaju normiran i intuitivan način oblikovanja složenih programskih sustava i potiču jasno izražavanje ideja pri oblikovanju, što je ključno u suradničkom okruženju razvoja moderne programske potpore.

Nakon svojih početaka u devedesetim godinama prošlog stoljeća, UML je prošao velike promjene da bi se bolje prilagodio stvarnim potrebama. Prve ozbiljnije primjene u praksi počele su Unificiranim procesom (UP), koji je koristio obrasce uporabe kao poveznicu svih faza razvoja programske potpore. Osim dijagrama obrazaca uporabe korišteni su i ostali dijagrami, posebice dijagrami razreda, koji su prilagođeni objektno-orijentiranim programskim jezicima i pružaju osnovu za objektno orijentiranu analizu i oblikovanje (engl. *Object-Oriented Analysis and Design, OOAD*).

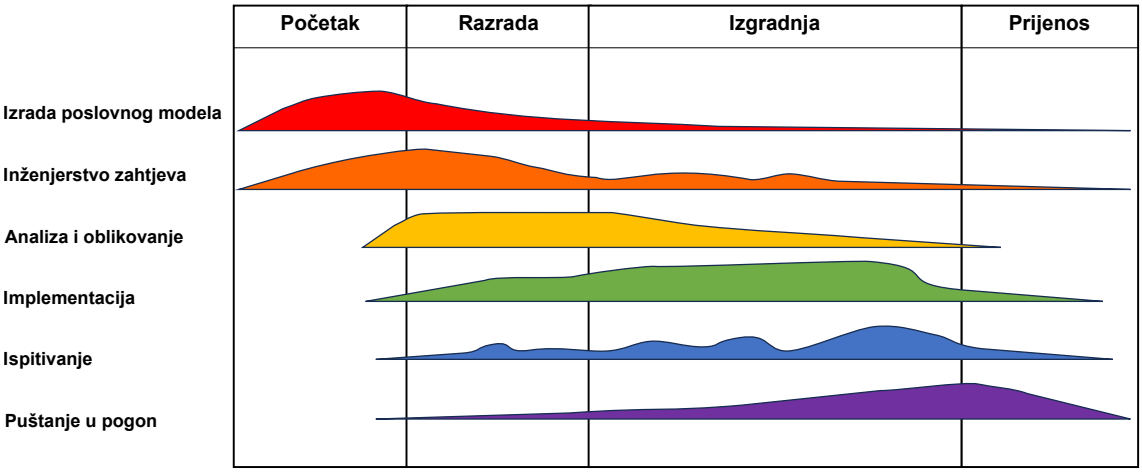
Pojavom agilnih metodologija kao što su ekstremno programiranje (XP), Lean i rani Scrum, koji su inicijalno poticali brzu implementaciju bez naglaska na početnom oblikovanju i dokumentaciji, primjena UML dijagrama u praksi je zastala. Međutim, zbog porasta složenosti programskih sustava razvojni inženjeri uvidjeli su potrebu za modeliranjem za vrijeme razvoja programske potpore te je praksa korištenja UML dijagrama ponovno oživjela. Mnoge tvrtke i razvojni timovi usvojili su UML kao glavni jezik za modeliranje, a UML dijagrami postali su uobičajen način komuniciranja s dionicima i klijentima. Također, UML je doprinio kodificiranju i normiranju različitih metodologija modeliranja koje su prije uzrokovale nesporazume. U tom se smislu posebno ističe agilno modeliranje, u kojem se UML primjenjuje fleksibilno da bi se bolje razumjeli zahtjevi sustava i omogućila prilagodba za vrijeme evolucije projekta.

Budući da UML ostaje jedini normirani jezik za modeliranje programskih sustava, neizostavan je dio akademskih programa za obrazovanje računalnih i programskih inženjera. Učenje UML-a pomaže studentima u razvoju vještine jasnog i sažetog izražavanja ideja, što je ključno za suradnju s kolegama i dionicima. Osim toga, UML omogućuje modeliranje složenih sustava na različitim razinama apstrakcije, potiče apstraktno razmišljanje i pomaže studentima u razbijanju velikih sustava na upravljive komponente i razumijevanju odnosa između njih. Uporabom UML-a studenti također dolaze u dodir s alatima i tehnikama koji se primjenjuju u industriji, što ih čini spremnijima za poslovno okruženje i konkurentnijima u području programskog inženjerstva. Ukratko, UML čini most između teorijskog znanja i stvarne primjene tako što priprema studente za uspješne karijere u razvoju programske potpore.

2.1 Unificirani proces (UP) i UML

UML dijagrami igraju ključnu ulogu u Unificiranom procesu (engl. *Unified Process, UP*), a to je iterativna i inkrementalna metodologija razvoja programske potpore koja se temelji na načelima objektnog oblikovanja. UP ističe disciplinirani pristup programskom inženjerstvu te je usmjeren na izradu programskih sustava visoke kvalitete.

Unificirani proces organiziran je u četiri faze: početak (engl. *inception*), razradu (engl. *elaboration*), izgradnju (engl. *construction*) i prijenos (engl. *transition*). Svaka faza uključuje niz aktivnosti poput prikupljanja zahtjeva, analize i oblikovanja, implementacije i ispitivanja, te rezultira nizom artefakata poput modela obrazaca uporabe, modela razreda i programskog koda. Na slici 2.1 ilustriran je tijek aktivnosti i njihov intenzitet po pojedinim fazama.



Slika 2.1: Aktivnosti i faze Unificiranog procesa

UML pruža vizualni jezik za modeliranje programskog sustava tijekom cijelog procesa razvoja. Koristi se za komunikaciju i dokumentiranje zahtjeva te modela oblikovanja i ponašanja sustava na normiran i lako razumljiv način. Različiti UML dijagrami, kao što su dijagrami obrazaca uporabe, dijagrami razreda, sekvencijski dijagrami i dijagrami stanja, koriste se u različitim fazama Unificiranog procesa da bi se modelirali različiti aspekti sustava.

2.1.1 Dijagrami obrazaca uporabe

Dijagrami obrazaca uporabe koriste se u UP-u da bi se zabilježili i vizualizirali funkcionalni zahtjevi sustava iz perspektive njegovih korisnika. Oni služe kao važno sredstvo za povezivanje svih faza procesa razvoja.

Na samom početku dijagrami obrazaca uporabe pomažu u prepoznavanju i utvrđivanju funkcionalnosti sustava na visokoj razini apstrakcije te primarnih aktora (korisnika i vanjskih sustava) koji komuniciraju sa sustavom. To pruža jasno razumijevanje opsega sustava i početnih zahtjeva.

U fazi razrade dijagrami obrazaca uporabe dalje se razrađuju da bi uključivali detaljnije opise načina uporabe, aktore i njihove međusobne odnose te pomažu u utvrđivanju granica sustava i preciziranju zahtjeva. Obrasci uporabe također postaju osnova za utvrđivanje ponašanja sustava u kasnijim fazama.

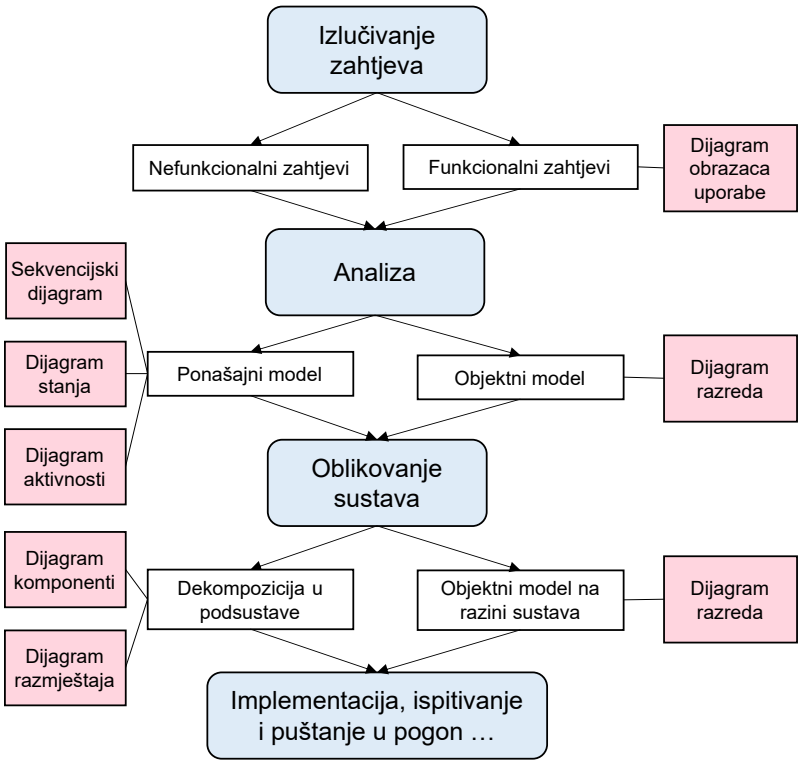
U fazi izgradnje dijagrami obrazaca uporabe vode proces razvoja tako što služe kao referenca za implementaciju i ispitivanje pojedinih načina korištenja. Razvojni inženjeri koriste ih da bi funkcionalnosti sustava odgovarale utvrđenim zahtjevima.

U fazi prijenosa dijagrami obrazaca uporabe pomažu u validaciji i verifikaciji sustava. Iz njih se mogu izvoditi ispitni slučajevi da bi sustav ispunjavao željenu funkcionalnost.

2.1.2 Objektno orijentirana analiza i oblikovanje

Objektno orijentirana analiza i oblikovanje (engl. *Object-Oriented Analysis and Design, OOAD*) važan je koncept u UP-u, kao pristup izradi programske potpore koji se usmjerava na modeliranje i oblikovanje programskih sustava korištenjem načela objektno-orijentiranog programiranja. OOAD uključuje prepoznavanje objekata, njihovih atributa, metoda i njihovih interakcija da bi se stvorio sveobuhvatan plan za izgradnju programske potpore.

UML dijagrami zajedno pružaju sveobuhvatan pristup OOAD-u i koriste se za modeliranje statičkih i dinamičkih aspekata sustava, kao što je ilustrirano na slici 2.2. Oni omogućuju razvojnim inženjerima i dionicima vizualizaciju, analizu i oblikovanje programskih sustava sustavno te osiguravaju da konačni proizvod ispunjava zahtjeve korisnika, primjenjuje robusne dizajnerske principe i da je prilagodljiv promjenama za vrijeme ciklusa razvoja.



Slika 2.2: Korištenje UML dijagrama u objektno orijentiranoj analizi i oblikovanju

Dijagrami obrazaca uporabe koriste se za bilježenje i prikazivanje funkcionalnih zahtjeva programskog sustava iz perspektive njegovih korisnika. Tijekom OOAD-a, oni pomažu u prepoznavanju funkcionalnosti sustava visoke razine i prikazivanju toga kako vanjski aktori interagiraju sa sustavom putem različitih obrazaca uporabe. Pružaju korisnički usmjeren pogled na željeno ponašanje sustava i polazišnu točku za razvoj drugih dijagrama.

Sekvencijski i komunikacijski dijagrami koriste se za prikazivanje dinamičkog ponašanja, tj. interakcija između objekata i aktora. Posebno su korisni za specificiranje detaljnog ponašanja obrazaca uporabe, uključujući redoslijed poziva metoda i razmjenu poruka između objekata te pružaju uvid u tijek ponašanja za vrijeme izvođenja određenih obrazaca uporabe.

Dijagrami razreda koriste se za modeliranje statičke strukture sustava. Predstavljaju razrede, njihove atribute, metode i odnose između razreda. U UP-u dijagrami razreda igraju ključnu ulogu u utvrđivanju podatkovnog modela sustava i služe kao temelj za objektno orijentirano oblikovanje. U fazi analize oni služe za utvrđivanje osnovnih razreda, njihovih atributa i odnosa. Kako se model razvija, dijagrami razreda postaju sve detaljniji, uključujući nasljeđivanje, asocijacije i višestrukosti.

Dijagrami stanja koriste se za modeliranje dinamičkog ponašanja objekata i komponenata sustava. Prikazuju kako objekt prelazi između različitih stanja kao odgovor na događaje. Ti su dijagrami posebno korisni za modeliranje složenih sustava koji se mogu nalaziti u različitim stanjima, poput npr. upravljačkih sustava.

Dijagrami aktivnosti vizualiziraju tijek aktivnosti ili procesa unutar sustava. U OOAD-u koriste se za modeliranje poslovnih procesa, tijeka izvođenja obrazaca uporabe ili interne logike složenih operacija. Dijagrami aktivnosti prikazuju redoslijed aktivnosti, točke odlučivanja, paralelizam i kontrolu tijeka unutar sustava te pomažu u specifikaciji ponašanja obrazaca uporabe i procesa.

Dijagrami komponenti i dijagrami razmještaja koriste se u kasnijim fazama, kao što su izgradnja i prijenos, za modeliranje fizičkog rasporeda komponenata sustava i njihovih odnosa prema sklopovskoj i programskoj infrastrukturi. Dijagrami komponenata pružaju prikaz komponenata sustava na visokoj razini, a dijagrami razmještaja ilustriraju kako su te komponente fizički raspoređene na infrastrukturi.

Ukratko, u sklopu OOAD-a koriste se različiti UML dijagrami koji omogućuju učinkovitu komunikaciju, analizu, oblikovanje i implementaciju tijekom cijelog ciklusa razvoja programske potpore. U tablici 2.1 sistematizirani su ključni elementi primjene UML dijagrama po svakoj od generičkih aktivnosti razvoja programske potpore.

Tablica 2.1: Primjena osnovnih vrsta UML dijagrama u različitim aktivnostima razvoja programske potpore

	Specifikacija programske potpore	**Analiza i oblikovanje**	**Implementacija**	**Ispitivanje**	**Evolucija**
Dijagrami obrazaca uporabe	Razrada funkcionalnih zahtjeva	Utvrđivanje komponenti/podsustava i njihove međusobne interakcije	Provjera usklađenosti trenutnog stanja i opsega sustava sa zadanim	Utvrđivanje ispitnih scenarija	Dokumentacija i komunikacija s dionicima
Sekvencijski dijagrami	Modeliranje interakcije u obrascima uporabe	Otkrivanje međuovisnosti komponenti na temelju interakcije	Provjera usklađenosti trenutnog stanja i opsega sustava sa zadanim	Razvoj ispitnih slučajeva	Dokumentacija i komunikacija s dionicima
Dijagrami razreda	Utvrđivanje i razumijevanje ključnih entiteta iz domene	Oblikovanje arhitekture sustava	Smjernice programerima za implementaciju	Utvrđivanje potencijalnih problema ili pogrešaka vezanih za međuovisnost dijelova programa	Razumijevanje strukture i učinka promjena na druge dijelove sustava
Dijagrami stanja	Razrada ponašanja dijelova sustava unutar scenarija obrazaca uporabe	Oblikovanje kontrolne logike sustava	Referenca za programere pri implementaciji kontrolne logike	Utvrđivanje rubnih slučajeva koji dovode do neočekivanog ponašanja	Procjena učinka promjena na dinamičko ponašanje sustava
Dijagrami aktivnosti	Detaljno modeliranje ponašanja pojedinačnih obrazaca uporabe	Prikaz tijeka složenih algoritama i modeliranje upravljačke logike	Referenca za implementaciju algoritama i upravljačke logike	Oblikovanje ispitnih slučajeva vezanih za grananja i točke odlučivanja	Dokumentacija i komunikacija s dionicima
Dijagrami komponenti	Razumijevanje interakcije glavnih komponenti sustava u ispunjavanju zahtjeva	Dekompozicija sustava – raspodjela odgovornosti, utvrđivanje sučelja i enkapsulacija	Referenca za implementaciju sučelja komponenti i integraciju	Utvrđivanje ključnih komponenti i njihovih ovisnosti	Dokumentacija postojeće arhitekture sustava – utvrđivanje međuovisnosti
Dijagrami razmještaja	Planiranje infrastrukture sustava	Vizualizacija i planiranja razmještajne konfiguracije sustava	Donošenje odluka vezanih za sklopovske zahtjeve, skalabilnost i performanse sustava	Utvrđivanje kritičnih dijelova sustava u kontekstu performansi i pouzdanosti	Dokumentacija za održavanje sustava i planiranje proširenja

2.2 Agilni razvoj i UML

Agilni pristup daje prednost funkcionalnosti nad opsežnom dokumentacijom, što čini izazov u korištenju UML dijagrama. Budući da tradicionalno modeliranje često uključuje formalan pristup koji nije uvijek prikladan za potrebe modernog razvoja programske potpore, na početku su agilne metode poput Ekstremnog programiranja (XP), Leana i Scruma preferirale minimalno modeliranje i neformalne dijagrame te izbjegavale UML.

Primjerice, XP se temelji na minimalnoj dokumentaciji i korisničkim pričama te na korištenju koda kao glavnog izvora informacija. Lean je usmjeren na smanjenje nepotrebnih dokumenata i koristi jednostavne neformalne dijagrame kada je to potrebno. Scrum također koristi korisničke priče i neformalne dijagrame.

Unatoč prvotnom smanjenju važnosti formalnog modeliranja i UML dijagrama u agilnim metodologijama, neki su timovi s vremenom razvili jednostavnije modeliranje koje se uklapa u agilne principe te se ne stvara nepotreban teret. Ipak, umjesto na opsežnoj dokumentaciji, naglasak ostaje na isporuci funkcionalne programske potpore i prilagodbi promjenama.

UML dijagrami mogu biti korisni alati u modernom agilnom razvoju, koji osiguravaju jasnoću, olakšavaju komunikaciju među timovima te omogućuju bolje razumijevanje i planiranje za vrijeme razvoja programske potpore. Međutim, prvenstveno se trebaju upotrebljavati kao fleksibilni alati koji se prilagođavaju promjenjivim zahtjevima i pružaju dovoljno jasnoće da bi podržali učinkovitu komunikaciju i razvoj. Agilni timovi, u nastojanju da izbjegnu pretjeranu birokraciju, često stvaraju i ažuriraju UML dijagrame prema potrebi te se usmjeravaju na vrijednost i fleksibilnost.

2.2.1 Scrum i agilno modeliranje

Mnoge tvrtke koje primjenjuju Scrum kao metodologiju razvoja programske potpore s vremenom su prihvatile lakši i fleksibilniji pristup korištenju UML dijagrama, koji je često poznat i pod nazivom „Agilno modeliranje” (engl. *Agile Modelling*) ili „Agilni UML” (engl. *Agile UML*) . Taj pristup gleda na modeliranje kao na suradnički i iterativni proces, koji teži stvaranju modela koji precizno zadovoljavaju trenutne potrebe i mogu se razvijati kako projekt napreduje.

Da bi olakšalo agilno i učinkovito modeliranje programske potpore, agilno modeliranje pruža smjernice i prakse koje ističu komunikaciju i suradnju između članova tima, dionika i korisnika. Isto tako, potiče uporabu jednostavnih i prilagodljivih tehnika modeliranja te razmjenu ideja i povratnih informacija. Osim toga, agilno modeliranje ističe važnost modeliranja s jasnim ciljem na umu, bilo da se radi o bilježenju zahtjeva, oblikovanju arhitekture sustava ili provjeri pretpostavki. Potiče stvaranje usmjerenih, sažetih modela koji ostaju relevantni za trenutačne potrebe projekta.

Za razliku od tradicionalnog modeliranja, koje često uključuje stvaranje sveobuhvatnih modela unaprijed, agilno modeliranje preferira iterativan i inkrementalan pristup. Modeli se kontinuirano usavršavaju kako projekt napreduje te se razvijaju na temelju povratnih informacija i novih saznanja. Takav pristup pomaže u izbjegavanju pretjeranog ili nedovoljnog modeliranja sustava te osigurava usklađenost s promjenama u zahtjevima projekta.

U agilnom modeliranju, UML dijagrami prije svega su alat za olakšavanje komunikacije i suradnje među članovima tima, a tek se zatim koriste kao dokumentacija. Umjesto stvaranja svih mogućih UML dijagrama, tim se usmjerava na one najvažnije za trenutačni sprint ili iteraciju. Na primjer, dijagrami obrazaca uporabe i sekvencijski dijagrami mogu se izraditi da bi se zabilježili zahtjevi i ponašanje sustava, a dijagrami razreda modeliraju ključne razrede i njihove odnose. Također, potiče se stvaranje UML dijagrama u stvarnom vremenu pomoću alata za suradničko modeliranje, čime se osigurava zajedničko razumijevanje ponašanja sustava i izbjegava prevelika dokumentacija. UML dijagrami generiraju se „u zadnji čas” (engl. *just-in-time*), prema potrebi, što sprječava pretjerano gomilanje dokumentacije i osigurava da su relevantni i korisni za tekući sprint ili iteraciju.

Sveukupno, u usporedbi s tradicionalnim modeliranjem, agilno modeliranje fleksibilniji je i prilagodljiviji pristup modeliranju u razvoju programske potpore. Ono ističe suradnju, iterativan i inkrementalan pristup te sposobnost prilagodbe promjenama. Mnoge tvrtke smatraju da je taj agilni pristup uporabi UML dijagrama učinkovit u uspostavi ravnoteže između modeliranja i agilnosti. Njime se osigurava zajedničko razumijevanje bez pretjerane dokumentacije, a detalji pristupa ovise

o potrebama projekta, ciljevima, preferencijama tima i vještinama.

2.2.2 Disciplinirana agilna isporuka (DAD)

Disciplinirana agilna isporuka (engl. *Diciplined Agile Delivery, DAD*) sveobuhvatan je i fleksibilan okvir za agilni razvoj programske potpore. DAD je evoluirao iz Unificiranog procesa (UP) i Agilnog unificiranog procesa (AUP) te je naslijedio UP-ov naglasak na iterativnom i inkrementalnom razvoju te najboljim praksama u arhitekturi prilagodivši ih promjenjivom okruženju suvremenog razvoja programske potpore.

DAD-ovi prepoznaju izazove UP-ove sveobuhvatnosti koja ga čini manje prikladnim za agilne okoline te stoga DAD nudi pragmatičniji, kontekstualno vođen i prilagodljiviji pristup isporuci programske potpore. Nadalje, DAD nadograđuje temelje AUP-a tako da u isto vrijeme teži biti agilnom inačicom UP-a i integrirati različite agilne metodologije, prakse i životne cikluse u sveobuhvatan okvir.

Jedna od ključnih razlika između DAD-a i UP/AUP-a leži u visokoj razini fleksibilnosti i prilagodljivosti koju DAD nudi. Organizacije mogu prilagoditi svoj pristup da bi odgovarao specifičnim potrebama i ograničenjima njihovih projekata. Za razliku od toga, UP i AUP više su skloni davati recepte po kojima se projekt treba izvesti te su manje prilagodljivi različitim projektima.

U kontekstu modeliranja UML-om DAD prepoznaje UML dijagrame kao ključan alat za modeliranje i komunikaciju. Pristup DAD-a prema modeliranju UML-om prilagođen je kontekstu te potiče fleksibilnost u odabiru praksi UML-a koje odgovaraju specifičnim potrebama projekta, kao što su odabir dijagrama, razina detalja i formalnost. Na primjer, složeni projekti mogu zahtijevati detaljnu dokumentaciju UML-om, a manji timovi mogu preferirati neformalne skice.

Prilagodljivost okvira i njegova usklađenost s agilnim načelima ističu važnost UML dijagrama koji doprinose ciljevima projekta i učinkovitoj komunikaciji te DAD čine prikladnim za različite projektne scenarije. UML dijagrami u DAD-u prije svega olakšavaju komunikaciju i suradnju među članova tima i dionicima tako što im pomažu u razgovorima, usavršavanju i provjeri ideja.

DAD zagovara iterativan i inkrementalan pristup modeliranju UML-om u skladu s agilnim principima prilagodljivosti i odgovora na promjenjive zahtjeve. Tim se pristupom izbjegavaju zamke iscrpnog dokumentiranja UML-om unaprijed, koje može zastarjeti. Nadalje, DAD potiče integraciju modeliranja UML-om s drugim agilnim praksama i artefaktima i osigurava njihovu praktičnu primjenjivost i usklađenost s glavnim ciljem: isporukom funkcionalne programske potpore koja zadovoljava potrebe korisnika.

Disciplinirana agilna isporuka razlikuje se od tradicionalnog UML-modeliranja i agilnog modeliranja u Scrumu zbog svojeg prilagodljivog i kontekstualno usmjerenog pristupa modeliranju UML-om. Dok se tradicionalno modeliranje UML-om obično smatra rigidnim i usmjerava se na dokumentaciju, s naglaskom na potpunosti umjesto prilagodljivosti, modeliranje UML-om u DAD-u ističe se po svojoj iznimnoj fleksibilnosti. DAD prilagođava prakse UML-a specifičnim potrebama projekta, potiče učinkovitu komunikaciju i suradnju putem UML dijagrama, te promovira iterativni i inkrementalni pristup modeliranju u skladu s agilnim načelima.

U usporedbi s agilnim modeliranjem u Scrumu, koje je uglavnom usmjereno na prakse UML-a koje izravno podržavaju Scrum aktivnosti poput planiranja sprinta i upravljanja backlogom, DAD-ov je pristup širi i prilagodljiviji te pruža podršku različitim vrstama projekata i metodologijama. Disciplinirana agilna isporuka (DAD) često se smatra skalabilnijom od agilnog modeliranja, posebno ako su u pitanju veći i složeniji projekti. Tim se okvirom pružaju smjernice za koordinaciju i usklađivanje više timova, što ga čini prikladnim za organizacije sa širokim portfeljem projekata ili

one koje razvijaju programske sustave velikog opsega.

2.3 Alati za modeliranje UML-om

Za izradu UML dijagrama postoji skup različitih alata, od jednostavnih alata za crtanje, poput LibreOffice Draw[1], Inkscape[2] i Microsoft Visio[3], sve do specijaliziranih programskih paketa posebno namijenjenih za izradu UML dijagrama, poput alata Visual Paradigm[4] i Enterprise Architect[5].

Korištenje specijaliziranih alata za izradu UML dijagrama naspram generičkih alata za crtanje ima nekoliko prednosti. Prije svega, specijalizirani alati namijenjeni su upravo za izradu UML dijagrama i obično sadržavaju sve potrebne elemente, oblike i notaciju specifične za UML. Imaju i prilagođene funkcionalnosti za specifične tipove dijagrama i modeliranje, što olakšava precizno i dosljedno modeliranje složenih koncepata. Tako specijalizirani alati omogućuju brži i učinkovitiji rad u usporedbi s generičkim alatima, koji zahtijevaju više truda za postizanje istih rezultata. Konačno, gotovo svi specijalizirani alati omogućuju određenu razinu automatizacije, kojom se pojednostavnjuje proces modeliranja, kao npr. generiranje programskog koda iz dijagrama, reverzni inženjering (izvlačenje modela iz postojećeg koda), provjera dosljednosti i drugo.

Nažalost, na tržištu ne postoji mnogo alata otvorenog koda specijaliziranih za modeliranje UML-om, a oni koji postoje uglavnom nisu potpuno u skladu s najnovijom normom UML-a ili se ne ažuriraju redovito. Zato se u praksi uglavnom koriste komercijalni alati, koji zahtijevaju plaćenu licencu. Međutim, većina komercijalnih alata nudi i besplatnu licencu (u edukacijske svrhe), s kojom je moguće modelirati sve UML dijagrame, ali bez naprednih mogućnosti automatizacije. U nastavku je navedeno nekoliko najčešće korištenih alata za modeliranje UML-om:

- **Visual Paradigm** (https://www.visual-paradigm.com/) – moćan alat za modeliranje UML-om koji podržava sve UML dijagrame i notaciju. Uključuje i niz dodatnih značajki poput generiranja koda, reverznog inženjeringa te suradnje članova tima pri izradi dijagrama. Besplatna licenca pokriva funkcionalnosti izrade UML dijagrama.

- **Enterprise Architect** (https://sparxsystems.com/products/ea/index.html) – sveobuhvatan alat za modeliranje UML-om koji podržava sve UML dijagrame i notaciju. Ovaj alat dodatno nudi šire pokrivanje cijelog procesa programskog inženjerstva te poslovnu analizu. Danas ima vrlo raširenu uporabu u industriji povezanoj s razvojem programske potpore. Dostupan je jedino s plaćenom licencom.

- **IBM Rational Software Architect Designer**(https://www.ibm.com/products/rational-software-architect-designer) – alat za podršku cijelom procesu razvoja programske potpore koji se temelji na načelima RUP-a (Rational Unified Process), IBM-ovoj inačici Unificiranog procesa. Premda mu to nije osnovna namjena, ovaj alat ima podršku za modeliranje UML-om te se koristi u razvoju programske potpore već mnogo godina (prethodnik mu je IBM Rational Rose). Podržava sve UML dijagrame i notaciju, kao i brojne dodatne značajke poput generiranja koda i timske suradnje.

- **MagicDraw** (https://www.3ds.com/products-services/catia/products/no-magic/magicdraw/) – još jedan sveobuhvatan alat koji pruža podršku za različite aspekte modeliranja i oblikovanja te je koristan za različite projekte koji zahtijevaju modeliranje

[1] https://www.libreoffice.org/discover/draw/
[2] https://inkscape.org/
[3] https://www.microsoft.com/en/microsoft-365/visio/flowchart-software
[4] https://www.visual-paradigm.com/
[5] https://sparxsystems.com/products/ea/index.html

složenih sustava, simulaciju, analizu performansi i drugo. Podržava modeliranje UML-om, SysML-om, modeliranje različitih vrsta baza podataka te modeliranje poslovnih procesa. Dostupan je isključivo uz plaćenu licencu.

- **Astah** (https://astah.net/) – alat koji je usmjeren na modeliranje UML-om te koji podržava sve UML dijagrame i notaciju. Uključuje i mnoge dodatne značajke poput generiranja koda i suradnje na izradi modela. Za studente je dostupna besplatna licenca uz registraciju s adresom e-pošte akademske ustanove.

- **ArgoUML** (https://github.com/argouml-tigris-org/argouml) – alat za modeliranje UML-om otvorenog koda koji podržava sve UML dijagrame i notaciju. Riječ je o laganom i fleksibilnom alatu koji je pogodan za manje do srednje velike projekte iako osvježavanja ne prate redovito sve promjene u normi UML.

Osim navedenih alata, koji se moraju instalirati na korisnikovo računalo kao desktop aplikacije, danas su sve raširenji alati u oblaku za online izradu UML dijagrama. Neki od trenutno najčešće korištenih su: Creately[6], LucidChart[7] i GenMyModel [8]. Dolaze uz različite načine licenciranja (uglavnom postoji i besplatna inačica), a osnovna im je prednost što su dostupni korisnicima na svim platformama (Windows, iOS, Linux, Android) i svim uređajima. Također, s obzirom na to da su izvorno implementirani u oblaku omogućuju vrlo jednostavnu suradnju na izradi dijagrama. Međutim, većina tih alata nije u potpunosti u skladu s važećom normom UML-a te ne nudi potpunu podršku za sve potrebne elemente i notaciju. U načelu, ti internetski alati nisu široko prihvaćeni u industriji koja se bavi ozbiljnim modeliranjem te se uglavnom koriste prethodno prikazani alati instalirani lokalno na računalo.

Konačno, mnoga razvojna okruženja podržavaju dodatke (engl. *extensions*) koji omogućuju korištenje UML-a za modeliranje ili vizualizaciju strukture programskog koda. No, to je najčešće ograničeno na generiranje dijagrama razreda ili komponenata na temelju programskog koda. Primjeri takvih dodataka su UMLet[9] i UML Designer[10] za Eclipse[11], Class Designer za Visual Studio[12] i ostali.

[6] https://creately.com/lp/uml-diagram-tool/
[7] https://www.lucidchart.com/pages/
[8] https://www.genmymodel.com/
[9] https://marketplace.eclipse.org/content/umlet-uml-tool-fast-uml-diagrams
[10] https://marketplace.eclipse.org/content/uml-designer
[11] https://www.eclipse.org/
[12] https://visualstudio.microsoft.com/

3. UML dijagrami obrazaca uporabe

UML dijagrami obrazaca uporabe (engl. *use case diagrams*) jedna su od najčešće korištenih vrsta dijagrama u programskom inženjerstvu. Njihova je glavna zadaća vizualno modeliranje funkcionalnosti sustava te prikaz interakcija između različitih aktora (engl. *actor*) i samog sustava. Zbog svoje jednostavnosti i korištenja visoke razine apstrakcije, dijagrami obrazaca uporabe lako su razumljivi i onim dionicima koji nisu po struci programski inženjeri (poput klijenata, članova tima i menadžera) te su vrlo korisni za komunikaciju zahtjeva utvrđenih na visokoj razini i utvrđivanje opsega sustava u ranim fazama razvoja. Oni pomažu u analizi i razumijevanju odnosa između različitih funkcionalnosti i aktora te u uočavanju potencijalnih područja poboljšanja ili nedostajuće funkcionalnosti. Osim toga, služe kao osnova za detaljnije modeliranje sustava, poput dijagrama aktivnosti, sekvencijskih dijagrama i dijagrama stanja.

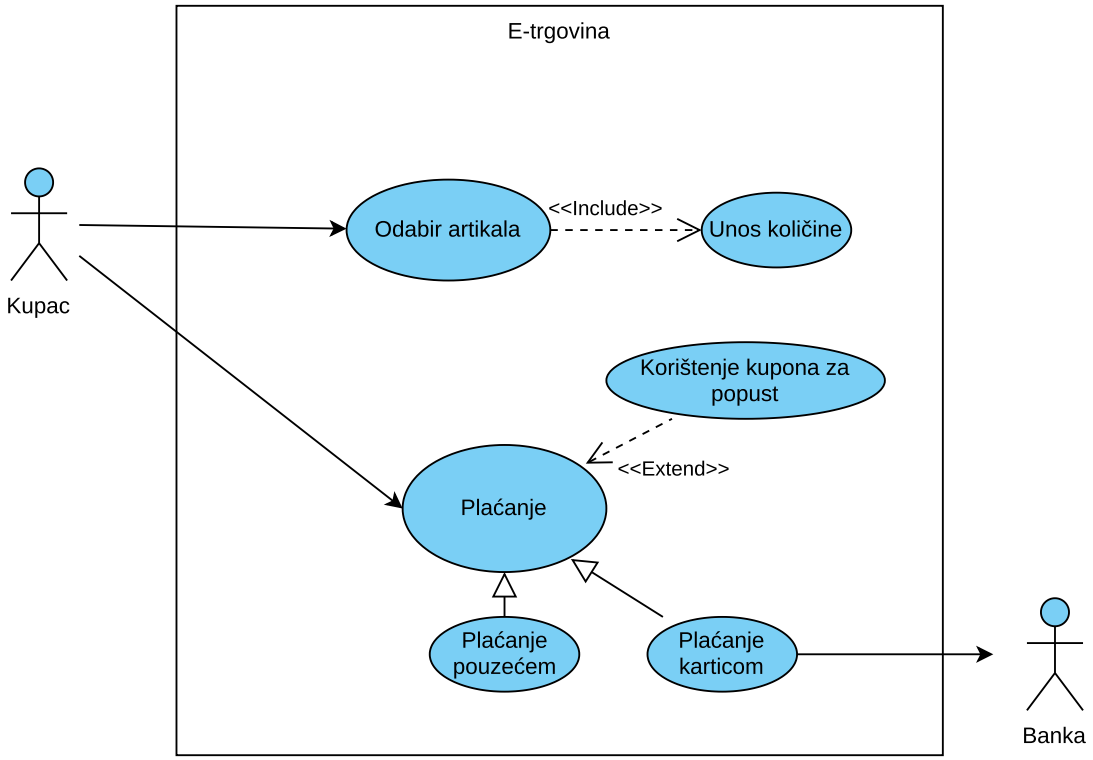
3.1 Definicija i osnovni elementi

Kao što je već spomenuto, UML dijagrami obrazaca uporabe vrsta su ponašajnih dijagrama koja se koristi za **vizualni prikaz skupa funkcionalnosti - obrazaca uporabe** (engl. *use case*) koje neki sustav treba ili može izvoditi **u suradnji s jednim vanjskim korisnikom sustava ili više njih** (aktora). Osnovni elementi dijagrama obrazaca uporabe, prisutni na svakom dijagramu, su:

- **Aktori** – vanjski entiteti koji stupaju u interakciju sa sustavom. Aktori mogu biti osobe, drugi sustavi ili organizacije. U dijagramu su najčešće predstavljeni ikonom čovječuljka (engl. *stickman*), ali se mogu koristiti i drukčije vizualne oznake koje pobliže predstavljaju aktora (npr. logo). Dobro razumijevanje uloge aktora važno je da bi se točno modelirali i analizirali zahtjevi i interakcije sustava.

- **Obrasci uporabe** – funkcionalnosti, ciljevi ili zadaci koje sustav treba izvršavati ili podržavati. Obrasci uporabe najčešće su prikazani ovalima koji sadržavaju kratak opis funkcionalnosti. Također, obrasci uporabe mogu imati i redni broj ili sličnu alfanumeričku oznaku koja je povezana s tekstnim opisom i koja omogućuje bolje snalaženje.

- **Veze** – interakcije između aktora i obrazaca uporabe, kao i odnosi između obrazaca uporabe. Za prikaz veza, koriste se različite vrste linija koje pobliže opisuju vrstu veze. Aktor može biti povezan s više obrazaca uporabe, a obrazac uporabe može uključivati više aktora.



Slika 3.1: Primjer dijagrama obrazaca uporabe

- **Granica sustava** – pravokutnik određenog naziva koji obuhvaća obrasce uporabe da bi se naznačio opseg sustava koji se modelira. Aktori se postavljaju izvan ove granice, čime se ističe da su izvan sustava.

Na slici 3.1 prikazan je primjer jednog jednostavnog dijagrama obrazaca uporabe za e-trgovinu koji sadržava sve osnovne elemente. U nastavku će se detaljnije objasniti svaki od osnovnih elemenata.

3.1.1 Aktori

U kontekstu dijagrama obrazaca uporabe i modeliranja sustava aktori se, na temelju svojih uloga i načina na koji stupaju u interakciju sa sustavom, mogu klasificirati kao aktivni ili pasivni:

- **Aktivni aktori** (engl. *primary actors*) su oni aktori koji pokreću interakcije sa sustavom. Aktivno izvode radnje ili pokreću događaje koji navode sustav na reagiranje ili izvršavanje određenog obrasca uporabe. Aktivni aktori često su vanjski korisnici ili drugi sustavi koji zahtijevaju funkcionalnost sustava da bi dovršili svoje zadatke. Primjeri aktivnih aktora uključuju kupce koji kupuju u e-trgovinama, ljudskog operatera koji upravlja strojem ili vanjski sustav koji šalje podatke sustavu koji se modelira.

- **Pasivni aktori** (engl. *secondary actors*) su oni koji ne pokreću interakcije sa sustavom, ali na njih utječu radnje ili odgovori sustava. Oni primaju informacije, usluge ili izlaze iz sustava, ali ne pokreću izravno obrasce uporabe. Pasivni aktori mogu biti drugi sustavi, uređaji ili čak ljudi koji su primatelji usluga ili izlaza sustava. Primjeri pasivnih aktora uključuju primatelja e-pošte u sustavu za slanje pošte, sustav baze podataka koji prima ažuriranja iz glavnog sustava ili uređaj za prikaz koji prikazuje izlaz iz sustava.

U dijagramima obrazaca uporabe i aktivni i pasivni aktori prikazuju se ikonom čovječuljka, a razlikuju se po načinu na koji je usmjerena veza pridruživanja u odnosu na obrazac uporabe s kojim su povezani. Više riječi o tome bit će u idućem potpoglavlju.

3.1.2 Veze

U UML dijagramima obrazaca uporabe postoje različite vrste odnosa između elemenata: između obrazaca uporabe i aktora, između aktora međusobno ili između samih obrazaca uporabe. Ti odnosi pomažu ilustrirati interakcije i međuzavisnosti unutar sustava i pridonose boljem razumijevanju zahtjeva na sustav, tj. željenog ponašanja sustava. Pritom se razlikuju četiri vrste veza:

- **Pridruživanje** (engl. *association*) – najčešća vrsta veze na dijagramima obrazaca uporabe koja označava da aktor sudjeluje u interakciji s obrascem uporabe.
- **Uključivanje** (engl. *include*) – označava da jedan obrazac uporabe (glavni obrazac uporabe) uključuje funkcionalnost drugog obrasca uporabe (uključeni obrazac uporabe). Koristi se kako bi se pokazalo da je ponašanje uključenog obrasca uporabe umetnuto u ponašanje glavnog obrasca uporabe. Ovaj odnos pomaže u modeliranju zajedničkog ponašanja koje dijele više obrazaca uporabe te potiče ponovno korištenje obrazaca uporabe.
- **Proširenje** (engl. *extend*) – ukazuje na to da jedan obrazac uporabe (obrazac proširenja) proširuje ili dopunjava ponašanje drugog obrasca uporabe (glavni obrazac uporabe) pod određenim uvjetima. Obrazac proširenja izvodi se samo ako su za vrijeme izvođenja glavnog obrasca uporabe zadovoljeni navedeni uvjeti. Ovaj se odnos koristi za modeliranje opcionalnog ili uvjetnog ponašanja u sustavu.
- **Generalizacija** (engl. *generalization*) – modelira odnos između dvaju aktora ili između dvaju obrazaca uporabe. Općenito, ova vrsta veze označava odnos u kojem specifičniji obrazac uporabe ili aktor nasljeđuje svojstva i ponašanje općenitijeg obrasca uporabe ili aktora. Ovaj odnos pomaže u modeliranju apstrakcije, specijalizacije i nasljeđivanja u dijagramima obrazaca uporabe te smanjuje redundantnost i potiče ponovnu uporabu. Generalizirani aktor ili generalizirani obrazac nisu nužno apstraktni u smislu da nemaju pridruženu vlastitu funkcionalnost.

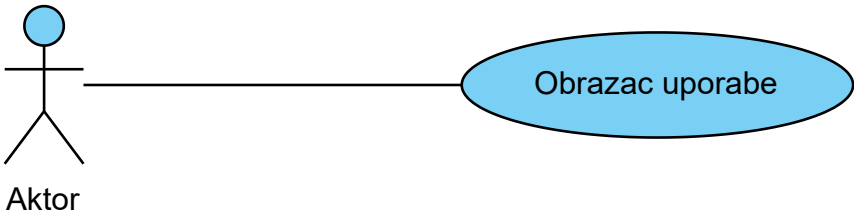
U nastavku će biti više riječi o načinu modeliranja svake od navedenih veza s primjerima korištenja.

3.1.2.1 Pridruživanje

Veza pridruživanja na UML dijagramima obrazaca uporabe **prikazuje interakciju između aktora i obrasca uporabe**. To znači da aktor sudjeluje u određenom obrascu uporabe, a veza pridruživanja pokazuje komunikaciju ili suradnju između njih. Veza je predstavljena ravnom linijom koja povezuje aktora s obrascem uporabe, kao što je prikazano na slici 3.2.

Osim označavanja toga koji je aktor povezan s kojim obrascem uporabe, veza pridruživanja može dati i dodatne informacije o ulozi aktora prema obrascu uporabe te o tome koliko instanci (primjeraka) aktora može stupiti u interakciju s određenim obrascem uporabe ili obrnuto.

Da bi se ta interakcija modelirala što preciznije, tj. utvrdilo ima li aktor aktivnu ili pasivnu ulogu prema obrascu uporabe s kojim je povezan, pridruživanju se zadaje **smjer** (prikazuje se strelicom). Kada je riječ o aktivnom aktoru, smjer pridruživanja ide od aktora prema obrascu uporabe, a kada je riječ o pasivnom aktoru, obrnuto. U složenijim obrascima uporabe aktor može



Slika 3.2: Pridruživanje

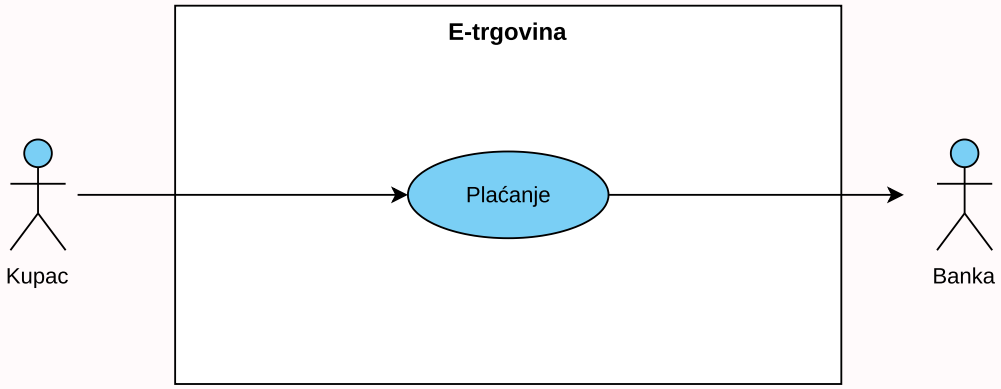
u okviru istog obrasca uporabe imati i aktivnu i pasivnu ulogu, a tada se na vezi pridruživanja ne prikazuje smjer (ravna crta bez strelica). Razlika između aktivnog i pasivnog sudjelovanja aktora u obrascu uporabe ilustrirana je u primjeru 3.1.

Primjer 3.1 — Aktivni i pasivni aktori. Modelirajte sljedeće dvije inačice procesa plaćanja u e-trgovini:

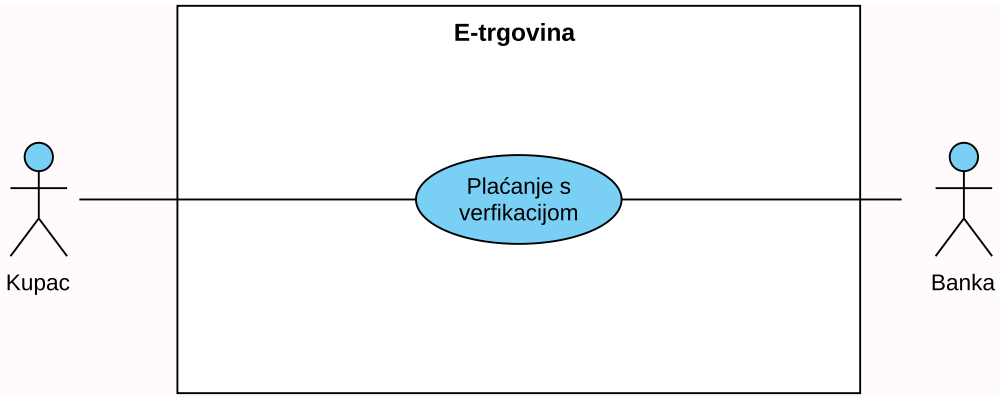
A) Jednostavna inačica – Kupac započinje proces plaćanja odabirom opcije „Plati". Zatim se aplikacija e-trgovine za provedbu plaćanja spaja s bankom te se provodi transakcija. Pretpostavite da banka odgovara isključivo na upite *web*-aplikacije, tj. sama ne inicira izvršavanje ni jednog dijela funkcionalnosti.

B) Složena inačica – Istovjetna prethodnoj inačici, ali uz sljedeći dodatak: provođenje transakcije zahtijeva dodatnu verifikaciju od strane kupca. Verifikacija se provodi tako da banka pošalje poruku PUSH kupcu, koji mora na nju odgovoriti da bi se provela naplata.

Rješenje je prikazano na slikama 3.3 i 3.4.



Slika 3.3: Model inačice A



Slika 3.4: Model inačice B

Komentar:

U objema inačicama imamo dva aktora: „Kupac" i „Banka", koji sudjeluju u obrascu uporabe „Plaćanje". U inačici A, „Kupac" je aktivni aktor jer inicira postupak, a „Banka" je pasivni aktor jer isključivo odgovara na upite. U inačici B situacija je nešto složenija. „Kupac" je i dalje taj koji inicira plaćanje (aktivna uloga), ali u drugom dijelu obrasca u kojem se provodi verifikacija, on čeka na PUSH poruku od „Banke" i na istu odgovara (pasivna uloga). Slično je i s „Bankom", u prvom dijelu ona ima pasivnu ulogu i čeka upit, ali u postupku verifikacije postaje aktivni aktor koji inicira poruku PUSH.

Važno je napomenuti da se u praksi, osobito u ranim fazama modeliranja sustava, često u potpunosti zanemaruje utvrđivanje aktivnih i pasivnih uloga te su u tom slučaju sve veze pridruživanja na dijagramu neusmjerene.

U dijagramima obrazaca uporabe, **višestrukost** (engl. *multiplicity*) koristi se za prikaz broja instanci jednog elementa koji se može povezati s jednom instancom drugog elementa. U kontekstu odnosa između aktora i obrazaca uporabe, višestrukost može pružiti informacije o tome koliko instanci aktora može stupiti u interakciju s određenim obrascem uporabe ili obrnuto.

Višestrukost se navodi kao brojčana oznaka na vezi pridruživanja. Moguće je specificirati točan broj instanci (npr. 10), ali i raspon od – do (npr. 0 – 10). Moguće je navesti i neograničen broj instanci korištenjem oznake *. Ako se višestrukost ne navede, podrazumijeva se da iznosi jedan na objema stranama veze. Primjer 3.2 prikazuje nekoliko slučajeva korištenja višestrukosti.

Primjer 3.2 — Višestrukost. Za neki kviz modelirajte tri moguće situacije:

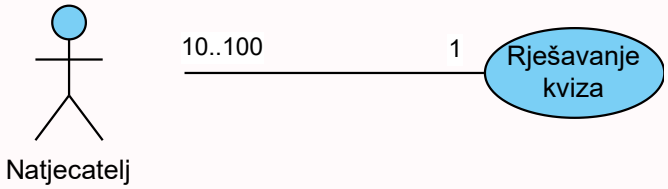
A) U kvizu sudjeluje točno 32 natjecatelja (kviz neće početi dok se ne skupe svi natjecatelji). Svaki natjecatelj ima pravo pristupiti kvizu samo jednom.

B) U kvizu smije sudjelovati od najmanje 10 do najviše 100 natjecatelja. Svaki natjecatelj ima pravo pristupiti kvizu samo jednom.

C) U kvizu smije sudjelovati neograničen broj natjecatelja. Svaki natjecatelj smije pristupiti kvizu do najviše tri puta (uzima se najbolji rezultat).

Rješenje je prikazano na slikama 3.5, 3.6 i 3.7.

Slika 3.5: Model inačice A



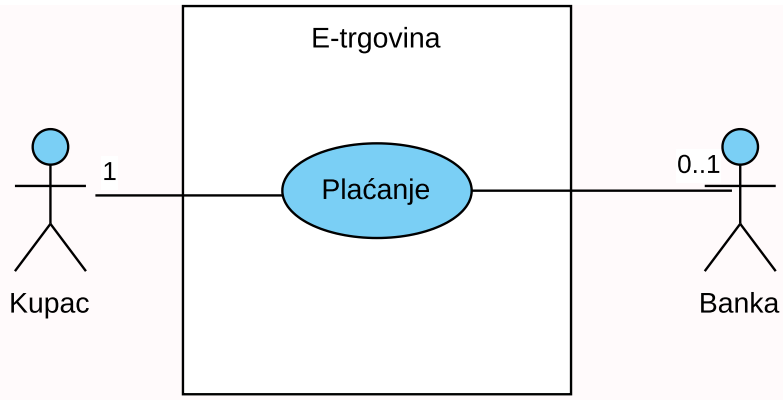
Slika 3.6: Model inačice B

Slika 3.7: Model inačice C

Nadalje, višestrukost se može koristiti i za označavanje **opcionalnog sudjelovanja** aktora u nekom obrascu uporabe. Višestrukost od nula do jedan ili od nula do neodređeno ukazuje na to da je odnos između aktora i obrasca uporabe opcionalan, tj. moguće je da nijedna instanca aktora nije povezana s obrascem uporabe, kao što je prikazano u primjeru 3.3. Ako je višestrukost utvrđena kao jedan ili od jedan do neodređeno mnogo, to implicira da se barem jedna instanca aktora mora povezati s obrascem uporabe, što odnos čini **obaveznim**.

Primjer 3.3 — Višestrukost – opcionalno sudjelovanje aktora. Prikažite na dijagramu obrazaca uporabe plaćanje u e-trgovini, uz napomenu da plaćanje može ići preko banke, ali je moguće i plaćanje gotovinom izravno dostavljaču pri preuzimanju narudžbe.

Rješenje je prikazano na slici 3.8.



Slika 3.8: Prikaz opcionalnog sudjelovanja aktora u obrascu uporabe

Komentar:

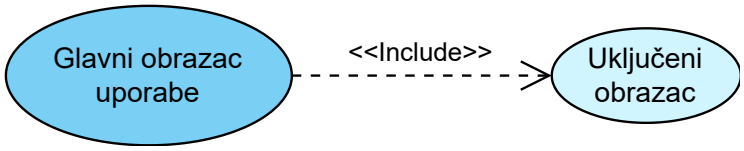
U ovom je slučaju sudjelovanje banke u obrascu „Plaćanje" opcionalno što se prikazuje korištenjem oznake višestrukosti 0..1 na strani aktora „Banka". Nadalje, dostavljača ne prikazujemo kao aktora na dijagramu jer ovdje modeliramo isključivo aplikaciju e-trgovine, a dostava i plaćanje u gotovini odvijaju se izvan aplikacije. ■

Za kraj je važno napomenuti da uporaba višestrukosti u UML dijagramima obrazaca uporabe nije obavezna te u praksi nije tako česta kao što je to kod ostalih UML dijagrama, poput npr. dijagrama razreda. Osnovni je razlog to što se dijagrami obrazaca uporabe najčešće koriste za modeliranje u vrlo ranom stadiju projekta, kada su još mnogi konkretni detalji implementacije, kao što su ograničenja u broju sudionika i sl., nepoznati ili nedorečeni. Nadalje, sama specifikacija UML-a nije jednoznačna u utvrđivanju značenja višestrukosti u kontekstu obrazaca uporabe, pa tako primjerice za slučaj broja instanci aktora nije precizirano odnosi li se broj koji označava višestrukost na pojedinu instancu interakcije ili na sve interakcije koje se događaju istodobno ili se barem dijelom vremenski preklapaju. Na primjer, ako postoje oznake višestrukosti vezane za ograničenja na plaćanje u sustavu e-trgovine, specifikacija UML-a ne utvrđuje precizno odnose li se ona na jedan slučaj plaćanja, ili na ukupan broj plaćanja koji se može obaviti u nekom razdoblju, nego je zadaća dionika da odrede razumnu interpretaciju. Međutim, korištenje oznaka višestrukosti može pružiti vrijedne informacije o kvantitativnim aspektima odnosa između aktora i obrazaca uporabe te se preporučuje njezino korištenje kada god je to moguće jer pomaže u boljem razumijevanju zahtjeva i ograničenja sustava.

3.1.2.2 Uključivanje

U dijagramima obrazaca uporabe odnos uključivanja (engl. *include*) koristi se za modeliranje situacije u kojoj ponašanje jednog obrasca uporabe (osnovni obrazac uporabe) uključuje (sadržava) ponašanje drugog obrasca uporabe (uključeni obrazac uporabe). Prikazuje se isprekidanom linijom s oznakom «include» i strelicom koja pokazuje od glavnog prema uključenom obrascu uporabe, kao što je prikazano na slici 3.9.[1]

[1] Napomena: neki alati, kao npr. Visual Paradigm koji je korišten u izradi dijagrama u ovom priručniku, automatski ispisuju «Include» (s velikim početnim slovom).



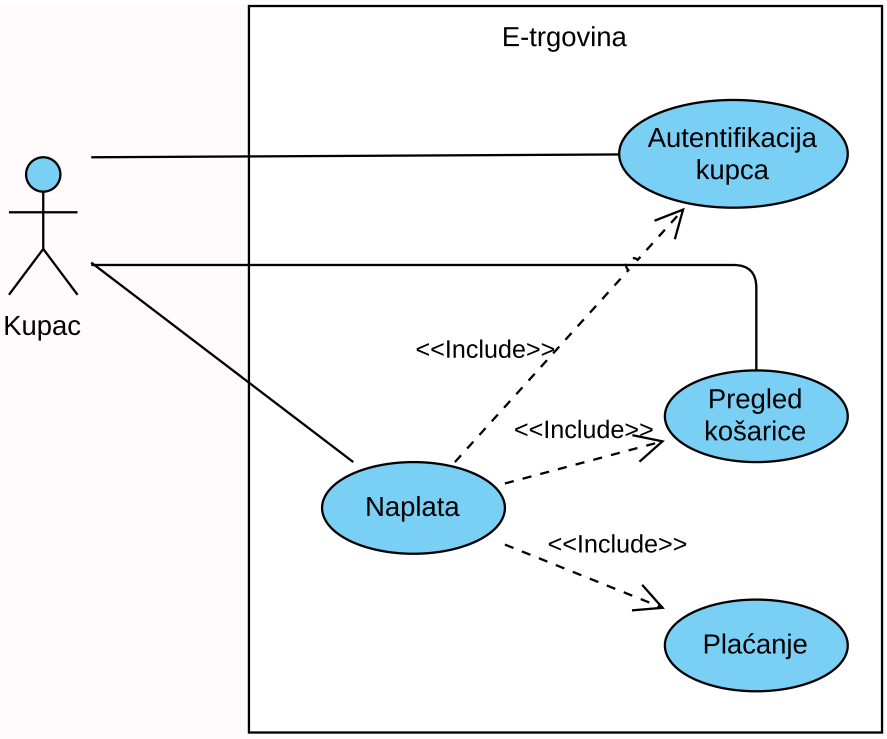
Slika 3.9: Uključivanje

Važno je istaknuti da odnos uključivanja podrazumijeva da će se **uključeni obrazac uporabe uvijek izvršiti kada se izvršava osnovni obrazac uporabe**, tj. ponašanje uključenog obrasca uporabe smatra se integralnim dijelom osnovnog obrasca uporabe. Na dijagramu se ne specificira vremenski trenutak u kojem će se uključeni obrazac uporabe izvršiti, kao ni redoslijed izvršavanja ako ima više uključenih obrazaca uporabe. Ovaj odnos promovira **ponovnu uporabu i modularnost** u sustavu tako što omogućuje dijeljenje zajedničke funkcionalnosti među više obrazaca uporabe i tipično se koristi u sljedećim situacijama:

- **Funkcionalna dekompozicija (podzadaci ili podprocesi)** – ako se neki složeniji obrazac uporabe sastoji od nekoliko manjih zadataka ili procesa koji se mogu smatrati obrascima uporabe sami po sebi, moguće je modelirati te podzadatke kao zasebne obrasce uporabe te ih povezati vezom uključivanja s osnovnim obrascem. Ovaj pristup pomaže razbijanju složenih obrazaca uporabe na jednostavnije i lakše upravljive dijelove.
- **Zajednička funkcionalnost** – kada više obrazaca uporabe dijeli funkcionalnost ili skup radnji, zajedničko ponašanje izdvaja se u zaseban obrazac uporabe koji se onda povezuje vezom uključivanja sa svakim relevantnim osnovnim obrascem uporabe. Tako se izbjegava ponavljanje istog opisa ponašanja u više obrazaca uporabe, čime dijagram postaje sažetiji i lakši za održavanje. Tipični bi primjeri bili zapisivanje, sigurnost ili provjera valjanosti podataka.
- **Modularnost i održivost (engl. *maintainability*)** – komplementarno s prvim dvjema točkama, radi poboljšanja modularnosti i lakoće održavanja dijagrama obrazaca uporabe (i programske potpore koja se na temelju njega razvija), razlaganje složene funkcionalnosti na manje dijelove olakšava uvođenje promjena i proširenja u budućnosti bez učinka na osnovnu funkcionalnost.

Primjer 3.4 — Uključivanje – funkcionalna dekompozicija. Modelirajte primjer e-trgovine u kojoj je, kada kupac pokrene naplatu, potrebno provjeriti njegov identitet, prikazati mu sadržaj košarice za konačnu provjeru te provesti plaćanje. Dodatno, moguće je da se kupac prijavi u sustav (uz autentifikaciju) i prije samog pokretanja naplate, a dopušteno je i pregledavati sadržaj košarice bez pokretanja naplate.

Rješenje je prikazano na slici 3.10.



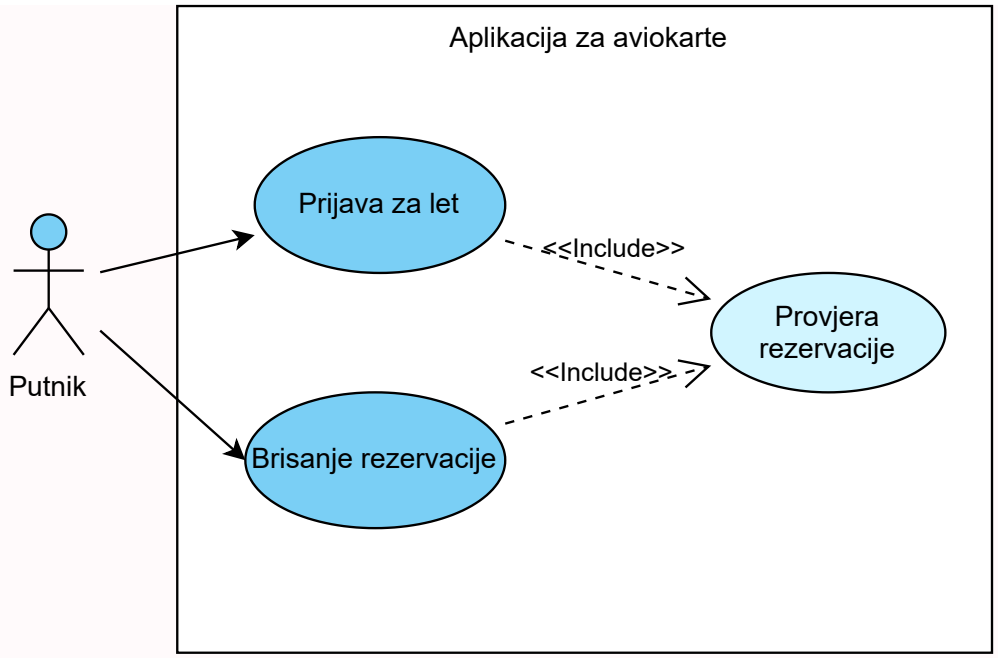
Slika 3.10: Primjer funkcionalne dekompozicije

Komentar:

U ovom je slučaju, obrazac „Naplata" složeni obrazac koji uključuje nekoliko manjih dijelova („Autentifikacija", „Pregled košarice" i „Plaćanje"). Također, „Autentifikacija" i „Pregled košarice" mogu biti i samostalne funkcionalnosti te je stoga nužno modelirati kao zasebne obrasce koji su onda uključeni u obrazac „Naplata". Samo „Plaćanje" može se, ali i ne mora izdvajati kao zasebni obrazac, ali se njegovim izdvajanjem poboljšava modularnost i olakšava daljnja nadogradnja (npr. uvođenje mogućnosti više vrsta plaćanja). ■

Primjer 3.5 — Uključivanje – zajednička funkcionalnost. Modelirajte primjer *web*-aplikacije za aviokarte u kojoj se kupac (s kupljenom kartom) može prijaviti za let ili obrisati rezervaciju. U oba slučaja potrebno je provjeriti rezervaciju.

Rješenje je prikazano na slici 3.11.



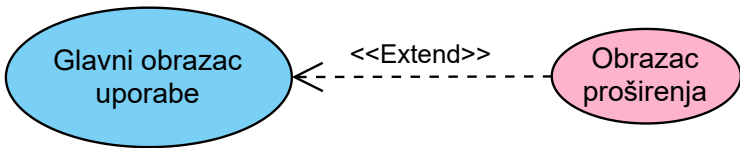
Slika 3.11: Primjer zajedničke funkcionalnosti

Komentar:

Budući da obje funkcionalnosti („Prijava za let" i „Brisanje rezervacije") imaju jedan zajednički dio („Provjera rezervacije"), dobra je praksa izdvojiti tu funkcionalnost u zasebni obrazac, čime se poboljšava ponovna iskoristivost. ■

3.1.2.3 Proširenje

U dijagramima obrazaca uporabe veza proširenja (engl. *extend*) koristi se za modeliranje situacije u kojoj jedan obrazac uporabe (proširujući obrazac uporabe) proširuje ili dopunjava ponašanje drugog obrasca uporabe (osnovni obrazac uporabe) pod određenim uvjetima. Prikazana je kao isprekidana linija s oznakom «extend» i strelicom koja pokazuje od proširenog prema glavnom obrascu uporabe, kao na slici 3.12.[2]



Slika 3.12: Proširenje

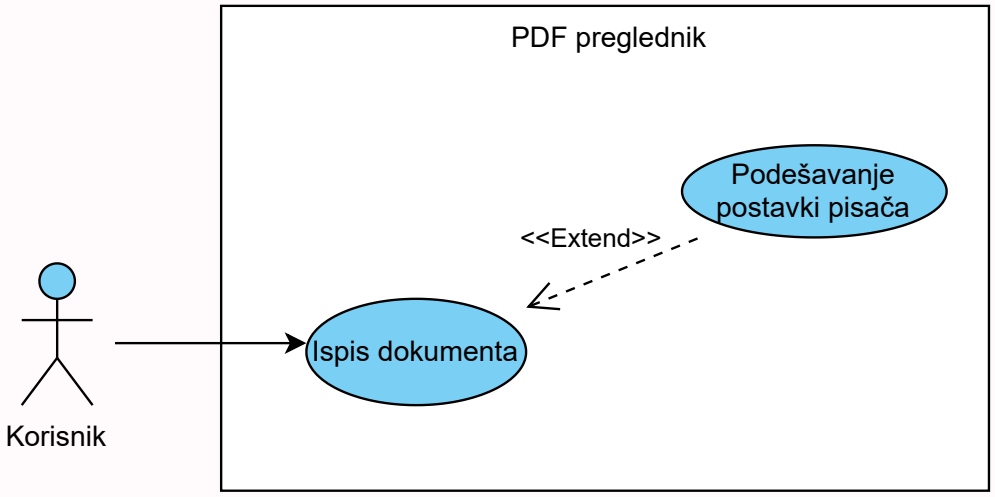
Proširujući obrazac uporabe predstavlja opcionalno ili uvjetno **ponašanje koje se ne izvršava uvijek kada se pokrene osnovni obrazac uporabe** što ga razlikuje od odnosa uključivanja, koji uvijek uključuje izvršavanje uključenog obrasca uporabe. Tipično se ovaj odnos primjenjuje u sljedećim situacijama:

[2]Napomena: neki alati, kao npr. Visual Paradigm koji je korišten u izradi dijagrama u ovom priručniku, automatski ispisuju «Extend» (s velikim početnim slovom).

- **Opcionalno (uvjetno) ponašanje** – ako obrazac uporabe ima opcionalno ponašanje koje se izvršava samo u određenim scenarijima ili pod specifičnim uvjetima i ograničenjima, kao što su korisničke uloge, konfiguracije sustava ili ulazni podaci, potrebno je razdvojiti osnovno ponašanje od opcionalnih dijelova. Opcionalno ponašanje treba prikazati kao zaseban, proširujući, obrazac uporabe i povezati ga vezom proširenja s osnovnim obrascem uporabe. Time se ističe varijabilnost u ponašanju sustava na temelju različitih okolnosti te dijagram postaje jasniji i modularniji.
- **Varijacije značajki** – u nekim slučajevima postoje različite varijacije značajki ili funkcionalnosti koje se mogu dodati ili ukloniti iz sustava bez učinka na njegovo osnovno ponašanje. U tom slučaju odnos proširenja olakšava razumijevanje i upravljanje različitim opcijama dostupnima u sustavu.

Primjer 3.6 — Proširenje. Modelirajte funkcionalnost ispisa dokumenta u PDF pregledniku pri čemu korisnik, ako želi, može dodatno podesiti neke postavke pisača.

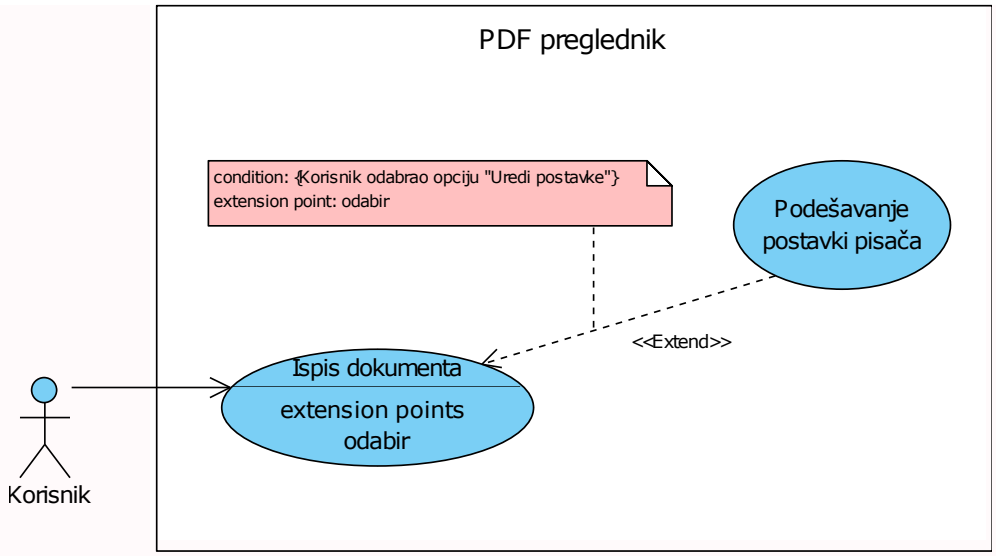
Rješenje je prikazano na slici 3.13.



Slika 3.13: Primjer proširenja funkcionalnosti ispisa dokumenta

Komentar:

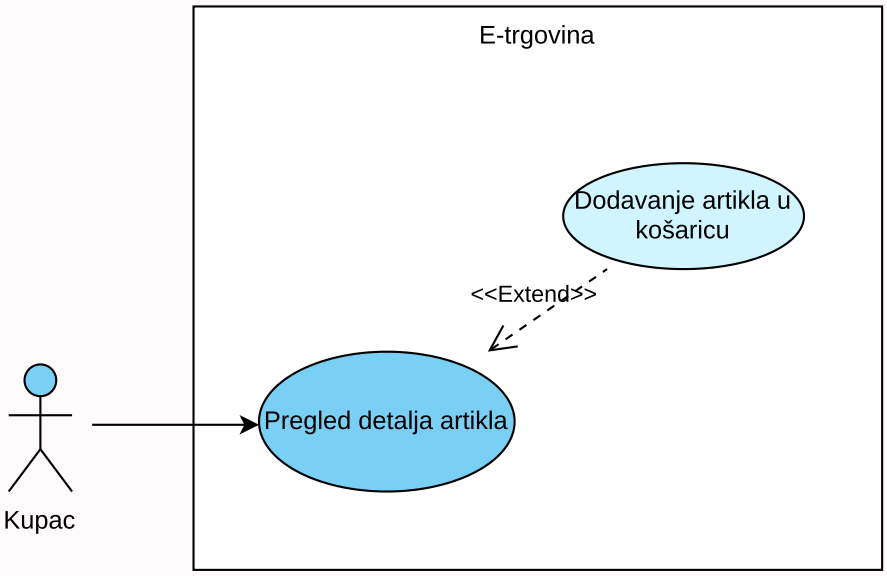
Pri modeliranju proširenja, mogu se (ali i ne moraju) eksplicitno navesti uvjet proširenja i točke proširenja. Na slici u nastavku prikazan je model istovjetan prethodnom u svemu osim što je dodan prikaz točke proširenja s uvjetom.



Slika 3.14: Primjer prikaza točke proširenja obrasca uporabe

Primjer 3.7 — Proširenje i uključivanje. Modelirajte sljedeće funkcionalnosti u aplikaciji e-trgovine. Kupac može pregledati detalje svakog artikla u ponudi te mu se pritom nudi i opcija dodavanja tog artikla u košaricu.

Rješenje je prikazano na slici 3.15.

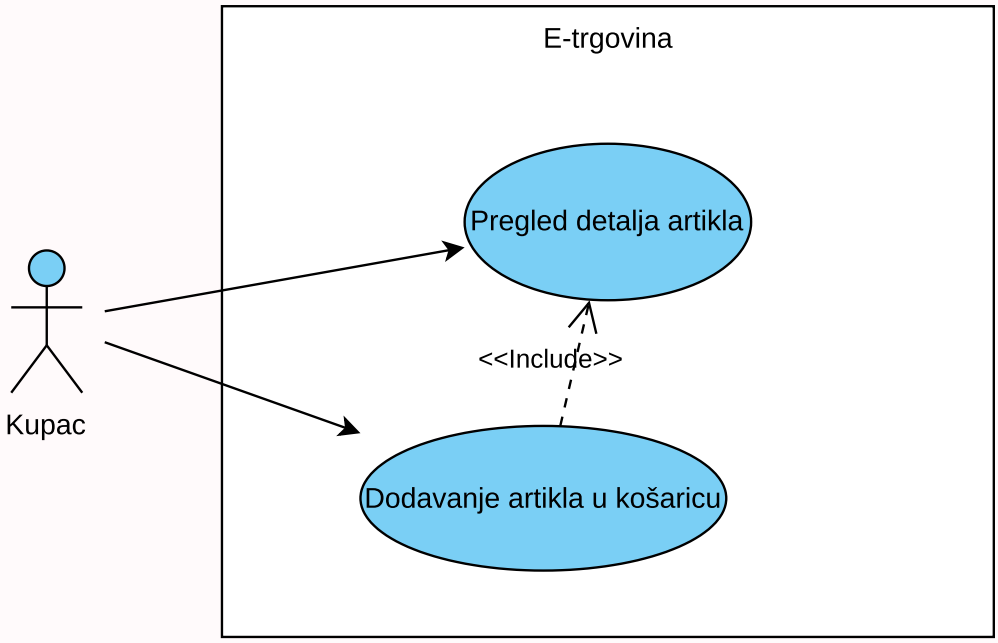


Slika 3.15: Primjer proširenja funkcionalnosti pregleda detalja artikla u e-trgovini

Komentar:

S obzirom na dani opis, model na prethodnoj slici najintuitivnije je rješenje. Međutim,

moguće je gledati na problem i nešto drukčije. Može se reći da su funkcionalnosti koje e-trgovina treba imati – mogućnost pregleda detalja svakog artikla i mogućnost dodavanja artikla u košaricu. Uz dodatnu napomenu (vezanu za sam tijek korištenja) da treba prvo odabrati artikl da bi se isti mogao dodati u košaricu. Ako se na problem gleda na ovaj način, onda su oba obrasca takoreći ravnopravna, a nužnost pregleda detalja artikla prije dodavanja u košarice modelira se vezom «include».



Slika 3.16: Primjer zamjene veze proširenja vezom uključenja

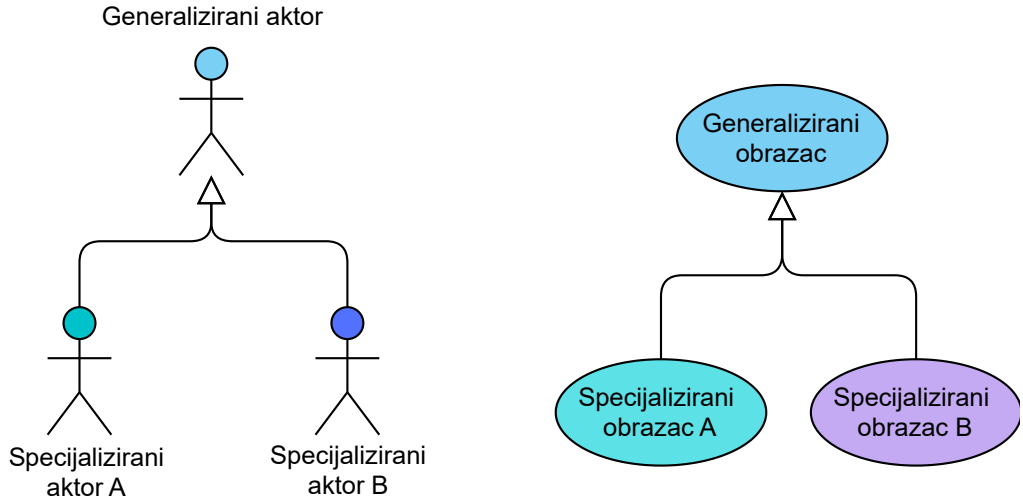
3.1.2.4 Generalizacija

U dijagramima obrazaca uporabe odnos generalizacije koristi se za modeliranje situacije u kojoj specifičniji (specijalizirani) obrazac uporabe ili aktor nasljeđuje svojstva i ponašanje općenitijeg (generaliziranog) obrasca uporabe ili aktora. Prikazuje se ravnom linijom sa šupljom strelicom koja pokazuje prema roditeljskom obrascu uporabe ili aktoru, kao na slici 3.17.

Ovaj odnos koristi koncepte apstrakcije i specijalizacije što pomaže u smanjenju redundancije i poticanju ponovne uporabe. Ovaj se odnos tipično primjenjuje u sljedećim situacijama:

- **Ponovna uporaba zajedničkog ponašanja** – ako više obrazaca uporabe dijeli niz zajedničkih svojstava ili ponašanja, ali ima jedinstvene karakteristike, zajedničko ponašanje predstavljeno je u općenitijem (generaliziranom, roditeljskom) obrascu uporabe, a specifično ponašanje svakog obrasca u specijaliziranim obrascima uporabe. Ovaj pristup pomaže u izbjegavanju ponavljanja zajedničkog ponašanja u više obrazaca uporabe te potiče ponovnu uporabu.

- **Apstrakcija i specijalizacija** – ako postoji skup obrazaca uporabe s različitim razinama apstrakcije, generalizacija se može koristiti da bi se obrasci organizirali u hijerarhiju, i to tako da apstraktniji obrasci uporabe budu na vrhu, a specijalizirani ispod. Ovakva organizacija čini dijagram razumljivijim i lakšim za održavanje jer ističe odnose između različitih razina apstrakcije.



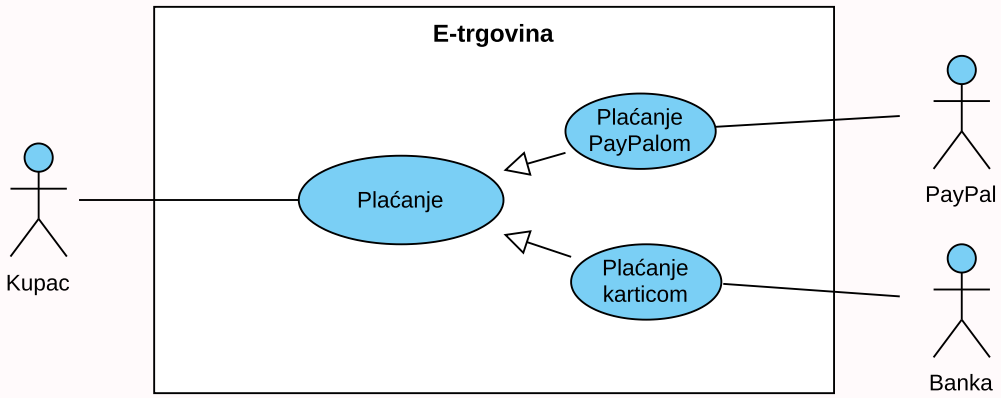
Slika 3.17: Generalizacija

- **Uloge aktora i nasljeđivanje** – ako postoje aktori koji dijele zajedničke atribute ili uloge, ali imaju jedinstvene karakteristike, generalizacija se koristi za modeliranje zajedničkih atributa ili uloga u općenitijem aktoru te jedinstvenih atributa ili uloga u specifičnijim aktorima. Ovakav pristup prvenstveno pomaže u izbjegavanju redundancije u prikazu uloga aktora. Također, organizacija aktora u hijerarhiju usko je povezana s nasljeđivanjem u dijagramu razreda, o čemu će biti više riječi u poglavlju 5.1.2.3 o generalizaciji na dijagramima razreda.

Općenito, korištenje generalizacije u dijagramima obrazaca uporabe pridonosi tome da dijagrami postanu modularniji i skalabilniji. Kada treba dodati nove obrasce uporabe ili aktore sličnog ponašanja, jednostavno se može stvoriti novi specijalizirani obrazac uporabe ili aktor koji nasljeđuje svojstva i ponašanje postojećeg roditeljskog obrasca uporabe ili aktora, što olakšava upravljanje promjenama i dodacima u sustavu. Sljedeća dva primjera prikazuju načine korištenja generalizacije nad obrascima uporabe i nad aktorima.

Primjer 3.8 — Generalizacija obrazaca uporabe. U e-trgovini kupac može platiti bankovnom karticom ili putem usluge PayPal.

Rješenje je prikazano na slici 3.18.



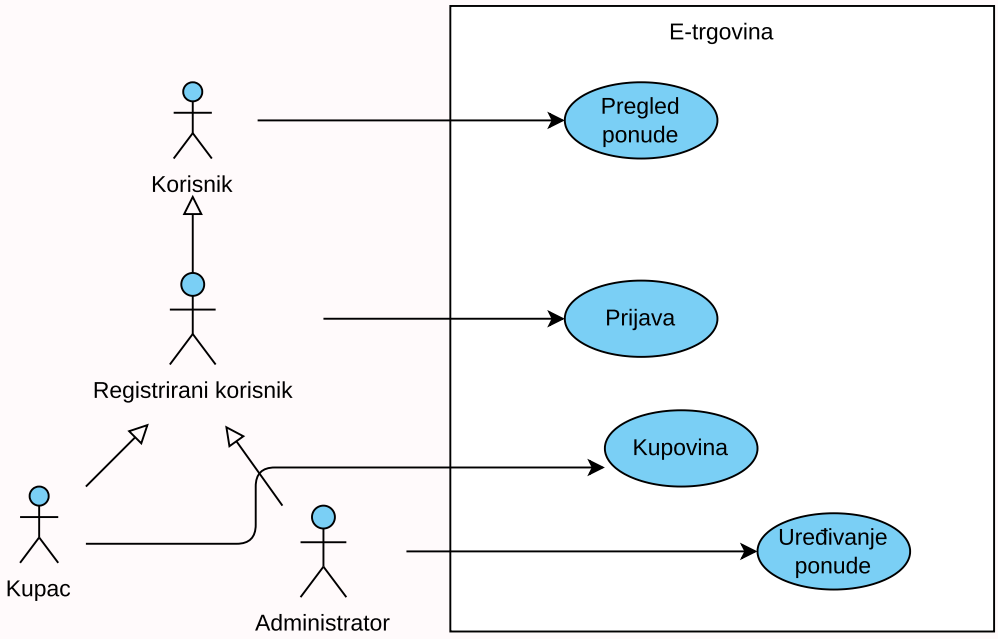
Slika 3.18: Primjer generalizacije različitih načina plaćanja

Komentar:

Iako nije netočno prikazati dvije mogućnosti plaćanja kao zasebne (i nepovezane) obrasce uporabe, bolji je pristup izgraditi hijerarhiju obrazaca uporabe jer se time ističe da su u suštini te dvije funkcionalnosti srodne (i da će pri implementaciji vrlo vjerojatno biti mnogo koda koji će one dijeliti). ■

Primjer 3.9 — Generalizacija aktora. U e-trgovini svaki korisnik može pregledati ponudu, ali se samo onaj registrirani može prijaviti sa svojim korisničkim podacima. Da bi obavio kupovinu, kupac mora biti prijavljen u aplikaciju. Nadalje, postoji i administrator aplikacije (koji također ima korisnički račun), ali on ne može kupovati, već je zadužen za uređivanje ponude.

Rješenje je prikazano na slici 3.19.



Slika 3.19: Primjer generalizacije aktora u e-trgovini

■

3.1.3 Granica sustava

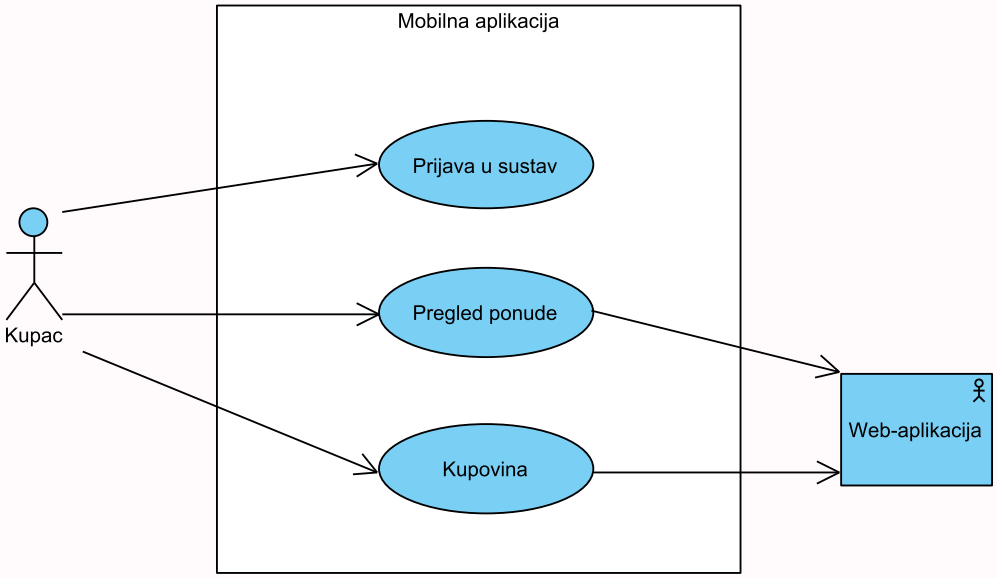
Na dijagramima obrazaca uporabe, granica sustava igra ključnu ulogu u utvrđivanju opsega i konteksta sustava koji se modelira. Granica sustava predstavljena je pravokutnikom koji obuhvaća skup obrazaca uporabe i vizualno razdvaja funkcionalnosti sustava od vanjskog okruženja. Obrasci koji se nalaze unutar granice predstavljaju specifične funkcionalnosti i značajke koje sustav pruža svojim korisnicima. Izvan granica sustava nalaze se aktori koji predstavljaju sve entitete koji komuniciraju sa sustavom i pokreću obrasce uporabe, poput korisnika ili drugih sustava.

Kada se modelira neki složeni sustav, koji se sastoji od više modula ili komponenti, svaki je takav modul moguće izdvojiti i modelirati zasebnim dijagramom. U tom slučaju na dijagramu kojim se detaljno modeliraju funkcionalnosti pojedinog modula modul se prikazuje kao zaseban sustav te granica sustava zapravo označava granice modula, dok se svi ostali moduli s kojima je

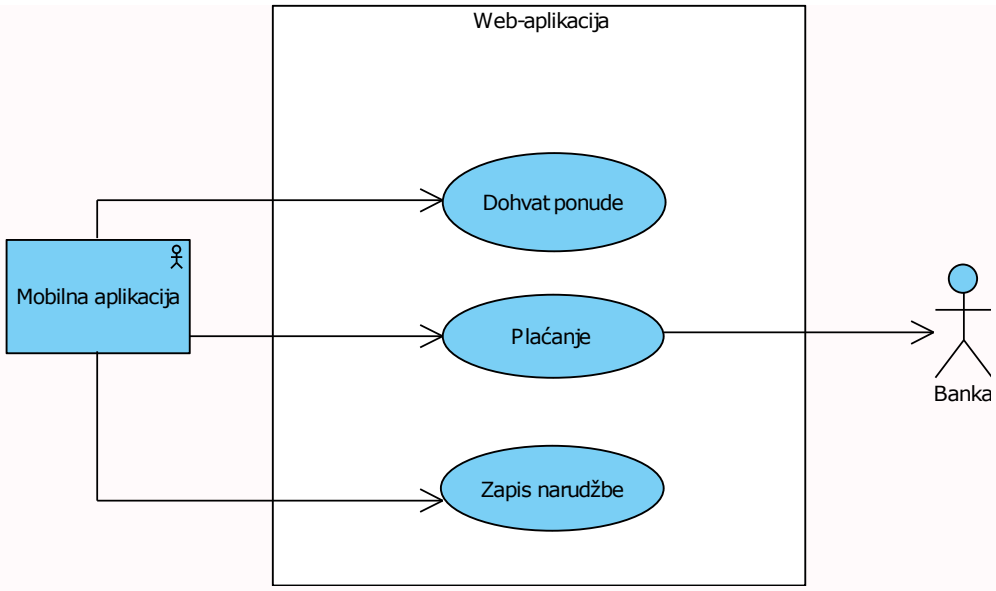
taj modul u interakciji prikazuju kao aktori i smještaju izvan te granice. Taj se pristup naziva još **modeliranje podsustava** ili **dekompozicija** te omogućuje razbijanje složenog sustava na manje, upravljive dijelove, od kojih svaki ima vlastitu granicu. Tako je moguće usmjeriti pozornost na specifične funkcionalnosti, interakcije i odnose unutar svakog podmodula, čime dijagram postaje razumljiviji, a složenost se smanjuje.

Primjer 3.10 — Modeliranje podsustava. Sustav e-trgovine sastoji se od dvaju modula: mobilne i *web*-aplikacije. Mobilnu aplikaciju koriste kupci da bi se prijavili u sustav, pregledali ponudu artikala te obavili kupovinu. Za funkcionalnosti pregleda ponude artikala i kupovine, mobilna aplikacija komunicira s *web*-aplikacijom koja ostvaruje funkcionalnosti dohvata ponude, zapisa narudžbe te provedbe plaćanja (putem banke). Kupac ne može izravno pristupiti *web*-aplikaciji.

Rješenje je prikazano na slikama 3.20 i 3.21.



Slika 3.20: Dijagram obrazaca uporabe mobilne aplikacije



Slika 3.21: Dijagram obrazaca uporabe *web*-aplikacije

Komentar:

Važno je uočiti da na svakom od prethodna dva dijagrama jedan dio sustava predstavlja aktora u odnosu na drugi dio sustava: *web*-aplikacija u odnosu na mobilnu aplikaciju i obrnuto. Da bi se vizualno istaknulo da se u pogledu *web*, odnosno mobilne aplikacije, radi o drukčijoj vrsti aktora nego što su to Kupac ili Banka, tj. da se radi o podsustavu, korištena je notacija aktora u obliku pravokutnika s imenom.

Utvrđivanjem granice sustava, dijagrami obrazaca pružaju usmjereniji i čitljiviji prikaz ponašanja sustava i njegovih odnosa s vanjskim aktorima. To svim dionicima olakšava razumijevanje interakcije i ovisnosti između različitih dijelova unutar većeg sustava te osigurava da fokus modeliranja ostaje relevantan za namjenu sustava te da se spriječe nepotrebne složenosti isključivanjem interakcija izvan granice.

3.2 Postupak izrade dijagrama

Dijagrami obrazaca uporabe najčešće se izrađuju na početku projekta i to na temelju opisa zahtjeva. Da bi taj postupak bio što uspješniji, a izrađeni dijagrami razumljivi dionicima i korisni za daljnje oblikovanje programske potpore, potrebno je primijeniti sistematični pristup. U nastavku su opisani osnovni koraci koje treba slijediti pri izradi tih dijagrama:

- **Utvrđivanje aktora** – izradu dijagrama najbolje je započeti utvrđivanjem vanjskih entiteta koji stupaju u interakciju sa sustavom. To mogu biti korisnici, drugi sustavi ili organizacije.
- **Utvrđivanje obrazaca uporabe** – na temelju opisa (funkcionalnih) zahtjeva za cijeli sustav, potrebno je odrediti specifične funkcionalnosti, ciljeve ili zadatke koje on treba obavljati. Na početku je poželjno zadržati višu razinu apstrakcije (npr. odrediti obrazac „Plaćanje”), a poslije se dopunjavaju konkretni detalji (npr. mogućnost više vrsta plaćanja). Za svaki obrazac uporabe preporučuje se napisati kratak tekstualni opis i usmjeriti se na glavnu radnju

ili cilj.

- **Povezivanje aktora i obrazaca uporabe** – u ovom koraku potrebno je povezati aktore s pripadajućim obrascima uporabe. Pri tome treba razmisliti o tome koji aktori pokreću koji obrazac i je li u pojedini obrazac uključeno više njih.

- **Određivanje granice sustava** – granica sustava pomaže istaknuti opseg sustava i razlikovati ga od vanjskih aktora (izvan granice sustava).

- **Organizacija obrazaca uporabe** – slične ili povezane obrasce uporabe treba grupirati zajedno. Takav pristup pomaže u prepoznavanju zajedničke funkcionalnosti i stvaranju šire slike o tome kako će sustav funkcionirati. Također, potrebno je razmotriti međusobne odnose između obrazaca uporabe (uključivanje, proširenje, generalizacija). Važno je istaknuti da nije nužno da svi obrasci budu prikazani na jednom dijagramu, već se po potrebi mogu razložiti na više njih (hijerarhijski, grupiranje po modulima itd.).

- **Pregled i poboljšanje** – nakon prethodnog koraka, potrebno je pregledati nastali dijagram i provjeriti odražava li točno opisane zahtjeve. Sada je i dobar trenutak za razmišljanje o tome treba li doraditi popis zahtjeva jer je moguće da se detaljnijim promišljanjem o sustavu uoče nejasnoće i nedostaci.

Dijagrami obrazaca uporabe najčešće se razrađuju u više iteracija u suradnji s dionicima, kao što su klijenti, voditelji projekata i članovi tima. Stoga trebaju biti jasni i pregledni da bi ih razumjeli svi dionici. Prema potrebi dijagrame treba revidirati da bi se uključile sve povratne informacije i poboljšala njihova točnost i jasnoća.

3.3 Primjena

Dijagrami obrazaca uporabe najčešće se koriste u ranim fazama procesa oblikovanja programske potpore i njihova je osnovna uloga razrada funkcionalnih zahtjeva sustava. Ponajviše se koriste u procesu prikupljanja i analize zahtjeva jer pomažu dionicima i razvojnim inženjerima u prepoznavanju i razumijevanju funkcionalnih zahtjeva sustava na visokoj razini apstrakcije. Stvaranjem vizualnog prikaza interakcija između sustava i njegovih aktora tim može razjasniti zahtjeve, utvrditi funkcionalnosti koje nedostaju i ukloniti nejasnoće.

Međutim, ti se dijagrami mogu koristiti tijekom cijelog životnog ciklusa projekta – u fazama oblikovanja, implementacije i ispitivanja razvojni inženjeri koriste ih da bi provjerili usklađenost trenutnog stanja i opsega sustava sa zadanim. U fazi oblikovanja arhitekture sustava dijagrami obrazaca uporabe pomažu arhitektima programske potpore u utvrđivanju cjelokupne strukture sustava. Koriste ih za prepoznavanje komponenti ili podsustava odgovornih za različite obrasce uporabe te određivanje odnosa i interakcija među njima. Ta faza može uključivati i poboljšanje (doradu) obrazaca uporabe da bi oblikovanje sustava bilo sveobuhvatno i kohezivno.

Također, dijagrami obrazaca uporabe korisni su u postupku oblikovanja korisničkog sučelja (engl. *user interface – UI*) jer pružaju uvid u interakcije između korisnika i sustava. Razvojni inženjeri mogu utvrditi ključne zadatke koje korisnici trebaju izvršavati te osigurati da je korisničko sučelje intuitivno i korisnički prijateljsko.

Nadalje, ispitni tim koristi dijagrame obrazaca uporabe za utvrđivanje ispitnih scenarija u procesu planiranja ispitivanja i razvoja ispitnih slučajeva. Proučavanjem interakcije između aktora i sustava tim može razviti ispitne slučajeve koji pokrivaju zadane funkcionalnosti i provjeriti zadovoljava li sustav svoje zahtjeve.

Konačno, dijagrami obrazaca uporabe mogu služiti kao sažet, vizualni prikaz funkcionalnosti sustava za potrebe dokumentacije, što je posebno važno u postupku planiranja budućih nadogradnji i evolucije sustava. Pomažu u komunikaciji zahtjeva i oblikovanja sustava različitim dionicima, poput klijenata, voditelja projekata i razvojnih inženjera. U tablici 3.1 nalazi se sažet i sistematiziran prikaz primjene obrazaca uporabe u aktivnostima programskog inženjerstva.

Tablica 3.1: Primjena obrazaca uporabe za vrijeme različitih aktivnosti programskog inženjerstva

Aktivnost	**Primjena**
Specifikacija programske potpore	Utvrđivanje i razumijevanje funkcionalnih zahtjeva sustava na visokoj razini apstrakcije.
Analiza i oblikovanje	Utvrđivanje komponenti ili podsustava odgovornih za različite obrasce uporabe te određivanje međusobnih odnosa i interakcija.
Implementacija	Provjera usklađenosti trenutnog stanja i opsega sustava sa zadanim.
Ispitivanje	Utvrđivanje ispitnih scenarija.
Evolucija	Dokumentacija i komunikacija s dionicima.

4. Sekvencijski UML dijagrami

Dijagrami obrazaca uporabe daju osnovni prikaz interakcije između vanjskih aktora i sustava i konkretnih funkcionalnosti koje sustav pruža. Međutim, da bi se pojedinosti te interakcije bolje razumjele, kao što su vremenski slijed ili uvjetno izvođenje, potrebne su dodatne informacije. Dijagrami obrazaca uporabe sami po sebi nisu dovoljno izražajni.

U svrhu detaljnije vizualizacije i analize dinamičkog ponašanja sustava koriste se sekvencijski UML dijagrami. Oni omogućuju detaljan prikaz slijeda događaja (interakcija) i objekata koji sudjeluju u tim interakcijama. Sekvencijski dijagrami korisni su u različitim fazama razvoja programske podrške, kao što su analiza zahtjeva, oblikovanje i dokumentacija. Premda se najčešće koriste za opisivanje tijeka pojedinih obrazaca uporabe na visokoj razini apstrakcije, mogu se upotrijebiti i za detaljniji prikaz tijeka izvođenja algoritama i procedura na nižoj razini apstrakcije. Također, sekvencijski dijagrami koriste se i za prikaz interakcije između objekata koji predstavljaju instance razreda. Sveukupno, ovi dijagrami pomažu razvojnim inženjerima i dionicima da bolje vizualiziraju i razumiju dinamičko ponašanje sustava, uoče potencijalne probleme te provjere zadovoljava li sustav željene zahtjeve.

4.1 Definicija i osnovni elementi

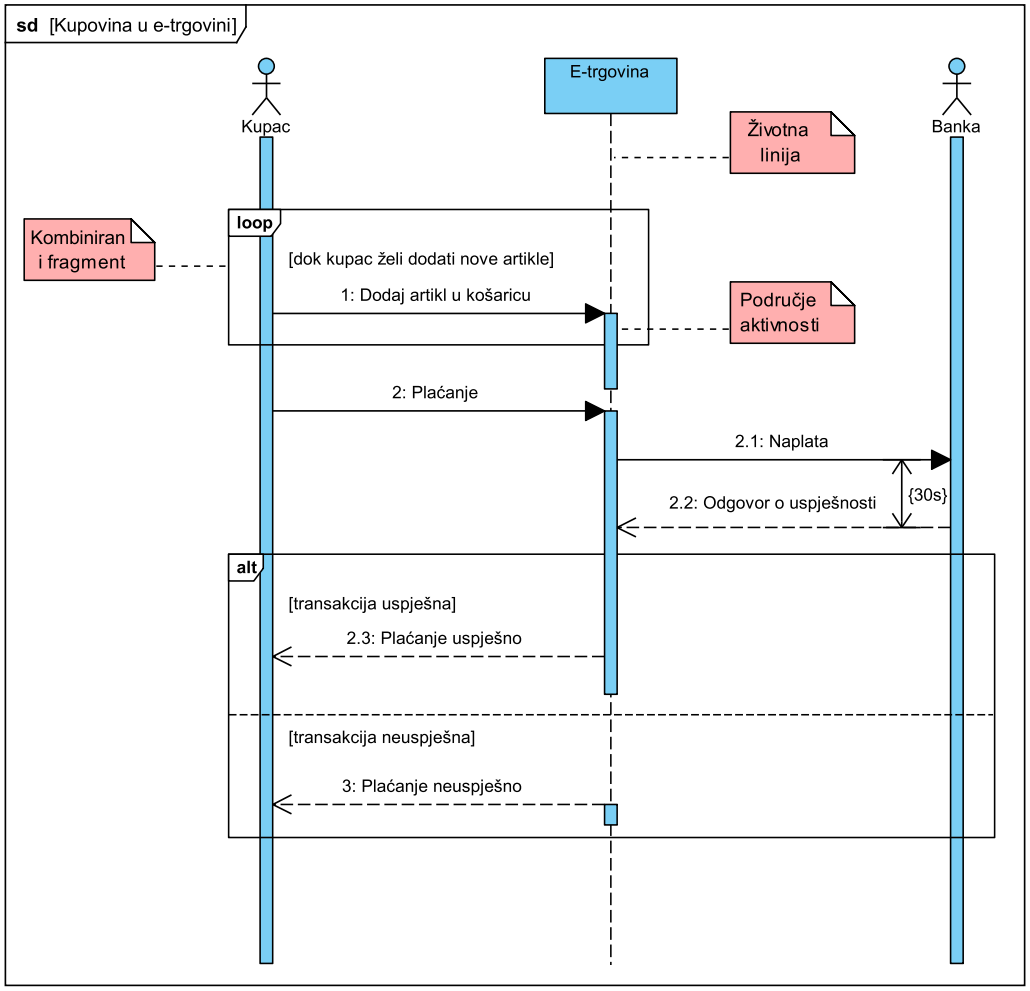
Sekvencijski UML dijagrami (engl. *sequence diagram*) ponašajni su UML dijagrami koji se koriste za prikazivanje slijeda interakcija, odnosno razmjene poruka, između objekata u sustavu tijekom vremena. Semantički su bogatiji od dijagrama obrazaca uporabe te uključuju veći skup elemenata i imaju nešto složeniju sintaksu. Osnovni elementi, prisutni na svakom sekvencijskom dijagramu, su:

- **Objekti** – predstavljaju aktore i komponente sustava (jedan objekt može predstavljati i cijeli sustav). Prikazuju se ikonom čovječuljka (ako se radi o aktorima) ili pravokutnikom s imenom objekta. Ako se dijagramom prikazuje komunikacija između instanci razreda, za objekte se navodi ime objekta i ime razreda razdvojeno, na primjer: „**imeObjekta: ImeRazreda**”. Ponekad se za ime objekta koristi samo ime razreda, ispred kojeg se nalazi znak dvotočke, na primjer: „**:ImeRazreda**”. U tom se slučaju radi o anonimnom objektu određenog razreda.
- **Životne linije (engl. *lifeline*)** – okomite isprekidane linije koje se protežu od objekta prema dolje i predstavljaju postojanje objekta tijekom niza interakcija. Prolazak vremena je u smjeru dolje. Prema zadnjoj inačici norme UML-a, 2.5.1., životna linija uključuje i objekt,

koji se naziva glava (engl. *head*) životne linije.

- **Područje aktivnosti (engl.*activation bar*)** – tanki pravokutnici postavljeni preko životnih linija koji predstavljaju razdoblje u kojem objekt aktivno sudjeluje u određenoj interakciji ili izvodi neku radnju.
- **Poruke** – predstavljaju interakciju ili komunikaciju između objekata. Prikazane su horizontalnim strelicama između životnih linija sudjelujućih objekata. Smjer strelice označava tijek poruke, a iznad strelice specificira se ime poruke, parametri i povratna vrijednost (opcionalno). Na nižoj razini apstrakcije poruke se mogu shvatiti i kao pozivi funkcija.
- **Kombinirani fragmenti** – koriste se za prikaz kontrolnih struktura, poput petlji, uvjetnih izraza ili paralelnog izvođenja, u nizu interakcija. Označeni su velikim pravokutnikom koji obuhvaća relevantne poruke i životne linije, s operatorom (npr., *alt* za alternative, *loop* za petlje) u gornjem lijevom kutu.
- **Vremenska ograničenja** – dodaju se porukama da bi se specificirali potrebni ili očekivani vremenski intervali između interakcija ili događaja. Najčešće se predstavljaju tekstnim bilješkama pored odgovarajućih poruka.

Na slici 4.1 prikazan je primjer jednog jednostavnog sekvencijskog dijagrama koji prikazuje



Slika 4.1: Primjer sekvencijskog dijagrama

tijek kupovine u e-trgovini i sadržava sve osnovne elemente. U nastavku poglavlja nalazi se detaljniji pregled svakog od navedenih elemenata s primjerima korištenja.

4.1.1 Poruke

Na sekvencijskim dijagramima različite vrste poruka koriste se za prikazivanje različitih vrsta interakcija ili komunikacije među objektima. Korištenjem poruka sekvencijski dijagrami mogu precizno modelirati složene interakcije i obrasce komunikacije koji se javljaju unutar sustava tako što pomažu pri vizualizaciji i analizi njegova dinamičkog ponašanja.

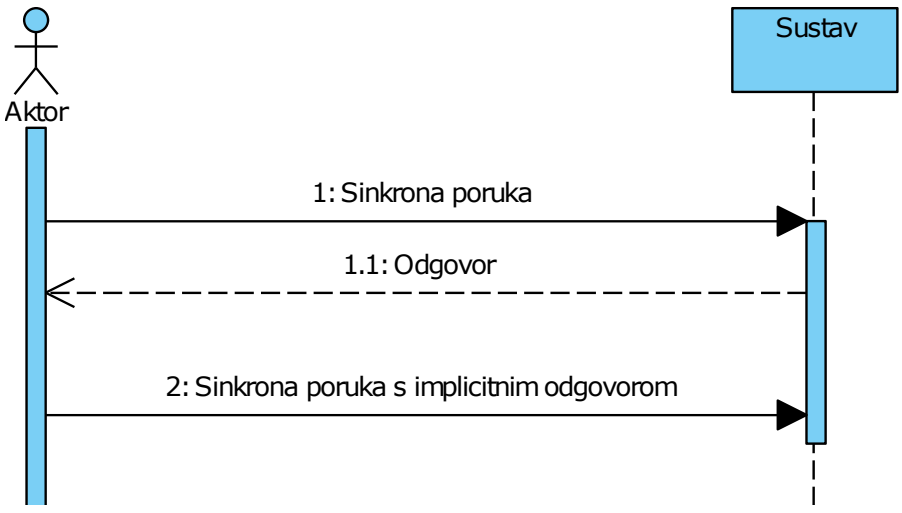
Svaka poruka naznačena je odgovarajućom vrstom strelice usmjerene od izvorišta (engl. *message caller*) prema odredištu (engl. *message receiver*) te potpisom poruke iznad strelice. Potpis poruke navedenog je oblika:

naziv_poruke(argumenti):povratna_vrijednost

U potpisu poruke opcionalno je sve osim naziva poruke, a za argumente i povratnu vrijednost može se navesti i tip (int, float, itd.). Tipično su poruke numerirane po vremenskom slijedu slanja u jednoj razini ili više njih (1., 2., 3... ili 1.1., 1.2. itd.)

Glavne vrste poruka na sekvencijskim dijagramima su:

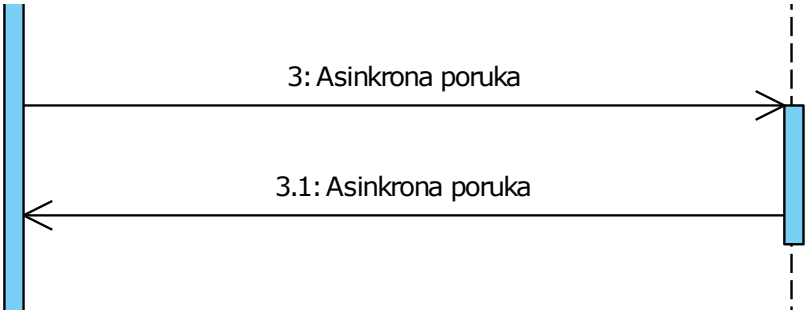
- **Sinkrone poruke** – predstavljaju poziv od jednog objekta do drugog, u kojem **pošiljatelj poruke čeka da primatelj završi svoju obradu** prije nego što nastavi s vlastitim izvođenjem. Sinkrone poruke prikazane su kao strelice s cjelovitom linijom i ispunjenim trokutastim vrhom. Strelica pokazuje od životne linije pošiljatelja do životne linije primatelja. Završetak izvođenja, tj. **odgovor na poruku** može, ali ne mora, biti eksplicitno naznačen. Poruka odgovora prikazuje se kao isprekidana horizontalna linija sa strelicom usmjerenom prema pošiljatelju izvorne poruke. Ako se po završetku izvođenja vraća povratna vrijednost koja utječe na daljnji tijek izvođenja, poruka odgovora svakako se treba prikazati. Međutim, poruka odgovora ne mora se prikazati (tzv. implicitan odgovor) ako se ne vraća nikakva vrijednost ili povratna vrijednost nema učinak na daljnji tijek izvođenja. Slika 4.2 ilustrira dva primjera slanja sinkrone poruke: s eksplicitnim i implicitnim odgovorom.



Slika 4.2: Sinkrone poruke i odgovori

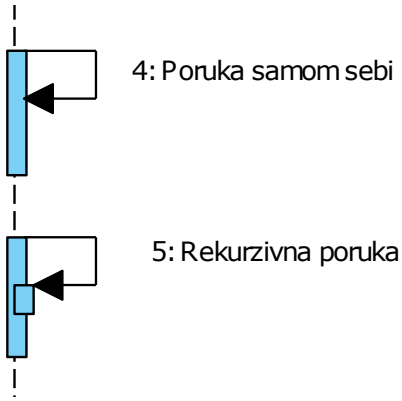
- **Asinkrone poruke** – predstavljaju poziv od jednog objekta do drugog, u kojem **pošiljatelj**

poruke ne čeka da primatelj završi svoju obradu i nastavlja s vlastitim izvođenjem odmah nakon slanja poruke. Asinkrone poruke prikazane su cjelovitom linijom s običnom strelicom, koja pokazuje od životne linije pošiljatelja do životne linije primatelja, kao na slici 4.3. Odgovor na asinkronu poruku može se prikazati kao nova asinkrona poruka, ali se može koristiti isti prikaz kao i za odgovor na sinkronu poruku.



Slika 4.3: Asinkrone poruke

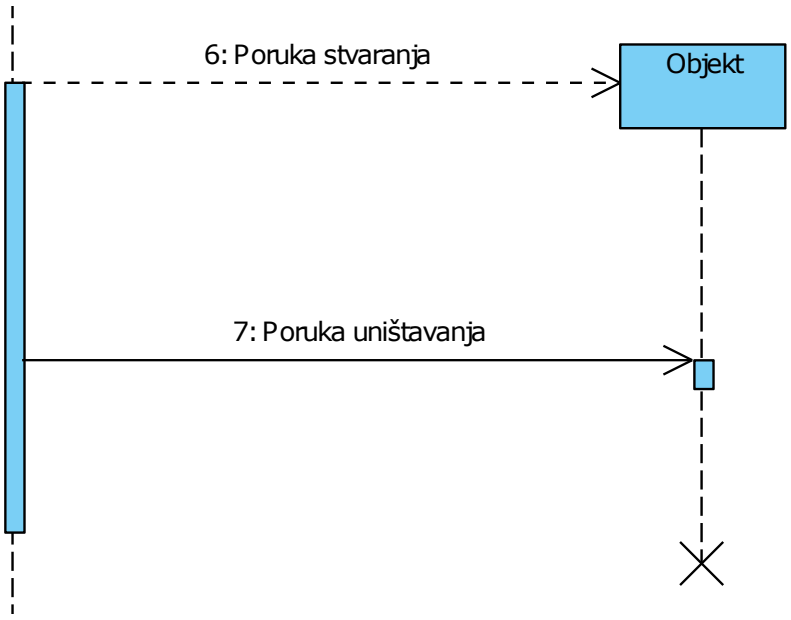
- **Poruke samom sebi i rekurzivne poruke** – predstavljaju situaciju u kojoj objekt poziva svoju metodu ili izvodi neku radnju na sebi, moguće i u rekurziji. Poruke prema sebi ilustrirane su kao cjelovita crta s popunjenom strelicom te stvaraju oblik U od trake aktivacije na životnoj liniji objekta natrag do iste te linije. Slika 4.4 prikazuje obje vrste poruka.



Slika 4.4: Poruka samom sebi i rekurzivna poruka

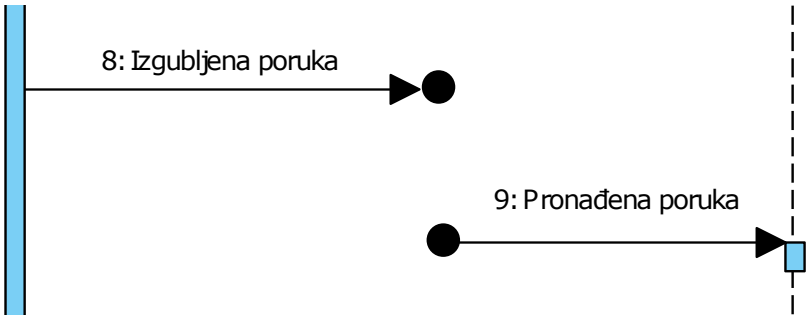
- **Poruke stvaranja i uništenja** – poruke stvaranja označavaju stvaranje ili instanciranje objekta. Prikazane su isprekidanom linijom s običnom strelicom; od životne linije pošiljatelja do novostvorenog objekta. Poruke uništenja predstavljaju brisanje ili završetak postojanja objekta u memoriji. Prikazane su ravnom linijom s običnom strelicom; od životne linije pošiljatelja do životne linije primatelja te se uz to, da bi se naznačilo uništenje objekta, dodaje oznaka veliko X preko životne linije primatelja. U potpisu ovih poruka ponekad stoje i stereotipi[1] «create» odnosno «destroy».

[1] **Stereotipi** su jednostavan i fleksibilan mehanizam proširenja unutar jezika UML. Postojećim sintaksnim elementima jezika UML dodaje se stereotip da bi im se dalo posebno značenje. Stereotipovi se pišu unutar dvostrukih šiljastih zagrada. Tipičan su primjer korištenja stereotipa veze uključivanja i proširenja na dijagramu obrazaca uporabe – isprekidanoj liniji koja ukazuje na ovisnost između dvaju obrazaca uporabe dodaje se stereotip «include», odnosno «extend» da bi se pobliže utvrdila vrsta veze između dvaju obrazaca.



Slika 4.5: Poruke stvaranja i uništenja

- **Izgubljene i pronađene poruke** – Izgubljene poruke su poruke koja su poslane, ali ih nije primio nijedan objekt ili nemaju određenog primatelja. Ilustrirane su kao ravna linija s trokutastom strelicom i malim popunjenim krugom na kraju, koji pokazuje prema praznom prostoru. Pronađene poruke su poruke koje je objekt primio, ali nemaju određenog pošiljatelja. Prikazane su kao ravna linija s trokutastom strelicom i malim popunjenim krugom na početku, koja pokazuje od praznog prostora do životne linije primatelja. Te dvije vrste poruka prikazane su na slici 4.6.

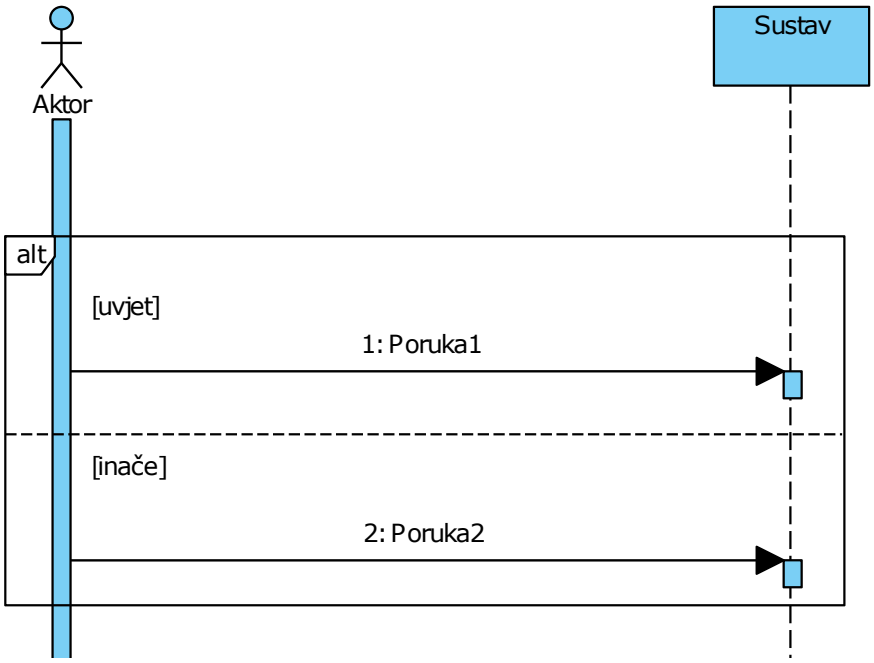


Slika 4.6: Izgubljene i pronađene poruke

4.1.2 Kombinirani fragmenti

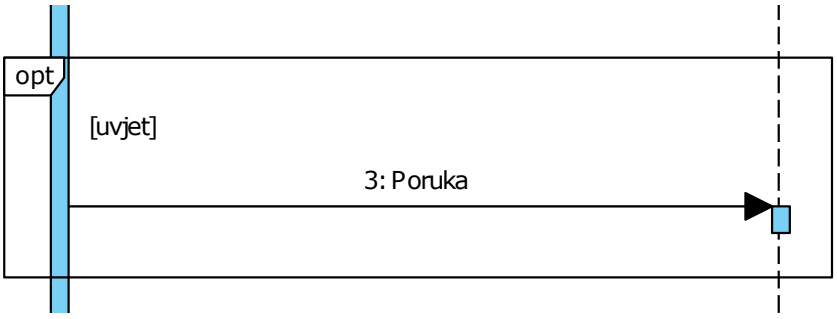
Kombinirani fragmenti koriste se u sekvencijskim dijagramima za predstavljanje kontrolnih struktura i modeliranje složenih interakcija koje uključuju uvjetnu logiku, petlje ili paralelno izvršavanje. Oni olakšavaju razumijevanje, analizu i provjeru dinamičkih aspekata sustava. Fragmenti se mogu ugnježđivati, a glavne vrste kombiniranih fragmenata u sekvencijskim dijagramima su:

- **alt (engl. *alternative*)** – predstavlja uvjetno izvršavanje (*if-then-else*)u kojem će se dogoditi jedan od nekoliko alternativnih skupova interakcija na temelju specifičnog uvjeta ili skupa uvjeta. Svaka alternativa odvojena je isprekidanom crtom unutar fragmenta, s odgovarajućim uvjetom napisanim pored nje. Izvršava se samo jedna alternativa, ovisno o tome koji je uvjet ispunjen. Primjer je prikazan na slici 4.7.



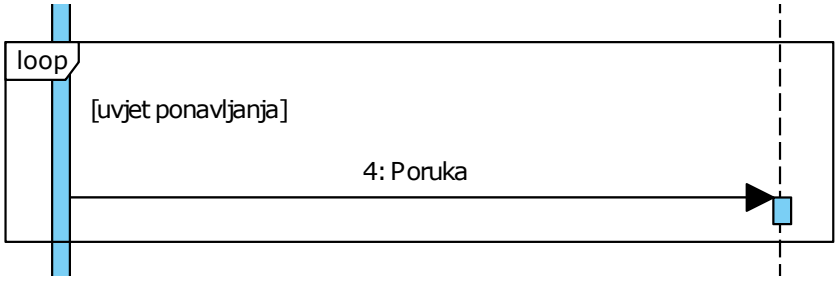
Slika 4.7: Kombinirani fragment alternative (*alt*)

- **opt (engl. *optional*)** – predstavlja opcionalno izvršavanje interakcija, što znači da se uključene interakcije događaju samo ako je ispunjen određeni uvjet. Slično je fragmentu alt, ali se koristi kada postoji samo jedna alternativa (jednostruki *if*). Primjer je prikazan na slici 4.8.



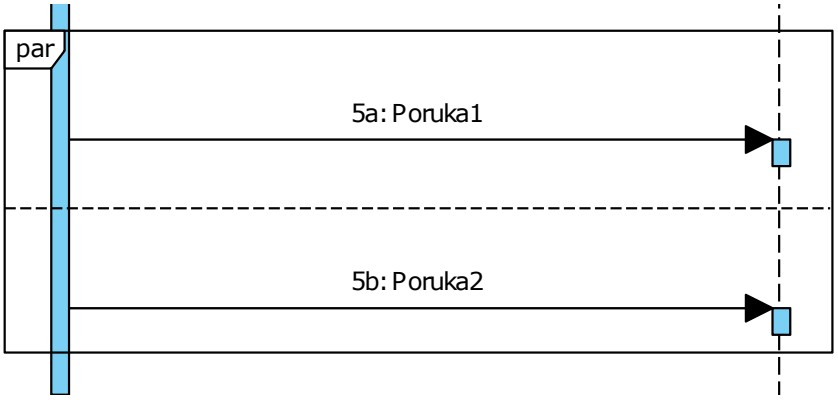
Slika 4.8: Kombinirani fragment opcionalnosti (*opt*)

- **loop** – predstavlja skup interakcija koje se ponavljaju više puta na temelju određenog uvjeta ili izraza petlje. Uvjet petlje ili izraz napisan je u gornjem lijevom kutu te vrste kombiniranog fragmenta, a ukazuje na broj iteracija ili kriterij zaustavljanja petlje. Primjer je prikazan na slici 4.9.



Slika 4.9: Kombinirani fragment petlje (*loop*)

- **par (engl. *parallel*)** – predstavlja paralelno (istodobno) izvršavanje više skupova interakcija. Svaka paralelna grana odvojena je isprekidanom crtom unutar fragmenta, a sve se grane izvršavaju simultano. Primjer je prikazan na slici 4.10.



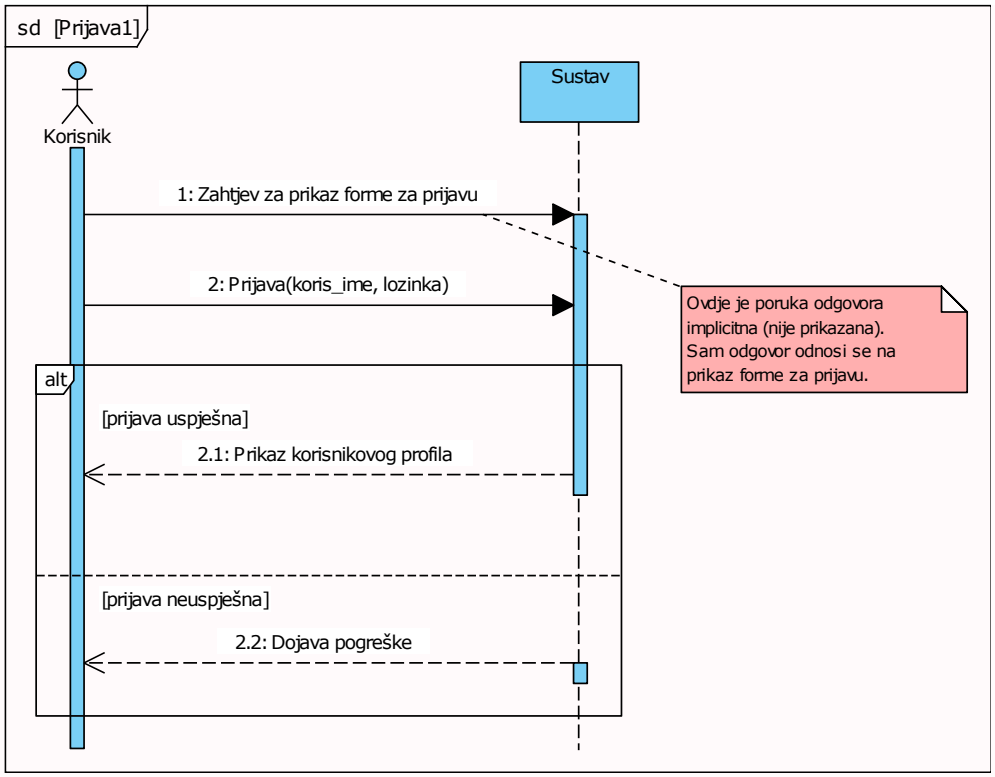
Slika 4.10: Kombinirani fragment paralelnog izvođenja (*par*)

Primjer 4.1 — Uvjetovani odgovor na poruku. Modelirajte proces prijave u sustav u dvjema inačicama:

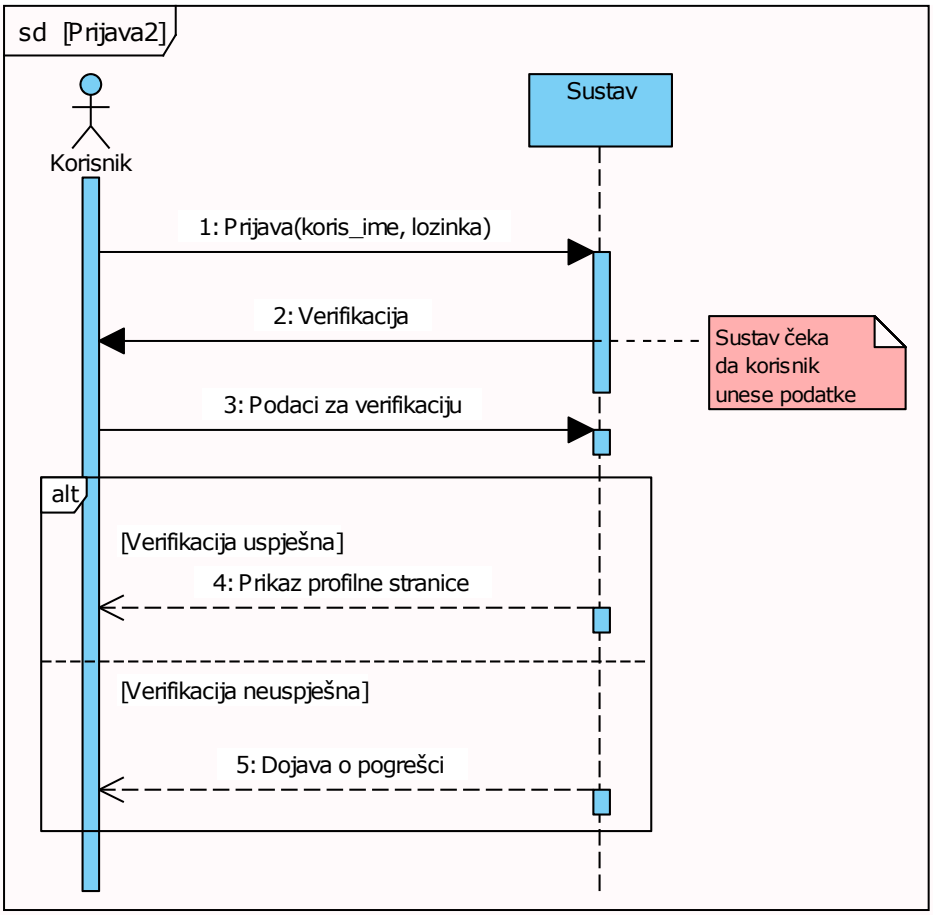
A) Korisnik najprije zatraži prikaz forme za prijavu. Nakon što mu sustav prikaže formu, on unosi podatke za prijavu (korisničko ime i lozinka). Ako je prijava uspješna, korisniku se prikazuje njegov profil. U suprotnom se dojavljuje pogreška.

B) Korisnik se prijavljuje u sustav sa svojim korisničkim imenom i lozinkom (kao i u prethodnoj inačici), ali je za prijavu potrebna i dodatna verifikacija (npr. verifikacija da nije bot). Za dodatnu verifikaciju sustav otvara poseban ekran u koji korisnik mora unijeti tražene podatke. Sustav čeka dok korisnik ne unese te podatke. Ako je verifikacija uspješna, korisniku se prikazuje profilna stranica, a u suprotnom poruka o pogrešci.

Rješenje je prikazano na slikama 4.11 i 4.12.



Slika 4.11: Sekvencijski dijagram procesa prijave u sustav – inačica A

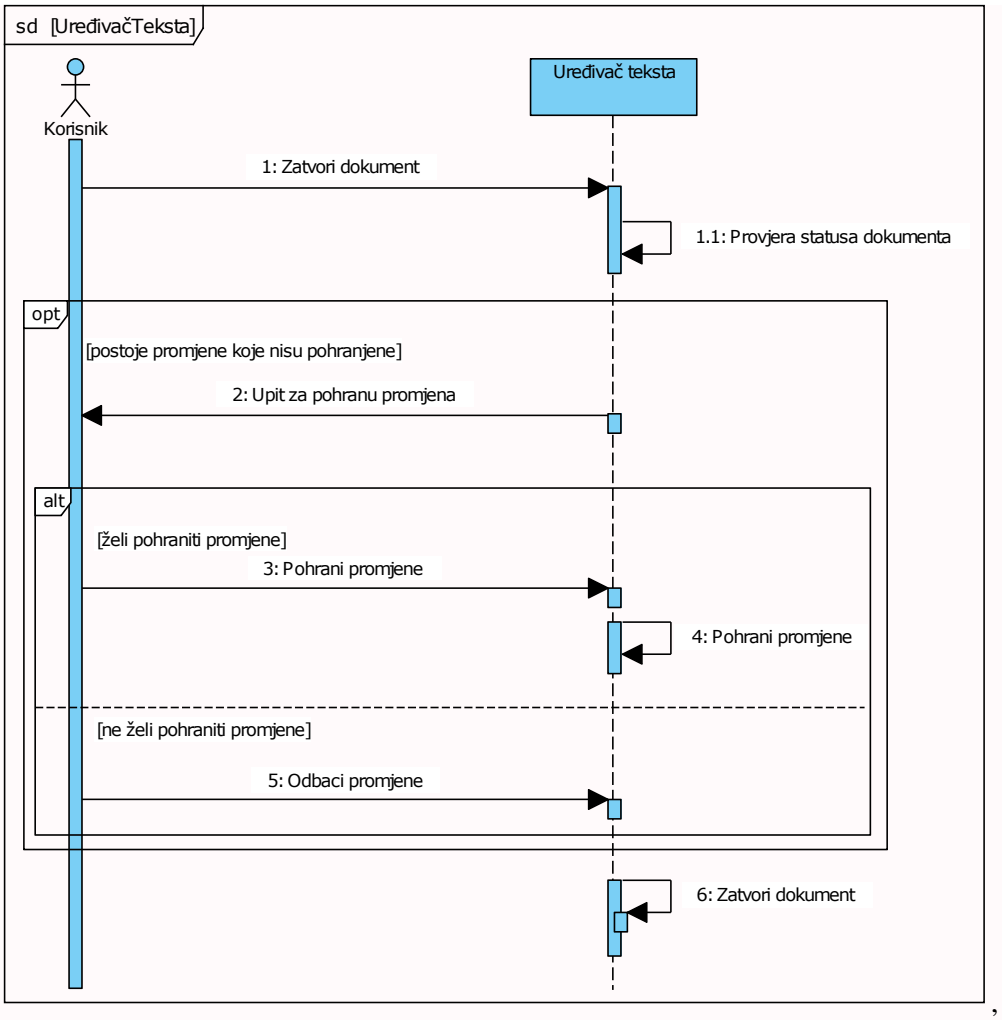


Slika 4.12: Sekvencijski dijagram procesa prijave u sustav – inačica B

Komentar: Povratne poruke nacrtane na početku pojedinačnih aktivacija u fragmentima (2.1 i 2.2 u inačici A te 4 i 6 u inačici B) predstavljaju povratnu poruku od ranije poslane sinkrone poruke (2 u inačici A, odnosno 3 u inačici B), samo rastavljenu po alternativama.

Primjer 4.2 — Uvjetno izvođenje. U programu za uređivanje teksta modelirajte situaciju u kojoj korisnik želi zatvoriti dokument. Program mora provjeriti status dokumenta, tj. ima li promjena koje nisu pohranjene. Ako ima, pita korisnika želi li ih spremiti. Korisnik to može prihvatiti i te se promjene spremaju, a može i odbiti pa se promjene zanemaruju. Dokument se nakon toga zatvara.

Rješenje je prikazano na slici 4.13.



Slika 4.13: Primjer uvjetnog izvođenja u programu za uređivanje teksta

Komentar:

Opt je, za razliku od *alta*, dobar izbor za vanjski okvir jer se kada nema nepohranjenih promjena neće dogoditi ništa. Unutar okvira *opt* mora biti okvir *alt* jer korisnik ima na izbor dvije opcije: „Pohrani" i „Odustani".

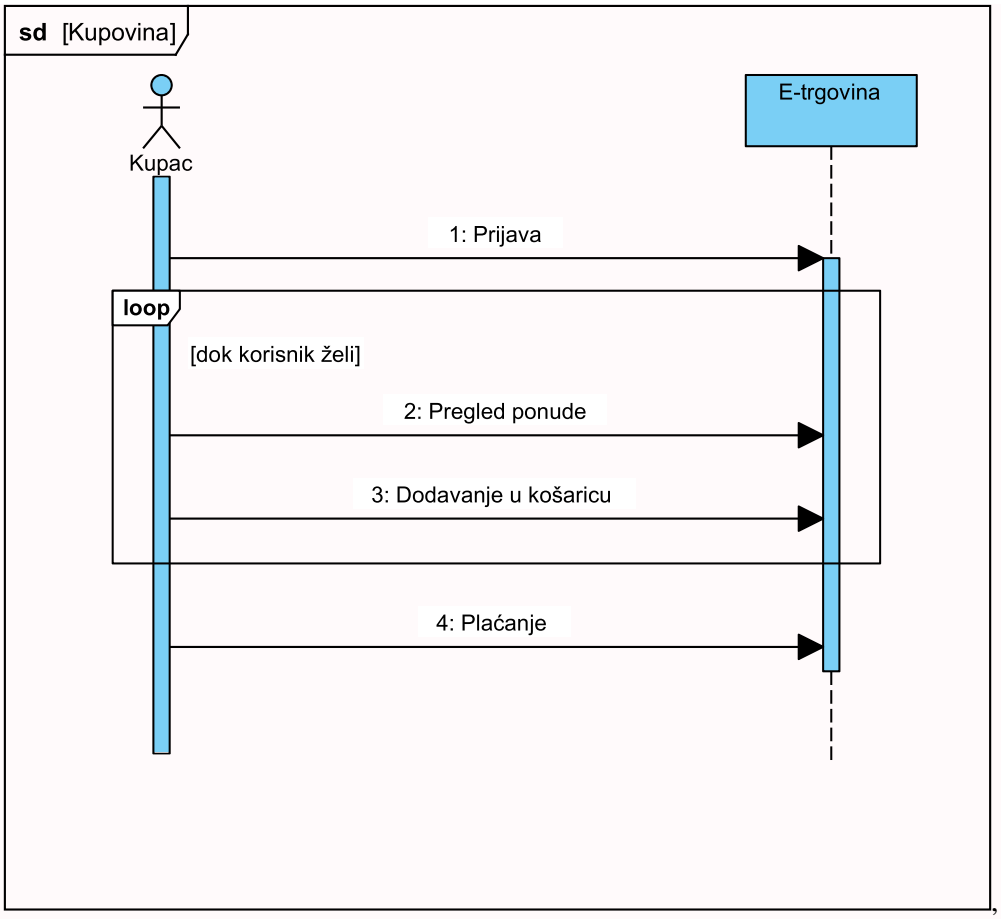
■

Primjer 4.3 — Izvođenje u petlji. Modelirajte sljedeća dva opisa na sekvencijskom dijagramu:

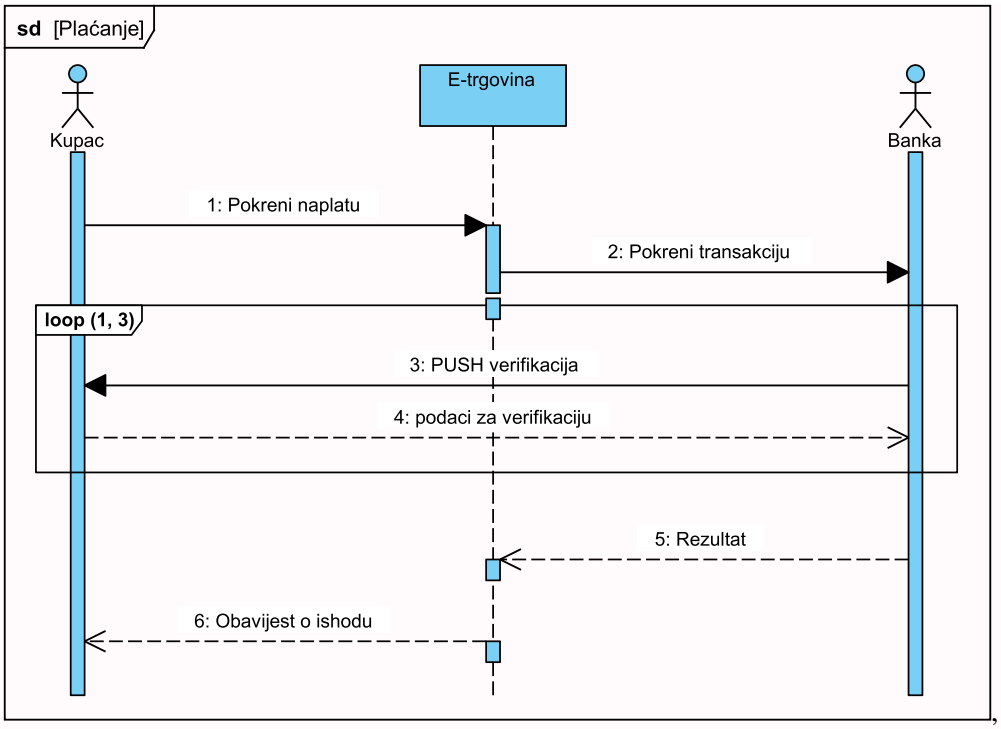
A) Korisnik nakon prijave u e-trgovini može pregledavati ponudu i dodavati artikle dok god to želi. Nakon što je završio s dodavanjem artikala, prelazi na naplatu.

B) Pri plaćanju u e-trgovini bankovnom karticom *web*-aplikacija pokreće transakciju s bankom, na što banka kupcu šalje PUSH poruku za verifikaciju. Poruka se šalje najviše tri puta. Transakcija zatim završava te se korisnika obavještava o rezultatu.

Rješenje je prikazano na slikama 4.14 i 4.15.



Slika 4.14: Primjer uvjetom ograničenog izvođenja petlje



Slika 4.15: Primjer izvođenja u petlji ograničenog brojem iteracija

Osim tih glavnih vrsta fragmenata postoji i nekoliko rjeđe korištenih vrsta:

- **break** – uvjetovani prekid u normalnom tijeku izvođenja. Ako je zadovoljen uvjet za prekid, izvršavaju se interakcije unutar fragmenta *break*, a one nakon fragmenta *break* se preskaču.
- **critical** – skup interakcija koje se moraju izvršiti uzajamno isključivo s drugim kritičnim regijama u sustavu (kritični odsječak) i osiguravajući da samo jedan objekt može pristupiti dijeljenom resursu u jednom trenutku. To se koristi za modeliranje sinkronizacije i zaštite pristupa resursu u konkurentnim sustavima.
- **neg** – skup interakcija koje se smatraju nevažećima ili se ne bi trebale dogoditi u sustavu. Obično se koristi za modeliranje scenarija s pogreškama ili situacija koje krše očekivano ponašanje sustava.
- **assert** – skup interakcija koje se uvijek moraju dogoditi, a ako se ne dogode, smatra se da je sustav u nevažećem stanju. Ovaj se fragment koristi za izražavanje važnih ograničenja ili invarijanti u sustavu.
- **ignore** – skup interakcija koje bi se trebale ignorirati za vrijeme analize ili evaluacije sekvencijskog dijagrama. To može biti korisno za usmjeravanje na specifične dijelove dijagrama ili pojednostavljenje složenih dijagrama za vrijeme analize.
- **consider** – suprotnost fragmentu *ignore*. Predstavlja skup interakcija koje bi se trebale razmotriti za vrijeme analize ili evaluacije sekvencijskog dijagrama, dok se sve druge interakcije izvan fragmenta ignoriraju.

4.1.3 Vremenska ograničenja

Vremenska ograničenja na sekvencijskim dijagramima koriste se za specificiranje potrebnih ili očekivanih vremenskih intervala između interakcija ili događaja. Osobito su korisna pri utvrđivanju vremenskih zahtjeva ili ograničenja performansi tako što omogućuju razvojnim inženjerima bolje razumijevanje i analizu vremenskog aspekta ponašanja sustava, prepoznavanje potencijalnih problema s performansama i osiguravanje da sustav zadovoljava svoje vremenske zahtjeve. Vremenska ograničenja mogu biti posebno važna za sustave koji rade u stvarnom vremenu (engl. *real time*) i za vremenski kritične sustave, u kojima je ispunjavanje specifičnih vremenskih zahtjeva nužno za ispravno funkcioniranje.

Na sekvencijskim dijagramima, vremenska ograničenja prikazana su tekstnim anotacijama, često unutar vitičastih zagrada { } i postavljena pored odgovarajućih poruka ili događaja. Razlikuju se dvije osnovne uporabe vremenskih ograničenja:

- **ograničenje proteklog vremena između dvaju događaja (engl. *duration constraints*)** – specificira očekivano (maksimalno) trajanje intervala između dvaju događaja ili poruka. Na primjer, ograničenje koje navodi da vrijeme između slanja poruke zahtjeva i primanja odgovora ne bi trebalo biti dulje od jedne sekunde, moglo bi biti napisano kao:

  $$vremenska\_razlika(zahtjev, odgovor) <= 1s$$

  Ograničenje trajanja najčešće se grafički ističe dodatnom okomitom strelicom koja povezuje dvije poruke na koje se ograničenje odnosi.

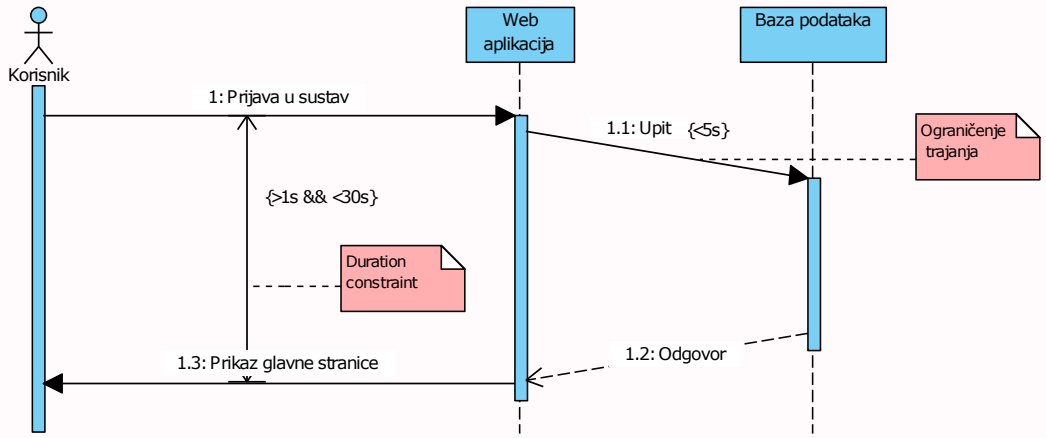
- **ograničenja trajanja jednog događaja (engl. *time constraint*)** – određuje potrebno ili očekivano trajanje pojedine radnje ili interakcije. Na primjer, ograničenje koje ukazuje da bi objekt trebao završiti određenu radnju unutar tri sekunde može biti napisano kao:

  $$trajanje(radnja) <= 3s$$

  Ograničenje vremena upisuje se uz samu poruku. Poruka ograničenog trajanja vrlo se često grafički prikazuje kao ukošena linija.

Primjer 4.4 — Vremenska ograničenja. Potrebno je modelirati sljedeću specifikaciju. Korisnik se prijavljuje u *web*-aplikaciju te se očekuje da će od slanja zahtjeva za prijavu do prikaza glavne stranice proći više od jedne sekunde, ali manje od 30 sekundi. Za provjeru korisničkih podataka *web*-aplikacija šalje upit bazi podataka. Upit se mora izvršiti za manje od pet sekundi.

Rješenje je prikazano na slici 4.16.



Slika 4.16: Primjer modeliranja vremenskog ograničenja pri prijavi u *web*-aplikaciju

Vremenska ograničenja mogu se izraziti i korištenjem složenijih vremenskih izraza (engl. *timing expressions*) koji mogu uključivati aritmetičke operacije ili usporedbe između različitih vremenskih vrijednosti. Na primjer, specifikacija:

$$vrem\_razlika(event1, event2) = 2 * vrem\_razlika(event3, event4)$$

opisuje ograničenje da bi vrijeme između dvaju događaja trebalo biti dvostruko dulje od vremena između dvaju drugih događaja.

4.2 Postupak izrade dijagrama

Prvi sekvencijski dijagrami izrađuju se na samom početku projekta kao nadopuna dijagramima obrazaca uporabe. Međutim, kao što je već i prije rečeno, ti dijagrami imaju znatno širu primjenu, pa se izrađuju i u kasnijim fazama projekta za vrijeme razrade detalja funkcioniranja pojedinih komponenti, razreda, sve do razine opisa tijeka izvođenja algoritama i procedura.

Izrada sekvencijskog dijagrama na temelju tekstualne specifikacije prirodnim jezikom uključuje nekoliko koraka da bi se osiguralo da dijagram točno predstavlja ponašanje i interakcije sustava:

- **Analiza i razumijevanje tekstualne specifikacije** – važan pripremni korak je razumijevanje zahtjeva sustava, interakcije i ograničenja.
- **Utvrđivanje objekata i aktora** – prvi je korak izrade dijagrama određivanje objekata, aktora i komponenti uključenih u interakcije opisane u tekstnoj specifikaciji. Aktori su obično vanjski entiteti u interakciji sa sustavom, dok objekti predstavljaju instance razreda ili komponenti unutar sustava. Svakom aktoru/objektu potrebno je pridružiti naziv i životnu liniju.
- **Utvrđivanje interakcije i poruka** – potrebno je odrediti tijek razmjene poruka (ili poziva metoda) koji se događaju između objekata i aktora u sustavu. Važno je obratiti pozornost na redoslijed u kojem se te interakcije događaju te na sve uvjete ili petlje koji utječu na tijek poruka. Svaku interakciju potrebno je modelirati odgovarajućom vrstom poruke (sinkrone poruke, asinkrone poruke itd.).

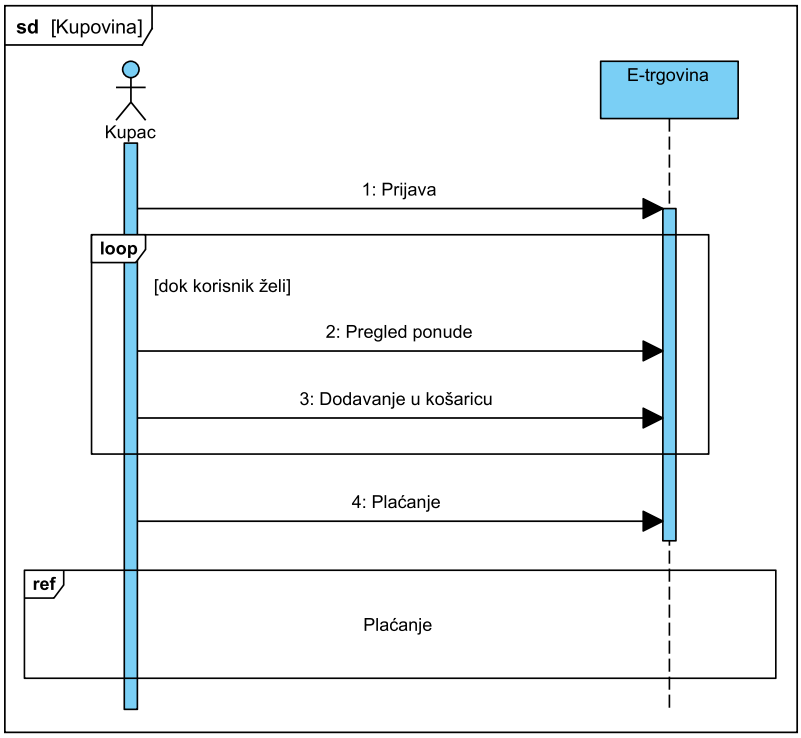
- **Uvođenje kombiniranih fragmenata** – nakon što su utvrđene osnovne poruke, potrebno je razmotriti mogućnosti uvjetnog izvođenja, alternativnog ishoda, ponavljanja, izvođenja u paraleli itd. te uključiti prikladne kombinirane fragmente (npr. alt, opt, loop).

- **Utvrđivanje područja aktivnosti** – za svaki objekt ili aktora uključenog u interakciju, nužno je nacrtati aktivacijske trake preko njihovih linija života da bi se prikazalo trajanje njihove aktivnosti ili izvršenja metode. Većina programskih alata za izradu UML dijagrama utvrđuje područje aktivnosti automatski, no svakako se preporučuje provjera rezultata automatskog iscrtavanja jer su moguće pogreške.

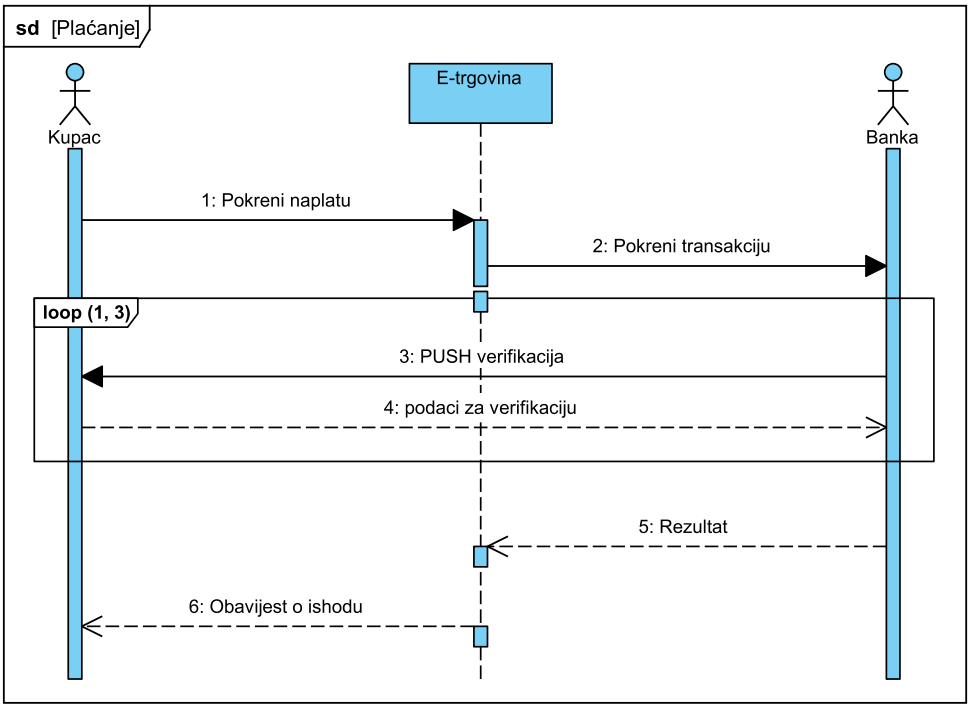
- **Specificiranje vremenskih ograničenja (opcionalno)** – ako tekstna specifikacija uključuje vremenska ograničenja, uputno je dodati vremenska ograničenja na odgovarajuća mjesta u sekvencijskom dijagramu.

- **Provjera i dorada dijagrama** – pri svakoj doradi važno je provjeriti predstavlja li taj sekvencijski dijagram ispravno zahtjeve i interakcije iz tekstne specifikacije. Izrada dijagrama iterativan je postupak i po potrebi se dorađuje da bi se poboljšala jasnoća, čitljivost i ispravnost. Također, potrebno je potvrditi dijagram sa zainteresiranim dionicima da bi se ustanovilo podudara li se on s njihovim razumijevanjem ponašanja sustava.

Konačno, treba istaknuti da je pri izradi dijagrama izrazito važno voditi računa o njegovoj preglednosti i izražajnosti. To znači da kada je riječ o složenijoj interakciji, nikako nije nužno na jednom dijagramu prikazati sve interakcije u sustavu. Umjesto toga, moguće je izraditi skup sekvencijskih dijagrama u kojem će se svaki dijagram usmjeriti na pojedini dio interakcije i detaljno je opisati, a dijagrami se po potrebi mogu referencirati jedan na drugog korištenjem **fragmenta interakcije** (engl. *interaction use*) **ref**. Jedan takav primjer prikazan je na slici 4.17, gdje se želi na visokoj razini apstrakcije opisati tijek kupovine u e-trgovini pa se izostavljaju detalji plaćanja, uz referencu na dodatni dijagram na slici 4.18, na kojem je proces plaćanja prikazan detaljnije.



Slika 4.17: Sekvencijski dijagram procesa kupovine u e-trgovini



Slika 4.18: Sekvencijski dijagram procesa plaćanja u e-trgovini

4.3 Primjena

Sekvencijski dijagrami korisni su u više faza procesa razvoja programske potpore zato što pružaju uvid u dinamičko ponašanje sustava i pomažu u komunikaciji, razumijevanju i validaciji. Najčešće se koriste u fazi analize zahtjeva za dokumentiranje i pojašnjavanja zahtjeva sustava. Preciznije, sekvencijski dijagrami koriste se za detaljniju razradu pojedinih obrazaca uporabe u vidu modeliranja interakcije između objekata i tijeka razmjene poruka u sustavu.

Tako razrađeni modeli koriste se kao podloga za razradu ostalih dijagrama i modela u idućim fazama procesa razvoja. Na primjer, u fazi oblikovanja sustava, sekvencijski dijagrami pomažu u utvrđivanju razreda, objekata i metoda potrebnih za implementaciju željene funkcionalnosti. Također, mogu se koristiti za modeliranje interakcija između komponenti tako što pomažu arhitektima programske potpore u planiranju opće arhitekture i organizacije sustava. Općenito, sekvencijski dijagrami pružaju dublji uvid u interakcije između objekata tako što otkrivaju njihove međuovisnosti te omogućuju prepoznavanje nesuglasica ili praznina u oblikovanju i olakšavaju rješavanje problema prije implementacije.

U fazi validacije i verifikacije sustava sekvencijski dijagrami se s obrascima uporabe koriste za osmišljavanje i razvoj ispitnih slučajeva.Tako olakšavaju provjeru ispravnosti komponenata i njihovih interakcija te toga da sustav ispunjava željene zahtjeve i ponaša se na očekivani način.

Općenito, sekvencijski dijagrami izvrstan su komunikacijski alat koji, ilustriranjem ponašanja sustava u vizualnom formatu, olakšava komunikaciju i suradnju među članovima tima, dionicima i klijentima. Pomažu osigurati zajedničko razumijevanje zahtjeva sustava, oblikovanja i očekivanog ponašanja te smanjuju rizik od nesporazuma ili nerazumijevanja i mogu se koristiti kao referenca za održavanje tijekom cijelog životnog ciklusa razvoja programske potpore. U tablici 4.1 nalazi se sažet i sistematiziran prikaz primjene sekvencijskih dijagrama u aktivnostima programskog inženjerstva.

Tablica 4.1: Primjena sekvencijskih dijagrama za vrijeme različitih aktivnosti programskog inženjerstva

Aktivnost	**Primjena**
Specifikacija programske potpore	Razrada tijeka obrazaca uporabe – modeliranje interakcije.
Analiza i oblikovanje	Uvid u interakcije između komponenti sustava i otkrivanje njihove međuovisnosti.
Implementacija	Provjera usklađenosti trenutnog stanja sa zadanim.
Ispitivanje	Osmišljavanje i razvoj ispitnih slučajeva.
Evolucija	Dokumentacija i komunikacija s dionicima.

5. UML dijagrami razreda

Dijagrami razreda vrijedan su alat u programskom inženjerstvu jer pružaju pregled strukture objektno orijentiranih sustava na visokoj razini, olakšavaju komunikaciju među dionicima, pomažu u oblikovanju i implementaciji programske podrške te podržavaju generiranje koda na temelju modela. Služe kao plan za razvojne inženjere da bi razumjeli i izgradili programske sustave na temelju utvrđenih razreda i njihovih veza.

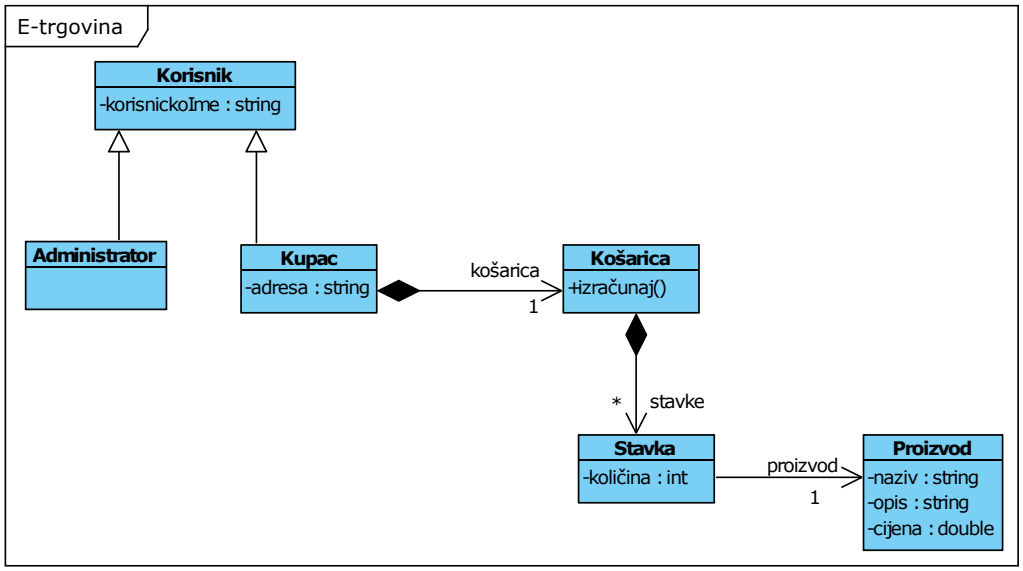
Zato su UML dijagrami razreda jedan od najčešće korištenih dijelova norme UML-a. Oni prikazuju razrede, njihove atribute i operacije te odnose među razredima poput nasljeđivanja, pridruživanja i ovisnosti. Pomažu u razumijevanju organizacije i hijerarhije razreda u sustavu te pružaju uvid u podatke i funkcionalnosti koje svaki razred posjeduje. Također, opisuju kako se razredi međusobno povezuju i dijele informacije. Budući da pružaju normirani način modeliranja i vizualizacije statičke strukture objektno orijentiranih sustava, koriste se za dokumentaciju sustava u svim fazama procesa razvoja: od oblikovanja do evolucije.

5.1 Definicija i osnovni elementi

UML dijagrami razreda (engl. *class diagrams*) statički su, strukturni UML dijagrami koji se koriste za vizualizaciju strukture objektno orijentiranih sustava. Prikazuju razrede, njihove atribute, operacije te odnose među razredima. Dijagrami razreda omogućuju razvojnim inženjerima i ostalim dionicima jasan uvid u organizaciju i interakciju razreda u sustavu, čime olakšavaju razumijevanje, planiranje i implementaciju programskog sustava. Osnovni elementi dijagrama razreda su:

- **Razred** (engl. *class*) – temeljni građevni blok dijagrama razreda koji predstavlja apstraktan koncept ili entitet u sustavu koji ima određene atribute i operacije. Razred se na dijagramu prikazuje kao pravokutnik podijeljen na tri dijela: naziv razreda, lista atributa i lista operacija.
- **Veze** (engl. *relationships*) – prikazuju odnose i interakcije između razreda u sustavu. Opisuju kako se razredi povezuju, dijele informacije i surađuju u ostvarivanju funkcionalnosti sustava. Osnovne vrste veza koje se često koriste u dijagramima razreda su pridruživanje, agregacija, kompozicija, generalizacija i ovisnost. Svaka se vrsta veze prikazuje različitom vrstom strelice, o čemu će biti više riječi u nastavku.

Primjer dijagrama razreda osnovnih elemenata iz domene e-trgovine prikazan je na slici 5.1.



Slika 5.1: Primjer dijagrama razreda

5.1.1 Razredi

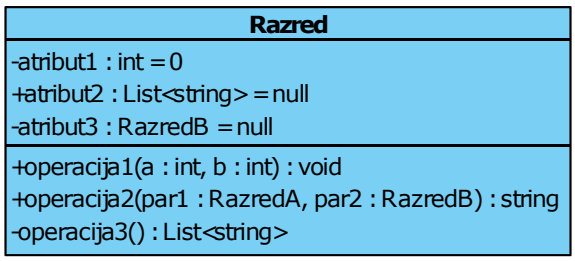
Razred je osnovna građevna jedinica u dijagramu razreda te predstavlja apstraktnu strukturu i karakteristike objekta u objektno orijentiranom programiranju. Smatra se predloškom na temelju kojeg se stvaraju primjerci (instance) pojedinačnih objekata. Razred opisuje svojstva, operacije i ponašanje objekta te definira njegove atribute i operacije. Na dijagramu razreda, razred se prikazuje kao pravokutnik podijeljen na tri dijela:

- **Naziv razreda** – ime razreda koje ga jednoznačno određuje. Naziv razreda smješta se u gornji dio pravokutnika i jedini je obvezan sadržaj svakog razreda.
- **Atributi** – svojstva koje razred posjeduje, tj. opisuju stanje ili karakteristike instance razreda (objekta) i navode se ispod naziva razreda.
- **Operacije** – utvrđuju operacije ili funkcionalnosti koje razred može izvršavati. Operacije se prikazuju u donjem dijelu pravokutnika.

U nastavku je detaljnije objašnjen način utvrđivanja atributa i operacija na dijagramu razreda.

5.1.1.1 Atributi

Za svaki atribut može se odrediti naziv, razina vidljivosti i tip podatka. Atributi mogu biti primitivnog tipa (bool, int, float itd.) ili tipa nekog drugog razreda. Sve osim naziva atributa je opcionalno,



Slika 5.2: Primjer razreda

a format navođenja atributa je sljedeći:

[oznaka_vidljivosti] naziv_atributa: [tip_podatka]

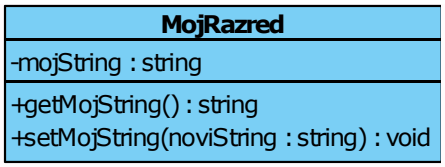
Na dijagramu razreda atributi mogu imati različite **razine vidljivosti**, koje određuju dostupnost atributa iz drugih dijelova sustava (iz drugih razreda). Razina vidljivosti označava se odgovarajućim simbolom na dijagramu (npr. +) koji odgovara ključnoj riječi u programskom jeziku (npr. *public*). Svaka razina vidljivosti ima svoju ulogu u upravljanju pristupom i zaštitom podataka. Odabirom pravilne razine vidljivosti za atribute može se osigurati pravilno upravljanje podacima i zaštita od neovlaštenog pristupa. Razine vidljivosti atributa su:

- **public (+)** – javna vidljivost atributa označava da je taj atribut dostupan izvan razreda u kojem je definiran, tj. može mu se pristupiti i mijenjati njegova vrijednost iz drugih razreda ili dijelova sustava[1].
- **private (-)** – privatna vidljivost atributa označava da je dostupan isključivo unutar razreda u kojem je definiran.
- **protected (#)** – atributi sa zaštićenom vidljivošću dostupni su unutar razreda, kao i izvedenih razreda (podrazreda) koji nasljeđuju taj razred. Ova razina vidljivosti omogućuje nasljeđivanje atributa i pristup njima u hijerarhijskoj strukturi razreda.
- **package/default (∼)** – paketna vidljivost (zadana vidljivost) označava da je atribut dostupan samo unutar istog paketa (ili modula) u kojem je definiran razred. Ako razredi pripadaju istom paketu, mogu pristupiti atributima s paketnom vidljivošću. Ponekad se simbol za paketnu vidljivost izostavlja, tj. ta se vidljivost podrazumijeva.

Primjer 5.1 — Enkapsulacija. Razred MojRazred ima atribut mojString, koji smije sadržavati isključivo alfanumeričke znakove. Potrebno je osigurati da se vrijednost atributa može dohvatiti i postaviti iz bilo kojeg drugog razreda, ali da atribut nikada ne sadržava neke druge znakove osim alfanumeričkih.

Rješenje je prikazano na slici 5.3.

[1] Osim ako su prisutni modifikatori **const** ili **final**, koji ograničavaju mogućnost promjene vrijednosti.



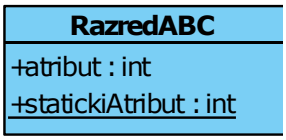

```
public class MojRazred {
    private String mojString;

    public String getMojString() {
        return mojString;
    }

    public void setMojString(String noviString) {
        if (noviString.matches("^[a-zA-Z0-9]+$")) {
            mojString = noviString;
        } else {
            System.out.println("Pogresan format!);
        }
    }
}
```

Slika 5.3: Primjer enkapsulacije korištenjem privatnog atributa

Komentar:

Atribut mojString može postati dohvatljiv iz bilo kojeg razreda samo na dva načina: postavljanjem javne (*public*) razine vidljivosti atributa ili definiranjem javnih metoda za postavljanje (tzv. *setteri*, od engl. *set*) i dohvaćanje podataka (tzv. *getteri*, od engl. *get*) za privatni atribut. Glavni je nedostatak javne vidljivosti atributa nemogućnost postavljanja bilo kakvih ograničenja na njegovu vrijednost. Suprotno tome, definiranjem javnih *gettera* i *settera* za privatni atribut moguće je zadati dodatna ograničenja i provjere za taj atribut. Dodatna je prednost takvog pristupa što korisnik ne mora znati detalje o ograničenjima vrijednosti atributa. Pristup u kojem se od korisnika skrivaju detalji implementacije naziva se **enkapsulacija** i važan je aspekt objektno orijentirane paradigme.

Posebna su vrsta atributa **statički atributi**, poznati još i pod nazivom „varijable razreda", koji su definirani na razini razreda, a ne objekta, te stoga imaju istu vrijednost za sve instance razreda. Njihov je naziv na dijagramu tipično podcrtan da bi se istaknuli u odnosu na ostale (nestatičke) atribute, a u kodu im se pristupa putem naziva razreda. Primjer je prikazan na slici 5.4.




```
public class RazredABC {
    public int atribut;
    public static int statickiAtribut;
}

public class Main {

    public static void main(String[] args) {
        // Pristupanje javnom atributu "atribut"
        ABC objekt = new RazredABC();
        objekt.Atribut = 10;
        System.out.println("'Atribut': " + objekt.Atribut);

        // Pristupanje javnom stati kom atributu "statickiAtribut"
        ABC.StatickiAtribut = 20;
        System.out.println("'StatickiAtribut': " + RazredABC.
            StatickiAtribut);
    }
}
```

Slika 5.4: Primjer definiranja i pristupanja statičkom atributu

5.1.1.2 Odgovornosti, operacije i metode

U kontekstu dijagrama razreda, pojmovi odgovornost (engl. *responsibility*), operacija (engl. *operation*) i metoda (engl. *method*) odnose se na različite aspekte razreda i njegovo ponašanje.

Odgovornost je zadatak, ponašanje, obveza ili uloga koju ima razred ili objekt unutar sustava. Odgovornosti mogu uključivati izvođenje određenih radnji, upravljanje podacima ili pružanje usluga drugim razredima. Odgovornosti se obično opisuju korištenjem opisnih fraza ili fraza s glagolima. Dobra praksa oblikovanja razreda jest da svaki razred ima samo jednu odgovornost.

Operacija je određena akcija ili ponašanje koje razred može obavljati. Jedna odgovornost najčešće se realizira s pomoću nekoliko operacija. Svaka operacija definirana je svojim nazivom, oznakom vidljivosti, popisom parametara i povratnim tipom u obliku:

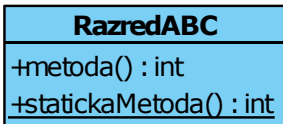
[oznaka_vidljivosti] naziv_operacije([parametar1, parametar2]): [povratni_tip]

Kao i kod atributa, operacije mogu imati različite razine vidljivosti, koje određuju njihovu dostupnost iz drugih dijelova sustava (iz drugih razreda). Primjenjuju se iste razine (oznake) vidljivosti kao i kod atributa, s istim značenjem. Parametri i povratni tip operacije mogu se izostaviti u ranim fazama oblikovanja, a poslije trebaju odgovarati točno onome što piše u izvornom kodu programa.

Metoda je konkretna implementacija operacije u nekom objektno orijentiranom programskom jeziku. Riječ je o stvarnom programskom kodu koji se prevodi i izvršava pri pozivanju operacije. U načelu se jedna operacija implementira kao jedna metoda, osim kada je riječ o hijerarhiji razreda kada podrazredi mogu nadjačati (engl. *overriding*) metode iz nadrazreda pa se tada de facto jedna operacija implementira u više metoda.

Na dijagramu razreda dodatno se označavaju dvije posebne vrste operacija (metoda): statičke i apstraktne. **Statičke operacije** dostupne su na razini razreda (pozivaju se putem imena razreda), po čemu se razlikuju od običnih operacija koje su dostupne za instance određenog razreda (pozivaju se

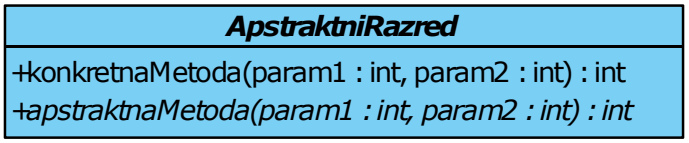
putem naziva instance). Statičke se operacije na dijagramu pišu podcrtano, kao što je prikazano na slici 5.5.




```
public class RazredABC {
    public int metoda(){
    return 10;
    }
    public static int statickaMetoda(){
    return 20;
    }
}

public class Main {
    public static void main(String[] args) {
        RazredABC objekt = new RazredABC();
        int rezultatMetode = objekt.metoda();
        System.out.println("Rezultat metode: " + rezultatMetode);

        int rezultatStatickeMetode = RazredABC.statickaMetoda();
        System.out.println("Rezultat stati ke metode: " +
            rezultatStatickeMetode);
    }
}
```

Slika 5.5: Primjer definiranja i pristupanja statičkoj metodi

Apstraktne operacije su operacije koje nemaju konkretnu implementaciju i te su na dijagramu razreda najčešće napisane kurzivom kao što je prikazano na slici 5.6 (iako prema UML-u 2.5. to više nije obvezno). Koriste se za definiranje zajedničkog sučelja ili ponašanja koje moraju imati razredi koji nasljeđuju apstraktan razred ili implementiraju sučelje. Razredi koji sadržavaju barem jednu apstraktnu operaciju (metodu) također su apstraktni, tj. njihovi se objekti ne mogu se instancirati. Više riječi o primjeni apstraktnih razreda u oblikovanju bit će u potpoglavlju o odnosu nasljeđivanja između razreda.



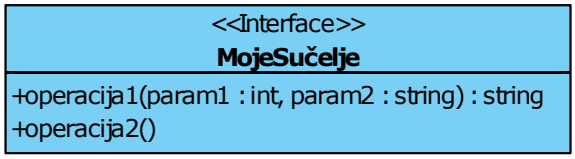

```
public abstract class ApstraktniRazred {
    public int konkretnaMetoda(int param1, int param2) {
        return param1 + param2;
    }

    public abstract int apstraktnaMetoda(int param1, int param2);
}
```

Slika 5.6: Primjer definiranja apstraktne metode

5.1.1.3 Sučelje

U objektno orijentiranoj programskoj paradigmi sučelje je koncept srodan razredu. Preciznije, sučelje specificira ugovor, tj. ponašanje koje razred treba pokazivati, ali ne definira način na koji se to ponašanje implementira.

Sučelje čini skup apstraktnih operacija koje se u programskom kodu zapisuju kao deklaracije metoda bez implementacije, a koje neki razred mora realizirati (implementirati). Jedan razred može implementirati više od jednog sučelja, a u nastavku će biti više riječi o implementaciji sučelja kroz razrede.

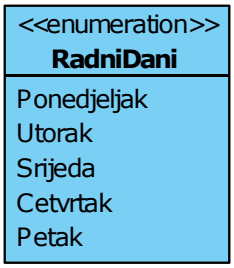


Slika 5.7: Primjer sučelja na UML dijagramu razreda

5.1.1.4 Enumeracija

Enumeracija (engl. *enumeration*) poseban je tip podataka u programskom jeziku koji omogućuje definiranje i korištenje fiksnog skupa diskretnih vrijednosti. U kontekstu dijagrama razreda, enumeracije se prikazuju kao posebni elementi, sa stereotipom «enumeration» ili «enum» i sadržavaju popis mogućih vrijednosti za određeni atribut ili varijablu.

Enumeracije su korisne u dijagramima razreda jer omogućuju jasno definiranje skupa mogućih vrijednosti koje određena varijabla ili atribut može imati. Tako dijagram razreda postaje jasniji i precizniji u prikazu ograničenja vrijednosti za određene elemente.



Slika 5.8: Primjer enumeracije na UML dijagramu razreda

5.1.2 Veze

Veze na dijagramu razreda igraju ključnu ulogu u modeliranju odnosa između različitih razreda u programskom sustavu i pružaju važne informacije o strukturi i dinamičkom ponašanju sustava. Razumijevanje i pravilno korištenje veza na dijagramu razreda ključno je za izgradnju dobro organizirane, modularne i održive programske potpore. U ovom se potpoglavlju nalazi pregled pet temeljnih vrsta veza: pridruživanja, agregacije, generalizacije, realizacije i ovisnosti, s primjerima korištenja u modeliranju objektno orijentiranih sustava.

5.1.2.1 Pridruživanje

Pridruživanje (engl. *association*) odnos je između dvaju razreda u kojem objekti jednog razreda mogu pristupati objektima drugog razreda, tj. njihovim javnim atributima i metodama. Preciznije,

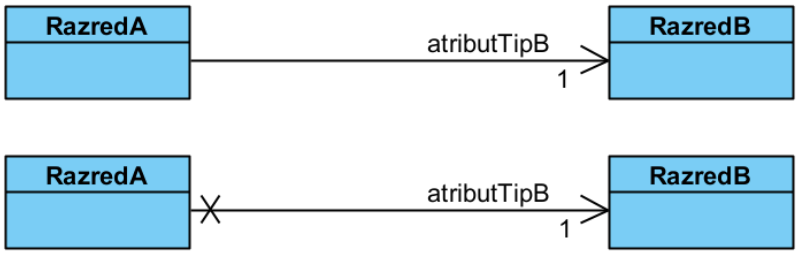
pridruživanje je odnos u kojem objekti jednog razreda imaju referencu na objekte drugog razreda. Pridruživanje se prikazuje ravnom linijom koja povezuje dva razreda, a može biti usmjereno ili neusmjereno, ovisno o potrebi.

S obzirom na to da objekt nekog razreda može imati referencu na različiti broj objekata drugog razreda, na vezi pridruživanja najčešće se pišu oznake **višestrukosti** (engl. *multiplicity*). Vrijede iste oznake kao i kod dijagrama obrazaca uporabe:

- **n** – objekt sadržava točno *n* referenci na objekte drugog razreda, npr. 1 ili 100
- **1..n** – objekt može sadržavati bilo koji broj referenci na objekte drugog razreda u rasponu od 1 do *n* (umjesto 1 može se zadati i neki drugi prirodni broj)
- **0..*** ili * – objekt može imati neograničen broj referenci na objekte drugog razreda, ali ne mora imati ni jednu.

Ako se višestrukost ne napiše, podrazumijeva se da postoji samo jedna referenca na objekt drugog razreda.

Usmjerena veza pridruživanja prikazuje se strelicom koja pokazuje prema razredu kojem se pristupa. Slika 5.9 ilustrira primjer u kojem RazredA ima atribut naziva „atributTipB”, koji sadržava točno jedan objekt razreda RazredB. Naziv atributa na koji se odnosi pridruživanje najčešće se piše na samoj vezi pridruživanja da bi ga se jasno istaknulo u odnosu na ostale atribute. Nadalje, na ovoj su slici prikazane dvije moguće notacije usmjerene veze pridruživanja. Prva notacija češća je u praksi (osobito u fazi modeliranja) iako nije potpuno točna prema normi UML-a, jer ne specificira ima li RazredB kakvo pridruživanje na RazredA (koje nije eksplicitno navedeno). Točniji je prikaz na kojem oznaka „x” eksplicitno kaže da ne postoji nikakvo pridruživanje u suprotnom smjeru, tj. RazredB nema nikakvog saznanja o RazredA. Konačno, na slici je dan i primjer izvornog koda u programskom jeziku Java koji bi odgovarao modelu, uz napomenu da je potrebno još programski osigurati na odgovarajući način (putem konstruktora i *settera*) provjeru da je u atributu uvijek sadržan točno jedan objekt tipa RazredB (model specificira višestrukost od „1”).

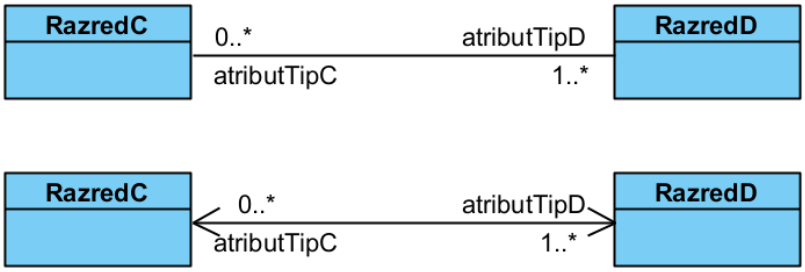



```
public class RazredA {
    public RazredB atributTipB;
}
```

Slika 5.9: Prikaz usmjerene veze asocijacije na dva načina s primjerom odgovarajućeg izvornog koda

Ako postoji obostrano pridruživanje, moguće je prikazati strelice na oba kraja ili ih u potpunosti izostaviti (češće u praksi). Primjer dvaju načina zapisa prikazan je na slici 5.10. U ovom primjeru objekt razreda RazredC ima referencu na jedan ili više objekata tipa RazredD (sadržanu u atributu razreda RazredC pod nazivom „atributTipD”), a objekt razreda RazredD ima referencu na

neograničen broj objekata tipa RazredC (sadržanu u atributu „atributTipC"). Na slici je prikazan i primjer izvornog koda u Javi koji odgovara zadanom modelu. U ovom kodu korištena je tipizirana lista (List<RazredD>) kao agregator više objekata u atributu, ali moguće je koristiti i neke druge slične tipove kao što su skupovi (Set) i sl. Također, ovisno o konkretnom programskom jeziku, potrebno je na odgovarajući način osigurati da kada je riječ o atributu „atributTipD" lista nikada nije prazna (model specificira višestrukost od „1..*").




```
public class RazredC {
    public List<RazredD> atributTipD;
}

public class RazredD {
    public List<RazredC> atributTipC;
}
```

Slika 5.10: Prikaz dvosmjerne veze asocijacije na dva načina s primjerom odgovarajućeg izvornog koda

Primjer 5.2 — Pridruženi razred. Potrebno je modelirati sljedeći zahtjev za rezervacijama sjedala u programskom sustavu aviokompanije. Kompanija želi pratiti rezervacije sjedala za svaki let tako da se točno vidi koji je putnik rezervirao koje sjedalo na kojem letu (svako sjedalo ima svoju oznaku, npr. 5A). Također, potrebno je voditi računa o tome da za svakog putnika i svaki let postoje dodatni podaci u sustavu, kao ime i prezime putnika, broj leta itd. (nije potrebno eksplicitno navoditi na dijagramu).

Rješenje je prikazano na slici 5.11.


```
public class Putnik {

    public List<Rezervacija> rezervacije;

}

public class Let {

    public List<Rezervacija> rezervacije;

}

public class Rezervacija {

    public Putnik putnik;
    public Let let;
    public int sjedalo;

}
```

Slika 5.11: Primjer Rezervacije kao pridruženog razreda razredima Putnik i Let

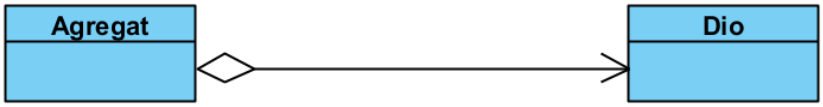
Komentar:

Iz navedenog opisa jasno je da moraju postojati razredi Putnik i Let jer modeliraju entitete za koje treba pamtiti određene podatke. Nadalje, postojanje podataka o rezervaciji leta ukazuje na potrebu za stvaranjem poveznice između razreda Putnik i Let. Međutim, zahtjev za praćenjem točne oznake sjedala koje je rezervirano znači da se razredi Putnik i Let ne mogu izravno povezati odnosom pridruživanja (nema se gdje zapisati oznaka sjedala). Iako je ovaj slučaj moguće riješiti na nekoliko načina (npr. korištenjem neke vrste liste u kojoj su elementi parovi – Map, Tupple, Dictionary itd.), najelegantnije bi bilo korištenjem oblikovnog obrasca **pridruženi razred** (engl. *association class*). Riječ je o dodatnom razredu povezanom s oba razreda koji su u odnosu pridruživanja, a koji omogućuje dodavanje specifičnih informacija (atribut) i ponašanja (operacija) koja se odnose na tu vezu, što rezultira jasnijim i detaljnijim prikazom odnosa između razreda. ■

5.1.2.2 Agregacija i kompozicija

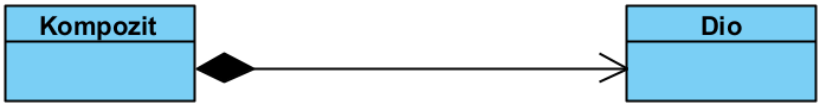
Agregacija i kompozicija dvije su specijalizirane vrste pridruživanja kojima se određuje odnos vlasništva ili sadržaja između razreda. Glavna razlika između agregacije i kompozicije leži u jačini veze i ovisnosti o životnom ciklusu glavnog razreda. Za obje vrste veza uobičajeno je korištenje oznaka višestrukosti, kao i kod običnog pridruživanja.

Agregacija (engl. *aggregation*) predstavlja odnos „cjelina i dijelovi” („HAS-A”) između razreda, u kojem jedan objekt jednog razreda (agregat) sadržava jedan objekt drugog razreda ili više njih. Odnos agregacije dopušta zajedničko vlasništvo (engl. *shared ownership*), tj. objekt jednog razreda može biti dio dvaju različitih agregata. Također, ne postoje nikakva ograničenja na životni vijek razreda koji je dio agregata, što znači da ako dođe do brisanja objekta razreda koji je agregat, objekti koje on agregira nastavljaju postojati. Agregacija se prikazuje linijom s otvorenim rombom na strani prema cjelini, slika 5.12. Agregacija može biti usmjerena ili neusmjerena.



Slika 5.12: Agregacija

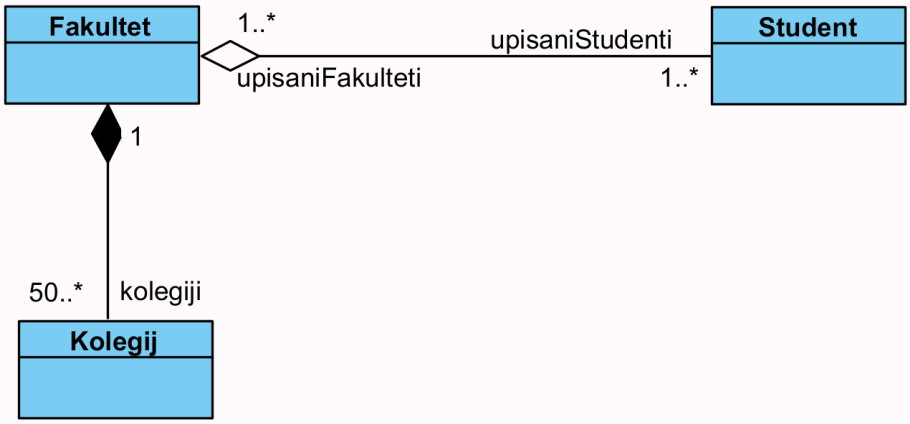
Kompozicija (engl. *composition*) jači je tip pridruživanja od agregacije. U ovom odnosu, dijelovi ne mogu postojati samostalno izvan cjeline. Preciznije, objekt može biti dio samo jednog kompozita te kada se briše kompozit, nužno se brišu i svi njegovi dijelovi. Kompozicija se prikazuje linijom s ispunjenim rombom na strani prema cjelini, slika 5.13.



Slika 5.13: Kompozicija

Primjer 5.3 — Agregacija i kompozicija. Potrebno je izraditi model prema sljedećem opisu. Na fakultet se upisuju studenti i održavaju se kolegiji. Student može upisati više od jednog fakulteta, ali mora biti upisan na barem jedan fakultet da bi imao status studenta. Svaki kolegij pripada isključivo jednom fakultetu i na svakom se fakultetu mora držati minimalno 50 kolegija.

Rješenje je prikazano na slici 5.14.



Slika 5.14: Primjer korištenja veza agregacije i kompozicije za modeliranje odnosa fakulteta, kolegija i studenata

Komentar:

S obzirom na to da student može upisati više od jednog fakulteta, između razreda Fakultet i Student jedine su moguće veze pridruživanje (slabija) ili agregacija (jača). Kompozicija nije moguća zbog dvaju razloga: (1) kompozicija ne dopušta zajedničko vlasništvo (upis više fakulteta) i (2) brisanjem kompozita nužno se brišu i svi dijelovi (student koji je upisan na više fakulteta bi prestankom postojanja jednog izgubio svoj status). S druge strane, u odnosu između razreda Fakultet i Kolegij treba koristiti kompoziciju da bi se istaknula pripadnost

pojedinog kolegija konkretnom fakultetu (kolegij pripada isključivo jednom fakultetu i nema smisla samostalno bez fakulteta).

U nastavku je i primjer izvornog koda u programskom jeziku Java koji prikazuje implementaciju razreda Fakultet. Moguće je primijetiti da se popis studenata stvara izvan razreda Fakultet i prenosi mu se kao parametar njegova konstruktora, a za stvaranje popisa kolegija zadužen je sam razred Fakultet. Tako se osigurava poštovanje pravila vlasništva kod agregacije i kompozicije. Za popis studenata odgovoran je neki drugi razred te se brisanjem objekta razreda Fakultet neće obrisati i popis studenata. Međutim, kolegiji su vezani za fakultet, pa je bitno osigurati da jedino Fakultet ima vlasništvo nad popisom kolegija što se postiže kombinacijom stvaranja popisa kolegija unutar konstruktora i pohranom tog popisa u privatnom atributu (za koji postoje javne metode *get* i *set*).

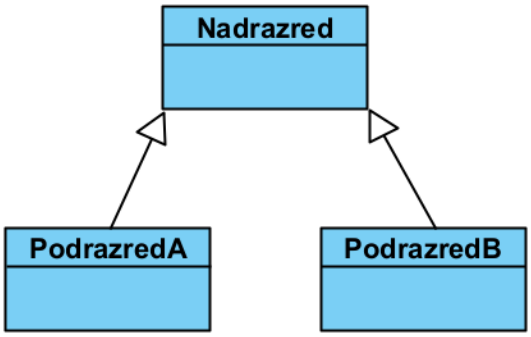
```
public class Fakultet {

    private List<Student> upisaniStudenti;
    private List<Kolegij> kolegiji;

    public Fakultet(List<Student> studenti){
    this.upisaniStudenti = studenti;
    this.kolegiji = stvoriKolegije();
    }

    private List<Kolegij> stvoriKolegije(){
    //formiranje popisa od najmanje 50 kolegija...
    }

    //Javni getteri i setteri za privatne atribute...

}
```

Slika 5.15: Izvorni kôd u programskom jeziku Java za razred Fakultet

5.1.2.3 Generalizacija

Generalizacija (engl. *generalization*) označava hijerarhijski odnos između razreda, u kojem podrazred nasljeđuje atribute i operacije (metode) od nadrazreda (engl. *base class*), ali može dodati i vlastite atribute i operacije te modificirati naslijeđene. Budući da podrazred nasljeđuje svojstva i ponašanje nadrazreda, smatra se podvrstom nadrazreda (odnos „IS-A”, naspram odnosa „HAS-A” kod agregacije). Iako nije eksplicitno specificirano u važećoj normi UML 2.5.1, odnos generalizacije u svim temeljnim aspektima potpuno odgovara konceptu **nasljeđivanja** (engl. *inheritance*) u objektno orijentiranoj programskoj paradigmi.

Odnos generalizacije, tj. nasljeđivanja, omogućuje podrazredu da ponovno koristi i proširi značajke nadrazreda, čime potiče ponovnu uporabu koda, modularnost i apstrakciju. Na UML dijagramima razreda, generalizacija se prikazuje strelicom s vrhom u obliku trokuta koji pokazuje od podrazreda prema nadrazredu. Strelica označuje smjer nasljedne veze, a trokut predstavlja specijalizaciju podrazreda iz nadrazreda. Primjer je prikazan na slici 5.16.

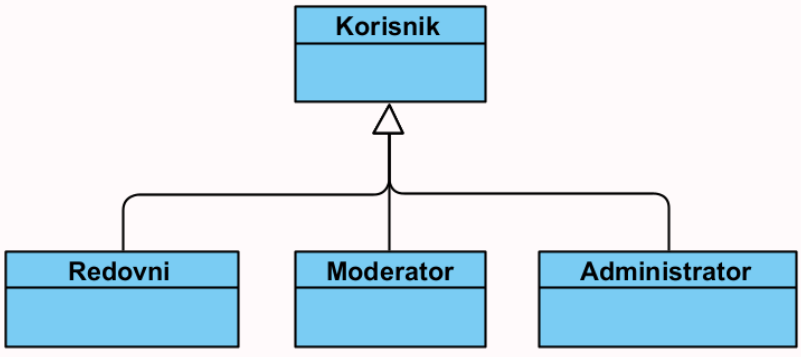


Slika 5.16: Generalizacija

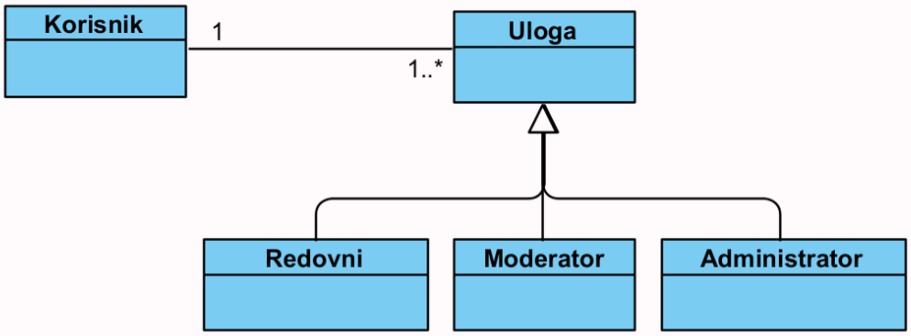
Svojstva generalizacije kao što su apstrakcija, polimorfizam i hijerarhijska organizacija pružaju nekoliko prednosti u razvoju programske potpore, uključujući modularnost i ponovnu uporabu koda. Na primjer, stvaranje apstraktnih razreda omogućuje definiranje zajedničkih značajki i ponašanja koja se mogu ponovno koristiti u više podrazreda, što smanjuje dupliciranje koda. Također, apstrakcija i hijerarhijska struktura olakšavaju organizaciju i razumijevanje odnosa između razreda. Nadalje, nasljeđivanje omogućuje specijalizaciju podrazreda i polimorfno ponašanje, u kojem se postojeće značajke nadrazreda mogu proširiti ili prilagoditi specifičnim potrebama podrazreda. Tako je moguće izgraditi fleksibilnije sustave koje je u budućnosti lakše održavati i nadograđivati.

Primjer 5.4 — Modeliranje uloga. Potrebno je modelirati tri uloge koje korisnik može imati na nekom forumu: redovni korisnik, moderator i administrator. Pri tome razmotrite razliku u mogućem rješenju ako: A) korisnik nikada neće promijeniti ulogu i B) korisnik s vremenom može promijeniti ulogu (npr. napredovati u moderatora) ili imati više uloga istodobno.

Rješenja obje inačice prikazana su na slikama 5.17 i 5.18.



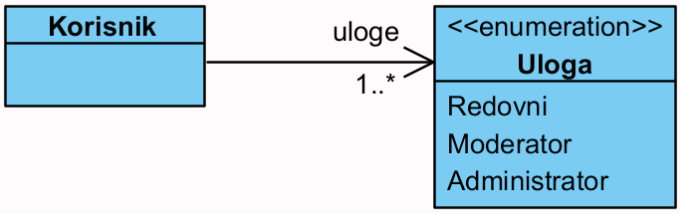
Slika 5.17: Rješenje inačice A – modeliranje uloga hijerarhijskom strukturom



Slika 5.18: Rješenje inačice B – modeliranje uloga obrascem Player-Role

Komentar:

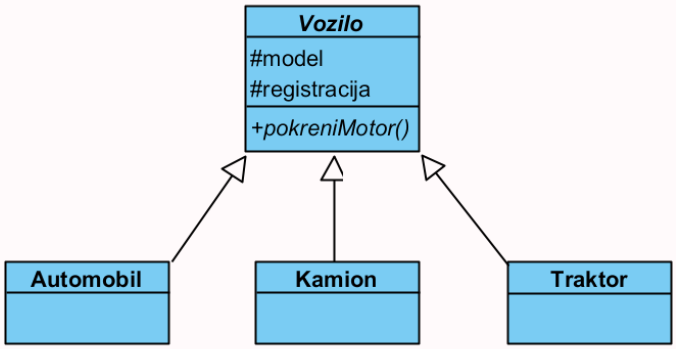
Ako se uloga može s vremenom promijeniti ili je istodobno moguće imati više uloga, rješenje u inačici A nije dobro jer pri nasljeđivanju nije moguće promijeniti razred niti istodobno naslijediti dva razreda. Rješenje prikazano na slici 5.18 implementacija je oblikovnog obrasca Player-Role, u kojem se uloga korisnika u sustavu razdvaja od entiteta korisnika. Nadalje, ako se pri oblikovanju uloga uoči da je potrebno isključivo pratiti informaciju o dopuštenim ulogama za nekog korisnika, ali ne postoje nikakva dodatna svojstva ili operacije vezane za pojedine uloge, onda nije nužno implementirati cijelu hijerarhijsku strukturu uloga, već je dovoljno definirati popis uloga kao enumeraciju povezanu s razredom Korisnik vezom pridruživanja. Primjer takvog rješenja prikazan je na slici 5.19.



Slika 5.19: Inačica C – modeliranje uloga enumeracijom

Primjer 5.5 — Liskovino načelo supstitucije (LSP). Potrebno je modelirati sljedeće razrede s odgovarajućim atributima i operacijama uz minimalno ponavljanje koda. U nekom voznom parku postoje automobili, kamioni i traktori. Za svako vozilo zadana je oznaka modela i registracija (u istom obliku za sve vrste vozila), ali svaka vrsta vozila treba na drukčiji način implementirati pokretanje motora.

Moguće rješenje prikazano je na slici 5.20.



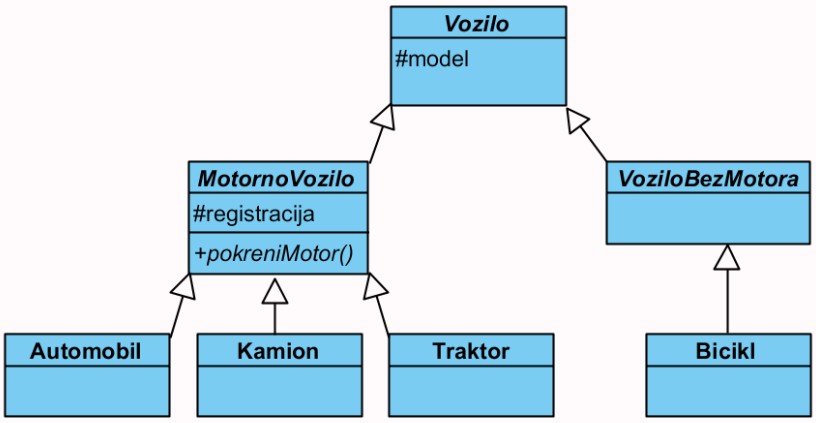
Slika 5.20: Modeliranje različitih vrsta vozila hijerarhijom razreda

Komentar:

U ovom je rješenju vidljivost atributa postavljena na „protected (#)" tako da su dostupni unutar hijerarhije i podrazumijevaju se javne metode *get* i *set* (implementirane u razredu Vozilo radi maksimalne ponovne iskoristivosti). Budući da svaka vrsta vozila pokreće motor na drukčiji način, metoda „pokreniMotor" definirana je u nadrazredu Vozilo kao apstraktna i svaki je podrazred mora implementirati na svoj način.

Proširenje opisa sustava: Naknadno su u sustav dodani i bicikli, ali oni nemaju registarsku oznaku ni motor. Potrebno je doraditi dijagram razreda.

Novo rješenje prikazano je na slici 5.21.



Slika 5.21: Hijerarhijska struktura razreda za prošireni opis vozila

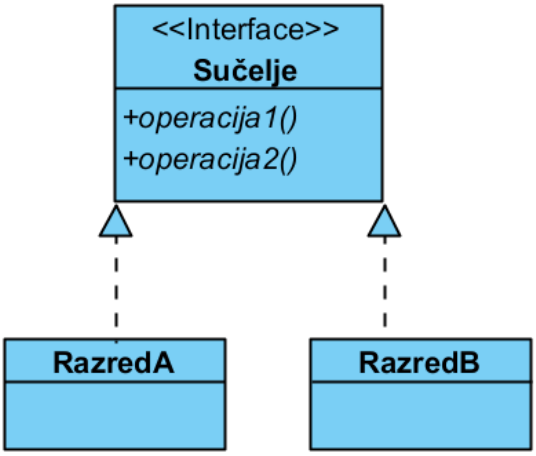
Komentar:

Kada je riječ o ovakvom proširenju, nikako nije dobro jednostavno dodati razred Bicikl koji nasljeđuje razred Vozilo u postojećem obliku jer bicikli nemaju registraciju i motor. Štoviše, potrebno je potpuno preraditi originalni nadrazred Vozilo da podrazredi ne bi kršili specifikaciju nadrazreda (poznato još i kao Liskovin princip supstitucije[a]).

[a] https://doi.org/10.1145/62139.62141

5.1.2.4 Ostvarivanje sučelja

Ostvarivanje sučelja (engl. *interface realization*) u kontekstu dijagrama razreda čini konkretizaciju i implementaciju operacija (metoda) definiranih u sučelju. Kada razred implementira sučelje, preuzima odgovornost za pružanje konkretne implementacije svih operacija koje su zadane u sučelju. Na dijagramu razreda, realizacija sučelja označava se isprekidanom strelicom s trokutastim vrhom koja povezuje razred koji ostvaruje sučelje i samo sučelje, slika 5.22.



Slika 5.22: Ostvarivanje sučelja

Realizacija sučelja koncept je vrlo sličan specijalizaciji apstraktnog razreda u konkretnom podrazredu, no postoji nekoliko temeljnih razlika. Apstraktni razred koristi se za opis nekog entiteta čiji će objekti (instance) imati svojstveno stanje opisano u sadržaju njihovih atributa. Nadalje, apstraktni razred može sadržavati i potpuno implementirane metode, dok sučelje sadržava isključivo deklaracije metoda bez implementacije. Treba napomenuti da neki moderni programski jezici omogućuju definiranje atributa i zadane (engl. *default*) implementacije metoda čime se sve više briše granica između sučelja i apstraktnih razreda. Ipak, sučelja se koriste za opis određenog ponašanja ili usluge koja ne mora nužno biti vezana za neki entitet i zbog toga je moguće da jedan razred implementira više sučelja. S druge strane, razred u suštini predstavlja programsku apstrakciju nekog entiteta (iz stvarnog ili virtualnog svijeta) te stoga većina programskih jezika dopušta nasljeđivanje isključivo jednog razreda.

Korištenje sučelja omogućuje modularnost i fleksibilnost u razvoju programske potpore konceptom polimorfizma. Objekti različitih razreda koji implementiraju isto sučelje mogu se tretirati na isti način što olakšava zamjenu jedne implementacije sučelja drugom bez učinka na druge dijelove sustava koji koriste to sučelje.

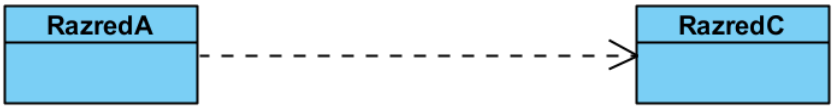
5.1.2.5 Ovisnost

U UML dijagramima razreda, ovisnost (engl. *dependency*) je odnos između dvaju razreda u kojem promjena u jednom razredu može utjecati na drugi razred. Riječ je o relativno labavu odnosu u usporedbi s drugim vezama, što znači da jedan razred na neki način ovisi o drugom razredu, ali ne postoji jaka strukturna ili ponašajna veza.

Ovisnost može nastati kada, na primjer, jedan razred koristi drugi razred kao tip lokalne varijable u metodi, tip parametra metode ili pak kao tip povratne vrijednosti metoda. Važno je istaknuti da je ovisnost najslabiji tip veze te ako istodobno postoji i neka jača veza između dvaju razreda (npr.

pridruživanje), ovisnost se ne prikazuje.

Ovisnosti se označavaju u UML dijagramima razreda korištenjem isprekidane strelice koja pokazuje od ovisnog razreda prema razredu o kojem ovisi, slika 5.23. Strelica označuje smjer ovisnosti. Obično se na strelici nalazi oznaka ili kratak opis koji pojašnjava prirodu ovisnosti, na primjer «use», «create» itd.

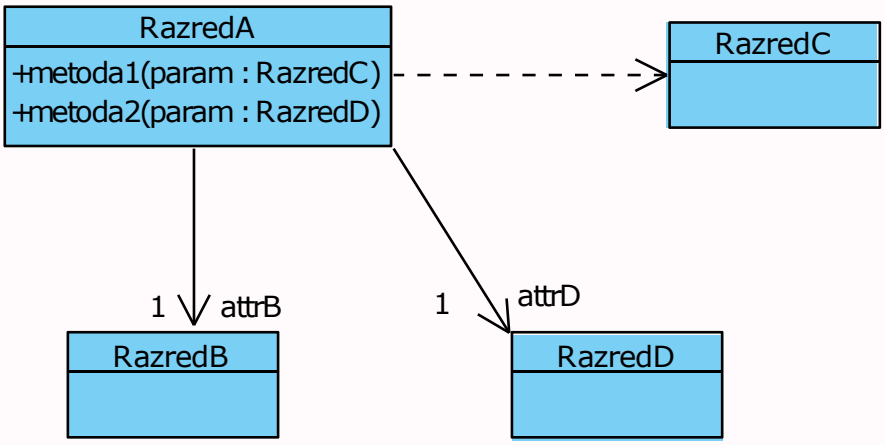


Slika 5.23: Ovisnost

Primjer 5.6 — Pridruživanje i ovisnost. Modelirajte dijagramom razreda sljedeći programski kôd:

```
public class RazredA{

    private RazredB attrB;
    private RazredD attrD;

    public void metoda1(RazredC param){...}
    public void metoda2(RazredD param){...}
}
```

Rješenje je prikazano na slici 5.24.



Slika 5.24: Modeliranje ovisnosti i pridruživanja

Komentar:

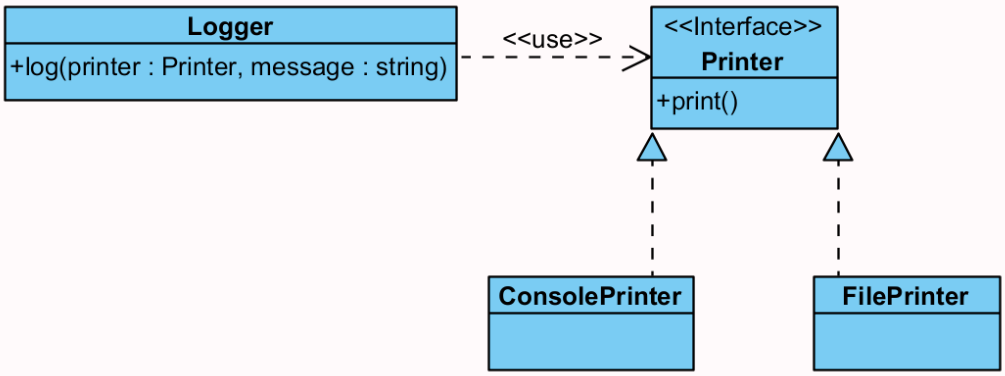
Ovisnost između razreda RazredA i RazredC nastaje zato što RazredA koristi RazredC kao tip lokalne varijable u metodi. Isto vrijedi i za odnos RazredA – RazredD, ali s obzirom na to da između razreda RazredA i RazredD postoji i odnos pridruživanja (atribut attrD iz RazredA je tipa RazredD), veza pridruživanja nadjačava vezu ovisnosti, pa se u tom slučaju ona ne prikazuje.

■

Ovisnost se vrlo često ne prikazuje u ranim fazama modeliranja kada još nisu poznati konkretni detalji implementacije razreda. No u kasnijim fazama, a osobito pri dokumentaciji gotovog proizvoda, prikaz ovisnosti važan je za razumijevanje učinka koji promjene u jednom razredu imaju na druge razrede. Ako razred o kojem neki drugi razred ovisi doživi promjenu, ovisni razred može trebati odgovarajuće izmjene. Ukupno gledano, ovisnosti pomažu u prikazu dinamičkih aspekata i veza između razreda u UML dijagramu razreda tako što pružaju uvid u interakcije i utjecaje među razredima u programskom sustavu.

Primjer 5.7 — Ubacivanje ovisnosti. Potrebno je modelirati razred Logger koji će zapisivati događaje u sustavu na odgovarajući izlaz. Za zapisivanje događaja primjenjuje se metoda *log*, a potrebno je osigurati i mogućnost dinamičke promjene načina ispisa (npr. na konzolu ili u datoteku).

Rješenje je prikazano na slici 5.25.



Slika 5.25: Primjer ubacivanja ovisnosti za bilježenje događaja u sustavu

Komentar:

U ovom je rješenju korišteno načelo **ubacivanja ovisnosti** (engl. *dependency injection*) kroz parametre metode *log*, koje u kombinaciji s vezanjem razreda Logger za sučelje Printer, umjesto za njegove konkretne implementacije u razredima ConsolePrinter i FilePrinter, omogućuje dinamičku promjenu ponašanja pri svakom pozivu metode *log*.

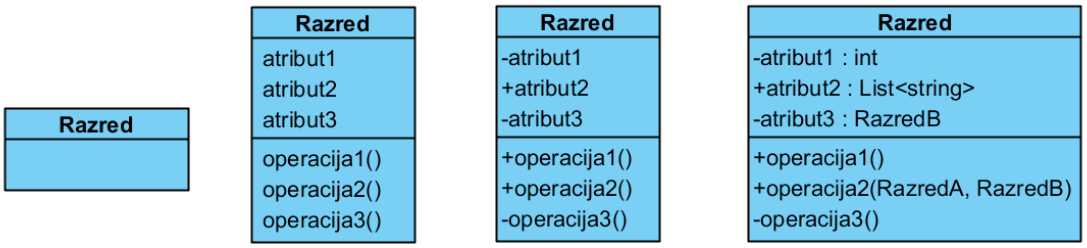
■

5.2 Postupak izrade dijagrama

Dijagrami razreda mogu se izrađivati u različitim fazama projekta i imati različitu namjenu. Na početku projekta potrebno je razraditi temelje arhitekture te tada dijagrami razreda pomažu u prepoznavanju ključnih entiteta, funkcionalnosti i njihovih međusobnih odnosa te služe kao alat za razumijevanje domene problema. Kasnije pak služe kao smjernica za implementaciju ili dokumentacija gotove programske potpore.

Ovisno o fazi u kojoj se izrađuje, razina apstrakcije na dijagramu razreda može varirati. Na višoj razini apstrakcije često se izostavljaju pojedini elementi dijagrama kao što su operacije, razine vidljivosti, povratni tipovi i sl. Na primjer, u ranim fazama razrade specifikacije programske potpore moguće je samo utvrditi nazive razreda, ili samo zadati nazive osnovnih atributa i metoda bez navođenja tipova, parametara, povratnih vrijednosti itd. To su tzv. „konceptualni modeli”,

čija je svrha razrada temeljne arhitekture u kratkom vremenu. U kasnijim fazama oblikovanja i implementacije moguće je nadopunjavati dijagram konkretnim detaljima, redom kako oni postaju poznati. Takvi dijagrami na kojima je dio atributa i metoda u potpunosti definiran, ali još nije sve fiksirano, često se nazivaju i specifikacijski dijagrami.



Slika 5.26: Model razreda na različitim razinama apstrakcije, od konceptualnog do specifikacijskog

Postupak izrade dijagrama razreda uvijek je potrebno prilagoditi potrebama i složenosti projekta. No, u većini slučajeva on obično uključuje sljedeće korake:

- **Utvrđivanje i analiza zahtjeva** – u prvom koraku potrebno je usmjeriti se na razumijevanje zahtjeva i funkcionalnosti sustava. Kao početna točka najbolje mogu poslužiti dijagrami obrazaca uporabe te popratni sekvencijski dijagrami i dijagrami aktivnosti (ako postoje).
- **Utvrđivanje razreda** – na temelju analize zahtjeva prepoznaju se temeljni razredi potrebni za modeliranje sustava. Oni bi trebali predstavljati entitete ili koncepte u stvarnom svijetu koji su bitni za funkcionalnost sustava.
- **Određivanje atributa** – za svaki razred potrebno je odrediti njegove atribute, odnosno podatke koje taj razred posjeduje. Minimalno je potrebno dati im smisleni naziv, a ako je moguće, definirati i tip podataka.
- **Određivanje odgovornosti i operacija** – nakon što su atributi određeni, potrebno je razmisliti o funkcionalnostima koje pruža svaki razred. Treba dodijeliti svaku odgovornost jednom razredu te na temelju toga odrediti operacije, odnosno metode. U ranim je fazama moguće izostaviti parametre koje prima metoda i njezin povratni tip.
- **Utvrđivanje odnosa** – potrebno je analizirati kako su razredi međusobno povezani i na temelju toga odrediti odnose između njih (pridruživanje, agregacija, nasljeđivanje, ostvarivanje sučelja).
- **Dodavanje detalja** – uključivanje dodatnih informacija u dijagram razreda: npr. stereotipi, ovisnosti i komentari čine dijagram jasnijim i razumljivijim.
- **Validacija i iteracija** – nakon izrade početnog dijagrama razreda, potrebno ga je pregledati da bi se provjerila njegova točnost i dosljednost sa specifikacijom zahtjeva. Po potrebi je moguće vratiti se na neki od prethodnih koraka i napraviti odgovarajuće korekcije i prilagodbe. Za izradu dobrog modela koji će poslužiti kao predložak za implementaciju najčešće je potrebno dorađivati i poboljšavati dijagram u više iteracija.

U konačnici, kada je programska potpora u potpunosti implementirana, dijagrami razreda generiraju se na temelju izvornog koda te sadržavaju potpuno utvrđene sve atribute i metode. Takvi

se dijagrami nazivaju još i implementacijski dijagrami te služe prvenstveno kao dokumentacija programske potpore. Budući da programska potpora najčešće sadržava stotine razreda, uputno je razdvojiti model na više dijagrama razreda koji će pokazivati pojedine dijelove sustava.

5.3 Primjena

Kao što je već spomenuto, dijagrami razreda imaju široku primjenu u procesu programskog inženjerstva i korisni su tijekom cijelog životnog ciklusa razvoja programske potpore.

Dijagrami razreda mogu se koristiti već od samog početka procesa razvoja programske potpore. U fazi analize zahtjeva služe kao alat za razumijevanje domene problema i preciziranje zahtjeva koje programski sustav treba zadovoljiti jer pomažu u prepoznavanju ključnih entiteta, funkcionalnosti i njihovih međusobnih odnosa.

Međutim, dijagrami razreda imaju glavnu ulogu u procesima oblikovanja i implementacije sustava, pod uvjetom da se implementacija radi u objektno orijentiranom programskom jeziku. Pri oblikovanju programske potpore omogućuju programerima da jasnije utvrde strukturu sustava, organiziraju razrede, atribute i metode te odrede njihove odnose. Pomažu i u vizualizaciji arhitekture sustava i omogućuju preciznije planiranje i implementaciju. Za vrijeme implementacije sustava dijagrami razreda programerima pružaju smjernice o tome kako implementirati razrede, metode i odnose određene u fazi oblikovanja. Pomažu u održavanju dosljednosti između specifikacije i stvarnog implementiranog koda te olakšavaju suradnju među programerima koji rade na različitim dijelovima sustava.

Dijagrami razreda razvijeni u procesu oblikovanja s implementacijskim dijagramima razreda pružaju temelje za daljnje ispitivanje i održavanje sustava. Daju jasnu sliku strukture programa i odnosa između komponenti što olakšava ispitivanje pojedinih funkcionalnosti i prepoznavanje potencijalnih problema ili pogrešaka. Također, dijagrami razreda pomažu pri održavanju i nadogradnji programske potpore jer olakšavaju razumijevanje strukture i učinaka promjena na druge dijelove sustava. U tablici 5.1 nalazi se sažet i sistematiziran prikaz primjene dijagrama razreda u aktivnostima programskog inženjerstva.

Tablica 5.1: Primjena dijagrama razreda za vrijeme različitih aktivnosti programskog inženjerstva

Aktivnost	**Primjena**
Specifikacija programske potpore	Analiza domene problema – utvrđivanje ključnih entiteta, funkcionalnosti i njihovih međusobnih odnosa.
Analiza i oblikovanje	Oblikovanje arhitekture sustava – organizacija razreda i određivanje njihovih atributa, metoda te međusobnih odnosa.
Implementacija	Smjernice programerima za implementaciju – održavanje dosljednosti između specifikacije i stvarnog implementiranog koda.
Ispitivanje	Razumijevanje strukture programa i odnosa između komponenti olakšava uočavanje potencijalnih problema ili pogrešaka.
Evolucija	Razumijevanje strukture i učinaka promjena na druge dijelove sustava.

6. UML dijagrami stanja

Dinamičko ponašanje sustava može se promatrati s aspekta komunikacije, aktivnosti i stanja. Sekvencijski dijagrami omogućuju modeliranje komunikacije i tijeka izvođenja, ali na njima nije moguće izravno prikazati stanje sustava ili dijelova sustava. U tu se svrhu upotrebljavaju UML dijagrami stanja koji omogućuju prikazivanje različitih stanja u kojima se sustav može nalaziti te nude vizualno sredstvo za razumijevanje dinamičkog ponašanja.

Dijagrami stanja jasno i strukturirano prikazuju kako sustav ili njegovi dijelovi prelaze iz jednog stanja u drugo kao odgovor na vanjske ili unutarnje događaje (okidače). To ih čini posebno korisnima za modeliranje sustava sa složenim obrascima ponašanja, u kojima njihova primjena pomaže u preciznijem utvrđivanju zahtjeva, olakšava proces oblikovanja i implementacije te ubrzava razvoj ispitnih slučajeva i otklanjanje pogrešaka. Oni doprinose pouzdanosti programske potpore i boljoj komunikaciji s dionicima te omogućuju bolje razumijevanje dinamike sustava.

Norma UML-a 2.5.1. definira značajan broj semantičkih elemenata koji se mogu koristiti na dijagramima stanja te specijalizirane podvrste dijagrama stanja, kao npr. dijagram stanja protokola (engl. *Protocol State Machine*), i u skladu s time razmjerno složena pravila sintakse. U ovom će priručniku naglasak biti na pregledu temeljnog skupa elemenata i primjerima njihova korištenja u praksi, dok se za naprednije značajke čitatelja upućuje na službenu dokumentaciju UML-a i dodatnu literaturu.

6.1 Definicija i osnovni elementi

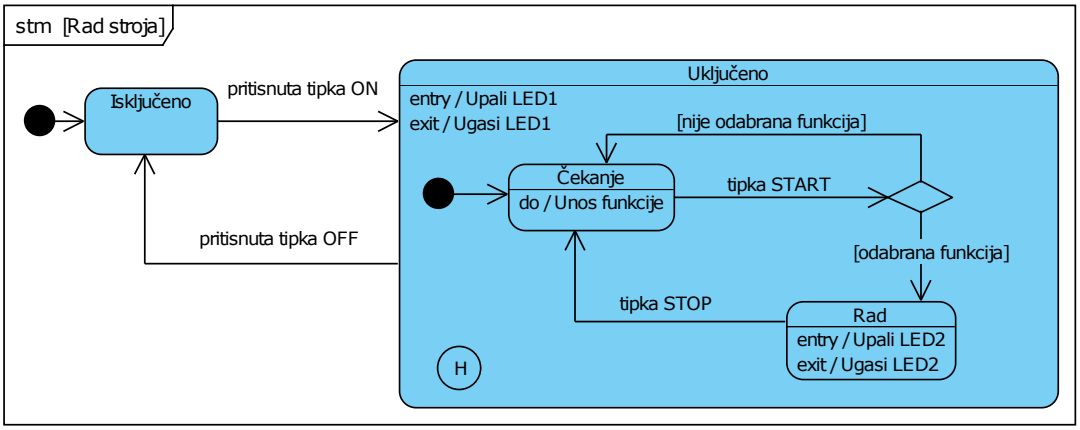
UML dijagram stanja (engl. *state machine diagram*) ponašajni je UML dijagram kojim se prikazuje diskretno ponašanje objekta ili sustava putem prelazaka između konačnog broja stanja, a često se za cjelokupni model stanja i prelazaka između stanja koristi izraz **stroj stanja** (engl. *state machine*). Ovaj dijagram je u osnovi sličan konačnom automatu iz teorije računarstva, no značajno semantički i sintaksno nadograđen. Dijagram stanja koristi se za modeliranje ponašanja entiteta tijekom vremena tako što ističe odgovor na događaje i okidače. Osnovni su elementi dijagrama stanja čvorovi, koji predstavljaju stanja i pseudostanja, te usmjereni bridovi, koji povezuju čvorove i predstavljaju prijelaze između stanja:

- **Stanje** (engl. *state*) – određeni način na koji objekt ili sustav može postojati, tj. modelira situaciju u kojoj vrijedi neki skup uvjeta ili se izvršava neko određeno ponašanje. Stanje može

predstavljati statičku situaciju, poput objekta koji čeka da se dogodi neki vanjski događaj, ali može modelirati i dinamičke uvjete, poput procesa izvođenja nekog ponašanja pri čemu se razmatra ulaz u stanje u kojem ponašanje započne i napušta ga čim se ponašanje završi. Pseudostanje je podvrsta stanja i koristi se za predstavljanje prolaznih (engl. *transient*) točaka u ponašanju sustava, kao npr. početna i završna pseudostanja, pseudostanja izbora itd. koja pomažu u modeliranju složenog upravljačkog tijeka.

- **Prijelaz** (engl. *transition*) – prikazuje se kao strelica, a služi za modeliranje trenutka u kojem objekt ili sustav prelazi iz jednog stanja u drugo kao odgovor na događaje ili okidače. Događaji su vanjski ili unutarnji poticaji koji pokreću prelaske iz jednog stanja u drugo i mogu biti korisničke radnje, promjene u okolini ili drugi ulazi vezani za sustav.

Od inačice norme UML 2.4 nadalje definirane su dvije vrste dijagrama stanja: **ponašajni dijagrami stanja** (engl. *behavioral state machine*) i **dijagrami stanja protokola** (engl. *protocol state machine*). Ponašajni dijagrami stanja općenita su vrsta dijagrama stanja i koriste se za opis stanja bilo kojeg entiteta, a primjer je dan na slici 6.1. Dijagrami stanja protokola specijalna su podvrsta koja se može koristiti za izražavanje protokola korištenja dijela sustava. Specifična sintaksa i pravila dijagrama stanja protokola neće se razmatrati u sklopu ovog priručnika.



Slika 6.1: Primjer UML dijagrama stanja koji prikazuje rad nekog stroja

U nastavku ovog poglavlja najprije će biti objašnjena sintaksa i semantika jednostavnih stanja i prijelaza, a zatim će se razmotriti složeniji slučajevi: ugniježđena stanja, paralelne regije i pamćenje.

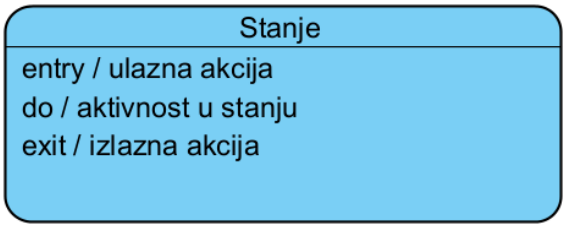
6.1.1 Stanja

Stanje je osnovna komponenta dijagrama stanja i igra ključnu ulogu u modeliranju dinamičkog ponašanja sustava. Riječ je o skupu specifičnih svojstava ili fazi ponašanja entiteta (npr. objekta, sustava ili komponente) koji vrijede sve dok je ispunjen određeni skup uvjeta. Stanje opisuje kako se entitet ponaša, koje su vrijednosti njegovih atributa i kako reagira na događaje.

Svako stanje predstavlja jedinstveni kontekst ili scenarij u kojem se entitet može nalaziti. Isto tako, stanje može imati određene akcije ili aktivnosti povezane s njim i definirane uvjete ili kriterije koji određuju kada će entitet prijeći iz jednog stanja u drugo. Stanje se na dijagramu prikazuje simbolom pravokutnika sa zaobljenim rubovima, a jedini je obvezni sadržaj njegov naziv, koji prenosi njegovo značenje ili svrhu, i pomaže u tome da dijagram bude razumljiviji.

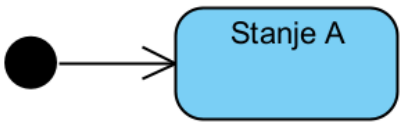
Unutar stanja, moguće je još definirati ulazne i izlazne akcije te aktivnost povezanu sa stanjem koja se izvršava neprekidno dok je entitet u tom stanju. Akcije se navode uz ključne riječi

entry i **exit**, a aktivnosti uz ključnu riječ **do**, kao što je ilustrirano na slici 6.2. Ulazna akcija definira što se događa kada entitet prelazi u stanje i izvršava se odmah po ulasku u stanje. Slično, izlazna akcija definira što se događa kada entitet izlazi iz stanja i izvršava se posljednja na izlasku iz stanja. Aktivnost u stanju izvršava se neprekidno dok je entitet u tom stanju. Riječ je o kontinuiranim radnjama ili ponašanju sve dok se entitet nalazi u određenom stanju. Iako ti elementi nisu obvezni, uključivanje akcija i aktivnosti u definicije stanja obogaćuje prikaz dinamičkog ponašanja u dijagramima stanja te omogućuju sveobuhvatniji i precizniji model ponašanja objekta ili sustava.



Slika 6.2: Stanje na UML dijagramu stanja

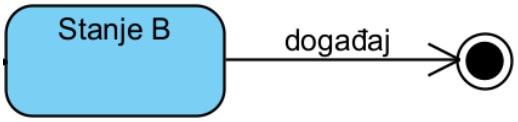
Pseudostanja su posebna skupina stanja koja čine prijelazna (engl. *transitional*) stanja u kojima se sustav zadržava kratkotrajno prije prelaska u neko od klasičnih stanja. Jedno je od najčešće korištenih pseudostanja **početno pseudostanje** (engl. *initial pseudostate*), prikazano kao ispunjeni crni krug, slika 6.3. Ono se koristi za označavanje početne točke stroja stanja, tj. označava gdje stroj započinje svoje izvođenje ili gdje počinje životni ciklus objekta ili sustava. Kada se stroj stanja aktivira, automatski prelazi iz početnog pseudostanja u stanje u koje ono vodi. Obično postoji jedno početno pseudostanje u dijagramu stanja, koje se putem prijelaza povezuje s jednim stanjem ili više njih.



Slika 6.3: Početno pseudostanje

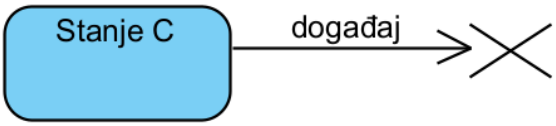
Stroj stanja može modelirati beskonačnu petlju u kojoj sustav stalno kruži između utvrđenih stanja, ali moguće je modelirati i sustav koji pod određenim uvjetima trajno prestaje s izvođenjem. Taj završetak može biti uvjetovan događajima unutar samog sustava koji vode u završno stanje ili promjenom konteksta izvođenja, kao npr. nestankom napajanja, uništavanjem objekta koji se modelira itd. U skladu s navedenim, za prikaz završetka stroja stanja moguće je koristiti završno stanje ili pseudostanje prekida.

Završno stanje (engl. *final state*) označava redoviti završetak skupa stanja, tj. životnog ciklusa određenog objekta. Završno stanje prikazuje se kao crni krug unutar bijelog kruga s crnim obrubom, kao na slici 6.4. Kada stroj stanja prijeđe u završno pseudostanje, to implicira da je proces završen i da daljnji prelasci više nisu mogući. Dijagram stroja stanja može imati jedno završno stanje ili više njih, ovisno o broju mogućih točaka završetka. Ovdje je bitno istaknuti da se prema važećoj normi UML-a 2.5.1 završno stanje ne smatra pseudostanjem, već pravim stanjem, iako je prema svojim karakteristikama slično svim ostalim pseudostanjima.



Slika 6.4: Završno stanje

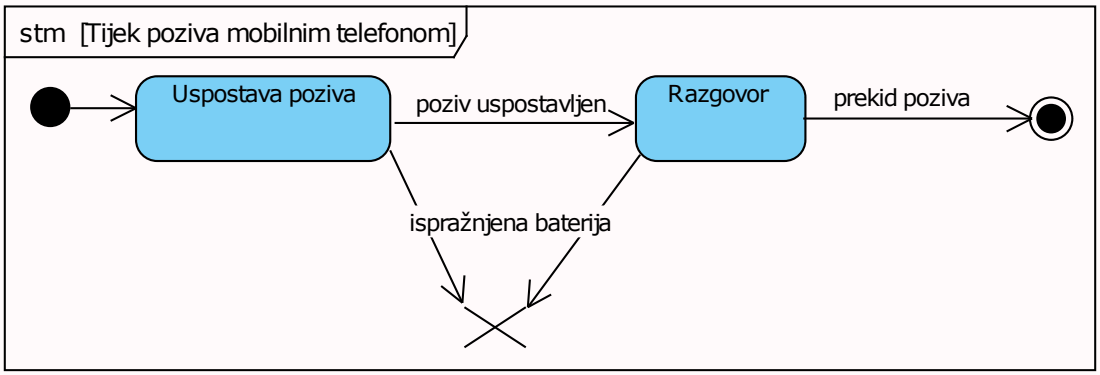
Pseudostanje prekida (engl. *terminate pseudostate*) označava da se izvođenje stroja stanja prekida i završava zbog promjena u kontekstu kojih dolazi do uništavanja objekta. Kod aktivacije prijelaza koji vodi u ovo pseudostanje stroj stanja ne izlazi iz bilo kojeg stanja niti izvodi bilo kakve izlazne akcije osim onih povezanih s prijelazom koji vodi do tog pseudostanja. Ulazak u pseudostanje prekida ekvivalentno je pozivanju funkcije „DestroyObjectAction" i prikazuje se simbolom „x", kao na slici 6.5.



Slika 6.5: Pseudostanje prekida

Primjer 6.1 — Završno stanje i pseudostanje prekida. Potrebno je prikazati tijek poziva mobilnim telefonom. Poziv je najprije u stanju uspostave. Kada se uspješno uspostavi, prelazi u stanje razgovora, a nakon što korisnik prekine poziv, on završava. Ako se baterija isprazni za vrijeme uspostave poziva ili razgovora, poziv se prekida.

Rješenje je prikazano na slici 6.6.



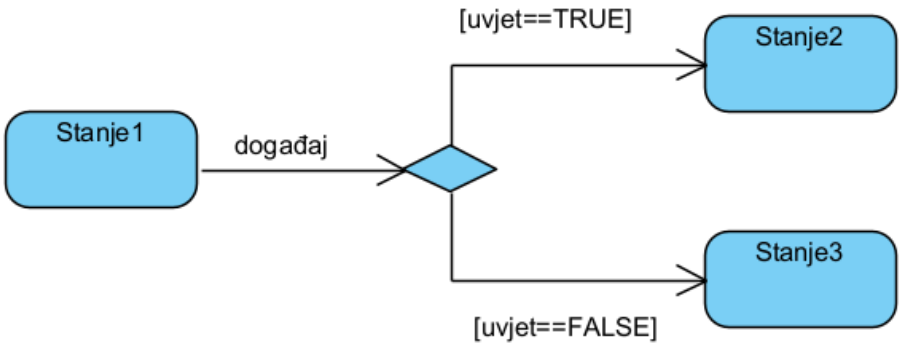
Slika 6.6: Dijagram stanja za uspostavu poziva i razgovor

Komentar:

Na dijagramu je istaknuta razlika između redovnog završetka poziva, u kojem se obavljaju sve aktivnosti vezane za završetak poziva, i naglog prekida poziva uzrokovanog pražnjenjem baterije što u ovom slučaju čini kritičnu promjenu u kontekstu izvođenja. ■

Pseudostanje izbora (engl. *choice*) upotrebljava se za modeliranje točaka odlučivanja unutar stroja stanja, tj. za prikazivanje grananja u različite puteve na temelju ispunjenja određenih uvjeta. Za prikaz se koristi simbol romba iz kojeg obično ima više izlaza, a svaki vodi prema drugom

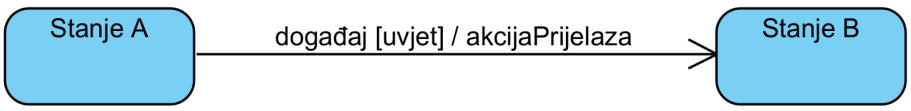
ciljnom stanju. Svaki prijelaz zaštićen je uvjetom (engl. *guard condition*), koji se zapisuje u uglatim zagradama, kao na slici 6.7. Pri donošenju odluke odabire se ona grana za koju je ispunjen zadani uvjet.



Slika 6.7: Pseudostanje izbora

6.1.2 Prijelazi

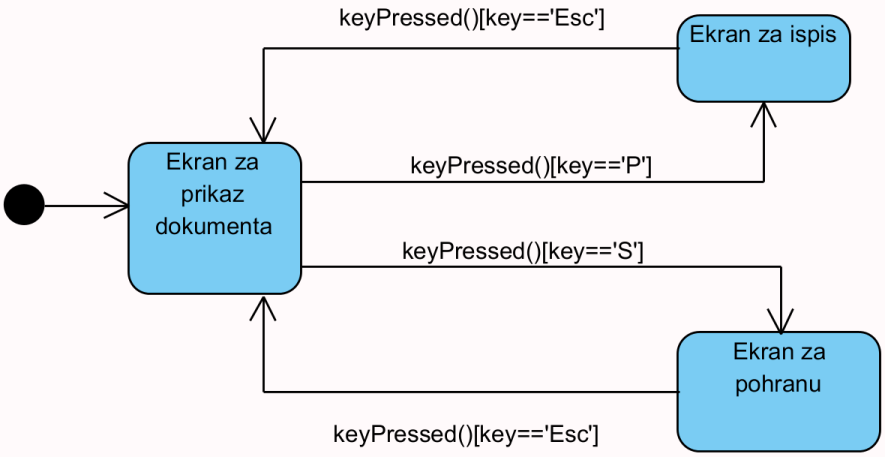
Prijelazi na UML dijagramima stanja upotrebljavaju se za prikaz toga kako objekt ili sustav prelazi iz jednog stanja u drugo kao odgovor na događaje. Osim u posebnim slučajevima kod složenih stanja, o čemu će biti riječi kasnije, okidač prelaska između stanja (engl. *trigger*) uvijek je neki događaj (npr. signal prekida, korisnička akcija itd.). Međutim, sama pojava događaja ne mora nužno značiti da će se prelazak zaista i dogoditi. Naime, uz okidač prelaska moguće je navesti i logički uvjet (engl. *guard*), koji također mora biti zadovoljen. Tek kada se dogodio događaj koji je okidač prelaska i kada su zadovoljeni zadani logički uvjeti, moguće je napraviti prelazak u iduće stanje, kao što je ilustrirano u primjeru 6.2. Po potrebi je moguće zadati i akcije koje se trebaju izvršiti pri prelasku u stanje, pa je sveukupna sintaksa prijelaza: **okidač[uvjet]/akcija** i zapisuje se iznad prijelaza, kao na slici 6.8.



Slika 6.8: Prijelaz između stanja

Primjer 6.2 — Uvjetovani prijelazi. U jednostavnom programu za uređivanje teksta na početku je aktivan ekran na kojem se prikazuje dokument. Pritisak tipke poziva metodu „keyPressed()" i ako je pritisnuta tipka „P", otvara se ekran za ispis dokumenta. Ako je pak pritisnuta tipka „S", otvara se ekran za pohranu dokumenta. Na ostale tipke nema reakcije. Iz ekrana za ispis i ekrana za pohranu vraća se na ekran za pregled pritiskom tipke „Esc".

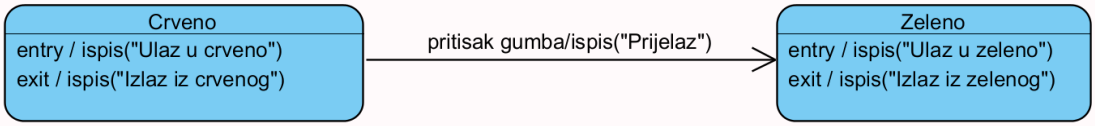
Rješenje je prikazano na slici 6.9.



Slika 6.9: Dijagram stanja za ekrane i pritiske tipki

Akcije prijelaza mogu biti ažuriranje vrijednosti nekih atributa prije ulaska u iduće stanje, slanje signala o prelasku sustava u novo stanje, izračun uvjeta vezanih za pseudostanje izbora itd. Ako stanje iz kojeg se izlazi ima zadane izlazne akcije (*exit*), a stanje u koje se ulazi ima zadane ulazne akcije (*entry*), akcija prijelaza odvija se nakon izlazne akcije, a prije ulazne akcije.

Primjer 6.3 — Redoslijed izvršavanja akcija. Ako je ponašanje nekog programa zadano dijagramom stanja na slici 6.10, što će se ispisati kada se pritisne gumb, ako je na početku program u stanju Crveno?

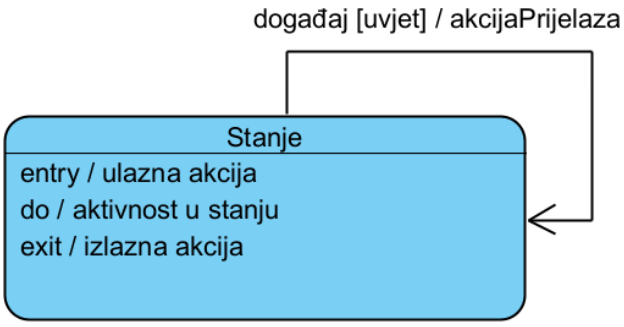


Slika 6.10: Dijagram stanja programa

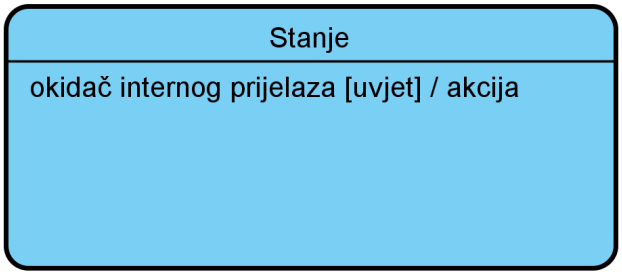
Rješenje:

Program će ispisati: „Izlaz iz crvenog | Prijelaz | Ulaz u zeleno”.

Kada je riječ o prijelazima natrag u isto stanje, moguća su dva osnovna slučaja: vanjski (engl. *external*) i unutarnji (engl. *internal*) prijelaz. Vanjski prijelaz, ilustriran na slici 6.11, znači da sustav izlazi iz trenutačnog stanja i zatim ponovno ulazi u to isto stanje. Pri tome obavlja redom izlaznu akciju, akciju prijelaza i ulaznu akciju, ako su one zadane. Nasuprot tome, unutarnji prijelaz, ilustriran na slici 6.12, znači da sustav ostaje u trenutačnom stanju i obavlja samo akciju prijelaza, ako je ona zadana.



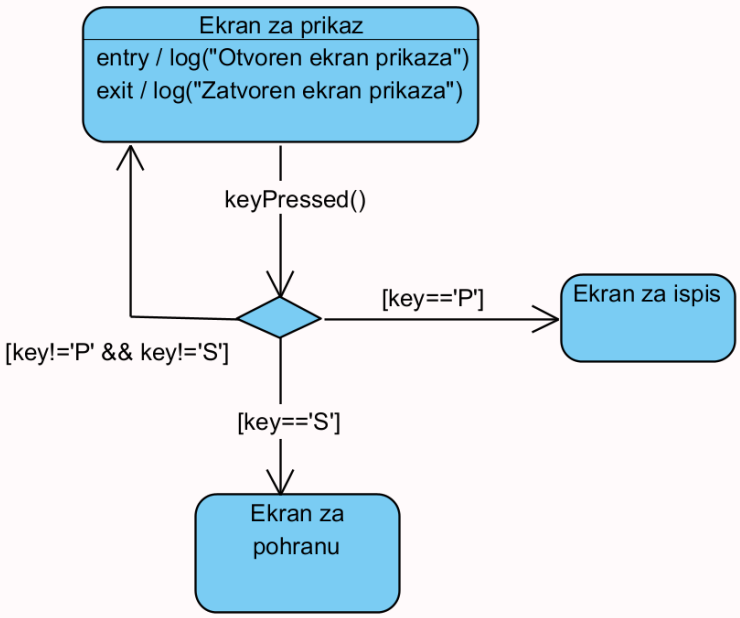
Slika 6.11: Vanjski prijelaz natrag u isto stanje



Slika 6.12: Unutarnji prijelaz

Primjer 6.4 — Vanjski prijelaz u isto stanje. Uporabom pseudostanja izbora modelirajte dio rješenja primjera 6.2 vezanog za prelaske iz ekrana prikaza dokumenta u ekrane za ispis ili pohranu (zanemarite povratak iz ekrana za ispis i pohranu na ekran za prikaz). Usto je još potrebno dodati i akcije upisa poruka „Otvoren ekran prikaza" i „Zatvoren ekran prikaza" u kontrolni dnevnik (engl. *log*) pri ulasku, odnosno izlasku iz stanja prikaza.

Rješenje je prikazano na slici 6.13.



Slika 6.13: Modeliranje kretanja između ekrana korištenjem pseudostanja izbora

Komentar:

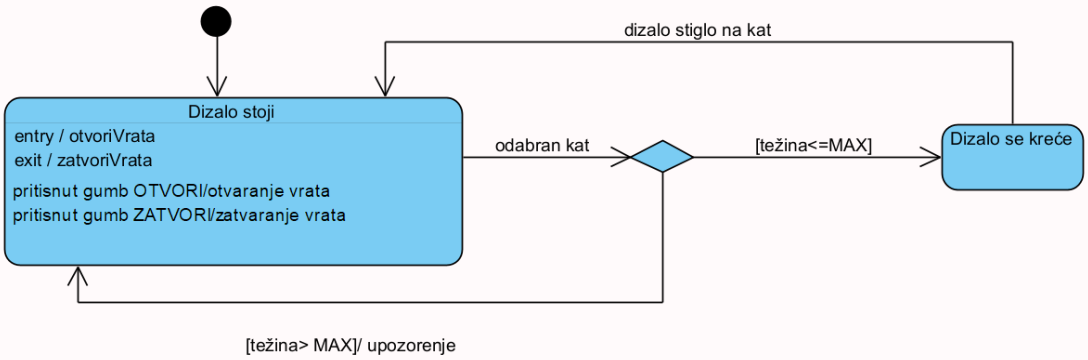
U ovom slučaju prijelaz na okidač „keyPressed()" razdvaja se na ulaz u pseudostanje izbora i izlazak u jednu od grana koje vode iz tog pseudostanja, ovisno o uvjetu koji je zadovoljen. Zbog toga je potrebno utvrditi i prijelaz natrag u isto stanje ako nije pritisnuta ni tipka „P" ni tipka „S", čega u originalnom rješenju nije bilo.

Razlika između originalnog i novog rješenja u tome je što se u originalnom rješenju ni jedan od dvaju prijelaza nije aktivirao ako nije bio zadovoljen odgovarajući uvjet (tipka), pa se u skladu s time nije niti izlazilo iz stanja prikaza. Sada se na okidač „keyPressed()" izlazi iz stanja „Ekran za prikaz" i ulazi u pseudostanje izbora, pa je potrebno utvrditi prijelaz natrag u stanje „Ekran za prikaz" ako nisu pritisnute tipke „P" ili „S". Također, to znači da će se na svaki „keyPressed()" aktivirati izlazna akcija koja će zapisati „Zatvoren ekran prikaza" te ako nisu pritisnute tipke „P" ili „S" ponovno će se ući u stanje „Ekran za prikaz" pa će se tada zapisati „Otvoren ekran prikaza".

Naravno, ako ne bi postojale ulazne i izlazne akcije, rješenje iz originalnog primjera i ovo rješenje rezultiralo bi identičnim ponašanjem sustava. ■

Primjer 6.5 — Vanjski i unutarnji prijelazi. Potrebno je modelirati jednostavno dizalo. Dizalo na početku stoji s otvorenim vratima sve dok se ne odabere kat. Tada se vrata zatvaraju i ako je težina manja ili jednaka najvećoj dopuštenoj, dizalo se kreće. Kada stigne na kat, zaustavlja se te se vrata otvaraju. Ako je težina veća od najveće dopuštene, oglašava se upozorenje, dizalo ne kreće i vrata se otvaraju. Također, dok dizalo stoji moguće je pritisnuti gumb za otvaranje vrata koji će otvoriti vrata, kao i gumb za zatvaranje vrata koji će zatvoriti vrata (dizalo neće krenuti sve dok nije odabran kat).

Rješenje je prikazano na slici 6.14.



Slika 6.14: Modeliranja stajanja i kretanja dizala

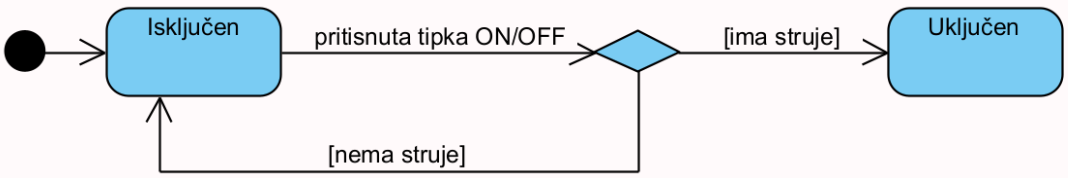
Komentar:

Pritisak gumba za otvaranje odnosno gumba za zatvaranje vrata ne vodi u promjenu stanja pa se ti okidači modeliraju kao unutarnji prijelazi stanja „Dizalo stoji". Akciju zatvaranja vrata moguće je pridružiti i okidaču „odabran kat" (zapis: odabran kat / zatvaranje vrata) bez promjene u značenju jer će se u oba slučaja vrata zatvoriti nakon odabira kata neovisno o tome kreće li se dizalo. Akciju otvaranja vrata nužno je modelirati kao ulaznu akciju stanja „Dizalo stoji" jer u opisu sustava piše da dizalo stoji s otvorenim vratima.

Kada je riječ o modeliranju grananja, status ispunjenja uvjeta može biti poznat u trenutku događaja koji dovodi do grananja, a to se još naziva **statičko grananje**, koje je ilustrirano primjerom 6.6. No, moguće su i situacije u kojima status ispunjenja uvjeta za odabir grane nije poznat u trenutku događaja koji dovodi do grananja, nego je potrebno izvršiti dodatne izračune, što se još naziva **dinamičko grananje**, a ilustrirano je primjerom 6.7.

Primjer 6.6 — Statičko grananje. Potrebno je modelirati jednostavan stroj koji je na početku isključen i uključuje se na pritisak tipke ON/OFF ako ima struje.

Rješenje je prikazano na slici 6.15.



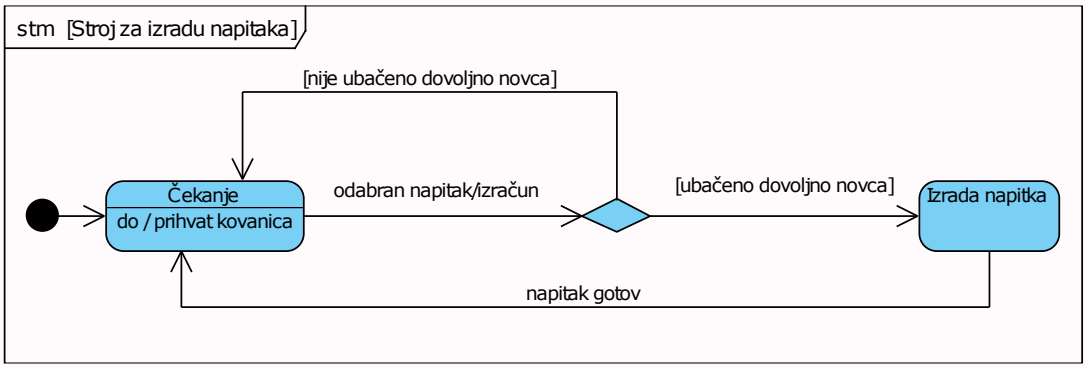
Slika 6.15: Uključenje i isključenje stroja modelirano statičkim grananjem

Komentar:

Ispunjenje uvjeta poznato je u trenutku događaja: struje ili ima ili nema, ne treba ništa računati.

Primjer 6.7 — Dinamičko grananje. Potrebno je modelirati jednostavan stroj za izradu napitaka. Stroj na početku čeka i prihvaća kovanice. Kada korisnik odabere napitak, stroj računa je li ubačeno dovoljno novca za odabrani napitak. Ako jest, stroj izrađuje napitak. Kada je napitak gotov, stroj ponovno prihvaća kovanice i čeka na odabir sljedećeg napitka. Ako je odabran napitak, a nije ubačeno dovoljno novca, stroj dalje čeka na ubacivanje dodatnih kovanica. Radi jednostavnosti pretpostavite da stroj ne vraća višak novca, ali pamti uneseni iznos dok se ne potroši na izrađeni napitak.

Rješenje je prikazano na slici 6.16.



Slika 6.16: Rad stroja za izradu napitaka modeliran dinamičkim grananjem

Komentar:

Ispunjenje uvjeta nije poznato u trenutku kada se odabere napitak te se prije donošenja odluke mora obaviti izračun.

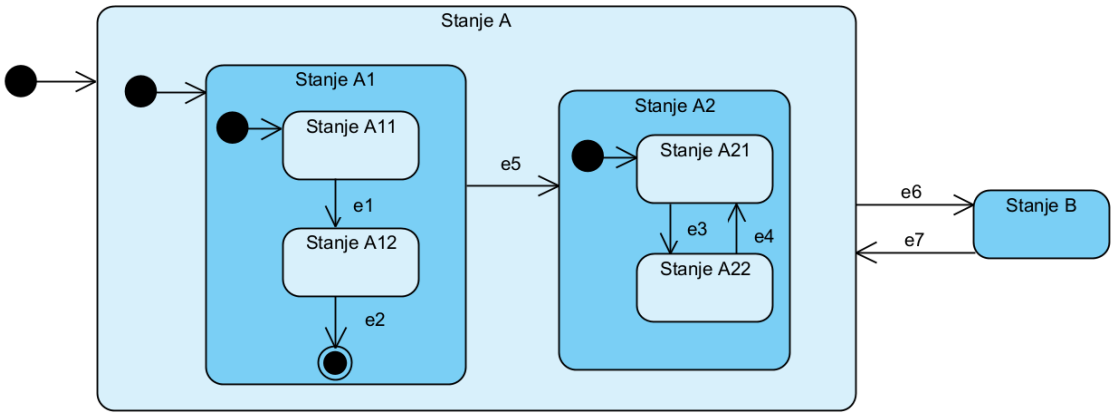
■

6.1.3 Složena stanja

Jezik UML nudi sintaksne elemente koji omogućuju opisivanje složenih sustava u detaljnijoj razradi pojedinih stanja. Tako je osim definiranja ulaznih i izlaznih akcija, aktivnosti i internih prijelaza, moguće razraditi i složenu strukturu stanja. To podrazumijeva razradu hijerarhije ugniježđenih podstanja, uvođenje svojstva pamćenja u stroj stanja te modeliranje ortogonalnih stanja.

Ugniježđena stanja (engl. *nested states*) omogućuju modeliranje ponašanja složenih sustava na strukturiran i modularan način. Tako je moguće izdvojiti manje skupine svojstava (atributa i ponašanja) sustava koji se mijenjaju i grupirati ih u zasebne cjeline – podstanja unutar vršnog stanja. Ako postoji više podstanja unutar jednog složenog stanja, potrebno je uporabom čvora početnog pseudostanja odrediti koje je od njih početno, kao što je ilustrirano na slici 6.17. Tumačenje tako definiranog stroja stanja glasi: ako je zadano neko složeno stanje, kao npr. Stanje A na slici, ulaskom u to stanje ulazi se i u jedno od njegovih podstanja (na svakoj razni). Ako nije drukčije naznačeno, npr. izravnim prijelazom koji vodi izvana u neko podstanje, uvijek se ulazi u ono podstanje na koje pokazuje čvor početnog pseudostanja.

Unutar složenog stanja, koje sadržava nekoliko podstanja, moguće je definirati i završetak tog stanja korištenjem čvora završnog stanja. Kada se dostigne završni čvor unutar stanja, to znači da stanje koje obuhvaća taj čvor završava i sustav može prijeći u iduće stanje.



Slika 6.17: Primjer ugniježđenih stanja

Osim korištenja ugniježđenih podstanja, za napredan opis dinamičkog ponašanja sustava potrebni su dodatni čvorovi pseudostanja: plitka i duboka povijest te čvorovi račvanja i grananja.

Povijesna pseudostanja, kao što su **plitka povijest** (engl. *shallow history*) i **duboka povijest** (engl. *deep history*), služe da bi se omogućilo pamćenje u složenim stanjima koja se sastoje od više razina ugniježđenih stanja. Temeljna je svrha tih pseudostanja pamćenje informacije o posljednjem aktivnom podstanju prije izlaska iz složenog stanja. Kada se ponovno uđe u složeno stanje, povijesni čvor omogućuje povratak u zapamćeno podstanje.

Pseudostanje plitke povijesti pamti zadnje aktivno podstanje samo na najvišoj razini ugniježđenih stanja. Međutim, pseudostanje duboke povijesti pamti cjelokupnu konfiguraciju stanja sve do najniže razine podstanja u hijerarhiji. Simboli koji predstavljaju ta pseudostanja prikazani su na slici 6.18, a ilustrativni primjeri 6.8 i 6.9 pobliže opisuju njihovu semantiku.

(a) Plitka povijest

(b) Duboka povijest

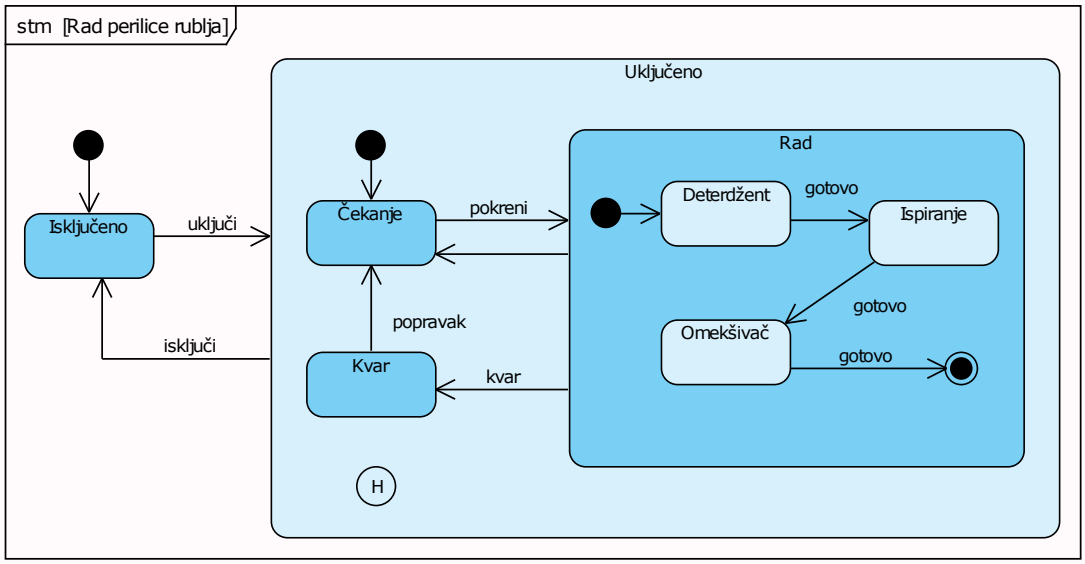
Slika 6.18: Povijesna pseudostanja

Primjer 6.8 — Plitka povijest. Ako je ponašanje jednostavne perilice rublja prikazano dijagramom stanja na slici 6.19, opišite što će se dogoditi u sljedeća tri slučaja:

A) Završi stanje Omekšivač.

B) Perilica se isključi dok je u stanju Kvar i zatim se ponovno uključi.

C) Perilica se isključi dok je u stanju Ispiranje i zatim se ponovno uključi.



Slika 6.19: Rad perilice rublja modeliran uz uporabu pseudostanja plitke povijesti

Rješenje:

A) Kada završi stanje Omekšivač, sljedeće stanje je završno stanje unutar stanja Rad te kada je ono dosegnuto, stanje Rad završava i aktivira se **implicitni prijelaz** (prijelaz bez događaja, prazna strelica) koji vodi u stanje Čekanje. Dakle, perilica će nakon što dovrši proces omekšavanja rublja, završiti s radom i čekati na ponovno pokretanje.

B) S obzirom na to da automat ima definirano pseudostanje plitke povijesti unutar stanja Uključeno, ako se perilica isključi, ona će nakon ponovnog uključivanja odmah otići u stanje Kvar.

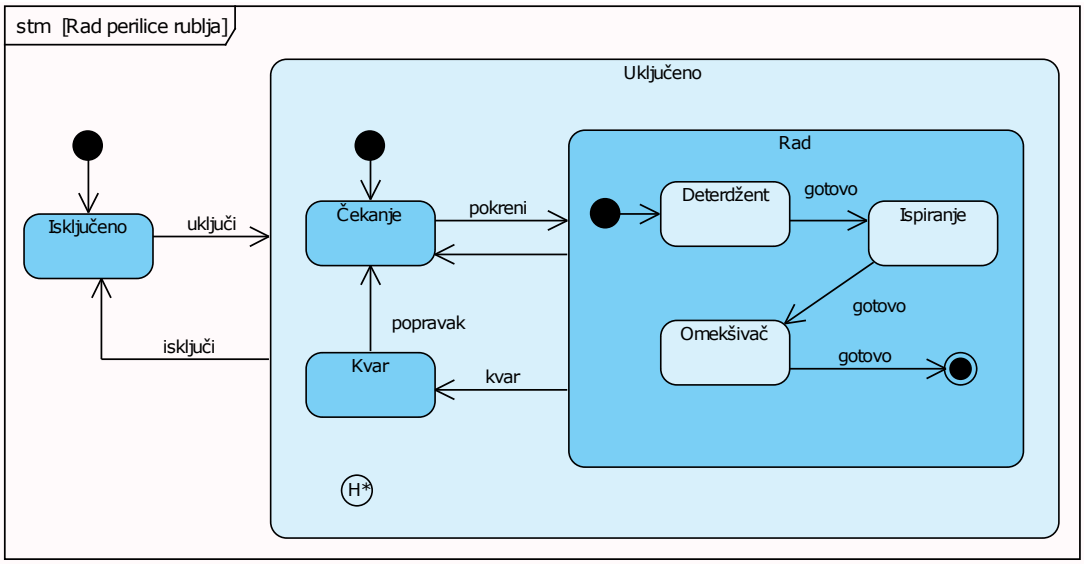
C) Budući da plitka povijest pamti samo zadnje stanje na prvoj razini, nakon isključivanja i ponovnog uključivanja, perilica će otići u stanje Rad, ali će započeti s ciklusom pranja otpočetka (plitka povijest ne pamti stanja na dubljim razinama).

■

Primjer 6.9 — Duboka povijest. Potrebno je izmijeniti model iz prethodnog primjera tako da perilica može nastaviti s ispiranjem ako se isključi dok je u tom stanju i zatim ponovno uključi.

Rješenje:

Pamćenje na svim razinama bit će moguće ako se pseudostanje plitke povijesti zamijeni pseudostanjem duboke povijesti, slika 6.20.



Slika 6.20: Rad perilice rublja modeliran uz uporabu pseudostanja duboke povijesti

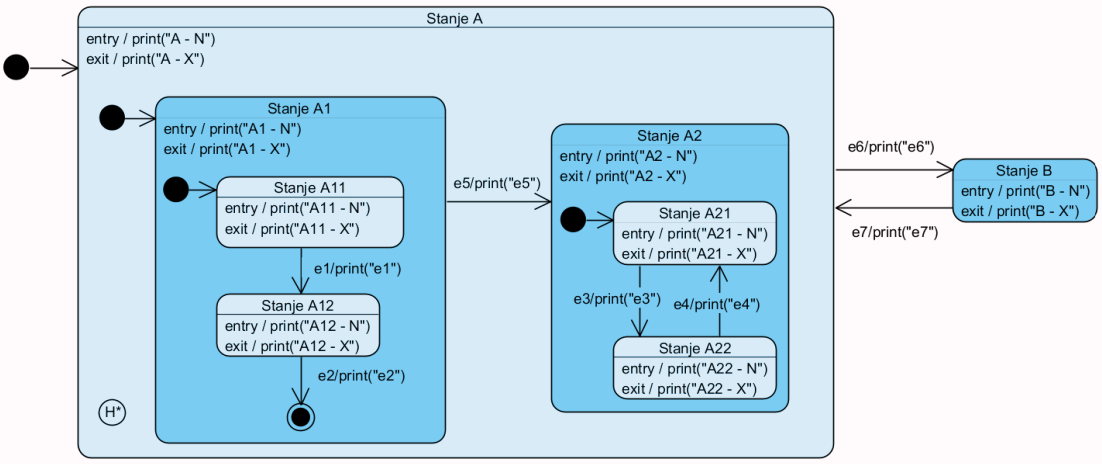
Kada je riječ o složenim stanjima s više razina ugniježđenih stanja moguće je i definirati ulazne i izlazne akcije na svim razinama. Kada se dogodi prelazak koji vodi van iz složenog stanja, potrebno je izvesti redom sve izlazne akcije od najniže razine prema najvišoj prije nego što se izvrši prelazak u sljedeće stanje. Ako unutar složenog stanja postoji pamćenje, pri ponovnom ulasku u to stanje potrebno je izvesti redom sve ulazne akcije počevši od najviše razine sve do ulaska u zapamćeno podstanje. Na primjeru 6.10 ilustriran je redoslijed izvođenja ulaznih i izlaznih akcija.

Primjer 6.10 — Ugniježđena stanja – redoslijed izvođenja. Ako je sustav opisan dijagramom na slici 6.21, što će se ispisati u sljedećim slučajevima:

A) Sustav počinje s izvođenjem.

B) Sustav je u podstanju A11 i dogodi se događaj e5.

C) Sustav je u podstanju A12 i dogode se redom događaji: e6 i e7.



Slika 6.21: Redoslijed izvođenja u ugniježđenim stanjima

Rješenje:

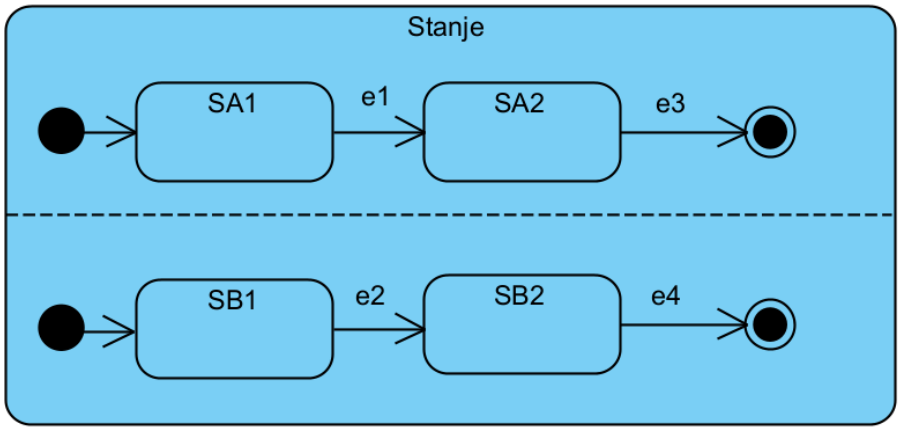
A) Ispis: „A - N | A1 - N | A11 - N” (oznaka „|” služi samo za razdvajanje pojedinih ispisa)

B) Ispis: „A11 - X | A1 - X | e5 | A2 - N | A21 - N”

C) Ispis: „A12 - X | A1 - X | A - X | e6 | B - N | B - X | e7 | A - N | A1 - N | A12 - N”

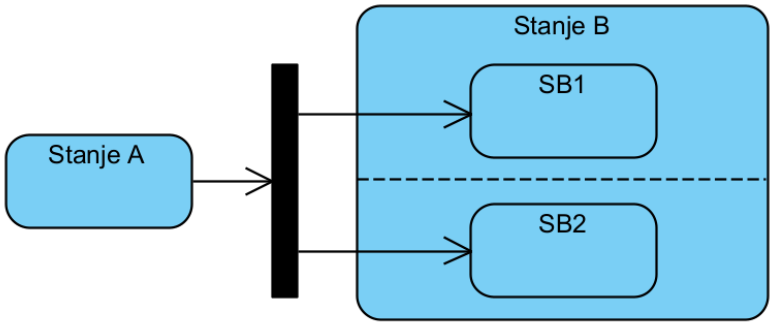
Ortogonalne regije u dijagramima stanja UML-a omogućuju modeliranje konkurentnog i paralelnog ponašanja kakvo se često susreće u stvarnim sustavima. U kombinaciji s ugniježđenim podstanjima, omogućuju modeliranje više neovisnih aspekata ponašanja istodobno te poboljšavaju preciznost i jasnoću opisa dinamičkog ponašanja sustava.

Ortogonalno stanje podijeljeno je na dva ili više dijelova razdvojenih isprekidanom linijom, kao što je prikazano na slici 6.22. U bilo kojem trenutku u svakoj je regiji aktivno točno jedno stanje. Ulaz u takvo stanje aktivira početna stanja svih regija. Za izlazak iz stanja u svim se regijama mora doseći završno stanje. Podstanja u različitim regijama neovisna su i mogu se doseći u različitim trenucima.

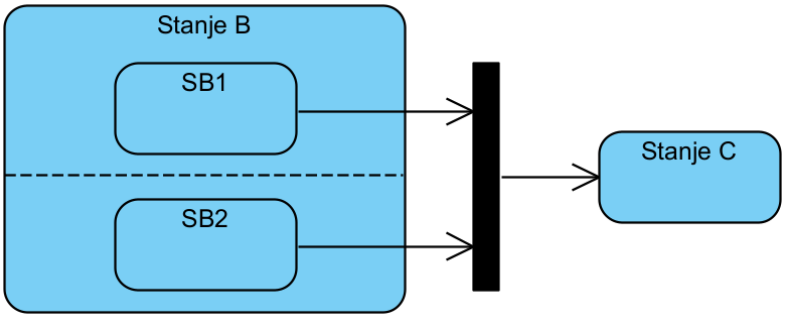


Slika 6.22: Ortogonalne regije

Za usklađivanje podstanja iz ortogonalnih regija mogu se koristiti pseudostanja račvanja (engl. *fork*) i sinkronizacije (engl. *join*), kao što je ilustrirano na slikama 6.23 i 6.24.



Slika 6.23: Račvanje



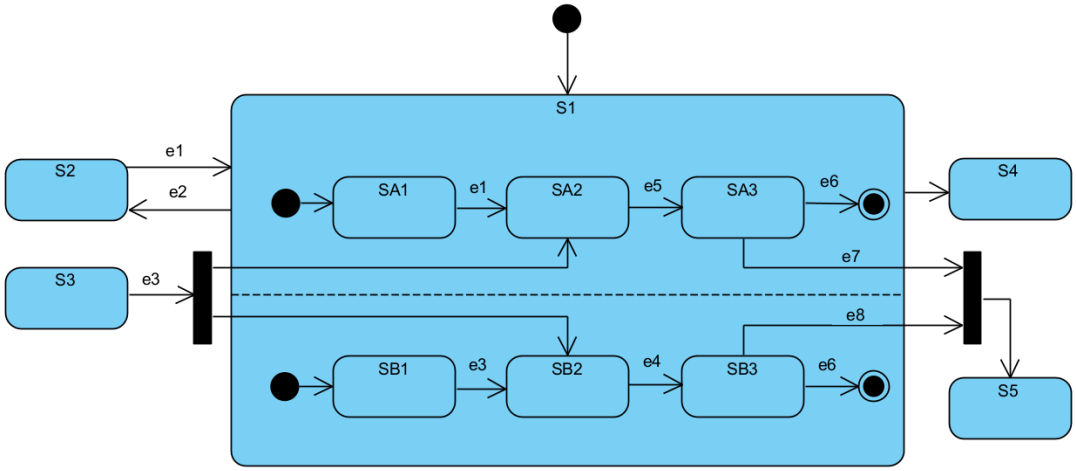
Slika 6.24: Sinkronizacija

Primjer 6.11 — Ortogonalne regije – redoslijed izvođenja. Za sustav modeliran dijagramom na slici 6.25 navedite redoslijed prolaska kroz stanja u sljedećim slučajevima:

A) Sustav počinje s radom i zatim se dogode redom: e1, e5, e3, e6, e4, e6.

B) Sustav se nalazi u stanju S3 i dogodi se e3.

C) Sustav se nalazi u stanju SA2 i SB3 i dogode se redom: e5, e7, e8.



Slika 6.25: Primjer dijagrama stanja s ortogonalnim regijama

Rješenje:

A) Sustav ulazi u stanje S1 te zatim istodobno u podstanja SA1 i SB1. Zatim se prolasci kroz regije odvijaju neovisno: događaj e1 vodi u SA2 (SB1 i dalje vrijedi), e5 u SA3 (SB1 i dalje vrijedi), e6 u završno stanje gornje regije, e3 u SB2, e4 u SB3 i s6 u završno stanje donje regije. Tek sada, kada su dosegnuta završna stanja u objema regijama, završava stanje S1 i aktivira se prelazak u S4.

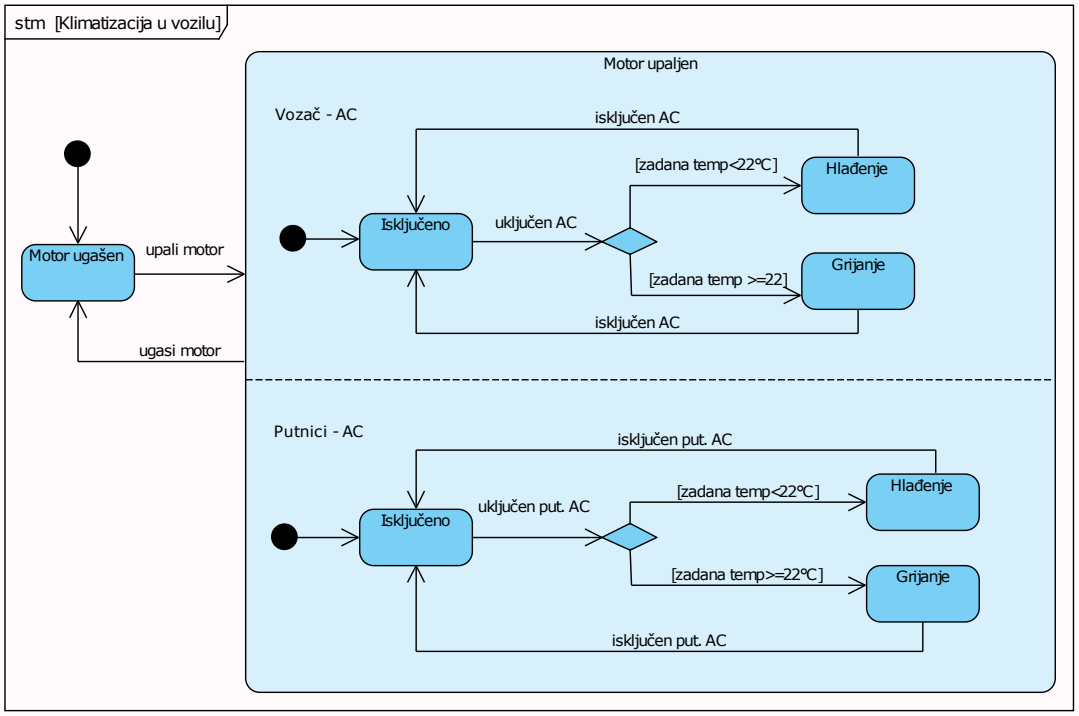
B) Sustav istodobno ulazi u podstanja SA2 i SB2 jer u njih vode prijelazi iz pseudostanja račvanja.

C) Događaj e5 vodi iz SA2 u SA3, a događaji e7 i e8 redom vode iz obiju regija u čvor sinkronizacije iz kojeg se nastavlja u stanje S5.

■

Primjer 6.12 — Ortogonalne regije – Klimatizacija u vozilu. Modelirajte sustav klimatizacije (AC) u vozilu za sljedeći opis sustava. Klimatizacija je podijeljena na dva dijela: vozački i putnički, te se njima može neovisno upravljati. Kada se motor upali, klimatizacija je uvijek isključena u vozačkom i putničkom dijelu. Kada se klimatizacija uključi (vozački ili putnički dio), ako je zadana temperatura niža od 22°C, uključit će se hlađenje, a inače će se uključiti grijanje.

Rješenje je prikazano na slici 6.26.



Slika 6.26: Rad klimatizacije u vozilu modeliran uz primjenu ortogonalnih regija

6.2 Postupak izrade dijagrama

Modeliranje ponašanja sustava ili dijela sustava UML dijagramima stanja složeni je postupak čiji detalji ovise o konkretnom problemu. U svim osim najjednostavnijim slučajevima, radi se o iterativnom procesu u kojem se model postupno razrađuje i poboljšava kako se povećava razumijevanje ponašanja dijela sustava koji se modelira. Pri izradi tih dijagrama uvijek je dobro kao početnu točku uzeti popis zahtjeva na sustav te postojeće dijagrame, kao npr. dijagrame obrazaca uporabe, sekvencijske dijagrame, dijagrame razreda itd. Na temelju tih početnih saznanja moguće je modelirati stanja i događaje u sustavu tako da se tipično slijedi određeni niz koraka opisan u nastavku:

- Utvrđivanje objekta i razumijevanje konteksta – na samom početku modeliranja važno je odrediti što će se konkretno modelirati (a što neće). Dakle, važno je odrediti koji se objekt, komponenta ili dio sustava želi modelirati dijagramom stanja te razumjeti njegovu ulogu, odgovornosti i stanja u kojima se može nalaziti.

- Određivanje osnovnog skupa stanja – nakon što je odlučeno što će se modelirati, potrebno je odrediti osnovni skup stanja u kojima se može naći objekt ili sustav koji se modelira. U ovom koraku nije nužno pokušavati odmah odrediti sva moguća stanja i podstanja, nego nekoliko temeljnih s pomoću kojih je moguće opisati promjenu osnovnih svojstava sustava koji se modelirati. Za svako stanje važno je odrediti ime koje ga dobro opisuje.

- Početno i završno stanje (ako je primjenjivo) – u ovom koraku potrebno je utvrditi koje je od prethodno utvrđenih stanja početno, tj. koje stanje čini početnu točku u kojoj se objekt

stvori ili inicijalizira. Prema potrebi, treba odrediti i završna stanja koja označavaju završetak životnog ciklusa objekta.

- Događaji i prijelazi – nakon što su utvrđena stanja, potrebno je odrediti događaje ili okidače koji mogu uzrokovati prelazak sustava iz jednog stanja u drugo. Događaji mogu biti akcije korisnika, obavijesti sustava ili promjene u okolini. Potrebno je i razmotriti je li za prelazak iz jednog stanje u drugo dovoljan samo događaj ili je nužno zadovoljiti i neki uvjet. Osim događaja i uvjeta, na prijelazu je moguće imati i neku akciju koja se događa pri samom prelasku sustava iz jednog u drugo stanje.

- Razrada stanja – ovisno o složenosti sustava koji se modelira, potrebno je razmotriti postojanje akcija koje se javljaju pri ulasku (*entry*) ili izlasku (*exit*) iz stanja, ili aktivnosti koja se izvršava za vrijeme cijelog stanja (*do*).

- Iterativna dorada – kada je gotova inicijalna verzija dijagrama stanja, potrebno je napraviti evaluaciju i razmotriti jesu li postojećim skupom stanja pokrivene promjene svih atributa (svojstava) sustava koji se modelira. Kada je riječ o složenim sustavima, često je potrebno uvesti ugniježđena podstanja, ortogonalne regije i pamćenje da bi se u potpunosti obuhvatilo ponašanje zadano specifikacijom. Nakon svake promjene u dijagramu potrebno je provjeriti odstupa li modelirano ponašanje od specifikacije i je li moguće doraditi postojeći model da bi se poboljšala njegova jasnoća i čitljivost.

Uz same dijagrame poželjno je po potrebi dodati i dodatni tekstni opis kojim se pojašnjavaju neki složeniji dijelovi modela. Također, kao i kada je riječ o drugim dijagramima, nije nužno uvijek sve prikazati na jednom dijagramu. Da bi se očuvala sažetost i preglednost, složena je stanja moguće detaljno razraditi na dodatnim dijagramima, koji se prilažu uz glavni.

Kako se sustav razvija ili se pojavljuju novi zahtjevi, nužno je ažurirati postojeće dijagrame stanja da bi se uključile promjene u ponašanju. Važno je i tijekom cijelog procesa oblikovanja komunicirati i s ostalim relevantnim dionicima, kao što su programeri, ispitni tim i stručnjaci iz domene, da bi se prikupile povratne informacije i osigurala usklađenost s očekivanim ponašanjem sustava.

6.3 Primjena

UML dijagrami stanja najviše se koriste u postupku oblikovanja sustava radi potpunijeg razumijevanja ponašanja dijelova sustava. Prvenstveno služe za modeliranje dinamičkog ponašanja komponenata sustava te ilustriraju kako objekti prelaze između stanja kao odgovor na različite događaje i tako pomažu u oblikovanju kontrolne logike sustava.

U kontekstu oblikovanja objektno orijentiranih sustava, dijagrami stanja pomažu u definiranju i specifikaciji ponašanja razreda, njihovih atributa i metoda. Pomoću njih je moguće detaljno razraditi ponašanje objekata pojedinog razreda prikazivanjem promjene stanja u odnosu na promjenu vrijednosti atributa i poziv metoda razreda. Također, dijagrami stanja mogu biti vrlo korisni u oblikovanju ponašanja korisničkih sučelja koja mijenjaju izgled i sadržaj (stanje) ovisno o korisnikovim akcijama (događaji).

Osim u postupku oblikovanja, dijagrami stanja mogu biti koristan alat i u ostalim fazama procesa razvoja programske potpore. Primjerice, u procesu izlučivanja i analize zahtjeva, dijagrami stanja mogu se upotrijebiti za detaljnije razrađivanje ponašanja objekata ili dijelova sustava unutar scenarija obrazaca uporabe. Omogućuju detaljno razumijevanje toga kako objekti reagiraju na različite događaje te tako pomažu u preciziranju i usavršavanju zahtjeva.

Nadalje, u fazi implementacije, dijagrami stanja služe kao referenca za programere i vode ih pri implementaciji logike povezane sa stanjima i kontrolnim strukturama u kodu. Isto tako, služe kao referenca pri oblikovanju ispitnih slučajeva kojima se provjerava ispravno ponašanje sustava ili objekata u različitim scenarijima i stanjima. Pomažu u utvrđivanju granica i rubnih slučajeva u kojima prelasci između stanja mogu dovesti do neočekivanog ponašanja.

Konačno, kada je riječ o uvođenju nadogradnji u sustav, dijagrami stanja pomažu u procjeni učinaka promjena na dinamičko ponašanje sustava te osiguravaju da izmjene ne dovedu do neočekivanih posljedica. Općenito se može ustvrditi da dijagrami stanja služe kao važna dokumentacija i referenca za programere tijekom cijelog procesa razvoja programske potpore tako što osiguravaju točno razumijevanje i implementaciju očekivanog ponašanja sustava. No, za razliku od dijagrama obrazaca uporabe ili dijagrama aktivnosti, njihova je sintaksa nešto složenija i zahtijeva određena predznanja iz područja oblikovanja sustava te nisu uvijek pogodni za korištenje među širim skupom dionika. U tablici 6.1 nalazi se sažet i sistematiziran prikaz primjene dijagrama stanja u aktivnostima programskog inženjerstva.

Tablica 6.1: Primjena dijagrama stanja za vrijem različitih aktivnosti programskog inženjerstva

Aktivnost	**Primjena**
Specifikacija programske potpore	Razrada ponašanja objekata ili dijelova sustava unutar scenarija obrazaca uporabe.
Analiza i oblikovanje	Oblikovanje kontrolne logike sustava – modeliranje dinamičkog ponašanja dijelova sustava u vidu stanja i prijelaza između stanja kao odgovor na različite događaje.
Implementacija	Referenca za programere pri implementaciji logike povezane sa stanjima i kontrolnim strukturama u kodu.
Ispitivanje	Pomažu pri oblikovanju ispitnih slučajeva – utvrđivanje granica i rubnih slučajeva u kojima prelasci između stanja mogu dovesti do neočekivanog ponašanja.
Evolucija	Procjena učinaka promjena na dinamičko ponašanje sustava.

7. UML dijagrami aktivnosti

UML dijagrami aktivnosti svestran su alat za modeliranje i vizualizaciju dinamičkog ponašanja sustava. Upotrebljavaju se za prikazivanje tijeka aktivnosti, akcija i odluka procesa u sustavu. Posebno su korisni za vizualizaciju složenih ponašanja, uključujući slijedne i paralelne aktivnosti, kao i točke odlučivanja i uvjetno grananje. Omogućuju programskim inženjerima razumijevanje, komunikaciju i oblikovanje složenih radnih tijekova, procesa i interakcija unutar programskog sustava.

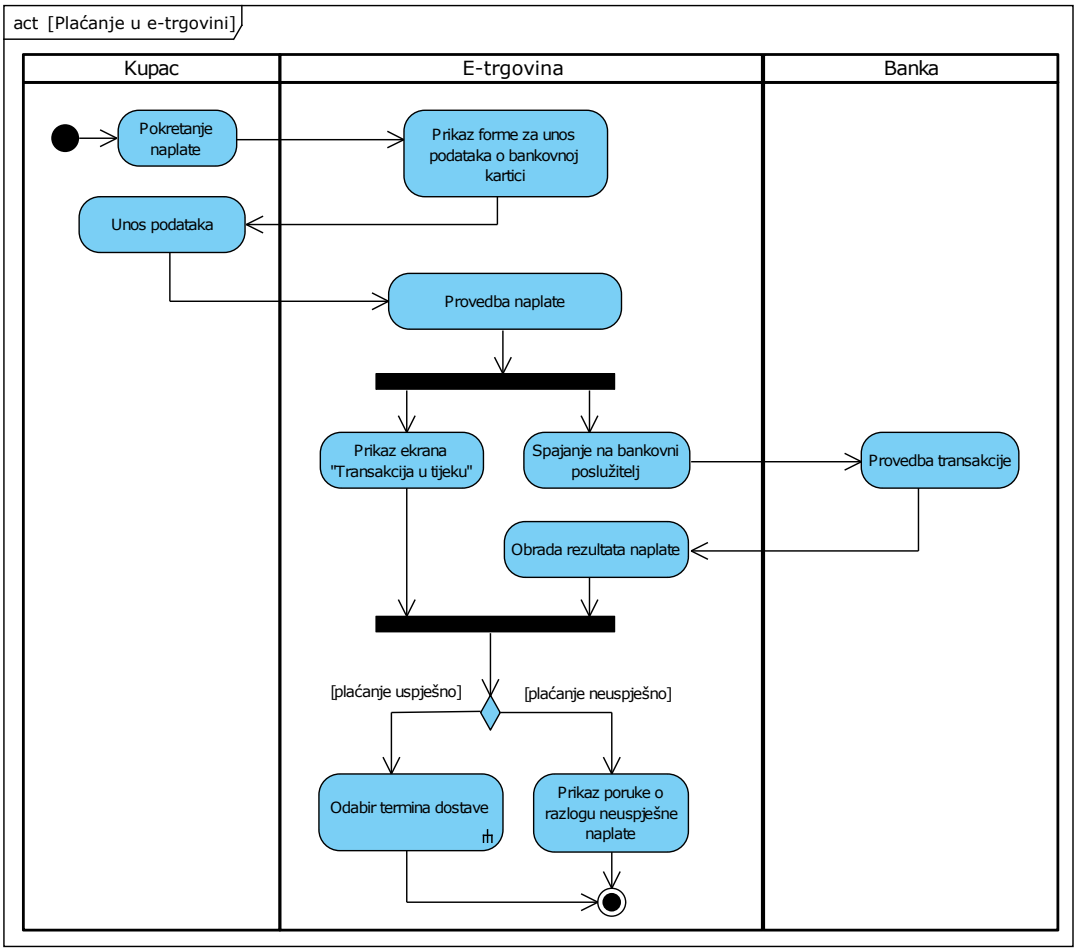
U programskom inženjerstvu, dijagrami aktivnosti nalaze primjenu u različitim fazama procesa razvoja programske potpore. Iako se najviše upotrebljavaju u prikupljanju zahtjeva za modeliranje tijeka interakcija korisnika sa sustavom, tj. detaljniju razradu obrazaca uporabe i scenarija, mogu se upotrijebiti za prikazivanje logike algoritama ili bilo kojeg dinamičkog aspekta programske potpore. Osim toga, često se rabe i za modeliranje poslovnih procesa i upravljanje radnim tijekovima.

Normom UML-a 2.5.1. definira vrlo velik broj semantičkih elemenata koji se mogu upotrijebiti na dijagramima aktivnosti i, u skladu s time, razmjerno složena pravila sintakse. U ovom će priručniku naglasak biti na pregledu temeljnog skupa elemenata i primjerima njihove uporabe u praksi, dok se za naprednije značajke čitatelja upućuje na službenu dokumentaciju UML-a i dodatnu literaturu.

7.1 Definicija i osnovni elementi

UML dijagrami aktivnosti (engl. *activity diagrams*) ponašajni su UML dijagrami koji se u programskom inženjerstvu upotrebljavaju za modeliranje i grafički prikaz dinamičkog ponašanja sustava. Na njima je prikazano izvođenje aktivnosti nizom akcija koje čine upravljačke tijekove i tijekove objekata, s naglaskom na slijed i uvjete tijeka. Pokretanje neke akcije uvjetovano je završetkom prethodne akcije ili više njih ili dostupnošću objekata i podataka. Osnovni elementi dijagrama aktivnosti su:

- **Aktivnost** (engl. *activity*) – određeno ponašanje ili proces unutar sustava. Aktivnosti se prikazuju kao niz čvorova povezanih usmjerenim bridovima koji modeliraju upravljački ili objektni tijek.
- **Čvor** (engl. *node*) – imenovani element koji predstavlja pojedinačni (nedjeljivi) korak



Slika 7.1: Primjer dijagrama aktivnosti

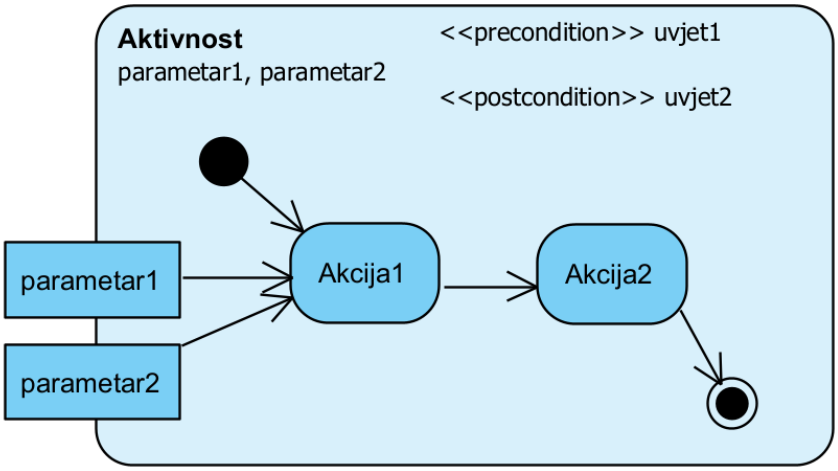
unutar aktivnosti. Postoje tri osnovne vrste čvorova: čvorovi akcije, upravljački čvorovi ili objektni čvorovi. Čvor akcije predstavlja izvođenje određene operacije ili izračuna. Upravljački čvorovi služe za utvrđivanje upravljačkog tijeka (grananje, paralelno izvođenje itd.). Objektni čvorovi upotrebljavaju se za prikaz tijeka objekata, tj. ukazuju na dostupnost instance određene vrste klasifikatora (razred, komponenta...) u određenom trenutku izvođenja aktivnosti.

- **Particija** (engl. *partition, swimlane*) – element dijagrama koji pruža način za vizualno grupiranje i organiziranje čvorova prema odgovornim aktorima, organizacijskim jedinicama ili dijelovima sustava. Particije pomažu razjasniti odgovornosti različitih sudionika i dijelova sustava prikazanih na dijagramu te poboljšavaju razumijevanje toga kako različiti dijelovi sustava surađuju.

Primjer dijagrama aktivnosti za provedbu naplate bankovnom karticom u e-trgovini prikazan je na slici 7.1.

7.1.1 Aktivnosti

Na UML dijagramima aktivnosti, aktivnost je koherentan i dobro utvrđen opis ponašanja ili procesa u nekom programskom sustavu koji podrazumijeva slijed akcija, odluka i drugih elemenata upravljačkog tijeka i tijeka objekata koji zajedno postižu određeni cilj ili funkcionalnost. Aktivnost

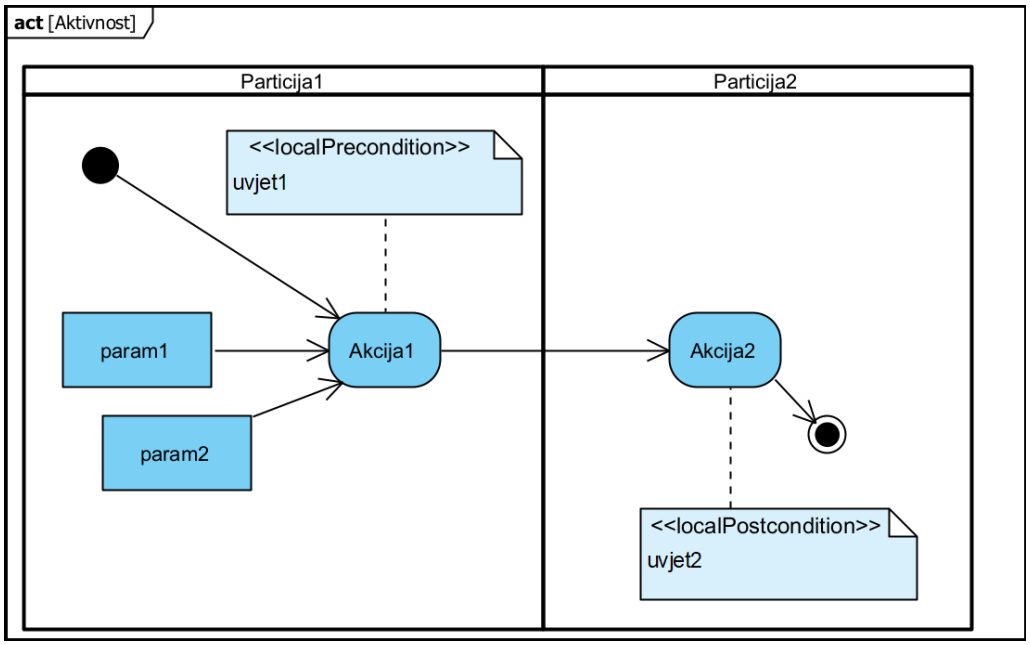


Slika 7.2: Prikaz aktivnosti korištenjem notacije pravokutnika sa zaobljenim vrhovima

kao cjelina obuhvaća razne elemente: čvorove, upravljačke čvorove i čvorove objekata (koji predstavljaju podatke) povezane usmjerenim bridovima. Ti elementi modeliraju tijek izvođenja aktivnosti i tijek podataka (objekata) između njih.

Aktivnost se može vizualno prikazati na dva načina: unutar pravokutnika sa zaobljenim vrhovima ili unutar okvira **act**. Prikaz aktivnosti unutar pravokutnika sa zaobljenim vrhovima, kao što je ilustrirano na slici 7.2, primjenjuje se kada se modeliraju jednostavnije aktivnosti koje sadržavaju manji broj čvorova. Naziv aktivnosti istaknut je podebljanim slovima u gornjem lijevom uglu, a uz njega je moguće zadati i parametre izvođenja aktivnosti te uvjete koji moraju biti zadovoljeni prije (engl. *precondition*) i nakon (engl. *postcondition*) izvođenja aktivnosti. Parametri izvođenja aktivnosti u osnovi su objektni čvorovi koji se povezuju s nekim od čvorova akcije, o čemu će više riječi biti u idućoj sekciji.

Drugi i razmjerno češći način prikaza aktivnosti jest unutar okvira **act** uz uporabu particija,



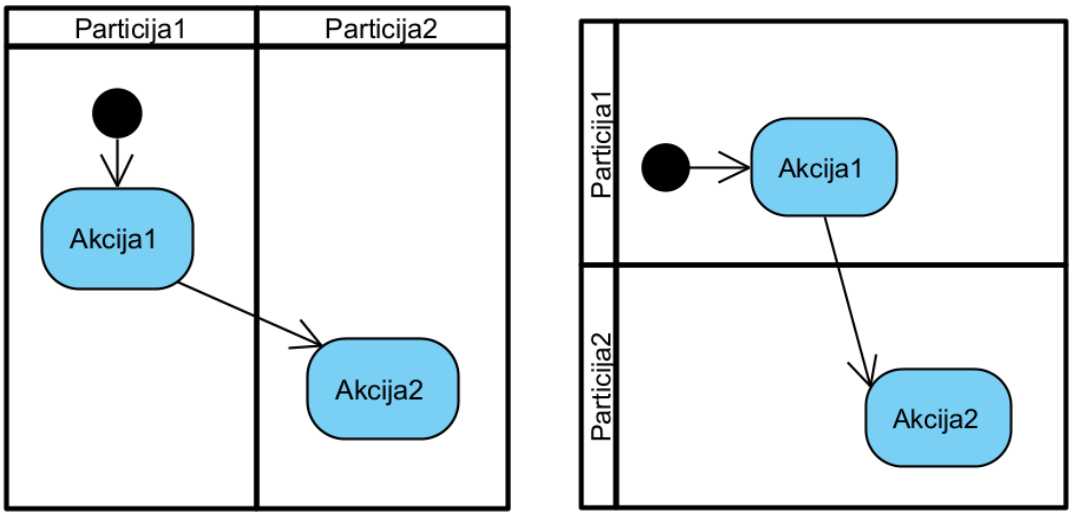
Slika 7.3: Prikaz aktivnosti uz korištenje particija unutar okvira **act**

pomoću kojih se akcije razdvajaju prema aktoru koji ih izvodi, dijelu sustava na kojem se izvode i slično. Takav je način prikaza uobičajen kada se radi o složenijim aktivnostima. Kod ovakvog načina prikaza parametri aktivnosti modeliraju se isključivo kao objektni čvorovi koji se povezuju s odgovarajućim čvorom akcije, a umjesto globalnih uvjeta koji trebaju biti zadovoljeni prije i nakon izvođenja aktivnosti, po potrebi se definiraju lokalni uvjeti (*localPrecondition* i *localPostcondition*), koji se izravno pridružuju akcijama prije, odnosno nakon kojih trebaju biti zadovoljeni. Primjer te notacije ilustriran je na slici 7.3.

7.1.2 Particije

Kada je riječ o prikazu složenijih aktivnosti u koje je uključeno više aktora ili dijelova sustava, poželjno je istaknuti tko/što izvodi koji dio aktivnosti i tako poboljšati čitljivost dijagrama. Na UML dijagramima aktivnosti za to služe particije (engl. *partitions, swimlanes*). Riječ je o vertikalnim ili horizontalnim podjelama unutar dijagrama aktivnosti koje razdvajaju i kategoriziraju akcije na temelju uključenih aktora ili komponenata. Korištenje particija pomaže u jasnom razumijevanju toga koji su aktori ili dijelovi sustava odgovorni za izvođenje kojih dijelova aktivnosti.

Particije se mogu prikazati kao vertikalne (slika 7.4a) ili horizontalne trake (slika 7.4b) sa zaglavljima u kojima su naznačena imena aktora ili komponenata koje predstavljaju. U samoj normi nema razlike u značenju ako se koriste horizontalne ili vertikalne particije, ali se u praksi najčešće upotrebljavaju vertikalne particije, i to za kategoriziraju aktivnosti prema aktorima ili komponentama sustava koje ih izvode, dok se horizontalne particije upotrebljavaju za razdvajanje aktivnosti na temelju njihovih funkcionalnih aspekata ili faza procesa. Svaka particija može predstavljati određenu fazu ili stupanj procesa.



(a) Vertikalne particije (b) Horizontalne particije

Slika 7.4: Prikaz particija na dijagramu aktivnosti u obliku vertikalnih i horizontalnih traka.

7.1.3 Čvorovi

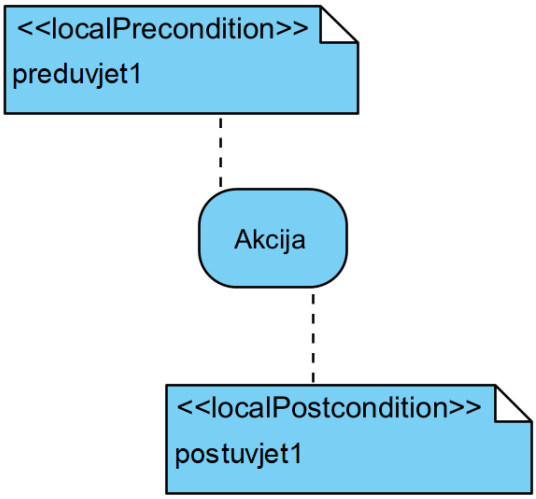
Na dijagramu aktivnosti postoje tri osnovne vrste čvorova: čvorovi akcije, upravljački čvorovi i objektni čvorovi. Svi se čvorovi međusobno povezuju usmjerenim bridovima koji se prikazuju kao ravne crte sa strelicom u smjeru tijeka izvršavanja aktivnosti. U nastavku će se detaljnije objasniti svaka vrsta čvora.

7.1.3.1 Čvorovi akcije

Akcija (engl. *action*) je imenovani element koji predstavlja pojedinačni nedjeljivi korak unutar aktivnosti, tj. koji se dalje ne razlaže unutar te aktivnosti. Čvor akcije modelira određenu radnju, korak u tijeku izvođenja aktivnosti, kao npr. odabir neke opcije unutar korisničkog sučelja, slanje poruke, čekanje na odgovor i sl. Akcije se prikazuju kao pravokutnici sa zaobljenim vrhovima te imenom akcije unutar pravokutnika, kao što je ilustrirano na slici 7.5.

Slika 7.5: Čvor akcije

Za neku akciju mogu biti zadani preduvjeti koji trebaju biti zadovoljeni da bi se ona mogla izvesti, kao i uvjeti koji trebaju biti zadovoljeni nakon njezina izvođenja. Za razliku od uvjeta koji vrijede za cijelu aktivnost, kao što je spomenuto u prethodnom potpoglavlju, ovi su uvjeti lokalni i bilježe se kao komentari sa stereotipom «localPrecondition», odnosno «localPostcondition», povezani s akcijom na koju se odnose crtkanom ravnom crtom, kao što je prikazano na slici 7.6.



Slika 7.6: Zadavanje uvjeta izvođenja akcije

Međutim, u samoj normi još uvijek nije precizno definirano kako modelirati slučajeve u kojima preduvjet ili uvjet nakon izvršavanja nije zadovoljen. tj. što se tada događa s tijekom aktivnosti. Upravo se zato uvjetovano izvođenje akcije najčešće ne modelira lokalnim uvjetima nego korištenjem upravljačkog čvora odluke, odnosno uvjetnog grananja, što će biti prikazano u idućoj sekciji.

Čvorovi akcije međusobno se spajaju usmjerenim bridovima (engl. *activity edge*) čime je utvrđen logički slijed izvođenja akcija unutar aktivnosti. Povezani čvorovi akcije čine jedan upravljački tijek (engl. *control flow*), kao što je ilustrirano na slici 7.7.

Slika 7.7: Povezivanje akcija bridovima u upravljački tijek

Osim za generičke akcije (radnje) čvorovi akcije koriste se i za modeliranje slanja i primanja signala (poruka, događaja) te vremensko čekanje. U tu je svrhu definiran poseban podskup čvorova akcije – **signali**, koji su posebno korisni u scenarijima u kojima su redoslijed aktivnosti i njihove interakcije kritični. Postoje tri vrste signala:

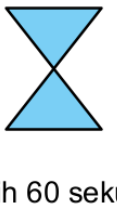
- **Slanje signala** (engl. *send signal action*) – označava da se u sklopu akcije šalje poruka na čiji primitak obično čeka druga akcija. Akcija slanja signala prikazuje se simbolom putokaza, a može imati i stereotip «send signal» ili slično, slika 7.8. Ova vrsta čvora upotrebljava se kada se želi istaknuti komunikacija ili prenošenje informacija između različitih dijelova sustava.

Slika 7.8: Čvor akcije slanja signala

- **Primanje signala** (engl. *accept event action, accept signal action*) – predstavlja primanje signala/poruke koja je prije poslana (akcija slanja signala), slika 7.9. Pokazuje da aktivnost čeka da određeni signal stigne prije nego što nastavi s izvođenjem. Primanje signala upotrebljava se za označavanje toga da je aktivnost pokrenuta dolaskom određenog signala.

Slika 7.9: Čvor akcije primanja signala

- **Vremensko čekanje** (engl. *accept time event action, wait time action*) – označava vremenski događaj (vremenski trenutak ili istek razdoblja) koji generira signal koji onda služi kao okidač za izvođenje sljedeće akcije, slika 7.10.



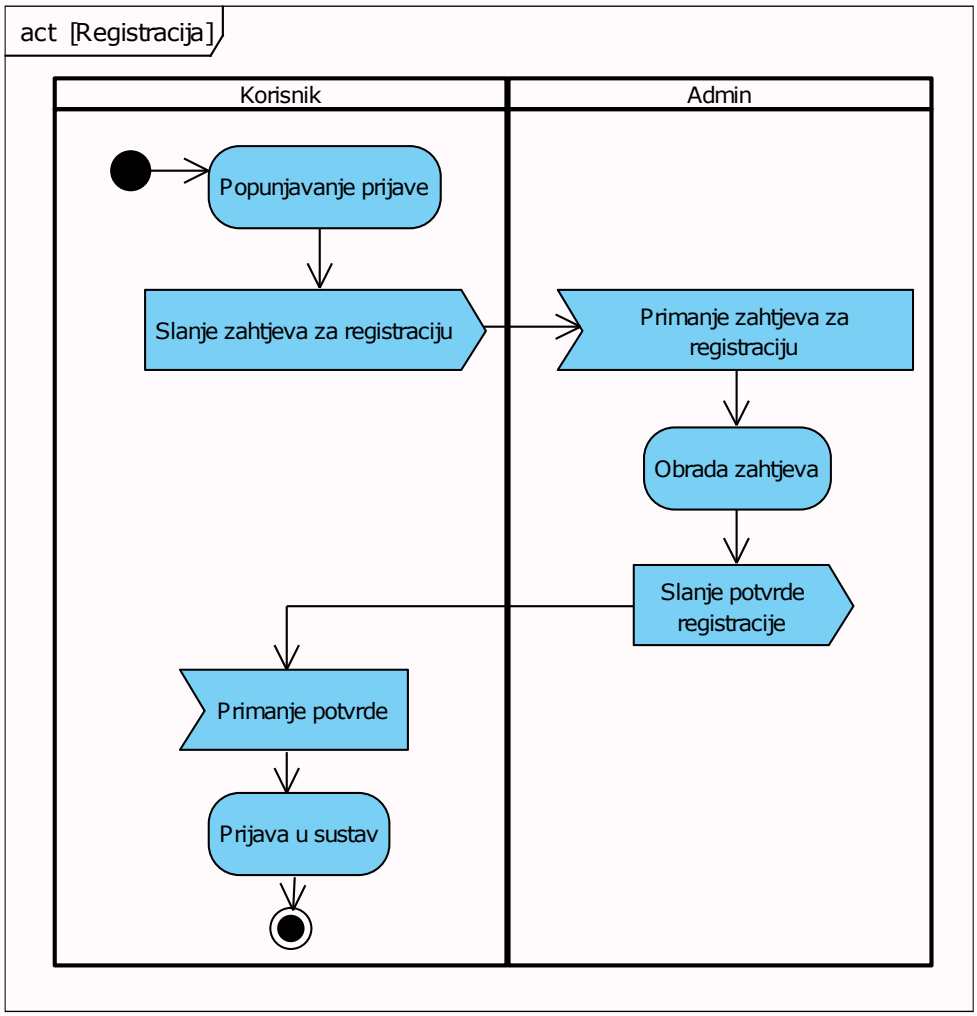
Slika 7.10: Čvor vremenskog čekanja

Za kraj je važno reći da s obzirom na to da su signali specijalna vrsta čvorova akcije, nije neispravno upotrijebiti obične čvorove akcije umjesto signala. Međutim, uporaba signala unosi

dodatne korisne informacije u dijagram i osobito se preporučuje u situacijama u kojima postoji komunikacija između aktora ili dijelova sustava u kojoj je važno istaknuti da jedna strana čeka na primitak poruke od druge ili pak na neki vremenski događaj.

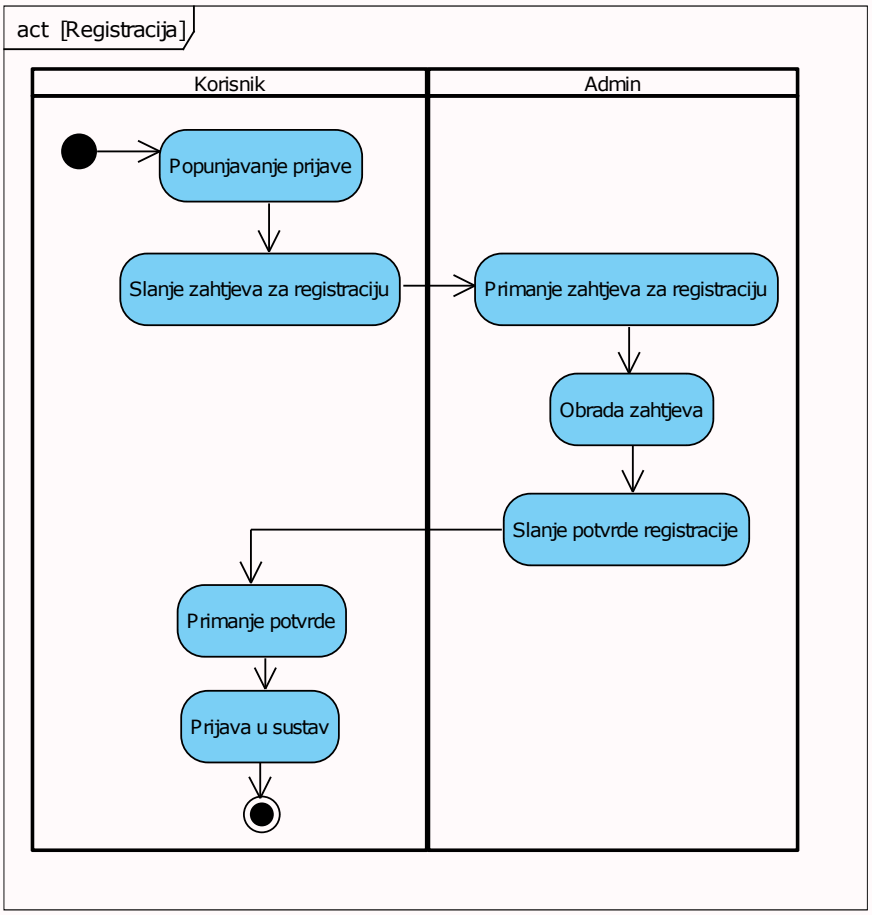
Primjer 7.1 — Primjena signala. Potrebno je modelirati sljedeći opis postupka registracije u nekom sustavu. Korisnik koji želi otvoriti račun u nekoj *web*-aplikaciji treba prvo popuniti prijavu i zatim poslati zahtjev za registraciju. Administrator sustava po primitku novog zahtjeva za registraciju obrađuje zahtjev i šalje korisniku potvrdu o registraciji (zanemariti slučajeve odbijanja). Kada korisnik primi potvrdu da je registriran, prijavljuje se u sustav.

Moguće rješenje – inačica 1: Zadani opis modelira se uz uporabu čvorova za slanje i primanje signala koji ističu postojanje komunikacije između različitih dijelova sustava, slika 7.11.



Slika 7.11: Rješenje postupka registracije uporabom čvorova za slanje i primanje signala.

Moguće rješenje – inačica 2: Zadani opis modelira se bez uporabe signala, samo s čvorovima akcije, slika 7.12.

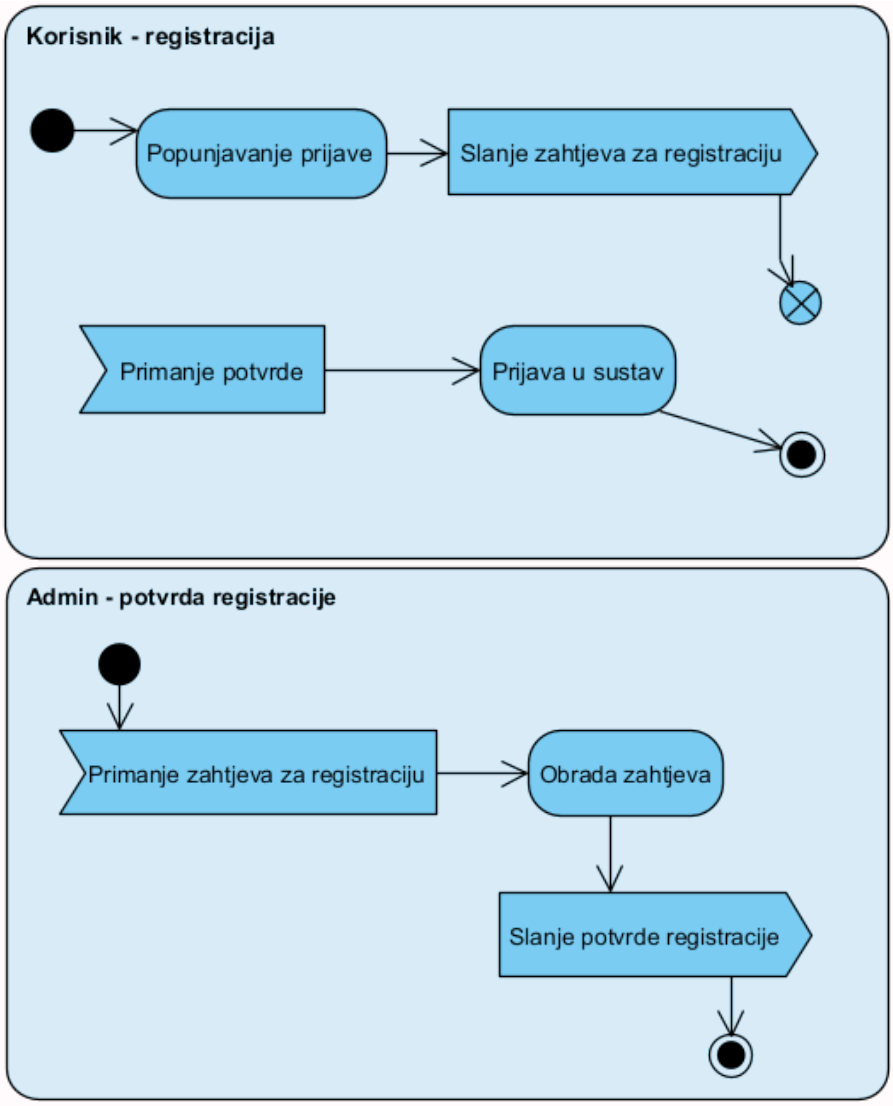


Slika 7.12: Rješenje postupka registracije bez uporabe signala.

Komentar:

Oba su ponuđena rješenja ispravna iako nude različitu razinu informacije. Slanje i primanje signala u osnovi su također akcije i nije neispravno koristiti isključivo čvorove akcije. Međutim, u prvom se rješenju ističe postojanje komunikacije između aktora u sustavu što dalje ukazuje na nužnost vođenja računa o načinu implementacije komunikacijskog kanala, dok kod drugog to nije toliko istaknuto i može se jedino zaključiti iz naziva samih akcija.

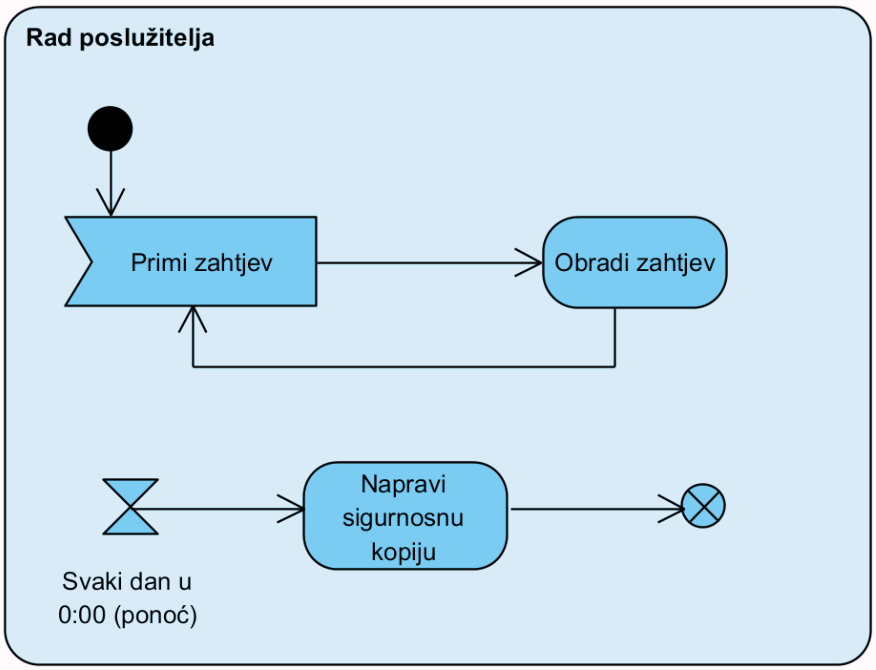
Nadalje, u prvom se rješenju ističe da jedan i drugi aktor moraju čekati na primanje određene poruke prije nego što nastave s radom. Kada bi se za svakog od tih aktora radio zaseban dijagram aktivnosti, kao što je to prikazano na slici 7.13, zamjena signala (osobito signala primanja) običnim čvorovima akcije ne bi imala jednako jasnu semantiku.



Slika 7.13: Prikaz postupka registracije korištenjem zasebnih aktivnosti za svakog aktora.

Primjer 7.2 — Vremensko čekanje – zadano vrijeme. Potrebno je modelirati sljedeći opis rada nekog poslužitelja. Poslužitelj čeka na dolazak zahtjeva. Kada zahtjev stigne, poslužitelj ga obradi i zatim čeka na novi zahtjev. Svaki dan, točno u ponoć, poslužitelj napravi i sigurnosnu kopiju sustava.

Rješenje je prikazano na slici 7.14.

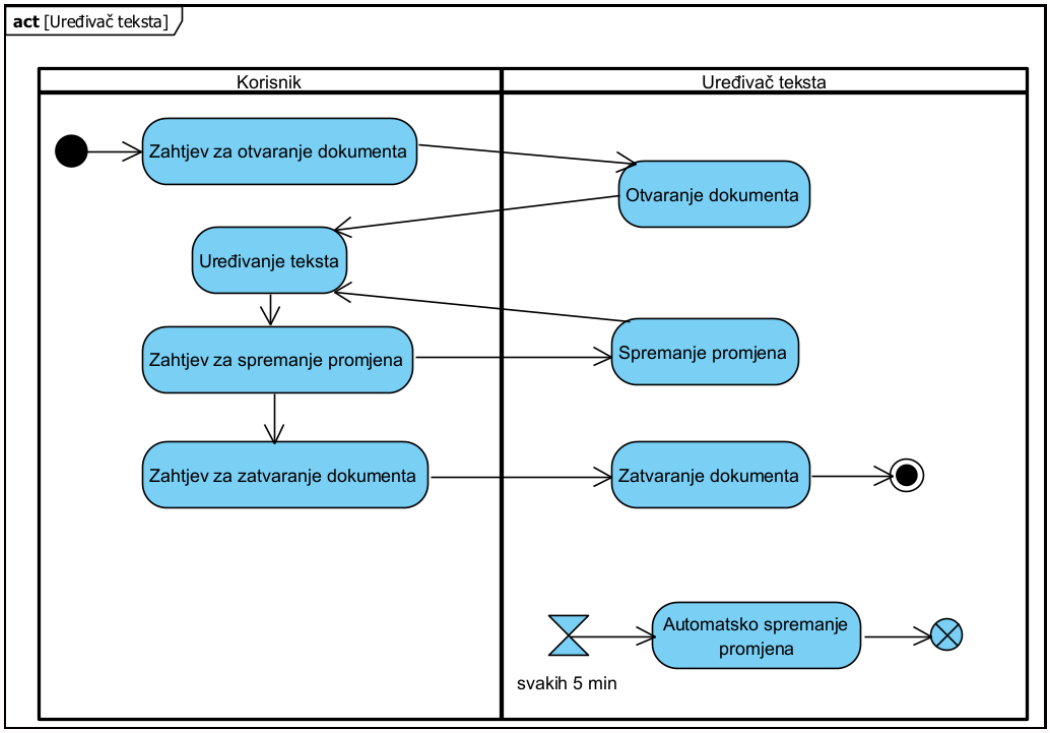


Slika 7.14: Model rada poslužitelja

Komentar: Reakcija na neki vremenski događaj najčešće se prikazuje kao zaseban tijek. Međutim, ovakav način prikaza ne znači nužno da će se akcija pokrenuta vremenskim okidačem izvesti u paraleli s ostatkom rada sustava. Ako se eksplicitno želi istaknuti paralelizam, potrebno je koristiti odgovarajuće upravljačke čvorove račvanja i sinkronizacije, što će biti objašnjeno u idućem potpoglavlju. ■

Primjer 7.3 — Vremensko čekanje – periodičko ponavljanje. Potrebno je modelirati rad korisnika u programu za uređivanje teksta. Korisnik prvo traži otvaranje dokumenta, uređivač teksta zatim otvara dokument i korisnik ga uređuje. Korisnik može zatražiti spremanje promjena, na što program sprema promjene i korisnik nastavlja dalje s uređivanjem teksta. Kada je gotov s uređivanjem, korisnik traži zatvaranje dokumenta i program ga zatvara. Usto, svakih pet minuta program automatski sprema promjene.

Rješenje je prikazano na slici 7.15.

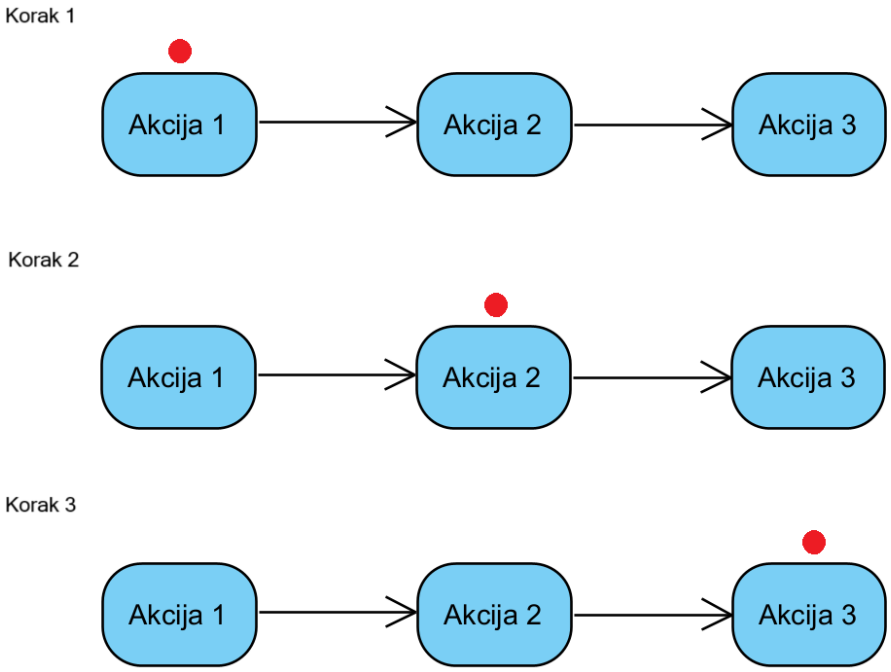


Slika 7.15: Rad korisnika u programu za uređivanje teksta.

7.1.3.2 Upravljački čvorovi

Osnovni upravljački tijek u kojem su akcije povezane bridovima nije dovoljan za modeliranje uvjetnog izvođenja, paralelizma, točaka sinkronizacije itd. U tu je svrhu definirana posebna skupina čvorova – **upravljački čvorovi** (engl. *control nodes*), od kojih su neki preuzeti iz Petrijevih mreža – matematičkog modela za opisivanje konkurentnih sustava. Upravljački čvorovi povezuju se s čvorovima akcije i tako proširuju semantiku i omogućuju modeliranje složenijih upravljačkih tijekova u kojima postoji grananje, ponavljanje, račvanje i skupljanje. Njihovo je korištenje ključno za prikazivanje dinamičkog ponašanja sustava.

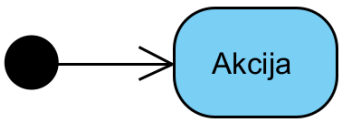
Prije nego što se objasne čvorovi iz skupine upravljačkih čvorova, potrebno je pojasniti koncept **značke** (engl. *token*), koji je također preuzet iz Petrijevih mreža, a koji omogućuje precizno i jednoznačno praćenje tijeka izvođenja. Na dijagramima aktivnosti značka putuje od jednog do drugog čvora po dijagramu i akcija se može izvesti tek kada do nje dođe značka. Nakon što se akcija izvela, ona prosljeđuje značku dalje prema idućem čvoru. Praćenjem kretanje značke moguće je znati trenutačno stanje aktivnosti, tj. napredak procesa. Ilustracija putovanja značke između čvorova akcije prikazana je na slici 7.16. Čvorovi akcije samo prosljeđuju značku dalje, međutim pojedini upravljački čvorovi imaju mogućnost proizvodnje novih znački ili potrošnju postojećih, o čemu će više riječi biti u nastavku.



Slika 7.16: Ilustracija tijeka putovanja značke (crveni krug) od jednog do drugog čvora. Napomena: značka je nacrtana samo ilustrativno. Na dijagramima aktivnosti značke se nikad ne crtaju, nego se njihovo postojanje implicira.

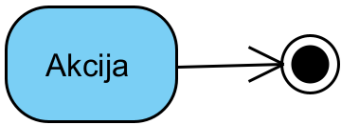
UML dijagram aktivnosti definira ukupno sedam upravljačkih čvorova:

- **Početni čvor** (engl. *initial node*) – označava početnu točku dijagrama aktivnosti, tj. točku od koje započinje upravljački tijek i aktivnost počne s izvođenjem. U pravilu, na dijagramu postoji samo jedan početni čvor spojen s jednim čvorom akcije i on označava početak izvođenja aktivnosti. Ipak, postoje i slučajevi u kojima na jednom dijagramu postoji više početnih čvorova, a tada se radi o postojanju više zasebnih tijekova. U kontekstu znački, početni čvor stvara jednu značku i prosljeđuje je čvoru akcije s kojim je povezan. Početni čvor prikazuje se kao ispunjeni crni krug, slika 7.17.



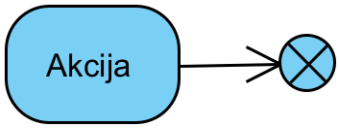
Slika 7.17: Početni čvor

- **Završni čvor** (engl. *activity final node*) – označava krajnju točku dijagrama aktivnosti, tj. završetak kompletne aktivnosti. Konkretno, završni čvor zaustavlja sve aktivne radnje u aktivnosti i uništava sve značke koje postoje unutar dijagrama. Prikazuje se kao ispunjeni crni krug unutar bijelog kruga s crnim obrubom, slika 7.18. Aktivnost može imati više završnih čvorova aktivnosti, a prvi dostignuti čvor zaustavlja sve tijekove u aktivnosti.



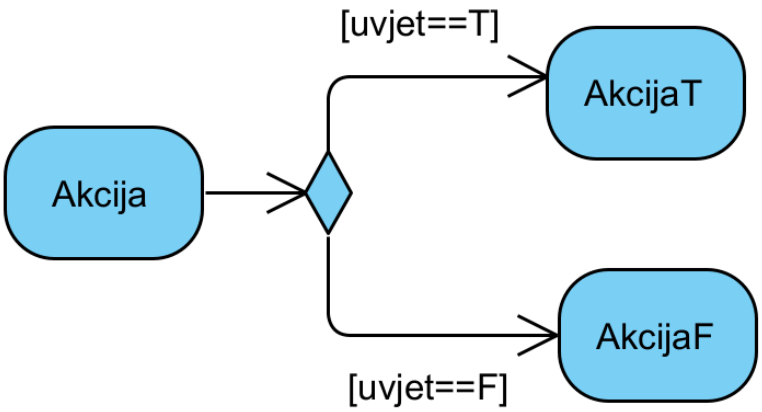
Slika 7.18: Završni čvor

- **Čvor završetka tijeka** (engl. *flow final node*) – označava kraj jednog tijeka, ali nema učinak na ostale tijekove. Sve značke koje dođu do ovog čvora bivaju uništene. Ovaj se čvor koristi kada na dijagramu postoji više paralelnih upravljačkih tijekova te se želi završiti samo jedan od njih. Označava se prekriženim kružnim simbolom, slika 7.19.



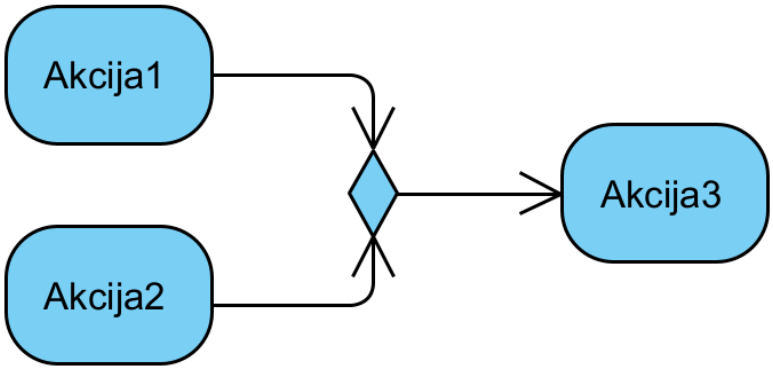
Slika 7.19: Čvor završetka tijeka

- **Čvor odluke** (engl. *decision node*) – označava točku odluke u kojoj se tijek aktivnosti dijeli na više alternativnih putanja na temelju ispunjavanja jednog uvjeta ili više njih. U čvor odluke ulazi jedna značka i iz njega izlazi onim putem za koji je ispunjen uvjet. Čvor odluke prikazuje se simbolom romba s jednim ulaznim bridom i više izlaznih, kao na slici 7.20. Za svaki izlazni brid potrebno je definirati uvjet (unutar uglatih zagrada).



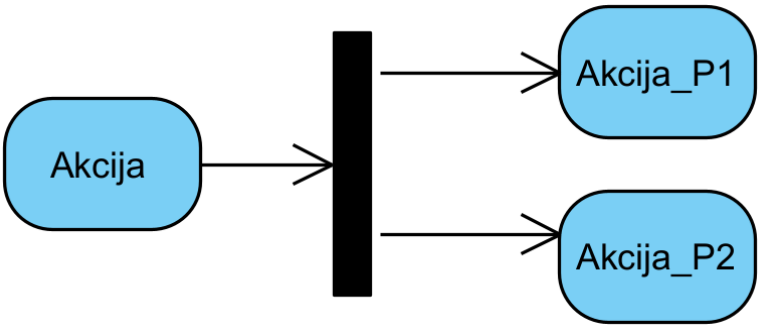
Slika 7.20: Čvor odluke

- **Čvor spajanja** (engl. *merge node*) – predstavlja točku u kojoj se višestruki tijekovi ponovno spajaju nakon što je donesena odluka. Uglavnom se upotrebljava nakon čvorova odluke za ujedinjavanje višestrukih upravljačkih tijekova koji su proizašli iz različitih grana čvora odluke, slika 7.21. Čvor spajanja prikazuje se simbolom romba s više ulaznih bridova i jednim izlaznim.



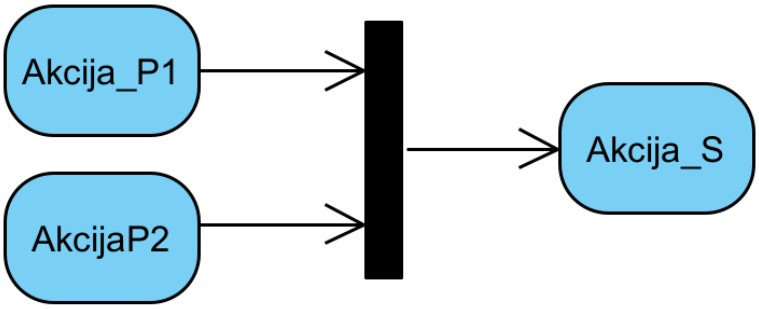
Slika 7.21: Čvor spajanja

- **Čvor račvanja** (engl. *fork node*) – označava točku u kojoj se jedan tijek razdvaja na više paralelnih tijekova, što omogućuje izvođenje više akcija istodobno. Račvanje se prikazuje kao deblja ravna crta u koju ulazi jedan brid, a izlazi više njih, slika 7.22. Ovaj čvor prima jednu značku i umnaža je onoliko puta koliko ima izlaznih tijekova.



Slika 7.22: Čvor račvanja

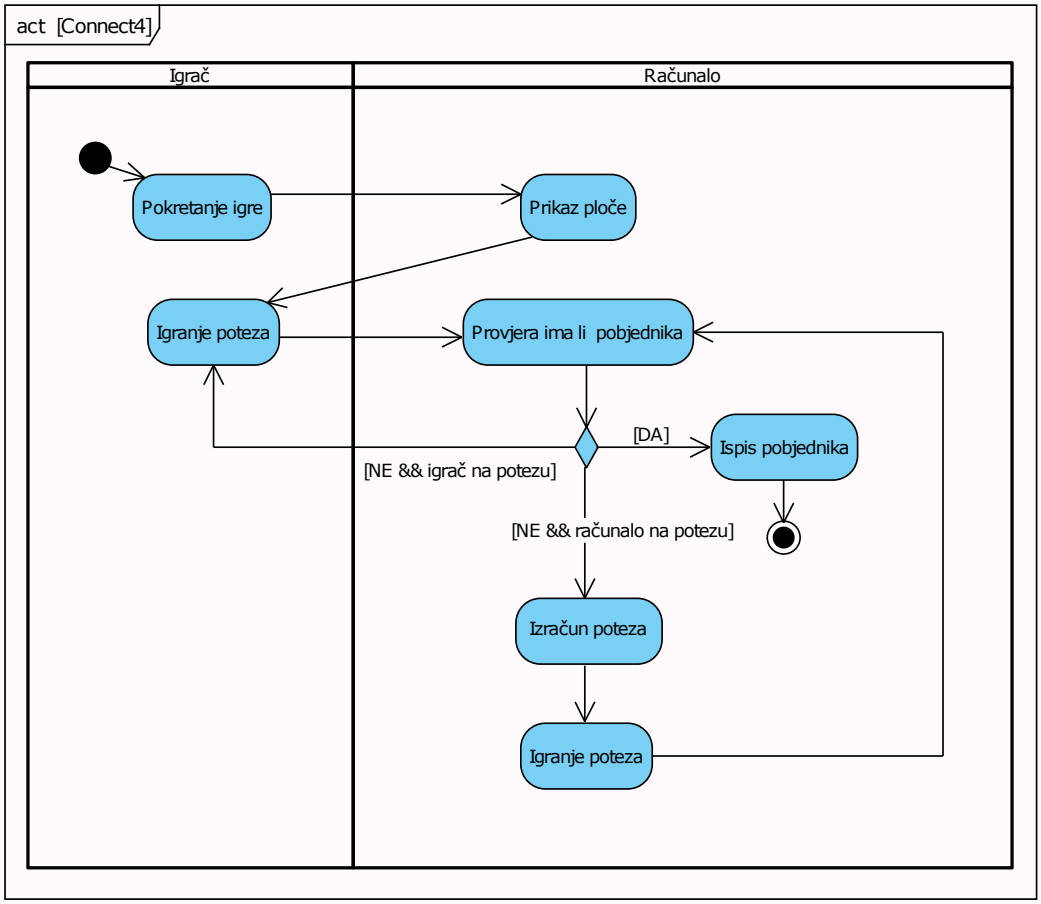
- **Čvor sinkronizacije** (engl. *join node*) – predstavlja sinkronizacijsku točku u kojoj se paralelni tijekovi izvođenja ponovno spajaju u jedan tijek. Nužan uvjet nastavka izvođenja jest da stignu značke iz svih ulaznih tijekova, pri čemu se one spajaju u jednu značku koja se onda šalje na izlaz. Sinkronizacija se prikazuje kao deblja ravna crta u koju ulazi više tijekova, a izlazi samo jedan, kao na slici 7.23.



Slika 7.23: Čvor sinkronizacije

Primjer 7.4 — Igra Connect4 – modeliranje uvjetnog grananja. Potrebno je modelirati osnovni tijek igre Connect4. Igrač i računalo naizmjenično rade poteze. Nakon svakog poteza računalo provjerava ima li pobjednika i, ako ima, igra završava. Inače, kada je računalo na potezu, ono najprije obavlja proračun najboljeg mogućeg poteza te zatim radi potez.

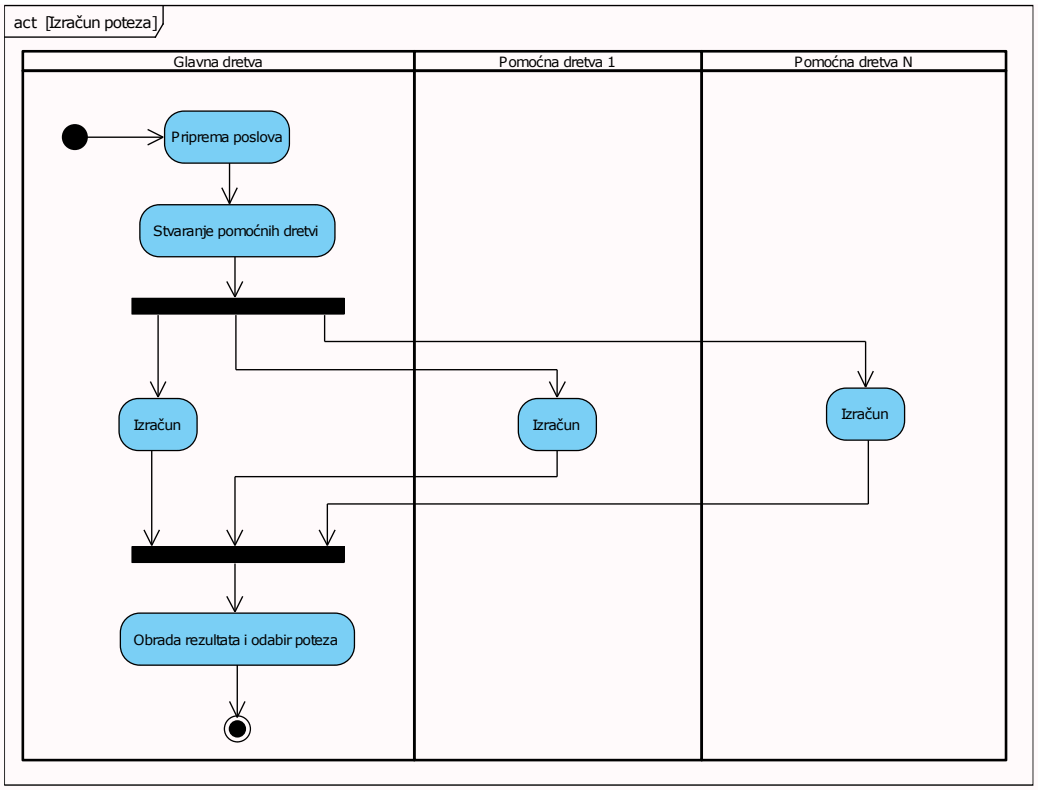
Rješenje je prikazano na slici 7.24.



Slika 7.24: Dijagram aktivnosti za igru Connect4

Primjer 7.5 — Izračun poteza – modeliranje paralelnog izvođenja. Potrebno je modelirati izračun poteza u igri Connect4, koji radi računalo. Izračun poteza paralelizira se na N pomoćnih dretvi. Glavna dretva najprije priprema poslove koje će obrađivati svaka dretva i nakon toga stvara pomoćne dretve. Zatim sve dretve (glavna i pomoćne) računaju svoj dio posla te nakon što su sve gotove, glavna dretva obrađuje rezultate i odabire potez. Pretpostavite da se podaci o poslovima i rezultati izračuna zapisuju u memoriju koja je dostupna svim dretvama.

Rješenje je prikazano na slici 7.25.



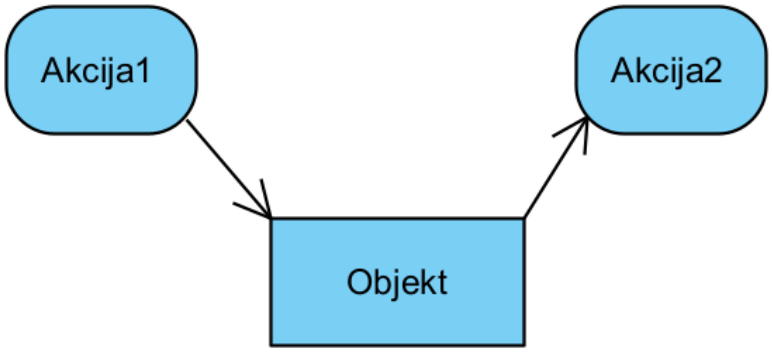
Slika 7.25: Izračun poteza računala za igru Connect4

7.1.3.3 Objektni čvorovi

Izvođenje aktivnosti često uključuje i različite radnje na objektima, kao što su stvaranje i uništavanje objekta, čitanje, izmjenu i sl. Iako se dijagram aktivnosti ponajprije usmjerava na radnje koje sustav izvodi – koje čine upravljački tijek, ništa manje važan nije ni **tijek podataka** koji su sadržani u objektima.

Prema važećoj normi UML-a objektni čvor je podvrsta čvora aktivnosti koji se upotrebljava za utvrđivanje tijeka objekata u aktivnosti. Njegovo postojanje ukazuje na to da bi primjerak određenog klasifikatora (razred, komponenta, predložak...), moguće u određenom stanju, mogao biti dostupan u određenoj točki aktivnosti. Čvorovi objekata mogu se upotrijebiti na različite načine, ovisno o tome odakle objekti dolaze i kamo idu.

Čvorovi objekata označavaju se kao pravokutnici. Ime koje označava čvor smješteno je unutar simbola, gdje ono označava vrstu čvora objekta ili ime i vrstu čvora u formatu „ime:vrsta”. Čvorovi objekata povezuju se s čvorovima akcije usmjerenim bridovima što predstavlja tijek objekata, odnosno prijenos podataka između akcija, slika 7.26. Čvorovi objekata predstavljaju podatke ili objekte koji se prenose.

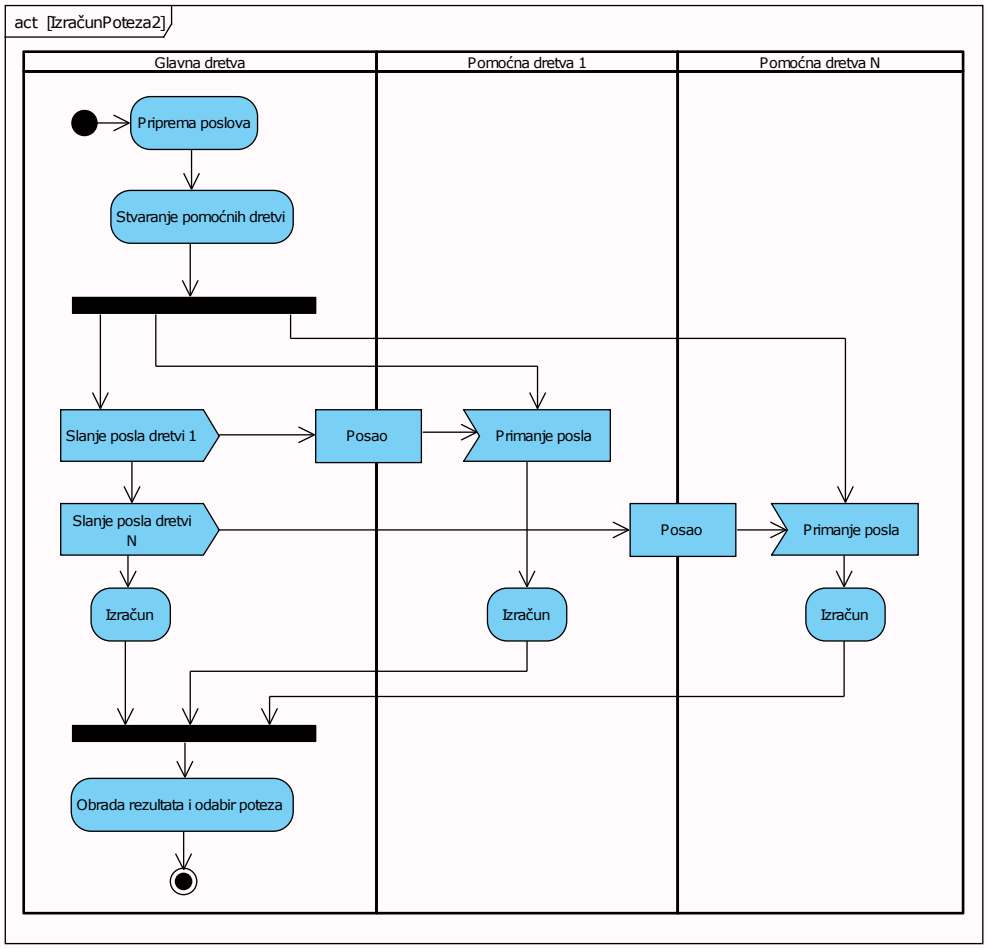


Slika 7.26: Povezivanje objektnog čvora s čvorovima akcije

U kontekstu protoka znački, značke također prolaze kroz objektne čvorove i predstavljaju kretanje podataka ili objekata između akcija. Objektni čvor s dolaznim tijekovima objekata prikuplja značke koje predstavljaju podatke, a akcija koja slijedi konzumira te značke, tj. obavlja operacije nad pristiglim podacima.

Primjer 7.6 — Connect4 – razmjena objekata. Doradite model izračuna poteza iz primjera 7.5 ako se uvede promjena da pomoćne dretve više ne čitaju poslove izravno iz dijeljene memorije, već im se posao šalje eksplicitno. Čitanje rezultata neka i dalje bude iz zajedničke memorije.

Rješenje je prikazano na slici 7.27.



Slika 7.27: Složeniji izračun poteza računala za igru Connect4

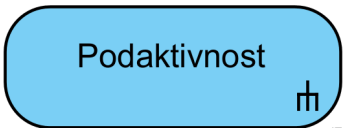
Osim osnovnih objektnih čvorova postoje i dodatni čvorovi kao što su priključnice (engl. *pins*) i spremnici podataka (engl. *central buffer, data store*). Oni omogućuju modeliranje naprednijih značajki, kao, npr. primanje objekata kao ulaznih parametara ili slanje objekata kao rezultat završetka akcije te upravljanje tijekovima iz više izvora i s više odredišta. Međutim, budući da se radi o naprednijim značajkama koje još uvijek nisu u potpunosti konzistentno utvrđene u važećoj normi UML-a, a i njihova uporaba u praksi nije česta, u ovom se priručniku one neće obrađivati, a zainteresirane se čitatelje upućuje na dodatnu preporučenu literaturu.

7.1.3.4 Podaktivnosti

Na UML dijagramima aktivnosti, čvorovi podaktivnosti upotrebljavaju se za prikaz uključivanja jedne aktivnosti unutar druge te promoviraju modularnost i ponovnu iskoristivost. Omogućuju zatvaranje složene logike ili često korištenih slijedova radnji u odvojene podaktivnosti. Čvorovi podaktivnosti pružaju način upravljanja složenošću velikih dijagrama razbijanjem na više upravljivih i razumljivih komponenata. Poboljšavaju jasnoću glavnog dijagrama tako što apstrahiraju detalje uključenih aktivnosti, zbog čega se cjelokupna struktura čini čišćom i razumljivijom.

Čvorovi podaktivnosti povezani su s ostalim čvorovima putem bridova, a njihova uporaba promovira najbolje prakse u oblikovanju programskih sustava, poput odvajanja odgovornosti

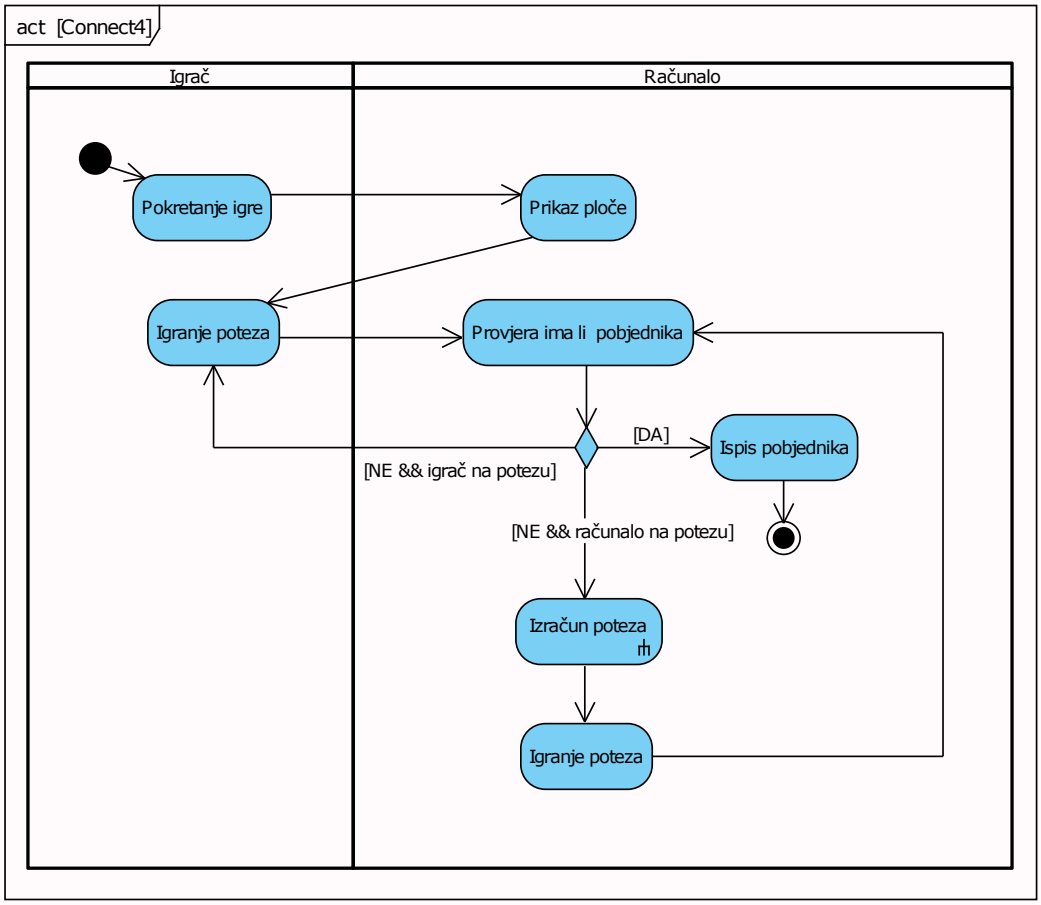
i stvaranja ponovno upotrebljivih građevnih blokova unutar ponašanja sustava. Simbol čvora podaktivnosti prikazan je na slici 7.28.



Slika 7.28: Simbol podaktivnosti

Primjer 7.7 — Connect4 – podaktivnost izračuna poteza. Objedinite model izračuna poteza iz primjera 7.5 s modelom cijele igre iz primjera 7.4.

Rješenje korištenjem čvora podaktivnosti prikazano je na slici 7.29.



Slika 7.29: Dijagram aktivnosti za igru Connect4 korištenjem čvora podaktivnosti za izračun poteza

Komentar: Potrebno je uočiti da se na novom dijagramu „Izračun poteza" više ne prikazuje simbolom akcije, nego simbolom podaktivnosti, koji referencira na dodatni dijagram u kojem je ta aktivnost detaljno prikazana. Tako se izbjegava dodavanje velikog broja novih elemenata na osnovnom dijagramu, što bi ga potencijalno učinilo manje preglednim i teže razumljivim. Pri modeliranju složenih aktivnosti uvijek je važno razmotriti što je u fokusu dijagrama, a sve ostalo izdvojiti na dodatne dijagrame. ■

7.2 Postupak izrade dijagrama

Izrada dijagrama aktivnosti složen je proces čiji detalji uvelike ovise o konkretnom problemu. Međutim, moguće je istaknuti nekoliko osnovnih koraka koji služe kao smjernica i mogu se primijeniti pri svakoj izradi. U nastavku su navedeni osnovni koraci izrade UML dijagrama aktivnosti:

- Određivanje konteksta i utvrđivanje aktivnosti – na samom početku nužno je jasno odrediti za koji se dio sustava, procesa ili funkcionalnosti izrađuje model. Ovisno o složenosti odabranog konteksta moguće je utvrditi jednu aktivnost ili više njih. Također, jedna složena aktivnost može se razbiti na više jednostavnijih. U načelu, svaka se aktivnost modelira na zasebnom dijagramu, a po potrebi se može naznačiti povezanost između dijagrama.

- Određivanje osnovnog skupa akcija – nakon što je odabrana aktivnost koja će se modelirati, potrebno je odrediti temeljne akcije koje će se izvršiti. Treba istaknuti da je izrada dijagrama iterativan proces i nije nužno u prvom koraku odmah pokriti sve detalje. Bitno je najprije postaviti osnovni okvir, a zatim ga nadograđivati.

- Utvrđivanje početnog i završnog stanja – potrebno je odrediti gdje aktivnost počinje i završava te u skladu s time postaviti početne i završne čvorove.

- Razrada upravljačkog tijeka – ako aktivnost uključuje donošenje odluka, potrebno je odrediti točke odlučivanja u kojima tijek može krenuti različitim putanjama na temelju određenih uvjeta. Također, potrebno je razmotriti postojanje dijelova koji će se izvoditi u paraleli te uvesti točke račvanja i sinkronizacije.

- Dodavanje čvorova objekata (ako je potrebno) – ako proces uključuje tijek podataka ili objekata, potrebno je dodati čvorove objekata i povezati ih s odgovarajućim čvorovima akcije u tijek objekata.

- Raspodjela po particijama (opcionalno) – ako se radi o složenijoj aktivnosti u čije je izvođenje uključeno više aktora ili komponenata, savjetuje se raspodjela akcija po particijama.

- Provjera i dorada – nakon oblikovanja osnovnog modela, važno je pregledati dijagram da bi se osigurala točnost i potpunost. Uvijek je nužno voditi računa o semantici dijagrama, tj. da dijagram točno prikazuje namjeravano ponašanje procesa ili sustava. Dijagram treba iterativno poboljšavati i revidirati na temelju povratnih informacija, promjena u procesu ili boljeg razumijevanja.

7.3 Primjena

UML dijagrami aktivnosti primjenjuju se u gotovo svim fazama procesa programskog inženjerstva zbog njihove sposobnosti prikaza dinamičkog ponašanja i tijeka aktivnosti unutar programskog sustava na sažet i lako razumljiv način.

U početnoj fazi razvoja, kada su u tijeku prikupljanje i razrada zahtjeva, dijagrami aktivnosti koriste se za detaljno modeliranje ponašanja pojedinačnih obrazaca uporabe. Oni pomažu u razumijevanju tijeka aktivnosti i interakcija između aktora i sustava.

Dijagrami aktivnosti koriste se i za modeliranje ponašanja i oblikovanje algoritama u procesu oblikovanja programskog sustava. Kada je riječ o razradi dinamičkog ponašanja sustava, dijagrami aktivnosti mogu se koristiti za modeliranje interakcija između komponenata, razreda ili modula.

Također, mogu biti vrijedan alat za detaljan prikaz tijeka složenih algoritama ili logike unutar metoda ili procesa.

Nadalje, detaljni dijagrami aktivnosti pružaju jasan plan implementacije različitih aktivnosti i zadataka te se koriste kao referenca pri pisanju programskog koda. Pomažu i u oblikovanju ispitnih scenarija koji su usmjereni na dinamičko ponašanje sustava tako što prikazuju očekivani slijed radnji, uključujući odlučujuće točke i alternativne tijekove. Osim navedenog, dijagrami aktivnosti koristan su alat za izradu korisničkih priručnika i edukacijskih materijala kojima se objašnjavaju koraci koje korisnik treba slijediti da bi u programskom sustavu mogao izvršiti određeni zadatak. Pri tome treba voditi računa o predznanju i iskustvu korisnika te po potrebi koristiti semantički jednostavnije i razumljivije elemente dijagrama a izostaviti one naprednije sa složenom sintaksom i semantikom.

Konačno, dijagrami aktivnosti mogu se koristiti i za vizualizaciju poslovnih procesa pri čemu pomažu u procesu analize i optimizacije poslovnih procesa, tako što prepoznaju uska grla, suvišnosti i prilike za poboljšanje. U tablici 7.1 nalazi se sažet i sistematiziran prikaz primjene dijagrama aktivnosti u različitim aktivnostima procesa razvoja programske potpore.

Tablica 7.1: Primjena dijagrama aktivnosti za vrijem različitih aktivnosti programskog inženjerstva

Aktivnost	**Primjena**
Specifikacija programske potpore	Detaljno modeliranje ponašanja pojedinačnih obrazaca uporabe – tijek aktivnosti i interakcija između aktora i sustava.
Analiza i oblikovanje	Detaljan prikaz tijeka složenih algoritama ili logike unutar metoda ili procesa.
Implementacija	Referenca za implementaciju algoritama i upravljačke logike.
Ispitivanje	Oblikovanje ispitnih slučajeva vezanih za grananja i točke odlučivanja.
Evolucija	Dokumentacija i komunikacija s dionicima.

8. UML dijagrami komponenti

UML dijagrami komponenti koriste se za modeliranje arhitekture programskog sustava na visokoj razini. Pružaju jasan i sažet način vizualizacije različitih komponenti ili građevnih blokova sustava i njihovih odnosa, te ih tako čine vrijednim alatom u programskom inženjerstvu.

Glavni su elementi dijagrama komponente, sučelja i veze između komponenti. Komponente predstavljaju modularne dijelove sustava poput programskih modula, knjižnica, radnih okvira i slično. Sučelja definiraju ugovor ili komunikacijski protokol između komponenti tako što specifiraju metode, operacije ili poruke koje izlažu.

UML dijagrami komponenti igraju važnu ulogu u oblikovanju sustava jer omogućuju analizu organizacije sustava, utvrđivanje ključnih komponenata i razumijevanje njihove interakcije. To pomaže u donošenju odluka u oblikovanju, optimizaciji strukture sustava te osiguranju njegove skalabilnosti, fleksibilnosti i održivosti.

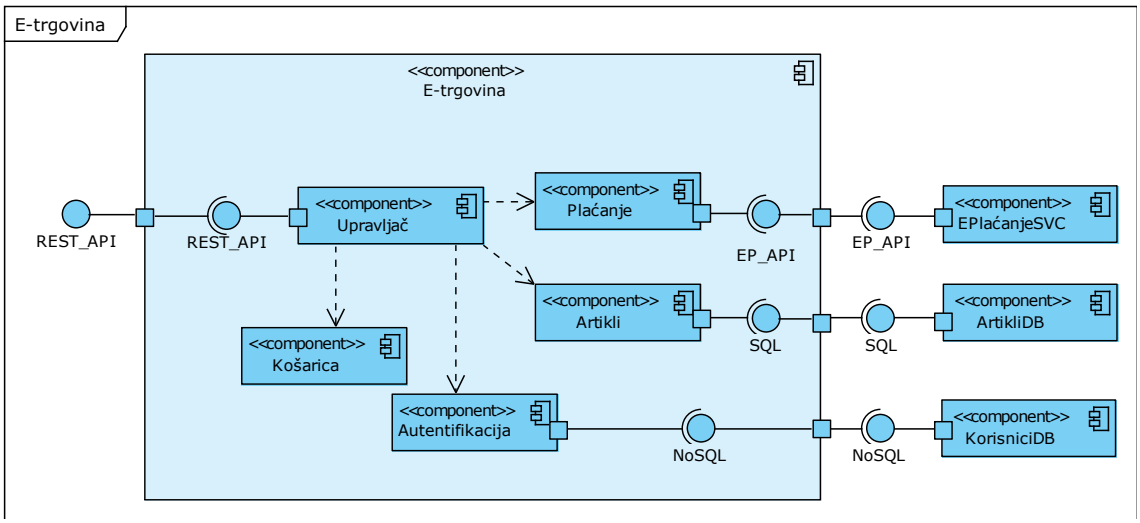
Osim toga, UML dijagrami komponenti pomažu u promicanju modularnosti i ponovne uporabe u razvoju programske potpore. Razdvajanjem sustava na kohezivne komponente postiže se bolja separacija odgovornosti, enkapsulacija i ponovna upotrebljivost koda. Komponente se mogu razvijati i ispitivati neovisno te tako olakšavaju paralelni razvoj i smanjuju ovisnosti između razvojnih timova.

8.1 Definicija i osnovni elementi

UML dijagrami komponenti (engl. *component diagrams*) vrsta su strukturnih UML dijagrama koji prikazuju organizaciju i odnose komponenti koje čine programsku potporu. Pružaju vizualni prikaz arhitekture sustava i ističu modularnu strukturu i interakcije između komponenti. Osnovni elementi svakog dijagrama su:

- **Komponente** – zasebna enkapsulirana cjelina programske potpore koja ostvaruje određenu funkcionalnost te komunicira s drugim komponentama putem dobro utvrđenih sučelja.
- **Sučelja** – imenovan skup javno vidljivih atributa i apstraktnih operacija koje komponenta pruža drugim komponentama ili ih zahtijeva.
- **Veze** – modeliraju odnose između komponenti i obuhvaćaju spojnice, delegacije i ovisnosti.

Primjer UML dijagrama komponenti prikazan je na slici 8.1.



Slika 8.1: Primjer UML dijagrama komponenti

U nastavku su detaljnije pojašnjeni osnovni elementi dijagrama komponenti.

8.1.1 Komponente

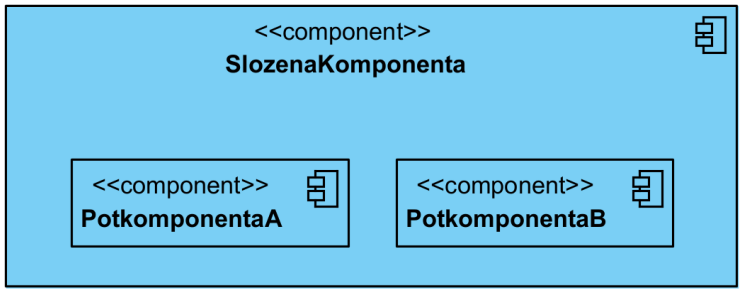
Komponente predstavljaju građevne blokove ili jedinice sustava, tj. služe za modeliranje fizičkih ili logičkih entiteta na različitim razinama apstrakcije: od razreda do većih programskih modula, knjižnica, podsustava i sl. Logičke komponente apstraktna su reprezentacija poslovne logike, usluga koje pruža neki sustav, repozitoriji podataka itd. Fizičke su komponente konkretni radni okviri ili knjižnice, kao npr. EJB (Enterprise Java Beans), .NET, pandas itd. Svim je komponentama zajedničko da enkapsuliraju određeni skup funkcionalnosti koji čini kohezivnu cjelinu i da komuniciraju s drugim komponentama isključivo putem vanjskih sučelja.

Komponente se prikazuju kao pravokutnici koji u gornjem dijelu sadržavaju naziv komponente te iznad njega stereotip «component». Također, moguće je u gornjem desnom uglu prikazati i malu ikonu koja predstavlja komponentu (simbol komponente iz verzije UML-a 1.4). Ako se prikaže ikona, moguće je izostaviti stereotip «component» ili ga zamijeniti nekim specifičnijim, namjenskim stereotipom (slika 8.2) koji pobliže opisuje vrstu komponente, kao npr.: «entity», «process», «service», «file», «library», «subsystem» itd.

(a) (b)

Slika 8.2: Primjeri prikaza komponente s (a) uobičajenim stereotipom i (b) namjenskim stereotipom

Komponenta može imati složenu strukturu, tj. unutar nje mogu biti ugniježđene druge potkomponente, kao što je prikazano na slici 8.3. Unutarnja struktura može dodatno sadržavati i druge elemente, poput razreda, sučelja, priključaka i spojnica, koji predstavljaju unutarnje funkcionalnosti i odnose komponenti.



Slika 8.3: Složena komponenta

8.1.2 Sučelja i priključci

Izolirana komponenta ima ograničenu korist i postaje relevantna tek kada uspostavlja komunikaciju s ostatkom svijeta. Kao apstraktna enkapsulirana cjelina koja može sadržavati brojne dijelove i imati složenu strukturu, komponenta je po definiciji zatvorena prema vanjskom svijetu, što znači da njezina unutarnja svojstva i funkcionalnosti nisu dostupni izvana. Da bi se moglo komunicirati s vanjskim svijetom, potrebno je eksplicitno utvrditi pristupne točke.

U tu se svrhu koriste **sučelja** (engl. *interface*) i **priključci** (engl. *port*). Sučelja definiraju ugovore i komunikacijske protokole koje komponente izlažu za interakciju s drugim komponentama u sustavu. Ona specificiraju skup operacija, metoda i poruka koje komponenta može pružiti ili zahtijevati od drugih komponenti. Priključci su točke kontakta s vanjskim svijetom, odnosno mjesta na kojima komponenta javno izlaže svoje određene unutarnje dijelove i omogućuje slanje i primanje poruka ili podataka između komponenti. Priključci se prikazuju kao mali pravokutnici na granici komponente, koji mogu biti imenovani ili neimenovani. Na priključke se mogu nadovezati sučelja, koja preciznije određuju način komunikacije komponente s vanjskim svijetom. Ona određuju hoće li komponenta pružati usluge drugim komponentama ili će koristiti i pozivati operacije drugih komponenti.

Kada komponenta ostvaruje skup funkcionalnosti koje druge komponente mogu koristiti, onda ona ima **ponuđeno sučelje** (engl. *provided interface*) putem kojeg izlaže funkcionalnost. To omogućuje drugim komponentama pristup i pozivanje izloženih operacija ili metoda. Ponuđeno sučelje prikazano je kao mali krug spojen s granicom komponente, uz koji stoji naziv sučelja.

S druge strane, kada komponenta za svoj rad koristi (poziva) operacije koje neka druga komponenta ostvaruje (izlaže putem ponuđenog sučelja), tada deklarira tzv. **zahtijevano sučelje** (engl. *required interface*). Zahtijevano sučelje prikazuje se kao polukružnica i ukazuje na ovisnost komponente (koja ga deklarira) o drugoj komponenti za pružanje potrebne funkcionalnosti, kao što je prikazano na slici 8.4.

Slika 8.4: Primjer ponuđenog i zahtijevanog sučelja

Do inačice UML-a 2.5 sučelja su se izravno spajala na komponentu bez priključaka, kao što je prikazano na slici 8.5. Ta se notacija još uvijek često koristi u praksi, ali bi se formalno trebala

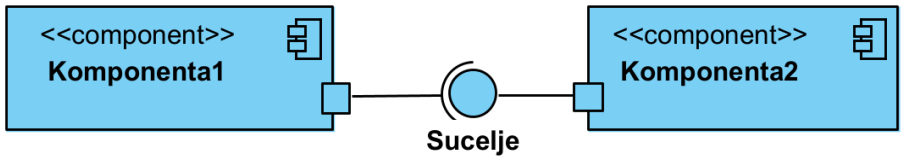
primjenjivati samo kada je riječ o jednostavnoj komponenti koja izravno implementira sučelje, istovjetno tome kako razredi implementiraju sučelja. Za složenije komponente, bez obzira na to prikazuje li se njihova interna struktura, sučelja se spajaju na komponentu putem priključaka.

Slika 8.5: Primjer notacije ponuđenog i zahtijevanog sučelja bez korištenja priključaka

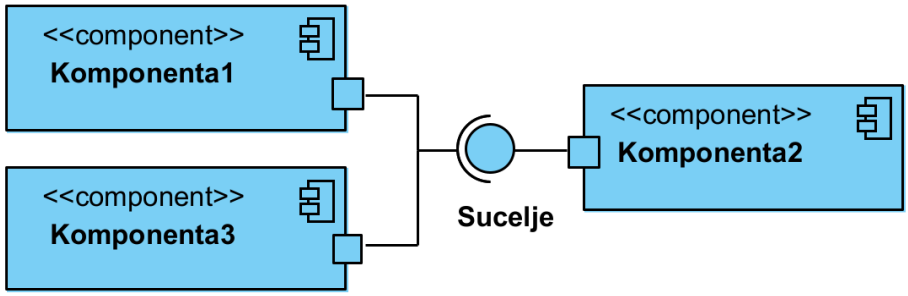
8.1.3 Veze

Veze između komponenti moguće je prikazati na tri načina: spojnicom, ovisnošću i delegacijom. Spojnice i ovisnosti koriste se za modeliranje izravne veze između dviju komponenti, a delegacija se koristi pri povezivanju komponenti sa složenom unutarnjom strukturom.

Spojnica (engl. *assembly connector*) na dijagramima komponenti koristi se za prikazivanje komunikacije ili interakcije između dviju komponenti. Ona je točka veze u kojoj se dvije komponente spajaju da bi omogućile protok informacija ili funkcionalnosti između njih. Spojnica se primarno koristi za povezivanje dvaju sučelja (zahtijevanoga s ponuđenim) i prikazuje se tzv. *ball-and-socket* notacijom, prikazanom na slici 8.6a. Moguće je i da jedna komponenta implementira sučelje koje koristi više komponenata, pa se tada spojnica prikazuje kao na slici 8.6b.



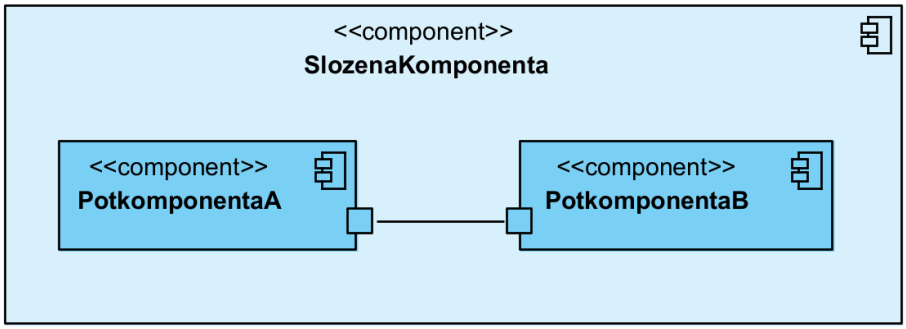
(a) Spojnica između sučelja dviju komponenti



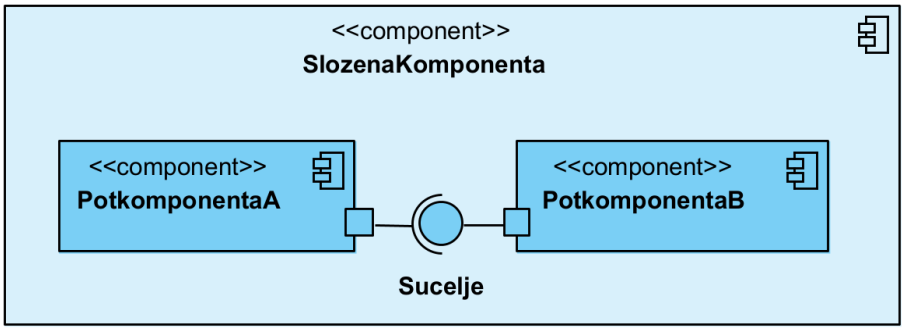
(b) Spojnica između ponuđenoga sučelja koje pruža jedna komponenta i dvaju istovrsnih zahtijevanih sučelja drugih dviju komponenti

Slika 8.6: Primjeri spojnice

Spojnica se može koristiti i za izravno povezivanje unutarnjih dijelova (potkomponenti), i to na dva načina. Ako unutarnje komponente nemaju definirana sučelja, povezuju se izravno priključci, kao što je prikazano na slici 8.7a. Ako unutarnje komponente imaju eksplicitno definirana sučelja, koristi se *ball-and-socket* notacija, kao što je prikazano na slici 8.7b.



(a) Spojnica između dviju unutarnjih komponenti s definiranim priključcima

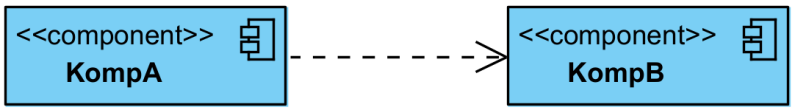


(b) Spojnica između dviju unutarnjih komponenti s definiranim sučeljima

Slika 8.7: Prikaz spojnice unutar složene komponente

Ovisnost (engl. *dependency*) prikazuju odnose između dviju komponenti u kojima jedna komponenta ovisi o drugoj za izvršavanje neke svoje funkcionalnosti. Preciznije, ovisna komponenta zahtijeva usluge, sučelja ili ponašanje koje pruža neka druga komponenta o kojoj ovisi.

Slično kao i na dijagramu razreda, ovisnost je vrlo apstraktna veza koja općenito ukazuje na to da promjena u jednoj komponenti može utjecati na drugu bez preciziranja detalja komunikacije (spajanja) komponenti, kao što je to npr. slučaj kada je riječ o spojnici između zadanih sučelja. Ovisnost se prikazuje iscrtkanim strelicama čiji smjer pokazuje iz ovisne komponente prema komponenti o kojoj ovisi, slika 8.8.



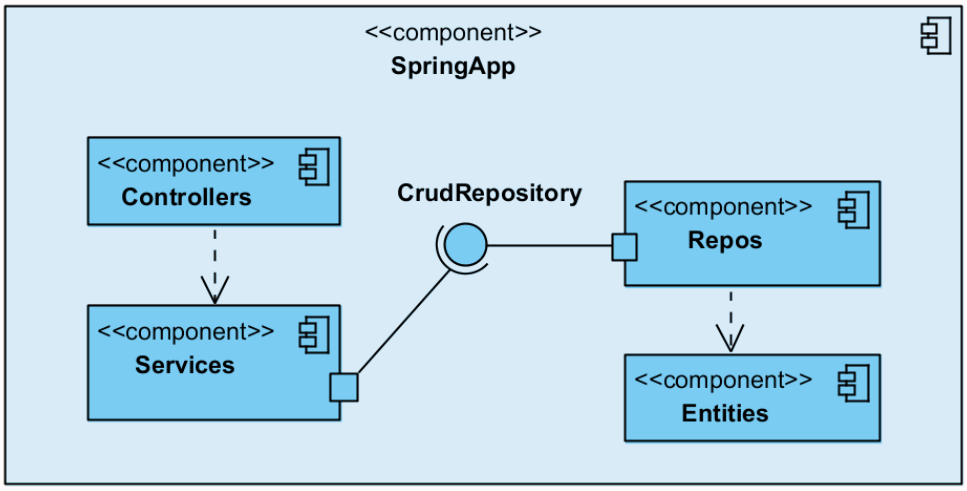
Slika 8.8: Prikaz ovisnosti između dviju komponenti.

Ovisnosti između komponenti utječu na modularnost, održavanje i fleksibilnost programskog sustava jer promjene u nekoj komponenti mogu zahtijevati promjene ili prilagodbe u drugim komponentama koje o njoj ovise. Ako detalji interakcije komponenti nisu poznati (ili nisu važni za istaknuti), nužno je barem naznačiti ovisnosti jer one pomažu u razumijevanju odnosa i međuovisnosti između komponenti tako što olakšavaju analizu, oblikovanje i održavanje sustava.

Primjer 8.1 — Sučelja i ovisnosti. Potrebno je modelirati unutranju strukturu aplikacije implementirane korištenjem radnog okvira Java Spring. Svi razredi aplikacije grupirani su u jednu od četiri potkomponente: Controllers, Services, Repos i Entities. Komponenta Controllers zadužena je za obradu HTTP zahtjeva na temelju kojih poziva operacije komponente Services.

Komponenta Services za izvođenje pojedinih operacija treba pristup podacima iz baze, za što koristi operacije komponente Repos. Komponenta Repos specifična je po tome što implementira operacije CRUD (*create*, *read*, *update*, *delete*), definirane u sučelju CrudRepository, pomoću kojih je onda moguće raditi s razredima iz komponente Entities.

Rješenje je prikazano na slici 8.9.



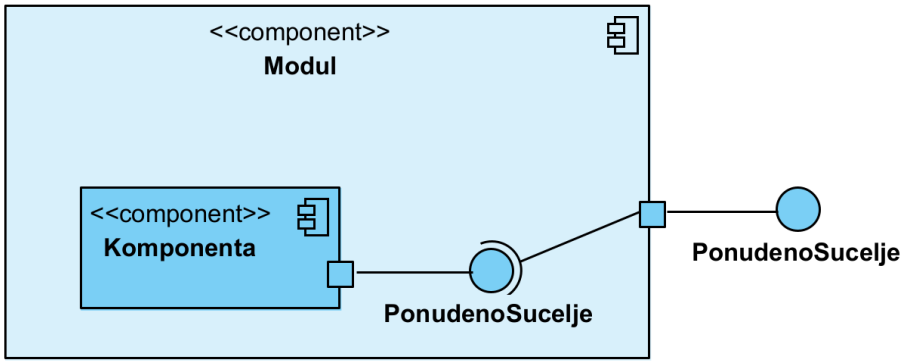
Slika 8.9: Unutarnja struktura aplikacije implementirane u radnom okviru Java Spring.

Komentar:

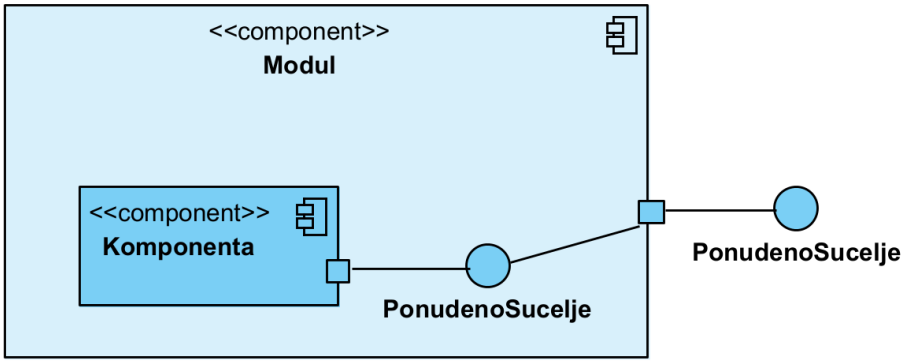
Budući da nije precizno utvrđeno kako komponente Controllers i Services međusobno komuniciraju, dovoljno ih je međusobno povezati vezom ovisnosti. Ista je situacija i između komponenti Repos i Entities. No, kada je riječ o komponentama Services i Repos, jasno je rečeno da komponenta Repos implementira sučelje CrudRepository, pa je tada nužno to sučelje i naznačiti te koristiti spojnicu (engl. *assembly connector*) za povezivanje tih dviju komponenata. Konačno, budući da ništa konkretno nije izrečeno o načinu na koji aplikacija prima HTTP zahtjeve i prosljeđuje ih komponenti Controllers, a fokus dijagrama je na unutarnjoj strukturi aplikacije, vanjska sučelja nisu modelirana. No, ako se želi, može se naznačiti ponuđeno HTTP sučelje na unutarnjoj komponenti Controllers i delegirati ga na priključak vanjske komponente (SpringApp).

Delegacija (engl. *delegation*) je odnos između dviju komponenti u kojem se odgovornost ili funkcionalnost delegira od jedne komponente drugoj. Koristi se pri prikazu unutarnje strukture komponente da bi se naznačilo da se ponuđeno ili zahtijevano sučelje neke unutarnje komponente otvara prema van (engl. *expose*).

Za delegiranje sučelja unutarnje komponente prema van vanjska komponenta mora utvrditi priključak kao točku interakcije s unutarnjom komponentom. Uz to se definira i sučelje na vanjskoj komponenti koje je istovjetno sučelju unutarnje komponente (koje se želi otvoriti prema van), i konačno se na priključak spaja unutarnje sučelje. Za spajanje s unutarnjim sučeljem može se koristiti notacija *ball-and-socket* ili se unutarnje sučelje može izravno spojiti na priključak. Primjeri delegacije za ponuđeno i zahtijevano sučelje, s obje vrste notacije, prikazani su na slikama 8.10 i 8.11.

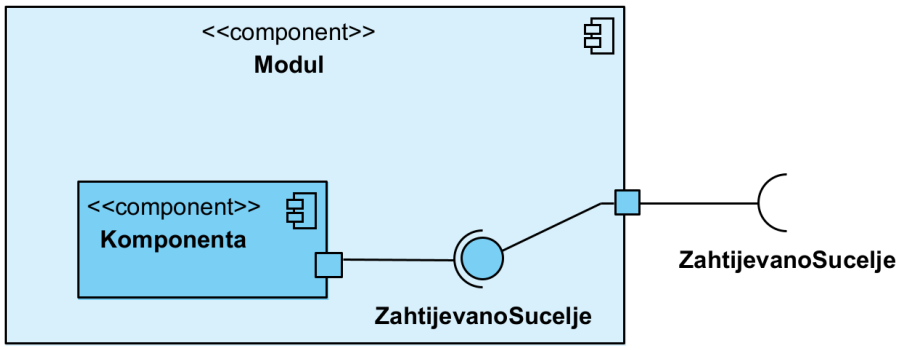


(a)

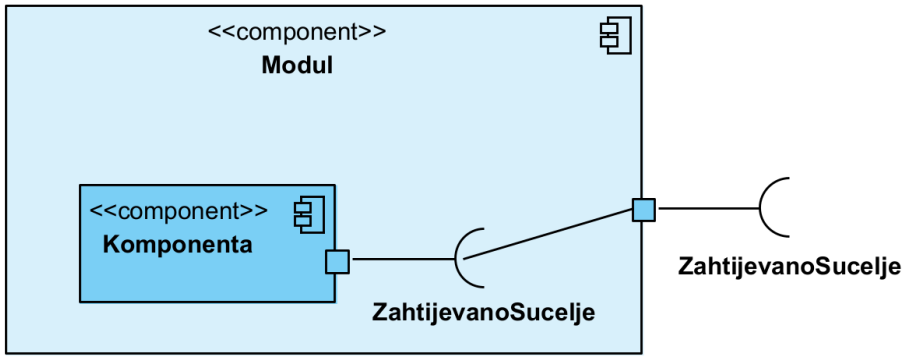


(b)

Slika 8.10: Delegacija ponuđenog sučelja: (a) notacija *ball-and-socket* i (b) izravno povezivanje na priključak



(a)

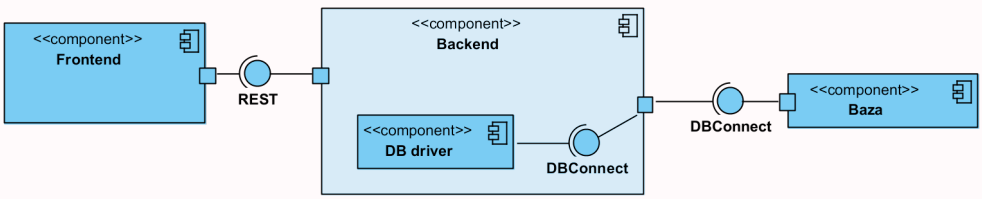


(b)

Slika 8.11: Delegacija zahtijevanog sučelja: (a) notacija *ball-and-socket* i (b) izravno povezivanje na priključak

Primjer 8.2 — Sučelja i delegacija. Potrebno je modelirati sustav koji se sastoji od tri dijela: korisničko sučelje (Frontend), pozadinska aplikacija (Backend) i baza podataka (Baza). Korisničko sučelje za komunikaciju s pozadinskom aplikacijom zahtijeva sučelje REST. Pozadinska aplikacija dohvaća sve podatke iz baze putem sučelja DBConnect. Za dohvat podataka iz baze pozadinska aplikacija sadržava dodatni modul DBDriver koji zna kako pristupiti sučelju DBConnect.

Rješenje je prikazano na slici 8.12.



Slika 8.12: Rješenje primjera sučelja i delegacije

Komentar:

Budući da komponenta Frontend zahtijeva sučelje REST, komponenta Backend mora ga

ostvariti. Kako nema dodatnih informacija o mogućim potkomponentama na koje bi se eventualno delegirala implementacija sučelja REST, dovoljno je prikazati da komponenta Backend ima priključak na koji je spojeno ponuđeno sučelje REST. Međutim, za komunikaciju s bazom podataka jasno je rečeno da komponenta Backend koristi modul DBConnect, pa se tu onda prikazuje delegacija zahtijevanog sučelja.

■

8.2 Postupak izrade dijagrama

Dijagrami komponenti mogu se, slično kao i dijagrami razreda, izrađivati u različitim fazama za vrijeme provedbe projekta. U fazi oblikovanja moguće je izraditi dijagram komponenti radi idejne razrade arhitekture sustava i utvrđivanja potencijalnih ovisnosti o vanjskim bibliotekama, radnim okvirima i sl. U fazi implementacije i ispitivanja dijagrami komponenti izrađuju se na temelju gotovog rješenja radi boljeg razumijevanja strukture i međuovisnosti sustava te dokumentiranja.

Ako se izrađuje dijagram za sustav koji je već implementiran, on se može generirati na temelju programskog koda, što podržava većina modernih razvojnih okruženja. No, u fazama oblikovanja, kada programski kôd još nije implementiran, dijagram komponenti izrađuje se iterativno na temelju idejnog rješenja, a preporučeni su koraci:

- Utvrđivanje i razumijevanje sustava – na početku je potrebno dobro utvrditi svrhu, funkcionalnost i opseg sustava, u čemu mogu pomoći tekstna specifikacija i neki od prethodno opisanih UML dijagrama (obrasci uporabe, dijagram razreda i sl.).
- Određivanje komponenti – sljedeći je korak određivanje ključnih komponenti sustava. Potrebno je razdijeliti sustav na kohezivne jedinice ili module koji u sebi sadržavaju specifičnu funkcionalnost. Komponente mogu predstavljati razrede, pakete, knjižnice, podsustave i sl.
- Definiranje sučelja – za utvrđene je komponente potrebno odrediti sučelja koja će komponente izložiti da bi komunicirale s drugim komponentama. U prvoj iteraciji to nije nužno, ali kasnije je svakako poželjno utvrditi operacije, metode ili poruke koje će komponente pružati ili tražiti za komunikaciju.
- Uspostavljanje odnosa – određivanje načina na koji komponente međusobno komuniciraju i ovise jedna o drugoj. U ovom se koraku utvrđuju ovisnosti te se komponente povezuju na adekvatan način: vezama ovisnosti, spojnicama ili delegacijom. Na svakoj je točki na kojoj komponenta izlaže neki dio svojih unutarnjih funkcionalnosti svakako nužno utvrditi i priključak.
- Primjena stereotipa (opcionalno) – za komponente specifične namjene moguće je definirati posebne stereotipe da bi se pružio dodatni kontekst ili specijalizacija. Stereotipi mogu pomoći u kategorizaciji komponenti prema njihovim ulogama, ponašanjima ili karakteristikama.
- Iterativna dorada – potrebno je pregledati dijagram komponenti i učiniti sve potrebne ispravke ili prilagodbe da bi se osigurala točnost i potpunost. S razvojem sustava ili promjenom zahtjeva, potrebno je redovito ažurirati dijagrame komponenti da bi odražavali trenutno stanje arhitekture sustava.

8.3 Primjena

Dijagrami komponenti mogu biti korisni u različitim fazama razvoja programske potpore, no najčešće se koriste u postupku razrade i oblikovanja sustava te kao način dokumentacije gotovog

sustava. U fazi oblikovanja pomažu u utvrđivanju komponenata koje čine programski sustav te omogućuju dekompoziciju sustava na modularne upravljive i ponovno upotrebljive jedinice. Nadalje, vizualizacija komponenata i njihovih odnosa olakšava raspodjelu odgovornosti, definiranje sučelja i osiguranje pravilne enkapsulacije. Analizom idejnog rješenja modeliranog pomoću dijagrama komponenti moguće je u vrlo ranoj fazi razumjeti učinak promjena na dijelove sustava te prepoznati potencijalne problematične točke.

Modeli utvrđeni korištenjem dijagrama komponenti u fazi oblikovanja služe kao referenca za fazu implementacije. Oni vode programere u implementaciji komponenata i njihove interakcije na temelju specificiranih sučelja i odnosa te pomažu osigurati usklađenost implementacije sa zadanom arhitekturom sustava. Također, dijagrami komponenti imaju ključnu ulogu u integraciji sustava, posebno u programskim sustavima velikog opsega jer pomažu u definiranju sučelja i komunikacijskih kanala između komponenti te olakšavaju razumijevanje toga kako se različite komponente spajaju u kohezivan sustav.

U procesu ispitivanja dijagrami komponenti pružaju uvid u strukturu i interakcije sustava te ih ispitni timovi mogu koristiti za utvrđivanje ključnih komponenti i njihovih ovisnosti, što je važno u razvoju učinkovitih ispitnih slučajeva. Nakon što je programski sustav dovršen, dijagrami komponenti prvenstveno se koriste u svrhu dokumentiranja sustava. Služe kao referenca za razumijevanje postojeće arhitekture sustava – strukture komponenata i njihove ovisnosti. U tom smislu pomažu u održavanju sustava i otklanjanju problema ili donošenju poboljšanja. Međutim, važni su i u procesu nadogradnje sustava jer pomažu u analizi strukture postojećeg programskog sustava i prepoznavanju mogućnosti za poboljšanje u vidu refaktoriranja ili optimizacije performansi.

Općenito, dijagrami komponenti moćan su alat u programskom inženjerstvu koji olakšava oblikovanje sustava, komunikaciju, dokumentaciju i održavanje te ih je stoga moguće koristiti u svakoj fazi razvoja, a posebice u svrhu komunikacije s dionicima. Na primjer, u fazi razrade zahtjeva mogu se koristiti za vizualizaciju i pojašnjenje funkcionalnih i nefunkcionalnih zahtjeva sustava jer pomažu u utvrđivanju glavnih komponenti sustava i razumijevanju njihove interakcije u ispunjavanju zahtjeva. Kao dijagrami visoke razine apstrakcije, s malo detalja i jednostavnom sintaksom, pružaju sveukupan pogled na arhitekturu sustava i omogućuju dionicima lako razumijevanje i analizu programskog sustava te provjeru usklađenosti arhitekture sa zahtjevima. U tablici 8.1 nalazi se sažet i sistematiziran prikaz primjene dijagrama komponenti u aktivnostima programskog inženjerstva.

Tablica 8.1: Primjena dijagrama komponenti za vrijeme različitih aktivnosti programskog inženjerstva

Aktivnost	**Primjena**
Specifikacija programske potpore	Utvrđivanje glavnih komponenti sustava i razumijevanje njihove interakcije u ispunjavanju zahtjeva.
Analiza i oblikovanje	Dekompozicija sustava i vizualizacija komponenata i njihovih odnosa – raspodjela odgovornosti, definiranje sučelja i osiguranje pravilne enkapsulacije.
Implementacija	Referenca za implementaciju vanjskih sučelja komponenti i integraciju sustava.
Ispitivanje	Utvrđivanje ključnih komponenti i njihovih ovisnosti.
Evolucija	Dokumentacija postojeće arhitekture sustava važna za održavanje i nadogradnju sustava.

9. UML dijagrami razmještaja

UML dijagrami razmještaja prikazuju fizičku arhitekturu programskog sustava, što podrazumijeva razmještaj programskih artefakata na sklopovskim čvorovima ili virtualnim okruženjima. Glavna je svrha dijagrama razmještaja u programskom inženjerstvu pružiti razumijevanje arhitekture razmještaja sustava.

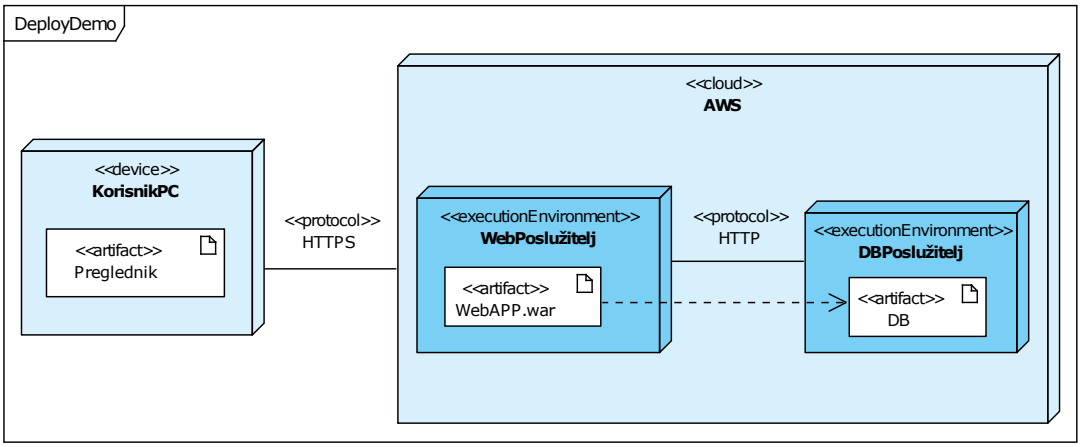
Prikazivanjem fizičke infrastrukture dijagrami razmještaja olakšavaju rasprave i donošenje odluka o sklopovskim zahtjevima, skalabilnosti, performansama i dodjeli resursa. Pomažu i u planiranju održavanja i nadogradnji sustava te olakšavaju koordinaciju i komunikaciju između timova za razvoj (engl. *development team*) i operacije (engl. *operations team*) te timova s pružateljem infrastrukture (engl. *provider*). Konačno, pomažu u prepoznavanju potencijalnih uskih grla ili pojedinačnih točaka kvara te omogućuju razvoj strategija za ublažavanje posljedica kvarova.

9.1 Definicija i osnovni elementi

UML dijagrami razmještaja (engl. *deployment diagrams*) također su vrsta strukturnih UML dijagrama koji prikazuju fizičku arhitekturu i konfiguraciju razmještaja programskog sustava. Oni ilustriraju raspodjelu programskih komponenti, izvršnih datoteka i knjižnica na sklopovske čvorove ili virtualna izvršna okruženja. Glavni elementi UML dijagrama razmještaja su sljedeći:

- **Čvorovi** (engl. *node*) – sklopovski uređaji ili računalni resursi na kojima se razmještaju programske komponente. To podrazumijeva fizičke i virtualne strojeve te različite resurse u oblaku.
- **Artefakti** (engl. *artifact*) – fizičke datoteke ili programski paketi koji se razmještaju na čvorove. Mogu uključivati izvršne datoteke, konfiguracijske datoteke, knjižnice ili bilo koje druge datoteke potrebne za funkcioniranje sustava (npr. ulazne datoteke s podacima za obradu).
- **Veze** – prikazuju način na koji čvorovi međusobno djeluju, komuniciraju ili dijele resurse te ovisnosti između artefakata.

Primjer dijagrama razmještaja prikazan je na slici 9.1, a u nastavku su detaljnije pojašnjeni njegovi osnovni elementi.



Slika 9.1: Primjer UML dijagrama razmještaja

9.1.1 Čvorovi

Temeljni i neizostavni element svakog dijagrama razmještaja su čvorovi (engl. *node*). Čvorovi su okruženja na koja se razmještaju i izvršavaju programske komponente. Čvorovi mogu biti fizički strojevi (računala) te različita virtualna okruženja, u koja se ubrajaju virtualni strojevi (engl. *virtual machine* – VM), operacijski sustavi (engl. *operation system* – OS), a danas je sve zastupljenije korištenje raznih vrsta virtualnih okruženja u oblaku, poput spremnika Docker i Kubernetes.

U osnovi, čvor je svako izvršno okruženje koje pruža kontekst (zaseban proces i adresni prostor) za izvođenje nekog programa. Čvorovi su enkapsulirane cjeline koje mogu komunicirati s drugim čvorovima putem priključaka korištenjem različitih komunikacijskih protokola (npr. HTTP, SSL i dr.).

Na dijagramu razmještaja čvorovi se prikazuju kao kvadri koji sadržavaju naziv čvora i stereotip. Naziv čvora sastoji se od dvaju dijelova:

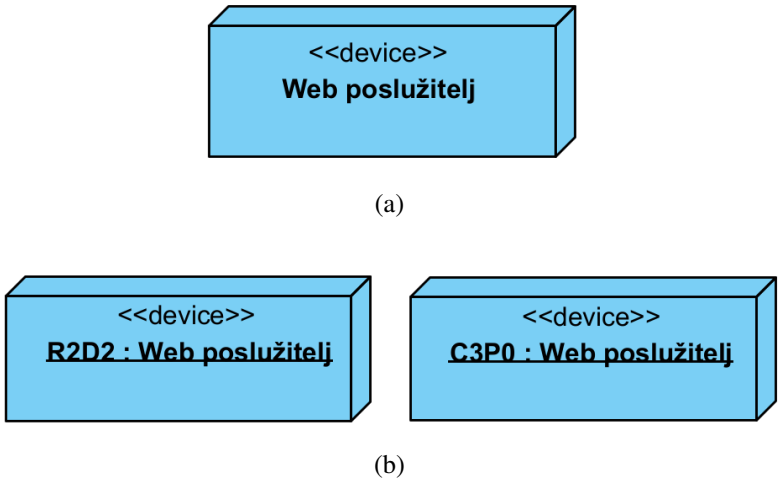
[Ime instance]: [Tip čvora].

Tip čvora nužan je podatak i jednoznačno određuje vrstu čvora, tj. njegovu zadaću (npr. *web*-poslužitelj). Ime instance koristi se kada se želi posebno istaknuti postojanje više instanci istog tipa, a inače se može izostaviti. Dijagrami razmještaja koji ne sadržavaju imena instanci nazivaju se još i **specifikacijski dijagrami razmještaja**, a dijagrami koji prikazuju više imenovanih instanci istog tipa nazivaju se **dijagrami razmještaja instanci**.

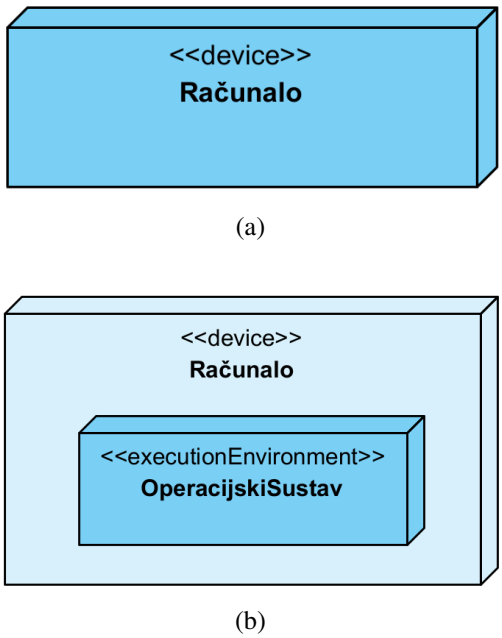
Stereotipi pružaju dodatne informacije ili kategorizaciju elemenata u dijagramu razmještaja i mogu ukazivati na ulogu ili vrstu čvora, artefakta ili veze. Uobičajeni stereotipi za čvor su «device» i «execution environment».

Stereotip «device» može se koristiti za fizička i virtualna računala te općenito u situaciji kada nije poznato o kakvom se tipu računala radi. Stereotip «execution environment» koristi se za opis programskog izvršnog okruženja, poput operacijskog sustava, koji pruža sve potrebne resurse za izvođenje neke aplikacije. Čvorovi sa stereotipom «execution environment» tipično su ugniježđeni unutar čvora sa stereotipom «device», kao što je prikazano na slici 9.3.

Međutim, uvođenje različitih virtualizacijskih tehnologija omogućilo je apstrahiranje temeljne računalne infrastrukture, uključujući fizičke strojeve i operacijske sustave, pa se danas infrastruktura nudi kao usluga u oblaku. Stoga, da bi se fizički realizirani dijelovi sustava jasnije razlikovali od onih koji koriste virtualizaciju, preporučuje se ograničiti korištenje stereotipa «device» za označavanje fizičkih čvorova, a za virtualna okruženja koristiti specifičniji stereotip poput «cloud»,



Slika 9.2: Prikaz čvora na razini specifikacije – zadan je samo tip čvora (a) i prikaz dviju imenovanih instanci istog tipa čvora (b).



Slika 9.3: Prikaz čvora na UML dijagramu razmještaja sa stereotipom «device» (a) i čvora s uobičajenim stereotipm «execution environment» ugniježđenog unutar čvora tipa «device» (b).

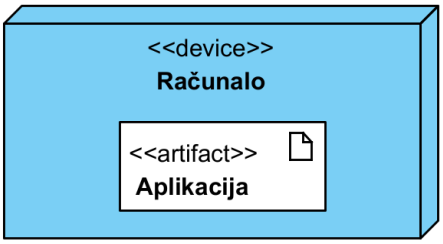
«VM», «container», «service» i slično.

Općenito, za stereotipe ne postoje formalna ograničenja pa se može koristiti bilo koji nenormirani stereotip, poput «web server», «database» i sl. ako pomaže u razumijevanju arhitekture razmještaja sustava.

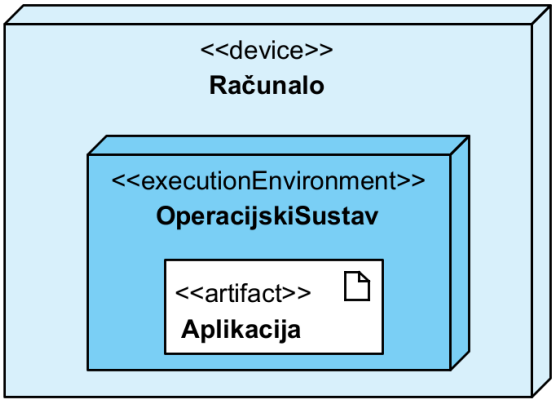
9.1.2 Artefakti

Artefakti su konkretna implementacija programskih (logičkih) komponenti, uobičajeno prikazanih na dijagramu komponenti. Riječ je o fizičkim datotekama koje se razmještaju na pojedine čvorove. Primjeri artefakata su izvršne datoteke, knjižnice, konfiguracijske datoteke i slično, a one mogu biti razmještene zasebno ili objedinjene u obliku programskih paketa kao npr. arhive .jar i .war.

Na dijagramima razmještaja artefakti se prikazuju kao pravokutnici sa stereotipom «artifact» te se tipično nalaze unutar čvora. Primjeri prikaza artefakata na dijagramu razmještaja nalaze se na slikama 9.4 i 9.5.



Slika 9.4: Prikaz artefakta na čvoru na UML dijagramu razmještaja



Slika 9.5: Prikaz ugniježđenog čvora s artefaktom na UML dijagramu razmještaja

Osim uobičajenog stereotipa «artifact», moguće je koristiti i druge stereotipe koji pobliže opisuju vrstu programskog artefakta, kao npr. «file», «executable», «library» i slično. Odabir stereotipa ovisi o konkretnom problemu i razini na kojoj se modelira sustav. U ranim fazama, kada još nisu poznati konkretni detalji implementacije, bolje je koristiti općenitije stereotipe, a pri dokumentaciji gotovog sustava, da bi se sustav opisao što preciznije, moguće je koristiti specifične stereotipe.

U ranim fazama specifikacije i oblikovanja sustava nije neuobičajeno na dijagramima razmještaja susresti i komponente, prikazane na isti način kao na UML dijagramu komponenti (vidjeti poglavlje 8.1.1.). Osnovni razlog je što u danom trenutku najčešće nije poznat konkretni artefakt koji će predstavljati realizaciju komponente. No, u tom se slučaju najčešće prikazuju samo vršne

komponente, a ne i njihove unutarnje strukture, te se najčešće izostavljaju sučelja, spojnice, delegacije i ostali detalji semantike UML dijagrama komponenti. Osim toga, na dijagramima razmještaja moguće je prikazati odnos manifestacije između komponenti i artefakata te međusobne ovisnosti, o čemu će biti više riječi u idućem potpoglavlju.

Primjer 9.1 — Primjeri modeliranja čvorova i artefakata. Potrebno je modelirati sljedeće četiri specifikacije:

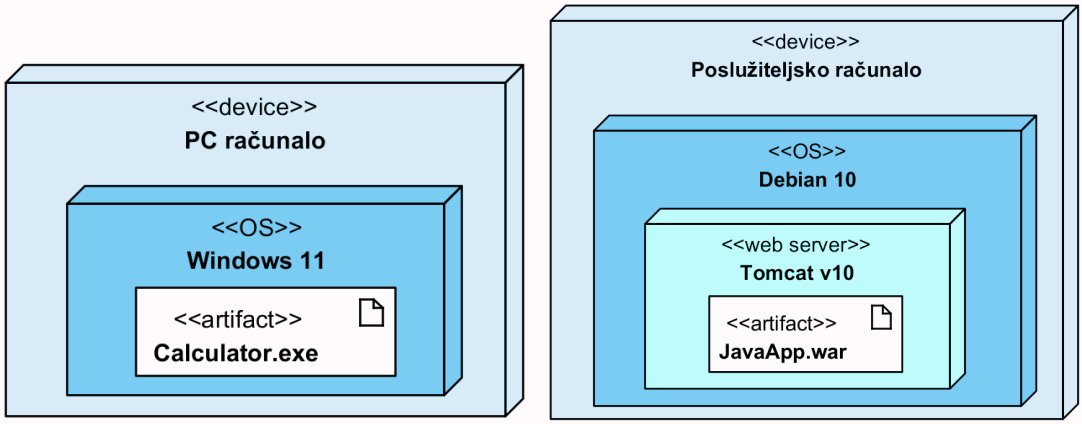
A) Aplikacija Calculator.exe pokrenuta je na PC računalu s instaliranim operacijskim sustavom Microsoft Windows 11.

B) *Web*-aplikacija zapakirana u datoteci JavaApp.war pokrenuta je korištenjem programskog rješenja za *web*-poslužitelj Tomcat v10. To je sve zajedno pokrenuto na poslužiteljskom računalu s instaliranim operacijskim sustavom Linux Debian 10.

C) Aplikacija napisana korištenjem programskog jezika Java v11 i radnog okvira Spring Boot 3.0.0. pokrenuta je kao slika Dockera (Docker Image) unutar spremnika Docker (Docker container), posebno prilagođenog za Spring Boot 3.0.0., a sve to korištenjem infrastrukture u oblaku Amazon AWS.

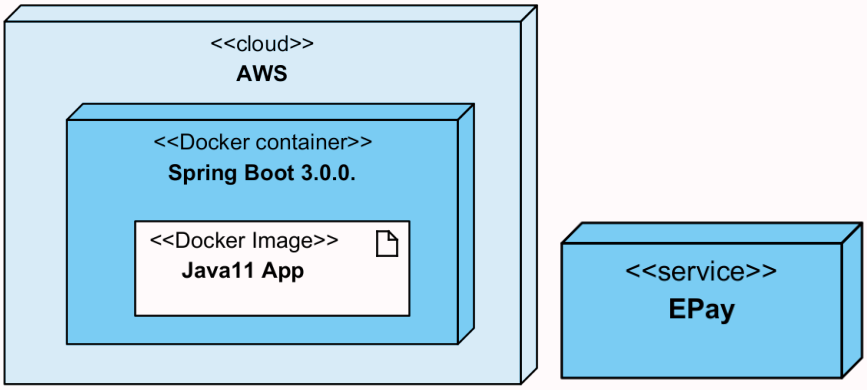
D) EPAy je usluga (servis) za plaćanje putem interneta dostupna putem protokola HTTP s normiranim API-jem.

Rješenja su prikazana na slici 9.6.



(a) Model specifikacije A

(b) Model specifikacije B



(c) Model specifikacije C

(d) Model specifikacije D

Slika 9.6: Rješenja primjera četiri specifikacije

Komentar:

U rješenju specifikacije A moguće je navesti i generički normirani stereotip «execution environment» za čvor koji predstavlja operacijski sustav, no stereotip «OS» pruža više informacija o čvoru, stoga je bolje rješenje. Isto je i kada je riječ o specifikaciji B. Za čvorove Debian i Tomcat ispravno je koristiti i normirani stereotip «execution environment», ali bi u tom slučaju bilo preporučljivo u samom imenu čvora navesti da se radi o operacijskom sustavu, odnosno *web*-poslužitelju.

U rješenju specifikacije C kao stereotip za čvor AWS koristi se «cloud» umjesto «device» jer se želi istaknuti da se ne upotrebljava vlastita fizička infrastruktura. Zbog načina na koji funkcioniraju Dockerovi spremnici, za izvođenje aplikacija implementiranih korištenjem radnog okvira Java Spring Boot nije potrebno navoditi operacijski sustav. Također, specifičnost Dockerovih spremnika je da se aplikacija (izvršni kôd sa svim konfiguracijskim datotekama i dodatnim programskim paketima o kojima on ovisi) pakira u jedan artefakt – Dockerovu sliku (Docker image), koji se izvršava unutar Dockerova spremnika.

Za specifikaciju D nisu navedeni nikakvi konkretni detalji osim da se radi o usluzi (engl.

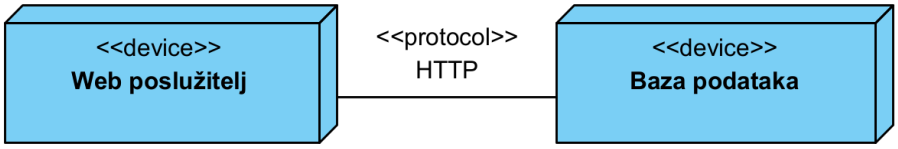
service) koja ima normirani *web* API te je u tom slučaju ispravno prikazati samo jedan čvor koji predstavlja tu uslugu. Riječ je tipičnom primjeru modeliranja vanjskih usluga, kao što su plaćanje, geo karte, e-pošta i sl., koje ostale *web*-aplikacije (kao npr. e-trgovina) koriste u svojem radu.

■

9.1.3 Veze

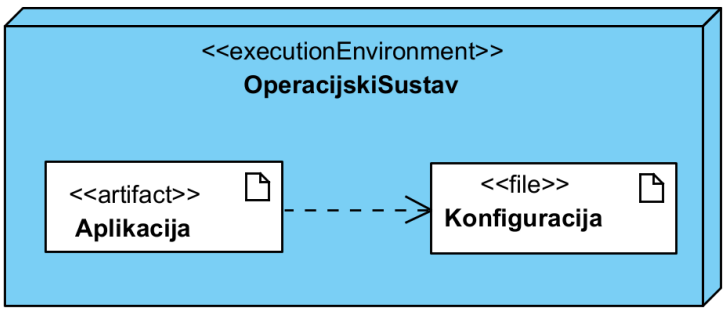
Na dijagramima razmještaja uobičajeno je prikazivati samo dvije vrste veza: pridruživanje između čvorova i ovisnost između artefakata. Dodatno, ako se želi istaknuti povezanost između logičke arhitekture prikazane na dijagramu razreda ili komponenata, moguće je prikazati i odnos manifestacije između artefakta i komponente koju taj artefakt implementira.

Pridruživanje (engl. *association*, *communication path*) između čvorova je komunikacijski kanal ili mrežna veza između dvaju čvorova. Ono se prikazuje ravnom crtom na kojoj se tipično navodi i komunikacijski protokol (i priključak) koji se koristi, kao npr. HTTP, što je ilustrirano na slici 9.7. Ako se želi istaknuti mogući broj instanci čvorova koji se koriste, na vezi se može navesti i brojnost, po istim pravilima kao i kada je riječ o dijagramu razreda.



Slika 9.7: Primjer povezivanja dvaju čvorova korištenjem protokola HTTP

Ovisnost (engl. *dependency*) između artefakata, koji mogu biti razmješteni na istom ili različitim čvorovima, označava da jedan artefakt ovisi o drugom ili zahtijeva drugi za ispravno funkcioniranje. Na primjer, artefakt koji predstavlja glavnu aplikaciju može ovisiti o određenim knjižnicama ili konfiguracijskim datotekama koje su razmještene kao zasebni artefakti na istom ili drugom čvoru. Ovisnost se prikazuje crtkanom strelicom koja pokazuje od artefakata koji ovisi prema artefaktu o kojem se ovisi, kao što je prikazano na slici 9.8.

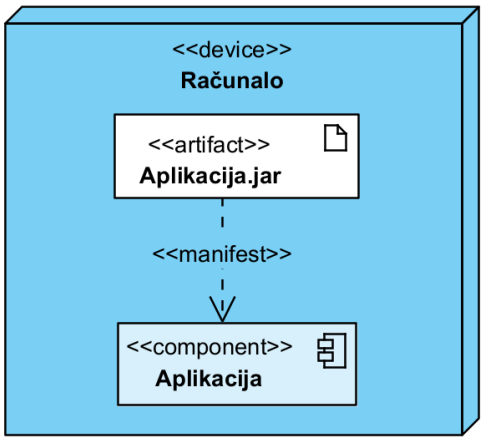


Slika 9.8: Primjer prikazivanja ovisnosti između dvaju artefakata pri čemu aplikacija koristi podatke iz konfiguracijske datoteke.

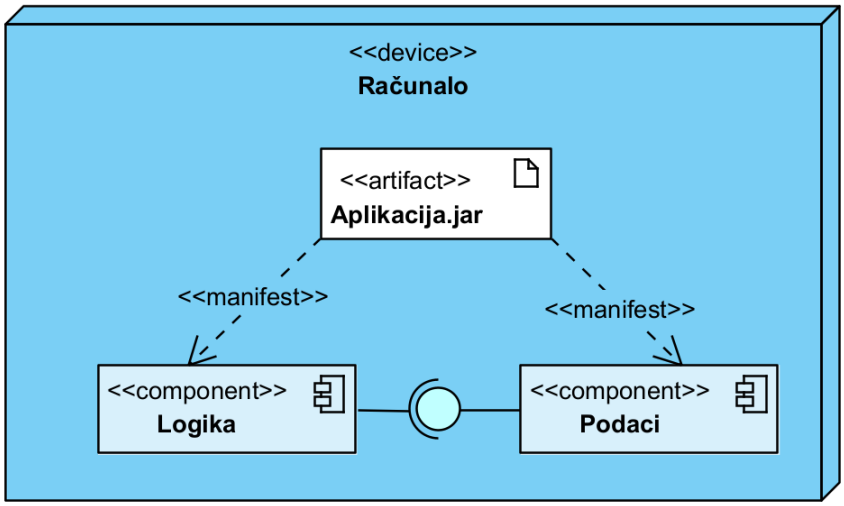
Iako nije nužno prikazati sve moguće ovisnosti između artefakata na dijagramu razmještaja, osobito ako to dovodi do nečitljivosti dijagrama, preporučuje se svakako istaknuti one ključne. Razumijevanje i prikazivanje ovisnosti na dijagramu razmještaja pomaže dionicima u utvrđivanju potrebnog redoslijeda razmještaja artefakata, rješavanju konflikata ili problema povezanih s

ovisnošću te osiguravanju ispravnog funkcioniranja sustava. Dodatno, na vezi ovisnosti moguće je dodati i stereotip da bi se pružile dodatne informacije o vrsti ovisnosti, kao npr. stvaranje, brisanje, korištenje itd.

Ponekad je na dijagramu razmještaja potrebno istaknuti povezanost između artefakta i komponente koju on implementira (modelirane na dijagramu komponenti). U tu se svrhu koristi odnos **manifestacije** (engl. *manifestation*). Manifestacija se prikazuje crtkanom strelicom sa stereotipom «manifest», usmjerenom od artefakta prema komponenti koju on implementira. Prikazivanje manifestacije uspostavlja vezu između logičkih komponenti i njihove fizičke implementacije u obliku specifične datoteke ili paketa. Jedan artefakt može implementirati jednu komponentu ili više njih, kao što je prikazano na slici 9.9.



(a)



(b)

Slika 9.9: Prikaz manifestacije ako jedan artefakt implementira jednu komponentu (a) te ako jedan artefakt implementira više komponenti (b).

Prikazivanje manifestacije komponenti u artefaktima pomaže dionicima u razumijevanju toga kako se logičko oblikovanje prenosi u fizičku implementaciju programskog sustava. Ta povezanost pomaže u konfiguraciji sustava, upravljanju verzijama i osigurava dosljednost između logičkog

oblikovanja i njegove fizičke realizacije. Ipak, radi očuvanja čitljivosti dijagrama nije nužno uvijek prikazati svaku manifestaciju. Ako se svaka manifestacija želi detaljno prikazati, poželjno je izdvojiti njezin prikaz na zaseban dijagram na kojem se mogu izostaviti čvorovi te prikazati samo artefakti i komponente.

9.2 Postupak izrade dijagrama

S obzirom na jednostavniju sintaksu i semantiku dijagrama razmještaja u odnosu na prethodno opisane UML dijagrame, postupak izrade dijagrama razmještaja također je nešto jednostavniji. U nastavku su opisani osnovni koraci:

- **Utvrđivanje čvorova** – preporučeni prvi korak je utvrđivanje fizičkih i virtualnih uređaja na kojima će se razmjestiti programske komponente. Potrebno je i razmotriti postojanje izvršnih okolina kao što su operacijski sustavi, virtualni spremnici i sl. Najbolje je započeti s vanjskim čvorovima, a zatim postupno dodavati ugniježđene. Za svaki je čvor potrebno odrediti odgovarajući stereotip.
- **Utvrđivanje artefakata i komponenti** – u sljedećem je koraku potrebno utvrditi programske artefakte (izvršne datoteke, knjižnice, konfiguracijske datoteke i sl.) te ih ispravno razmjestiti po čvorovima. Ako još nije poznat gotov artefakt, moguće je umjesto njega prikazati odgovarajuće komponente.
- **Povezivanje čvorova** – potrebno je utvrditi veze pridruživanja među čvorovima za koje je u specifikaciji sustava zadano da trebaju međusobno komunicirati i, ako je poznato, protokol komunikacije.
- **Modeliranje ovisnosti između artefakata** – po potrebi prikazati ovisnosti između artefakata, a moguće je (na istom ili zasebnom dijagramu) prikazati i odnos manifestacije između artefakata i komponenti.
- **Iterativno poboljšanje** – nakon što je dijagram razmještaja dovršen, potrebno ga je pregledati da bi se osigurala točnost i potpunost. Treba posvetiti posebnu pozornost provjeri toga je li konfiguracija razmještaja usklađena sa zahtjevima i ograničenjima sustava i po potrebi doraditi postojeći dijagram. Također, kada dođe do promjene zahtjeva i arhitekture, potrebno je ažurirati dijagram razmještaja. Ovisno o složenosti sustava, moguće je koristiti više dijagrama razmještaja da bi se očuvala preglednost i razumljivost.

Važno je napomenuti da je u praksi moguće susresti dvije vrste dijagrama razmještaja – dijagram razmještaja na razini specifikacije (engl. *specification-level deployment diagram*) i dijagram razmještaja na razini instanci (engl. *instance-level deployment diagram*).

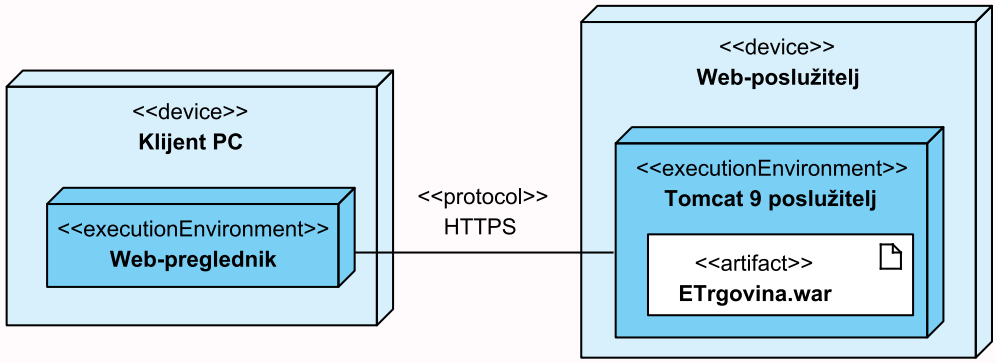
Dijagram razmještaja na razini specifikacije usmjerava se na prikazivanje cjelokupne strukture i veza između programskih artefakata i čvorova na visokoj razini apstrakcije. On prije svega predstavlja konceptualni pogled na razmještajnu konfiguraciju. Ovaj tip dijagrama razmještaja ističe statičku arhitekturu razmještaja tako što prikazuje kako su artefakti dodijeljeni čvorovima te između koje vrste čvorova postoje veze, ne uzimajući u obzir broj određenih instanci. Obično se koristi u ranim fazama dizajna i arhitekture sustava.

Dijagram razmještaja na razini instanci pruža detaljniji i konkretniji pogled na arhitekturu razmještaja uzimajući u obzir stvarne instance artefakata i čvorova. Pogodan je pri razmatranju performansi sustava tako što uzima u obzir čimbenike poput skaliranja, redundancije i uravnoteženja opterećenja jer prikazuje dinamičke aspekte konfiguracije razmještaja, uključujući više instanci komponenti, replikaciju ili grupiranje.

Primjer 9.2 — Modeliranje na razini specifikacije i instanci. Potrebno je modelirati sustav e-trgovine u kojem je *web*-aplikacija implementirana kao arhiva ETrgovina.war i pokrenuta na poslužitelju Tomcat 9 koji se izvodi na fizičkom računalu. Korisnik pristupa aplikaciji sa svojeg računala (PC) korištenjem nekog *web*-preglednika i protokola HTTPS.

Najprije treba razmotriti potrebne vrste čvorova i artefakata te ih prikazati na specifikacijskom dijagramu razmještaja, a zatim razmotriti konkretni slučaj u kojem radi ujednačavanja opterećenja postoje dva poslužiteljska računala te je na svakom od njih pokrenut poslužitelj Tomcat koji izvodi svoju kopiju *web*-aplikacije.

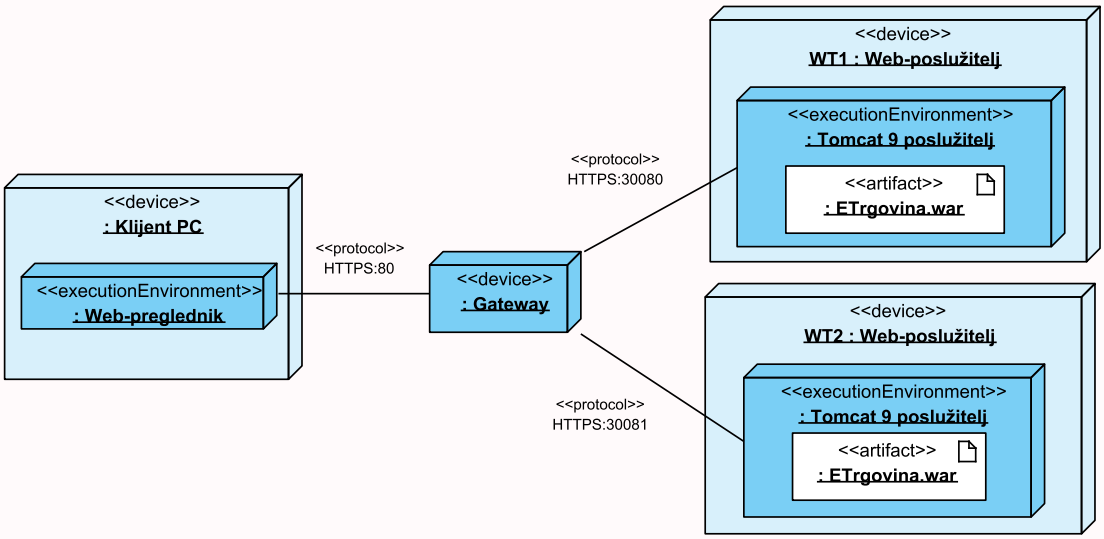
Rješenje je prikazano na slikama 9.10 i 9.11.



Slika 9.10: Rješenje za specifikacijski dijagram razmještaja

Komentar:

Budući da je protokol HTTPS utvrđen na razini aplikacijskog sloja, preciznije je rješenje za specifikacijski dijagram razmještaja u kojem su izravno povezani čvorovi *web*-preglednik i poslužitelj Tomcat 9 (sama računala povezuju se na nižem sloju, korištenjem npr. TCP/IP).



Slika 9.11: Rješenje za dijagram razmještaja instanci

Komentar:

Pri modeliranju konkretnih instanci potrebno je uočiti jedan nedostatak specifikacijskog

modela –ako se radi ujednačavanja opterećenja u pogon pušta više instanci (replika) aplikacije na više poslužitelja, potrebno je uvesti dodatni mehanizam koji će prosljeđivati korisničke upite. Stoga više nije moguće da se korisnik sa svojeg računala izravno spaja na poslužitelj, već je između potrebno postaviti dodatni uređaj, npr. pristupnik (engl. *gateway*) koji će dalje prosljeđivati upite na neki od poslužitelja. Nadalje, instancama nije nužno davati imena, osim ako ih se eksplicitno želi razlikovati (npr. točno određeno fizičko računalo).

■

9.3 Primjena

UML dijagrami razmještaja koriste se u različitim fazama procesa razvoja programske potpore radi oblikovanja arhitekture, planiranja infrastrukture te održavanja i nadogradnje sustava.

U fazi oblikovanja arhitekture programske potpore, dijagrami razmještaja pomažu u vizualizaciji i planiranju razmještajne konfiguracije sustava, uključujući dodjelu artefakata sklopovskim čvorovima. Oni su alat za komunikaciju među različitim dionicima uključenima u razvoj programske potpore, od programera, preko arhitekata i administratora sustava, sve do voditelja projekata. Budući da pružaju jasan i jednostavan vizualni prikaz razmještajne arhitekture sustava, olakšavaju rasprave, razumijevanje i suradnju između tehničkih i netehničkih članova tima.

Dijagrami razmještaja neizostavan su alat u planiranju infrastrukture te donošenju odluka vezanih za sklopovske zahtjeve, skalabilnost, performanse i održavanje sustava. Pružaju uvid u potrebne infrastrukturne resurse nužne za podršku programskom sustavu te pomažu u određivanju broja i vrsta potrebnih čvorova, kao i njihovih zahtjeva za povezivanje i komunikaciju. U ovoj fazi, dijagrami razmještaja pomažu razvojnom timu ostvariti suradnju s administratorima sustava ili pružateljima infrastrukture da bi se osigurala odgovarajuća infrastruktura za programski sustav.

Nakon što je sustav pušten u pogon, dijagrami razmještaja koriste se u svrhu održavanja sustava i rješavanja problema. Pomažu u utvrđivanju specifičnih čvorova na kojima su razmješteni programski artefakti te omogućuju programerima i administratorima sustava učinkovito pronalaženje i dijagnosticiranje problema. Dijagrami razmještaja pomažu u razumijevanju ovisnosti i veza između komponenti i čvorova, što olakšava aktivnosti održavanja sustava i rješavanje problema. U tablici 9.1 nalazi se sažet i sistematiziran prikaz primjene dijagrama razmještaja u aktivnostima programskog inženjerstva.

Tablica 9.1: Primjena dijagrama razmještaja za vrijeme različitih aktivnosti programskog inženjerstva

Aktivnost	**Primjena**
Specifikacija programske potpore	Planiranje infrastrukture sustava.
Analiza i oblikovanje	Vizualizacija i planiranje razmještajne konfiguracije sustava – raspodjela artefakata po sklopovskim čvorovima.
Implementacija	Donošenje odluka vezanih za sklopovske zahtjeve, skalabilnost, performanse i održavanje sustava.
Ispitivanje	Prepoznavanje kritičnih dijelova sustava u kontekstu performansi i pouzdanosti.
Evolucija	Dokumentacija za održavanje sustava i planiranje proširenja.

II Zadaci za vježbu

10. UML dijagrami obrazaca uporabe

10.1 Zadaci

■ **Zadatak 10.1 — Upravljanje hotelom.** Modelirajte pomoću dijagrama obrazaca uporabe sustav upravljanja hotelom zadan sljedećim opisom.

Gosti hotela mogu putem interneta pretražiti hotelsku ponudu i napraviti rezervaciju sobe. Pri rezervaciji sobe gosti mogu opcionalno rezervirati parkirno mjesto te dokupiti neke od sljedećih opcija: doručak, polupansion ili puni pansion. Recepcionar hotela može izdati sobu i napraviti naplatu. Pri izdavanju sobe recepcionar obavezno provjerava rezervaciju i programira ključ (karticu) za otvaranje sobe te pri naplati izdaje račun. Administrator sustava upravlja raspoloživim sobama, unosi parkirna mjesta te uređuje cjenik hotela. Upravljanje raspoloživim sobama odnosi se na dodavanje novih soba, brisanje postojećih te unos podataka o sobi (broj ležaja, dodatni sadržaj itd.).
■

■ **Zadatak 10.2 — Prodaja autobusnih karata.** Modelirajte pomoću dijagrama obrazaca uporabe sustav za prodaju karata na autobusnom kolodvoru zadan sljedećim opisom.

Sustav koriste blagajnici i putnici. Blagajnik može rezervirati mjesto ili prodati kartu putniku. Prodaja karte uključuje odabir odredišta i vremena polaska te naplatu. Naplata se može izvršiti u gotovini ili bankovnom karticom. Ako se naplata vrši karticom, putnik treba unijeti PIN te se zatim provodi transakcija spajanjem na bankovni poslužitelj. Opcionalno, pri prodaji karte blagajnik može izdati račun R1 ako putnik to zatraži. Osim na blagajni rezervaciju mjesta može napraviti i putnik samostalno putem SMS servisa. ■

■ **Zadatak 10.3 — Grozd računala.** Modelirajte pomoću dijagrama obrazaca uporabe načine korištenja grozda računala akademske zajednice.

Korisnici grozda članovi su akademske zajednice i administratori. Svi korisnici na grozdu mogu dodavati nove poslove za obradu, brisati postojeće te mijenjati parametre postojećih poslova. Izmjena parametra postojećih poslova uključuje brisanje postojećeg posla i stvaranje novoga. Isto tako, korisnici mogu slati podatke na grozd računala, kopirati ih i preuzimati ih s grozda računala.

Administratori grozda zaduženi su još i za dodavanje novih korisnika, promjenu podataka korisnika, brisanje postojećih korisnika te konfiguraciju sustava. Konfiguracija sustava podrazumijeva podešavanje postavki raspoređivača poslova i izrade sigurnosne kopije podataka. Izrada sigurnosne

kopije podataka posebna je vrsta kopiranja podataka. Administratori mogu upravljati i poslovima i podacima na grozdu. ■

■ **Zadatak 10.4 — Rent-a-car.** Potrebno je izraditi dijagram obrazaca *web*-aplikacije koja služi kao podrška radu *rent-a-car* tvrtke.

Korisnici koji iznajmljuju automobile mogu putem interneta napraviti ili otkazati rezervaciju vozila. Pri stvaranju rezervacije, korisnici mogu napraviti i rezervaciju za dodatak: GPS uređaj, dječju sjedalicu, krovne nosače za bicikle ili skije i prikolicu.

Kada korisnici dođu u poslovnicu *rent-a-car* tvrtke u kojoj trebaju preuzeti vozilo, zaposlenik u aplikaciji provjerava postoji li rezervacija vozila za određenog korisnika. Ako rezervacija ne postoji, zaposlenik *rent-a-car* tvrtke izrađuje rezervaciju za korisnika. Nakon što je rezervacija napravljena (ili je postojala od prije), može se pristupiti izdavanju vozila.

Pri izdavanju vozila zaposlenik *rent-a-car* tvrtke ispisuje obrazac o preuzimanju vozila, koji korisnik potpisuje. Da bi se obrazac ispisao, aplikacija se povezuje s upravljačkim programom za ispis. Potpisani se obrazac skenira i unosi u računalo, za što se aplikacija treba povezati s upravljačkim programom za skeniranje.

Pri povratku automobila zaposlenik izdaje račun što uključuje i naplatu terećenjem kreditne kartice korisnika, koja se provodi spajanjem na bankovni poslužitelj. ■

■ **Zadatak 10.5 — E-trgovina.** Potrebno je modelirati dijagramom obrazaca uporabe sustav e-trgovine zadan sljedećim opisom.

U e-trgovini mogu kupovati registrirani i neregistrirani korisnici. Svi korisnici mogu pregledati proizvode u ponudi i dodati ih u košaricu. Za dodavanje proizvoda u košaricu korisnik mora zadati količinu. On u svakom trenutku može pregledati sadržaj košarice i ukupni iznos te ima opciju uklanjanja proizvoda iz košarice ili promjene njegove količine.

Kada je gotov s dodavanjem proizvoda u košaricu, korisnik može napraviti narudžbu što uključuje i proces naplate. Platiti se može bankovnom karticom ili u gotovini (pouzećem). Pri plaćanju karticom, korisnik mora unijeti podatke o kartici te se nakon potvrde narudžbe plaćanje vrši provođenjem transakcije putem vanjskog servisa za plaćanje.

Kada je riječ o registriranim korisnicima, završene se narudžbe pohranjuju te se mogu ponoviti u budućnosti. Neregistrirani korisnici mogu se registrirati. ■

■ **Zadatak 10.6 — Portal za oglašavanje.** Modelirajte pomoću dijagrama obrazaca uporabe *web*-portal za oglašavanje.

Oglašavati se mogu pokretnine, nekretnine i usluge. Svi su oglasi dostupni javno bez potrebe za registracijom. Pri pregledu oglasa dostupna je tražilica koja omogućuje pretraživanje oglasa prema kategoriji oglasa i riječima iz naslova. Također, dostupna je i opcija pregleda svih oglasa od nekog oglašivača.

Da bi objavili svoj oglas, potencijalni oglašivači moraju se registrirati na portalu. Posebno se registriraju pravne, a posebno fizičke osobe (različiti podaci). Predviđene su tri kategorije članstva: Standard, Optimum i Premium s raznim pogodnostima. Nakon registracije, automatski se postavlja kategorija članstva Standard (besplatna), a ako korisnik želi prijeći u višu kategoriju, mora platiti članarinu. Uplata članarine obavlja se putem aplikacije, a za provedbu transakcije aplikacija se spaja na vanjski servis za plaćanje. Plaćanje članarine nužno je za oglašavanje u nekim kategorijama, kao npr. Auto-moto i Nekretnine. Uplatu članarine korisnik može obaviti na svom profilu ili pri objavi oglasa ako se oglašava u kategoriji za koju je nužno imati plaćenu članarinu.

Registrirani korisnici i administratori mogu se prijaviti u sustav pomoću korisničkog imena i lozinke. Korisnici imaju pravo upravljati svojim profilom što podrazumijeva uređivanje i brisanje

profila. Imaju pravo i upravljati svojim oglasima: objavljivati nove oglase te uređivati i brisati postojeće. Administratori mogu pregledati sve korisnike i pritom ukloniti korisnika i oglas.

■

■ **Zadatak 10.7 — Pametna Kuća.** Potrebno je modelirati mobilnu aplikaciju „Pametna Kuća" zadanu sljedećim opisom.

Mobilna aplikacija „Pametna Kuća" omogućuje korisnicima upravljanje na daljinu pametnim uređajima u vlastitu domu. Pri registraciji u aplikaciju, korisnik mora odmah unijeti barem jednu lokaciju na kojoj će biti pametni uređaji. Lokacija se može zadati korištenjem GPS-a ili ručno. Korisnik naknadno može dodati još lokacija te urediti ili obrisati one postojeće.

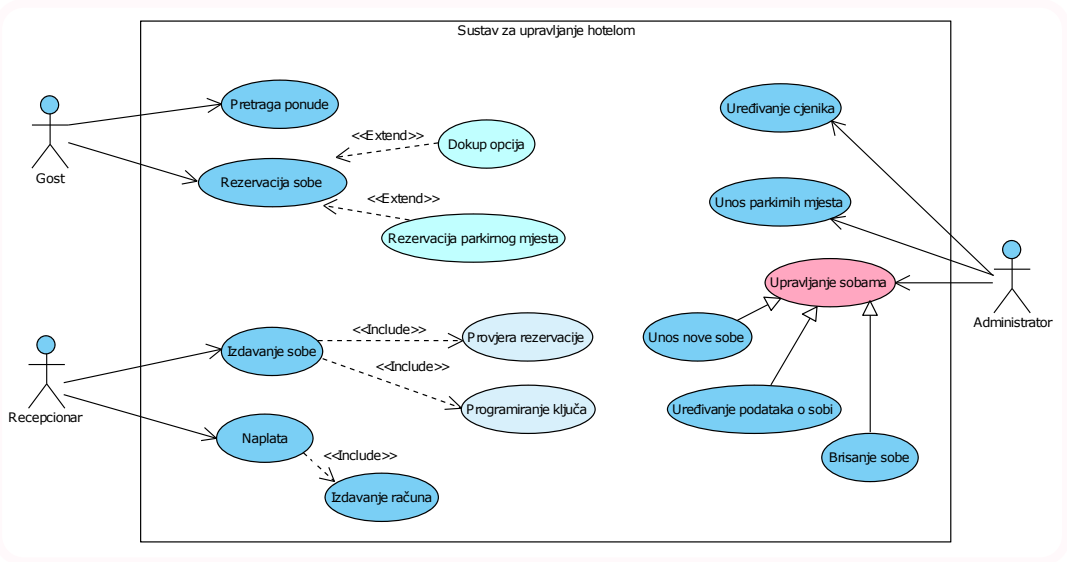
Nakon uspješne registracije i prijave u sustav korisnik može povezati pametne uređaje. Za povezivanje s uređajem korisnik odabire lokaciju te unosi jedinstveni kôd (koji dobiva kupnjom uređaja). Mobilna aplikacija šalje taj kôd vanjskoj usluzi za registraciju pametnih uređaja SmartDeviceRegService, od koje dobiva podatke za spajanje na uređaj te se zatim spaja na pametni uređaj, čime završava proces povezivanja novog uređaja u aplikaciju. Pri povezivanju novog pametnog uređaja korisnik ima mogućnost unosa nove lokacije ako ne želi koristiti ni jednu od postojećih.

Mobilna aplikacija razlikuje dvije vrste pametnih uređaja: mjerače i kućanske aparate. Kućanskim aparatima moguće je upravljati na daljinu, a za mjerače je moguće pregledati potrošnju energije. Također, korisnik može obrisati povezane uređaje.

Mobilna aplikacija sve podatke o korisniku, lokacijama i uređajima pohranjuje na poslužitelju BackendServer. ■

10.2 Rješenja

- **Zadatak 10.1 Upravljanje hotelom**

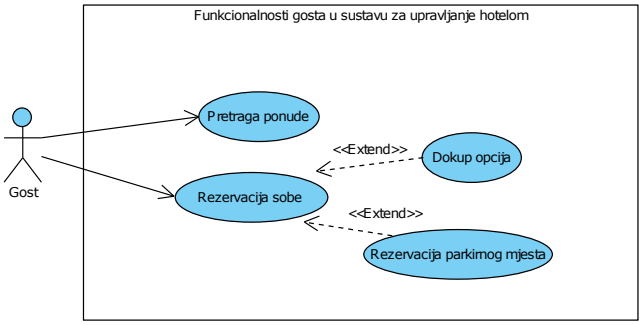


Slika 10.1: Dijagram obrazaca uporabe sustava za upravljanje hotelom

Komentar: Rezervacija parkirnog mjesta i dokup opcija ne moraju se nužno izvršiti pri rezervaciji sobe i stoga su ti obrasci s glavnim obrascem povezani vezom «extend». Međutim, pri izdavanju sobe nužno je provjeriti rezervaciju, pa je stoga odabrana veza «include». Nadalje, opcije doručka, polupansiona i punog pansiona i njihov dokup objedinjene su u jedan obrazac, što znači da bi pri implementaciji trebale biti ponuđene kao popis opcija iz kojeg korisnik odabire (označava) opcije koje želi.

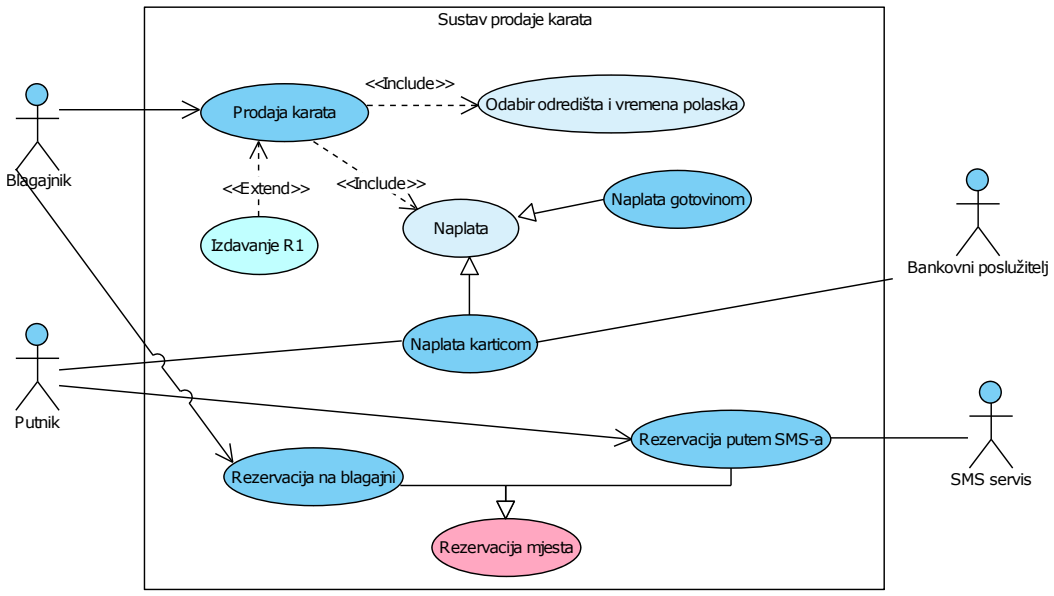
Izdavanje računa može se razmatrati kao cjeloviti obrazac ako se smatra da je potrebno obaviti više koraka u sustavu da bi se izdao račun. Međutim, budući da u tekstu zadatka nije precizno navedeno što sve točno podrazumijeva funkcionalnost izdavanja računa, može se smatrati i da je to samo jedan korak unutar tijeka obrasca Naplata, a u tom se slučaju ne mora nužno prikazivati kao zasebni obrazac.

Ovaj je dijagram moguće razlomiti u tri manja dijagrama, pri čemu svaki od njih prikazuje funkcionalnosti jednog od aktora. U nastavku se nalazi primjer takvog izdvojenog dijagrama za aktora Gost.



Slika 10.2: Dijagram obrazaca uporabe za funkcionalnosti gosta u sustavu za upravljanje hotelom

- **Zadatak 10.2 Prodaja autobusnih karata**

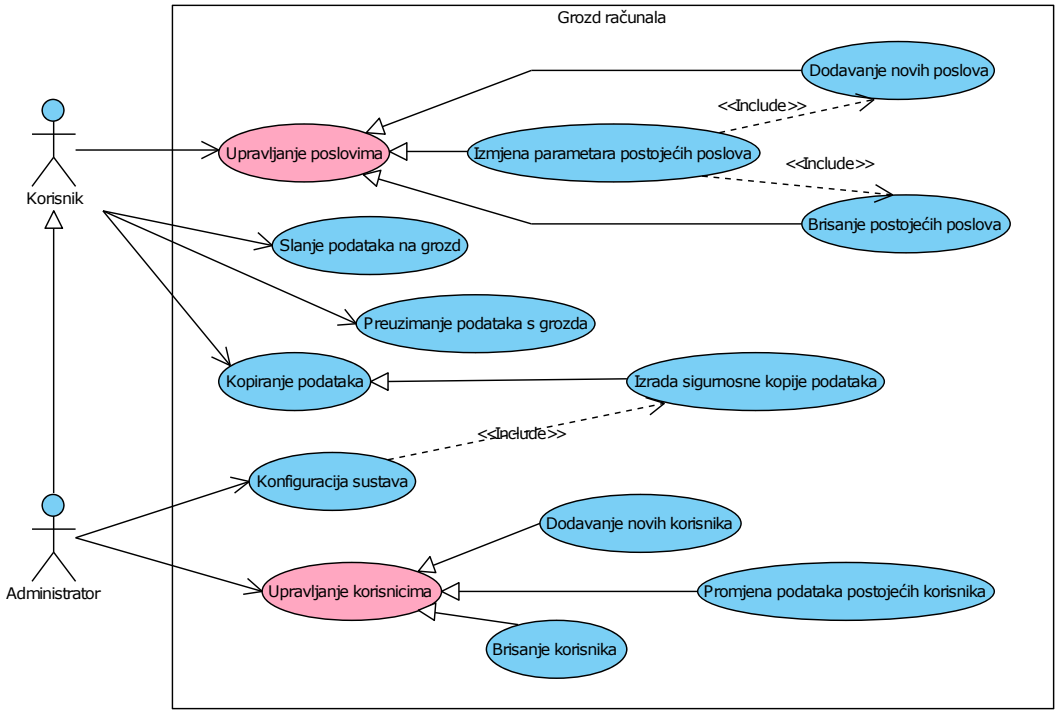


Slika 10.3: Dijagram obrazaca uporabe sustava prodaje autobusnih karata

Komentar: „Unos PIN-a" nije prikazan kao obrazac uporabe jer obuhvaća premali dio funkcionalnosti. Ta funkcionalnost modelirana je kroz naznaku sudjelovanja putnika u obrascu uporabe „Naplata karticom" (dvosmjerna asocijacija).

Ako bi se „Unos PIN-a" htjelo prikazati kao zasebni obrazac uporabe, on bio onda bio povezan s obrascem „Naplata karticom" vezom «include».

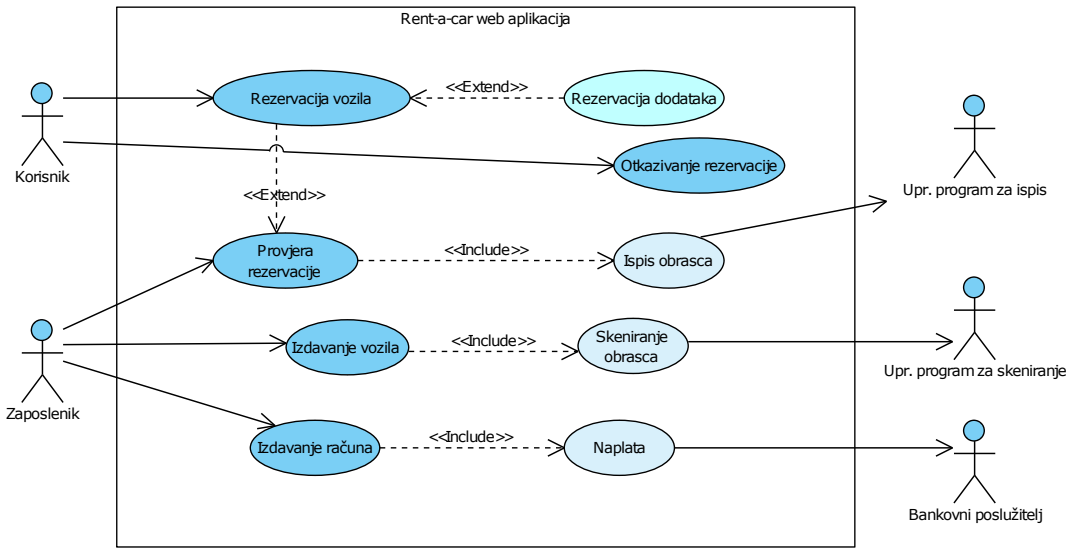
- **Zadatak 10.3 Grozd računala**



Slika 10.4: Dijagram obrazaca uporabe grozda računala

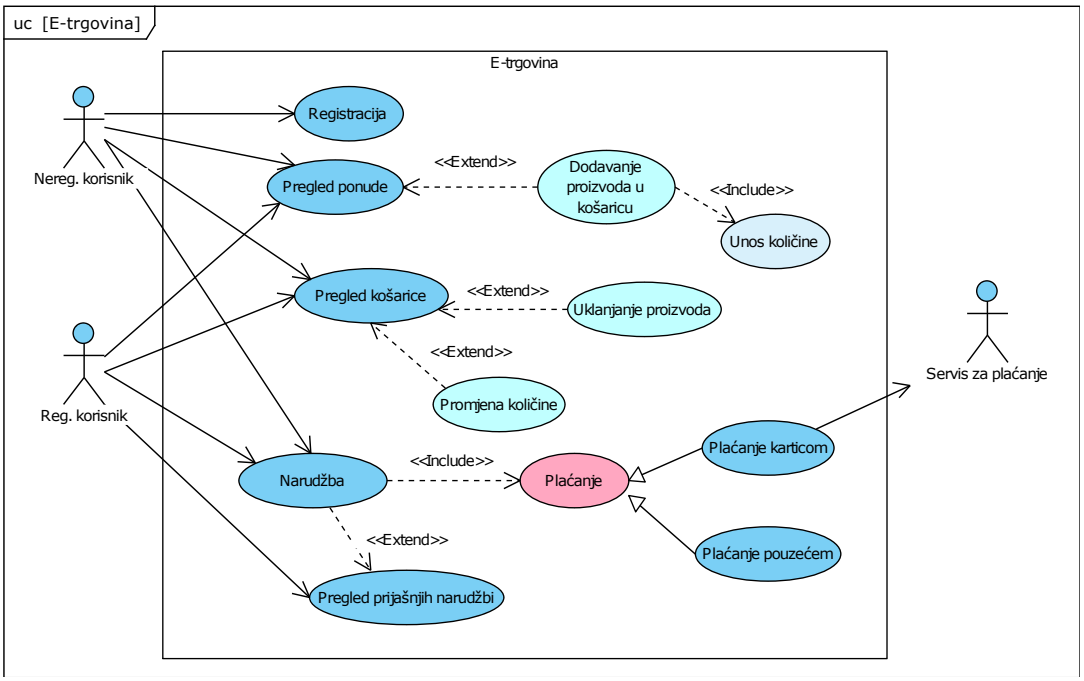
Komentar: Iako se obrazac uporabe „Upravljanje poslovima" izričito ne spominje u tekstu, obrasci vezani za stvaranje, uređivanje i brisanje (tzv. CRUD operacije) objekta najčešće se prikazuju kao specijalizacija obrasca Upravljanje. No, rješenje u kojem nema obrasca „Upravljanje poslovima" i s njim povezanih generalizacija jednako je valjano.

- **Zadatak 10.4 Rent-a-car tvrtka**



Slika 10.5: Dijagram obrazaca uporabe web aplikacije rent-a-car tvrtke

- **Zadatak 10.5 E-trgovina**



Slika 10.6: Dijagram obrazaca uporabe sustava e-trgovine

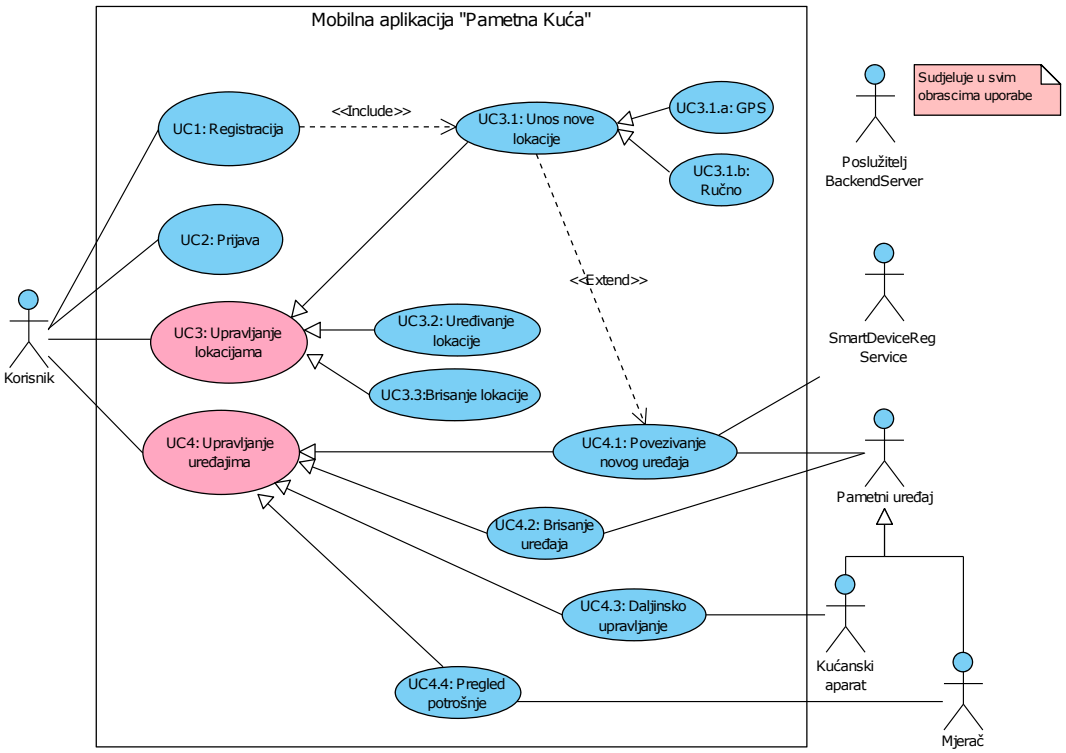
Komentar: Unos količine proizvoda pri dodavanju u košaricu ne mora se nužno prikazivati kao zaseban obrazac jer se u osnovi radi o maloj količini funkcionalnosti.

- **Zadatak 10.6 Portal za oglašavanje**

Slika 10.7: Dijagram obrazaca uporabe portala za oglašavanje

Komentar: Generalizirani obrasci vezani za rad s oglasom i upravljanje profilom mogu se izostaviti. Nadalje, brisanje korisničkog profila od strane korisnika i administratora ista je funkcionalnost, stoga je taj obrazac, koji je u osnovi specijalizacija upravljanja profilom, iskorišten i kao opcija obrasca „Pregled korisnika". Funkcionalnost plaćanja članarine mora biti dostupna i pri objavi oglasa (veza «extend» na obrazac „Objava oglasa"), tako da korisnik može uplatiti članarinu pri objavi oglasa ako ga želi objaviti u kategoriji koja se plaća, a nema već uplaćenu članarinu.

- **Zadatak 10.7 Pametna Kuća**



Slika 10.8: Dijagram obrazaca uporabe za mobilnu aplikaciju „Pametna Kuća”

Komentar: Generalizirani obrasci UC3 i UC4, vezani za upravljanje lokacijama i uređajima, mogu se izostaviti te se mogu prikazati samo njihovi specijalizirani obrasci.

11. Sekvencijski UML dijagrami

11.1 Zadaci

■ **Zadatak 11.1 — Upravljanje hotelom – upravljanje sobama.** Pomoću sekvencijskog dijagrama modelirajte funkcionalnost upravljanja sobama u sustavu za upravljanje hotelom iz zadatka 10.1. Osim osnovnog opisa uzmite u obzir i sljedeće.

Sekvenca počinje tako što administrator odabere opciju upravljanja sobama, na što mu sustav za upravljanje hotelom prikazuje sučelje za upravljanje. Administrator tada izvodi jednu od sljedećih aktivnosti: dodavanje nove sobe, uređivanje ili brisanje postojeće sobe. Za uređivanje ili brisanje, administrator treba najprije zatražiti pregled svih postojećih soba. ■

■ **Zadatak 11.2 — Upravljanje hotelom – rezervacija sobe.** Pomoću sekvencijskog dijagrama modelirajte postupak rezervacije sobe u sustavu za upravljanje hotelom iz zadatka 10.1. Osim osnovnog opisa uzmite u obzir i sljedeće.

Postupak započinje tako što gost pretražuje ponudu hotela postavljanjem upita za razdoblje u kojem želi rezervaciju i broj osoba. Sustav mu zatim ispisuje jednu ponudu ili više njih. Postupak se ponavlja sve dok gost ne pronađe odgovarajuću ponudu. Nakon toga gost odabire opciju rezervacije sobe, a sustav mu zatim prikazuje ponudu dodatnih mogućnosti: rezervacija parkirnog mjesta i dokup opcija. Ako želi, gost može odabrati rezervaciju parkirnog mjesta. Također, ako želi, može dokupiti jednu od opcija: doručak, polupansion ili puni pansion. Konačno, gost potvrđuje rezervaciju i sustav je pohranjuje. ■

■ **Zadatak 11.3 — Prodaja autobusnih karata.** Modelirajte sekvencijskim dijagramom postupak prodaje autobusnih karata u sustavu za prodaju karata iz zadatka 10.2. Osim osnovnog opisa uzmite u obzir i sljedeće.

Blagajnik najprije unosi u aplikaciju podatke o odredištu i datumu polaska za koje će prodati kartu. Aplikacija zatim ispisuje cijenu. Potrebno je modelirati oba načina plaćanja: gotovinom i karticom, kao i mogućnost izdavanja računa R1. Pri gotovinskom plaćanju putnik predaje novčanice blagajniku, a on u aplikaciji potvrđuje da je karta plaćena, aplikacija pohranjuje podatke te ispisuje kartu i račun. Pri kartičnom plaćanju blagajnik u aplikaciji pokreće naredbu naplate karticom, a putnik zatim provlači karticu i unosi PIN. Za provedbu naplate aplikacija se spaja na bankovni poslužitelj. Nakon uspješnog završetka transakcije aplikacija pohranjuje podatke te se ispisuju karta i račun. Zbog jednostavnosti pretpostavite da putnik uvijek unosi točan PIN i da transakcija

uvijek uspijeva. Modelirajte i mogućnost izdavanja R1 računa. Zanemarite mogućnost da putnik odustaje od kupnje karte zbog neprihvatljive cijene. ■

■ **Zadatak 11.4 — Rent-a-car tvrtka – provjera rezervacije i izdavanje vozila.** Pomoću sekvencijskog dijagrama modelirajte postupak provjere rezervacije i izdavanja vozila u *web*-aplikaciji *rent-a-car* tvrtke iz zadatka 10.4. Osim osnovnog opisa uzmite u obzir i sljedeće.

Zaposlenik najprije provjerava rezervaciju te ako ona ne postoji, stvara novu. Za stvaranje rezervacije najprije unosi razdoblje, na što mu sustav prikazuje popis dostupnih automobila. Zaposlenik odabire automobil i unosi korisnikove podatke te se rezervacija pohranjuje. U ovom je koraku moguće opcionalno rezervirati dodatke (GPS uređaj, dječju sjedalicu...) ako to klijent želi. Nakon toga zaposlenik sustavu daje naredbu za ispis obrasca za preuzimanje, a sustav šalje obrazac upravljačkom programu za pisač. Nakon što je obrazac ispisan i potpisan, zaposlenik sustavu daje naredbu za skeniranje obrasca, a sustav se spaja s upravljačkim programom za skener, dohvaća sken obrasca i pohranjuje ga lokalno. Upravljački programi za pisač i skener dio su operacijskog sustava računala, tj. nisu dio programskog sustava *rent-a-car*. ■

■ **Zadatak 11.5 — Portal za oglašavanje – objava oglasa.** Pomoću sekvencijskog dijagrama modelirajte postupak objave oglasa za koji je nužno imati plaćenu članarinu na *web*-portalu za oglašavanje iz zadatka 10.6. Osim osnovnog opisa uzmite u obzir i sljedeće.

Postupak počinje tako što oglašivač odabere opciju stvaranja novog oglasa te mu se prikaže forma za unos oglasa. Oglašivač popunjava podatke za oglas, na što sustav provjerava jesu li uneseni svi nužni podaci. Oglašivač može objaviti oglas tek kada su uneseni svi nužni podaci, a do tad ga sustav vraća na formu za unos oglasa. Nakon što su uneseni svi nužni podaci i korisnik odabere objavu oglasa, sustav provjerava status članstva. Ako oglašivač ima uplaćenu članarinu, oglas se pohranjuje. Inače se prikazuje forma za plaćanje. Oglašivač tada može odustati i poništiti oglas ili unijeti podatke za plaćanje. Ako su uneseni podaci za plaćanje, naplata se provodi putem vanjskog servisa za naplatu te se oglas pohranjuje. Transakcija naplate ne bi trebala trajati više od 20 sekundi, a sveukupno od unosa podataka za plaćanje do potvrde o pohrani oglasa ne bi trebalo proći više od 30 sekundi. Pretpostavite da je transakcija uvijek uspješna. ■

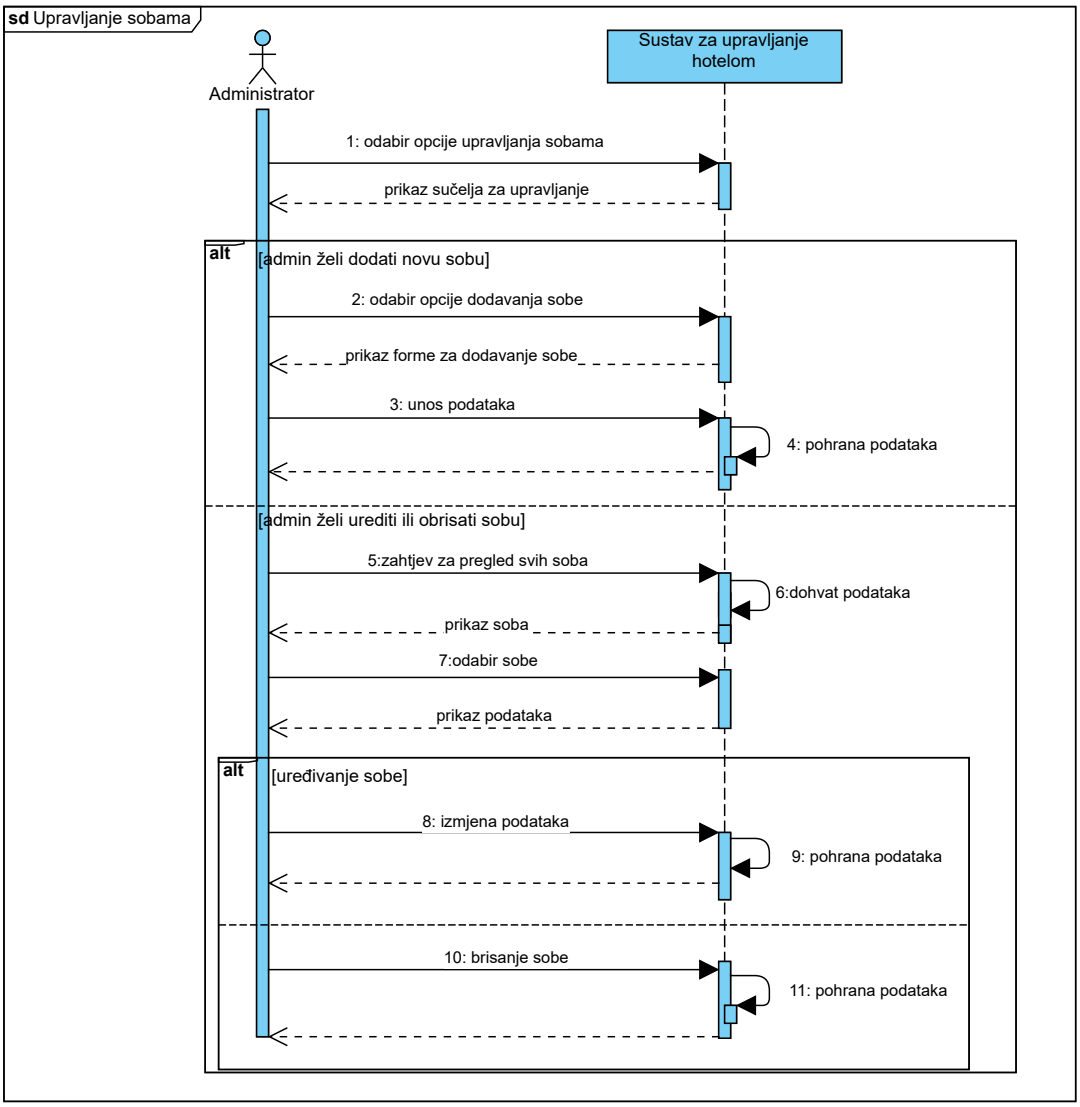
■ **Zadatak 11.6 — Pametna Kuća – povezivanje novog uređaja.** Pomoću sekvencijskog dijagrama modelirajte funkcionalnost povezivanja novog pametnog uređaja s mobilnom aplikacijom „Pametna Kuća" iz zadatka 10.7. Osim osnovnog opisa uzmite u obzir i sljedeće.

Proces povezivanja novog pametnog uređaja počinje tako što korisnik odabere opciju povezivanja novog uređaja. Zatim se zadaje lokacija (modelirati i mogućnost unosa nove lokacije). Nakon toga korisnik unosi kôd uređaja, aplikacija dohvaća podatke od vanjske usluge SmartDeviceRegService. Ako se usluzi pošalje neispravan kôd, proces registracije se prekida i aplikacija obavještava korisnika da kôd nije ispravan. U suprotnom se aplikacija pokušava spojiti na pametni uređaj najviše pet puta s istekom utvrđenog vremenskog ograničenja (engl. *timeout*) od pet sekundi. Ako ne dobije odgovor od uređaja, proces povezivanja s uređajem prekida se i aplikacija obavještava korisnika o neuspjeloj registraciji.

Pohranu podataka na poslužitelj BackendServer nije potrebno prikazivati. ■

11.2 Rješenja

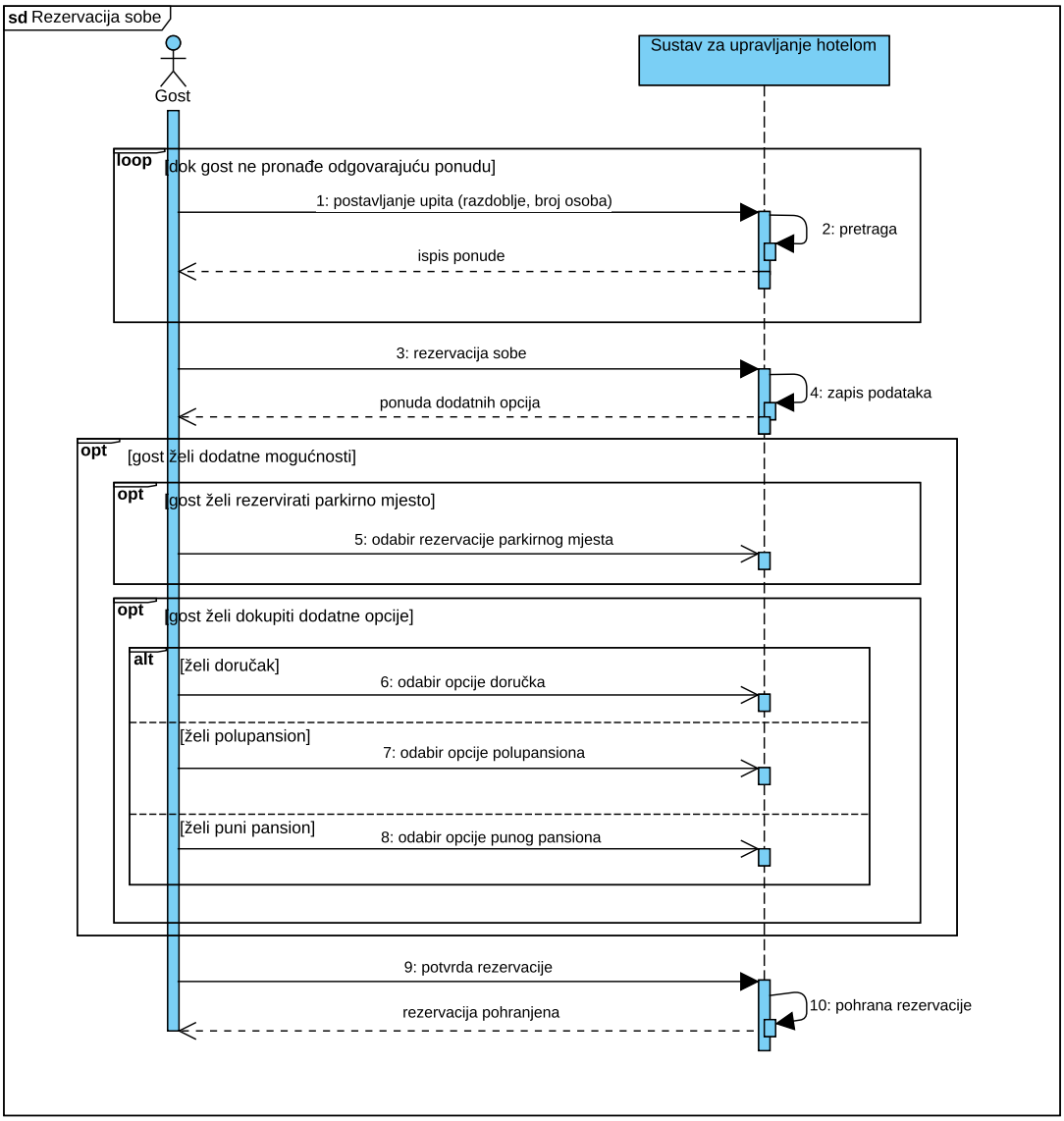
- **Zadatak 11.1 Upravljanje hotelom – upravljanje sobama**



Slika 11.1: Sekvencijski dijagram za postupak upravljanja sobama u hotelu

Komentar: Poruke odgovora u kojima se samo potvrđuje uspješnost provedbe neke akcije ne moraju nužno imati naziv (npr. odgovor na poruke 3 i 10).

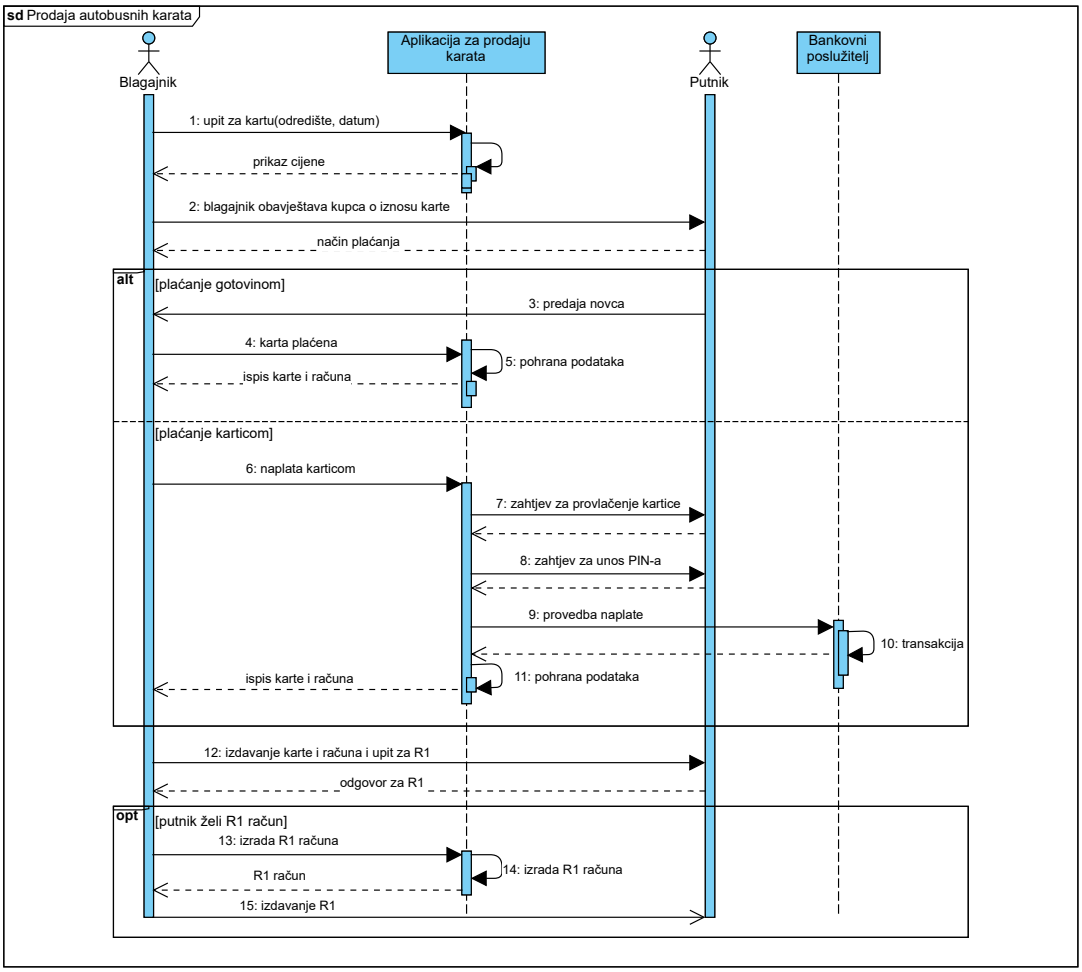
- **Zadatak 11.2 Upravljanje hotelom – rezervacija sobe**



Slika 11.2: Sekvencijski dijagram za postupak rezervacije sobe u hotelu

Komentar: Odabir opcija rezervacije parkinga, doručka, polupansiona i punog pansiona (poruke 5 – 8) prikazan je asinkronim porukama. To se može tumačiti tako što su sve raspoložive opcije gostu prikazane na istom ekranu i on ih po želji odabire (npr. stavlja kvačicu, ili odabire iz liste), a konačan odgovor od sustava stiže tek na kraju, kada je odabrana potvrda rezervacije. Međutim, moguće je umjesto asinkronih koristiti sinkrone poruke, ali tada bi se sekvenca tumačila tako da za svaku odabranu opciju korisnik odmah dobiva neki odgovor od sustava.

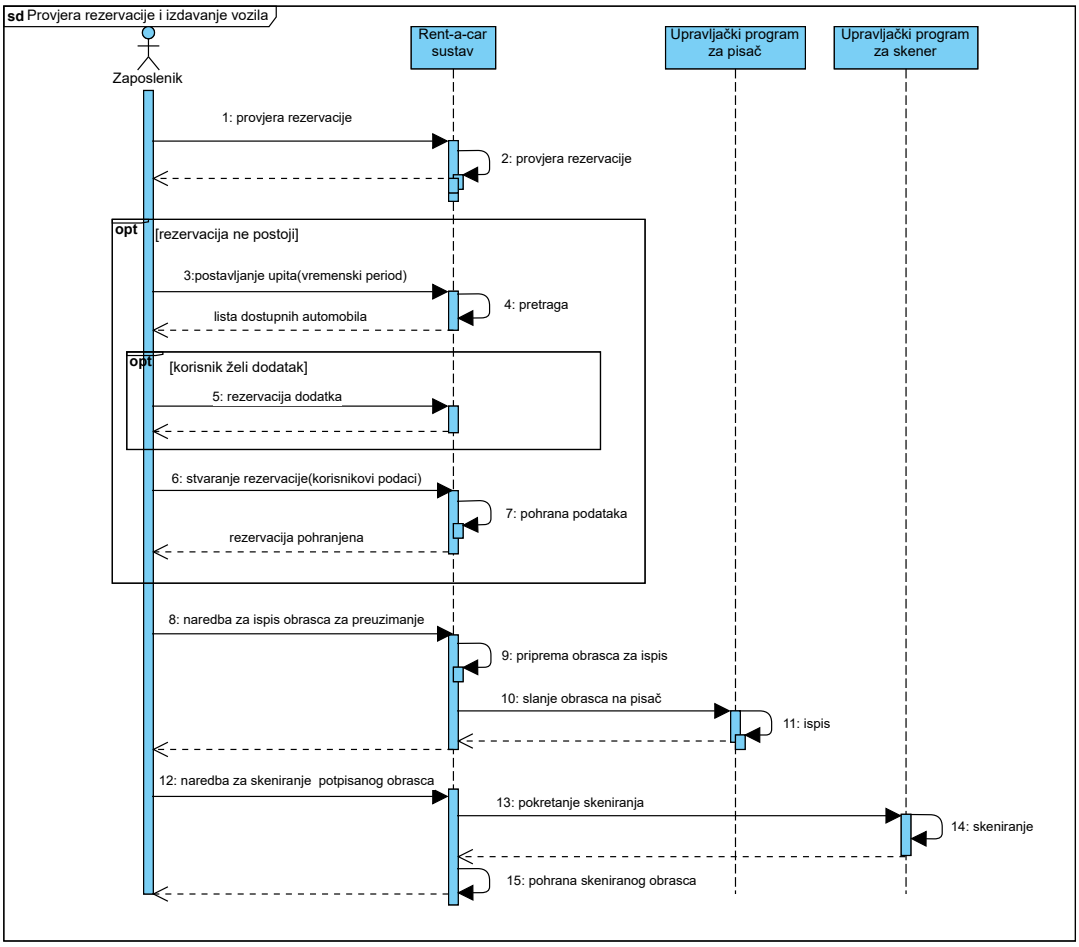
- **Zadatak 11.3 Prodaja autobusnih karata**



Slika 11.3: Sekvencijski dijagram za postupak prodaje autobusne karte

Komentar: Poruke 2, 12 i 15, u kojima je prikazana izravna (usmena) komunikacija između blagajnika i putnika, moguće je izostaviti jer se izvode neovisno o aplikaciji za prodaju karata (neće biti dio implementacije, tj. programske podrške). Međutim, one su ovdje prikazane zbog jasnoće tijeka samog postupka. Poruku 10, kojom je prikazano izvođenje transakcije na bankovnom poslužitelju, također je moguće izostaviti i umjesto nje samo prikazati odgovor na poruku 9 jer nije dio programske podrške aplikacije za prodaju karata.

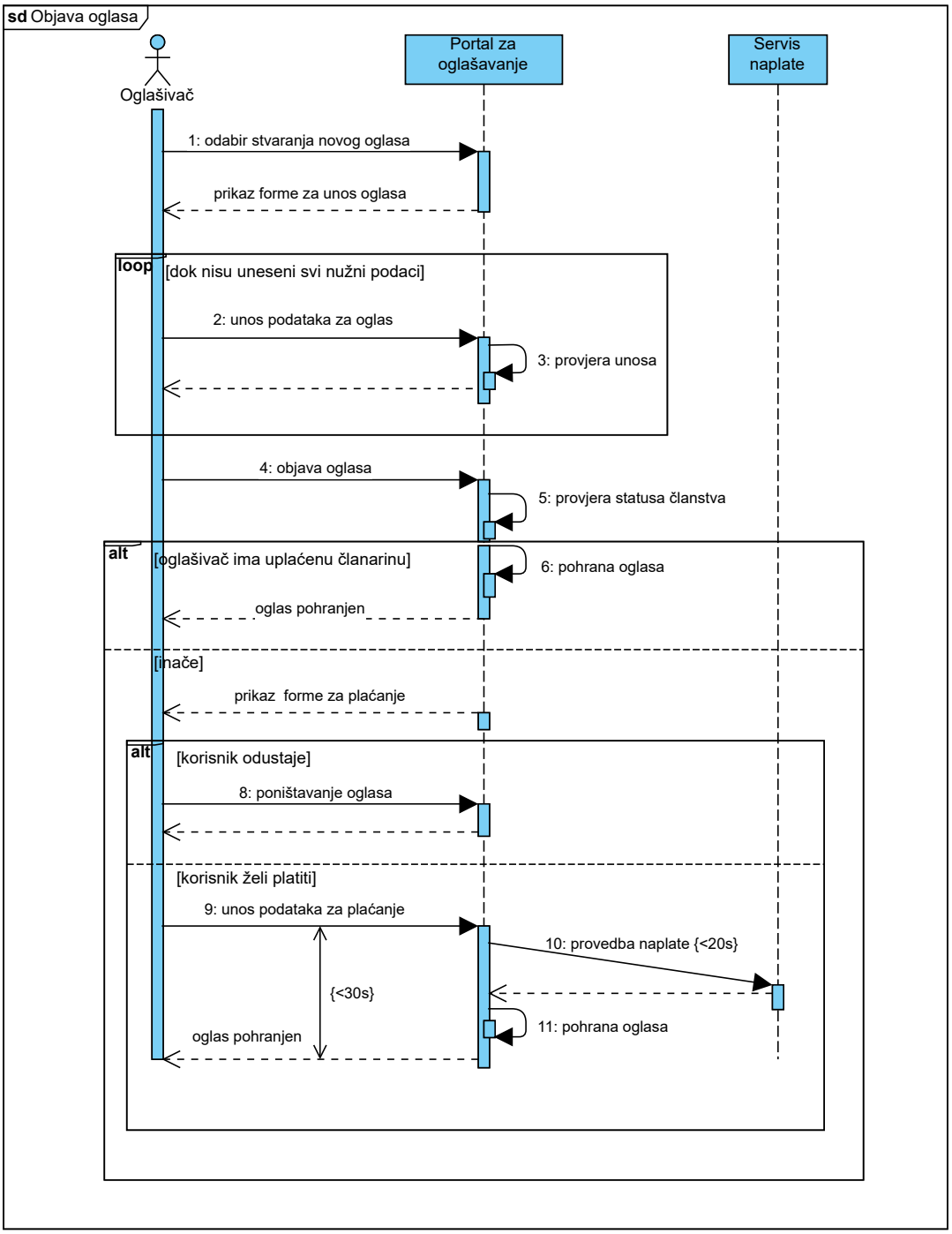
- **Zadatak 11.4 Rent-a-car tvrtka – provjera rezervacije i izdavanje vozila**



Slika 11.4: Sekvencijski dijagram za provjeru rezervacije i izdavanje vozila

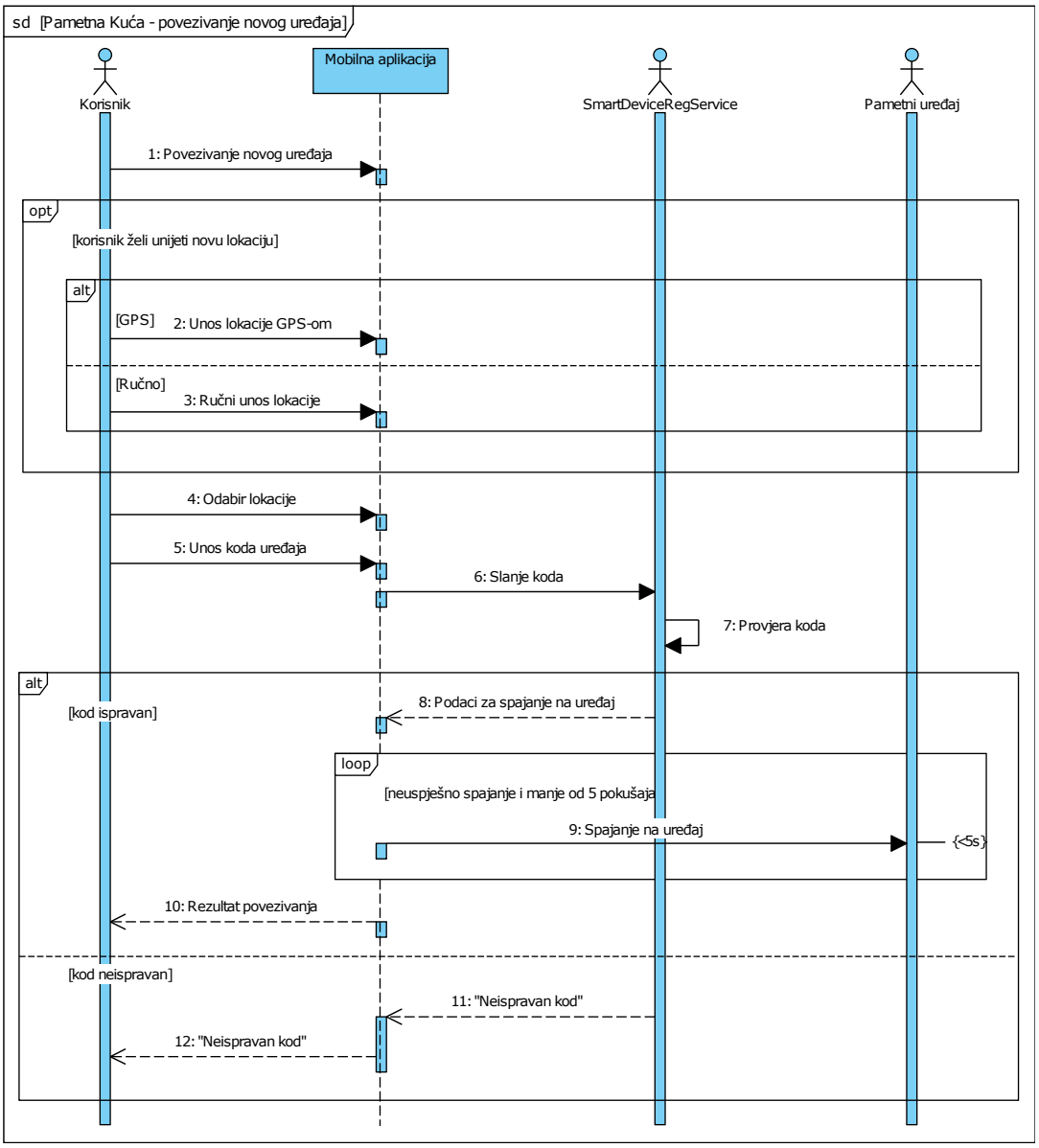
Komentar: Poruke 11 i 14 mogu se izostaviti te se umjesto njih može samo prikazati odgovor na poruke 10, odnosno 13 jer ispis i skeniranje nisu dio programske podrške sustava *rent-a-car*.

- **Zadatak 11.5 Portal za oglašavanje – objava oglasa**



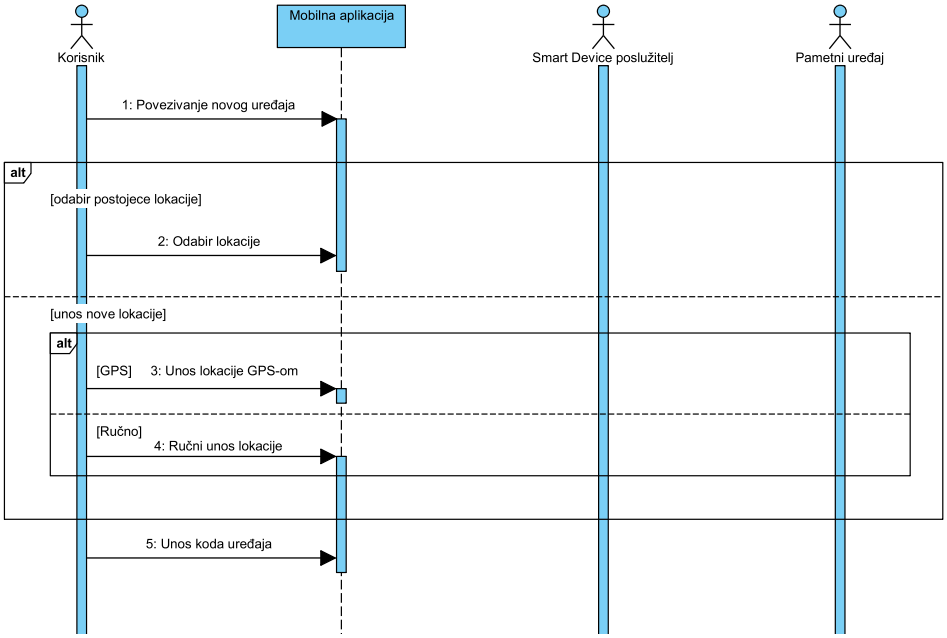
Slika 11.5: Sekvencijski dijagram za postupak objave oglasa na portalu za oglašavanje

- **Zadatak 11.6 Pametna Kuća – povezivanje novog uređaja**



Slika 11.6: Sekvencijski dijagram za postupak povezivanja novog uređaja u mobilnoj aplikaciji „Pametna Kuća”

Komentar: Opis tijeka zadavanja lokacije moguće je tumačiti i tako da se unosom nove lokacije automatski odabire ta lokacija. U tom je slučaju unos lokacije moguće modelirati korištenjem okvira *alt* kao na slici u nastavku:



Slika 11.7: Alternativno rješenje sekvencijskog dijagrama za unos lokacije

12. UML dijagrami razreda

12.1 Zadaci

■ **Zadatak 12.1 — Tvrtka za popravak informatičke opreme.** Modelirajte pomoću dijagrama razreda aplikaciju koja pomaže tvrtki koja se bavi popravkom informatičke opreme u praćenju popravaka.

U tvrtki je zaposleno više servisera i jedan voditelj. Oni su djelatnici tvrtke čiji se podaci brišu njezinim zatvaranjem. Za svakog djelatnika zapisani su sljedeći podaci: ime, prezime, OIB. Voditelj otvara radne naloge, a na svakom radnom nalogu radi najmanje jedan serviser.

Nakon otvaranja radnog naloga voditelj inicijalno procjenjuje trajanje radnog naloga i upisuje taj podatak u radni nalog. Jednom radnom nalogu pridružena je jedna stranka, u čije je ime radni nalog otvoren. Za svaku stranku evidentira se: ime, prezime i broj telefona. Jedan radni nalog može se sastojati od jednog radnog naloga ili više njih ovisno o složenosti popravaka.

Status radnog naloga može biti: „zaprimljen", „u obradi", „završen – uspješno" i „završen – neuspješno". Svaki radni nalog sadržava barem jednu servisnu akciju. Za svaku servisnu akciju evidentira se broj utrošenih čovjek-sati i cijena čovjek-sata i prema tome se izračunava cijena popravka za radni nalog. Akcije se dijele na: dijagnostiku, podešavanje i ugradnju. Ugradnja sadržava barem jednu računalnu komponentu koja se ugrađuje za vrijeme akcije. Svaka komponenta ima svoju cijenu. ■

■ **Zadatak 12.2 — Program za crtanje.** Modelirajte elemente jednostavnog programa za crtanje prema sljedećem opisu.

Osnovni je element programa za crtanje radna površina. Na radnoj se površini mogu crtati različiti oblici: linije, poligoni i elipse. Za svaki je oblik utvrđena RGB boja (boja koja se sastoji od R, G i B komponente). Linija je određena svojom početnom i završnom točkom. Poligon je određen skupom linija (barem tri) koje predstavljaju njegove stranice. Elipsa je određena skupom točaka koje omeđuju njenu površinu. Točke su određene koordinatama x i y.

Za sve oblike moguće su operacije transformacije: rotacija, translacija, smanjenje i povećanje. Elipse i poligone moguće je animirati. Animacija se utvrđuje dinamički, a moguće je odabrati između dvije vrste: FadeIn i FadeOut. Obje vrste animacije rade na isti način i za elipse i za poligone. ■

■ **Zadatak 12.3 — Programska potpora za licenciranje.** Modelirajte dijagramom razreda programsku potporu za licenciranje koja je opisana u nastavku.

Programska potpora za licenciranje omogućuje prodavaču licenciranje pojedinačnog programskog proizvoda, dijelova proizvoda i cijele linije proizvoda. Programski proizvod sastoji se od jednog dijela ili više njih (npr. prevoditelji za programske jezike, programski moduli za neku funkcionalnost i sl).

Linija proizvoda obuhvaća skup proizvoda srodnih funkcionalnosti i sve njihove dijelove. Proizvodi mogu pripadati samo jednoj liniji, a dijelovi proizvoda mogu istodobno biti dio više linija. Proizvodi iz iste linije mogu imati neke iste dijelove. Za svaku liniju proizvoda određen je osnovni proizvod za koji kupac obavezno mora imati licencu ako želi koristiti bilo što iz nje.

Kupac može kupiti licencu za proizvode i dijelove proizvoda, a istu licencu ima samo jedan kupac. Licenca sadržava ključ koji kupac dobiva i koji može biti alfanumerički kôd ili fizički USB ključ. Na osnovi vrste ključa pri kupnji licence generira se alfanumerički kôd ili vrši programiranje USB ključa.

Kupci se dijele na privatne i poslovne te svaki od njih ima svoj identifikacijski broj. Za privatne osobe pamte se ime, prezime, adresa e-pošte i kontakt telefon. Za poslovne osobe pamte se naziv tvrtke, adresa e-pošte i kontakt telefon. Za svakog se kupca utvrđuje adresa isporuke robe i adresa isporuke računa, koje sadržavaju naziv ulice i kućni broj, poštanski broj, grad i državu. ■

■ **Zadatak 12.4 — E-trgovina.** Modelirajte dijagram razreda sustava narudžbe i plaćanja u e-trgovini iz zadatka 10.5 na konceptualnoj razini. Osim osnovnog opisa uzmite u obzir i sljedeće.

Za svaki proizvod naveden je naziv, opis i cijena po jediničnoj količini. Pri izradi narudžbe sustav od korisnika traži unos sljedećih informacija: željeni datum i vrijeme dostave, adresu dostave, broj telefona i adresu e-pošte. Adresa dostave uključuje naziv ulice, broj kuće, naziv grada i poštanski broj. Kada je riječ o registriranim korisnicima, sustav već sadržava informacije o njihovoj adresi dostave, broju telefona i adresi e-pošte, ali im omogućuje da ih izmijene za tu narudžbu.

Sustav cijelo vrijeme prati status narudžbe. Status može imati jednu od tri diskretne vrijednosti: „U TIJEKU" – prije potvrde narudžbe, „POSLANA" – nakon potvrde narudžbe do dostave i „ISPORUČENA" – nakon uspješne dostave narudžbe. ■

■ **Zadatak 12.5 — Portal za oglašavanje.** Modelirajte dijagram razreda portala za oglašavanje iz zadatka 10.6 na konceptualnoj razini. Osim osnovnog opisa uzmite u obzir i sljedeće.

Kategorije oglasa su: Nekretnine, Auto-moto, Roba i Usluge. Svaki oglas sadržava naslov, opis, cijenu usluge, naziv oglašivača i kontakt te može sadržavati slike i kratke video materijale, dok ostali podaci mogu varirati od kategorije do kategorije, npr. za nekretnine treba navesti površinu nekretnine.

Potrebni podaci za registraciju fizičke osobe su: OIB, ime, prezime, fizička adresa, adresa e-pošte i kontakt telefon. Poslovni subjekt pri registraciji također mora navesti svoj OIB te još naziv tvrtke, sjedište tvrtke, adresu e-pošte i kontakt telefon. ■

■ **Zadatak 12.6 — Pametna Kuća.** Modelirajte dijagram razreda mobilne aplikacije iz zadatka 10.7 na konceptualnoj razini. Osim osnovnog opisa uzmite u obzir i sljedeće.

Korisnik pri registraciji unosi željeno korisničko ime, lozinku, adresu e-pošte te barem jednu lokaciju. Za svaku lokaciju na kojoj se nalaze pametni uređaji zadani su ime i koordinate. Korisnik može u jednom trenutku imati pohranjeno do najviše 10 lokacija.

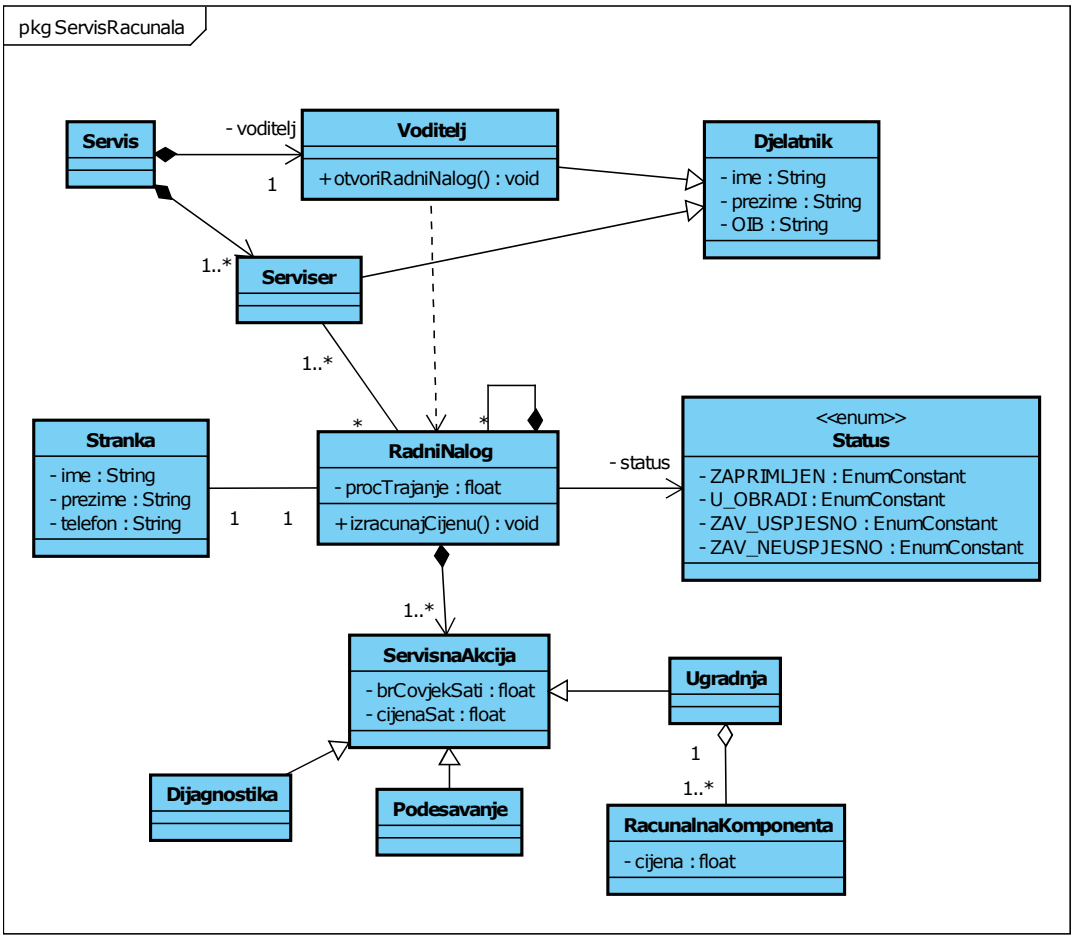
Korisnik može imati registrirano do najviše 100 pametnih uređaja. Za svaki povezani uređaj trajno se pohranjuju podaci za spajanje koje aplikacija dobije od usluge „SmartDeviceRegService".

Za mjerače je moguće pregledati potrošnju energije po danima unazad 365 dana, a za svaki se dan bilježi ukupni potrošeni iznos i mjerna jedinica. Kućanski aparati imaju jednu dostupnu funkciju ili više njih, a svaka je od njih određena svojim nazivom, opisom i trenutnim statusom. Status funkcije može biti „IZVODENJE", „DOSTUPNA", „NEDOSTUPNA". Funkcije je moguće pokrenuti, zaustaviti i vidjeti njihov trenutni status.

Nije potrebno modelirati komunikaciju s vanjskim poslužiteljem „BackendServer" i uslugom „SmartDeviceRegService". ■

12.2 Rješenja

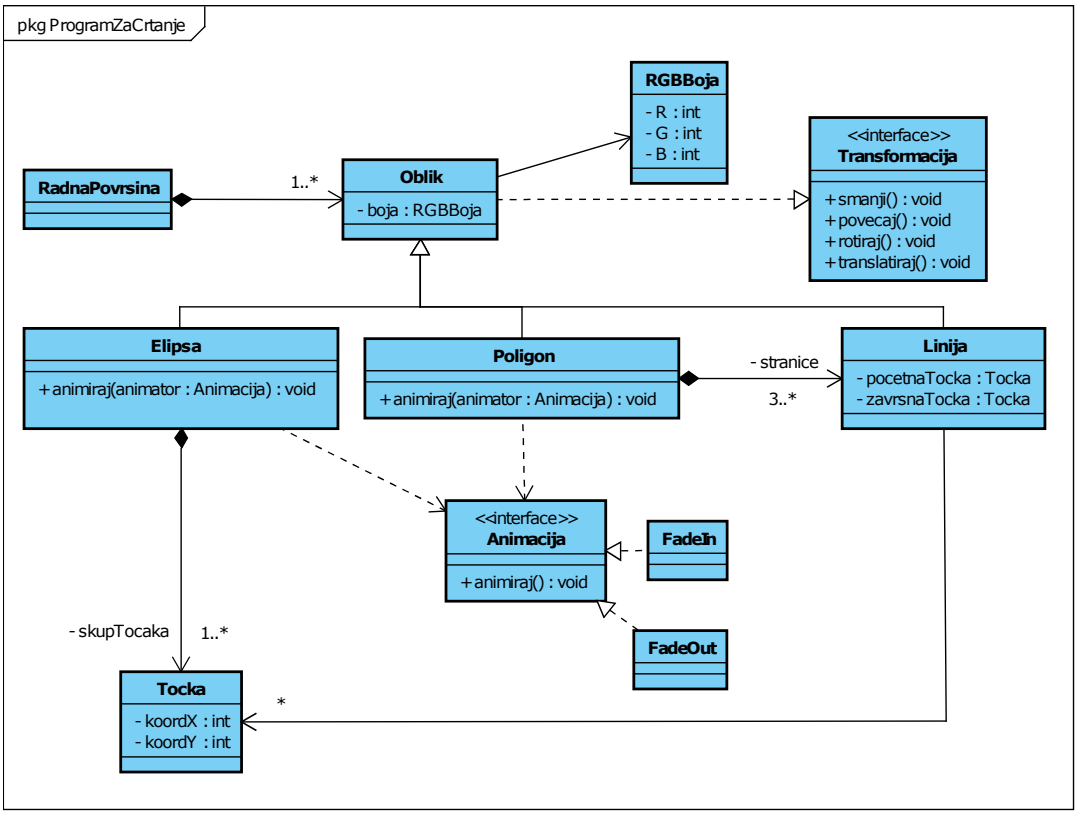
- **Zadatak 12.1 Tvrtka za popravak informatičke opreme**



Slika 12.1: Konceptualni dijagram razreda programske potpore tvrtke za popravak informatičke opreme

Komentar: Voditelj stvara radni nalog putem metode otvoriRadniNalog() i stoga je razred Voditelj povezan s razredom RadniNalog vezom ovisnosti. Nadalje, budući jedan radni nalog može sadržavati više podnaloga, razred RadniNalog ima refleksivnu vezu kompozicije na samog sebe.

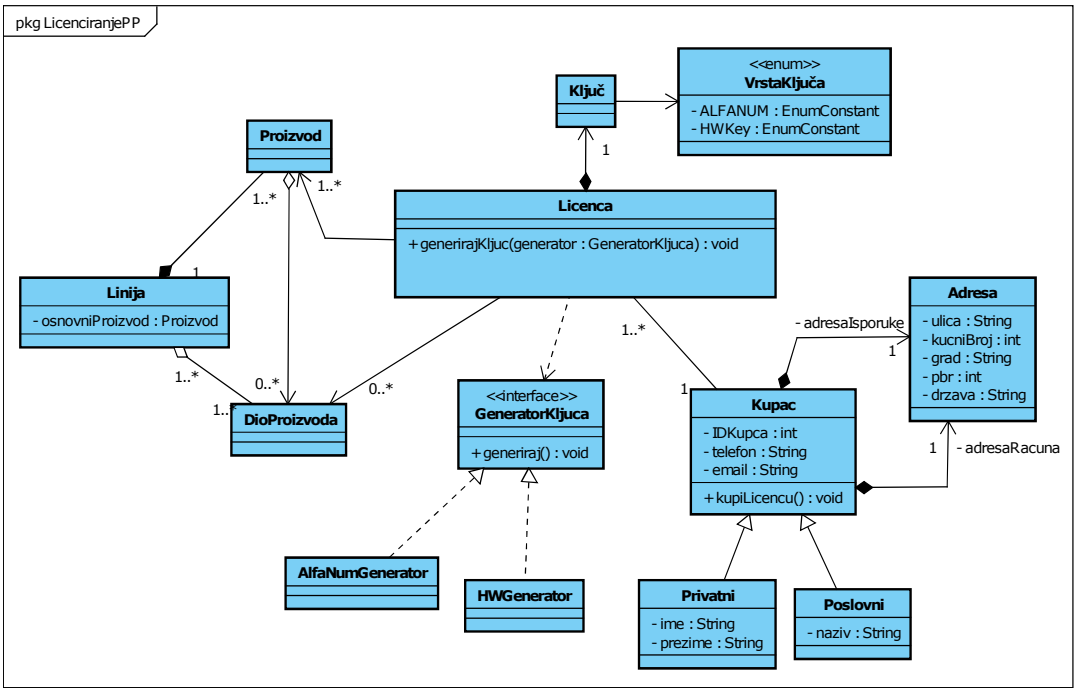
- **Zadatak 12.2 Program za crtanje**



Slika 12.2: Konceptualni dijagram razreda programske potpore za crtanje

Komentar: Budući da je za sve oblike zadano da se nad njima moraju moći izvesti operacije transformacije (rotacija, translacija, smanjenje i povećanje), te su operacije izdvojene u sučelje Transformacija, a svaki konkretni razred koji nasljeđuje razred Oblik implementirat će način primjene tih operacija nad sobom. Kada je riječ o animaciji, zadane su dvije vrste animacije koje rade na isti način i za elipsoide i za poligone te je stoga konkretna animacija implementirana u razredima FadeIn i FadeOut, a razredi Elipsoid i Poligon koriste tu implementaciju tako što pozivaju metodu animiraj iz sučelja Animacija te su zato povezani vezom ovisnosti s tim sučeljem.

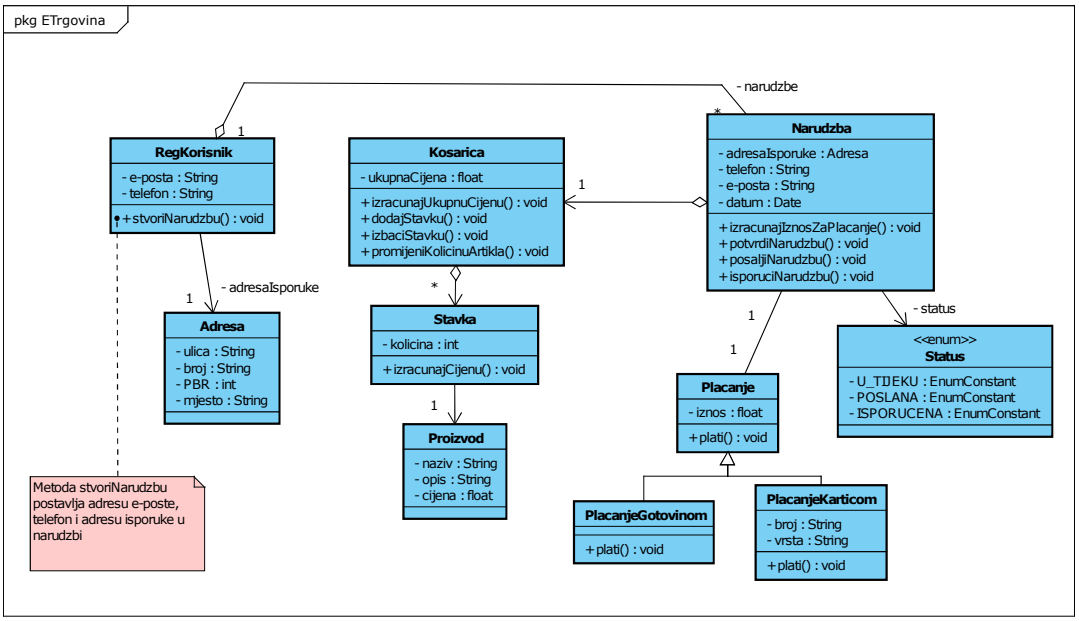
- **Zadatak 12.3 Programska potpora za licenciranje**



Slika 12.3: Konceptualni dijagram razreda programske potpore za licenciranje

Komentar: Budući da adresa sadržava nekoliko elemenata, nije ih preporučljivo pisati kao jedan niz znakova (String) nego su izdvojeni u cjelinu kao zaseban razred. Metoda kupiLicencu() razreda Kupac predstavlja odgovornost u domenskom modelu, međutim u konkretnoj implementaciji (npr. prema arhitekturi MVC) ta odgovornost može prijeći u upravljački sloj (Controller). Licenca sadržava ključ koji dobiva kupac i koji može biti alfanumerički kôd ili fizički USB ključ. Kako se način generiranja ključa razlikuje ovisno o njegovoj vrsti, ostvarene su dvije konkretne implementacije sučelja GeneratorKljuca: AlfaNumGenerator i HWGenerator.

- **Zadatak 12.4 E-trgovina**



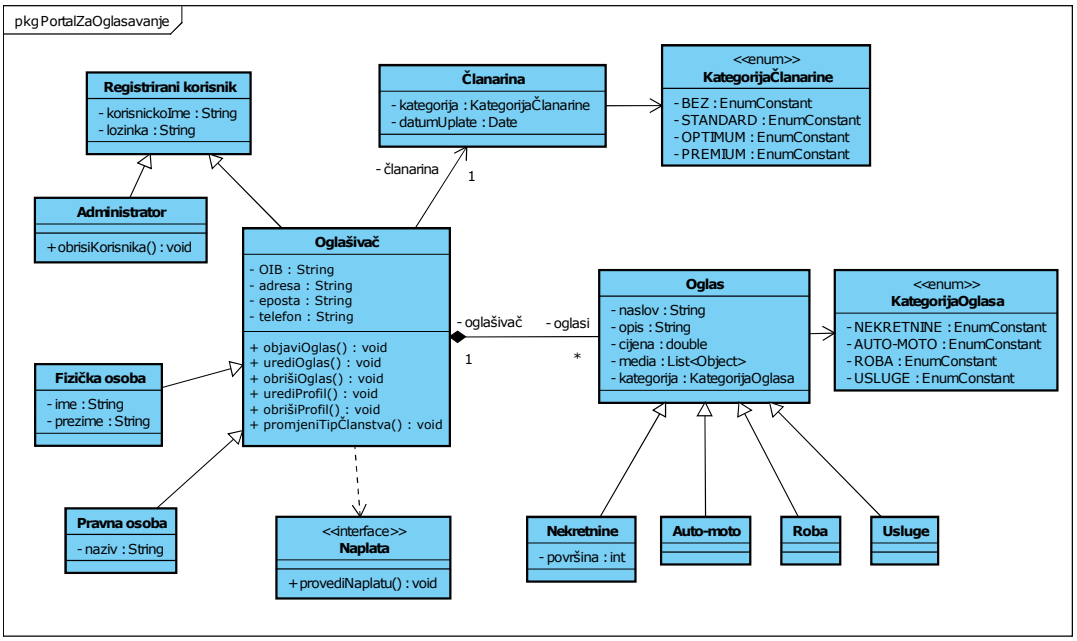
Slika 12.4: Konceptualni dijagram razreda programske potpore e-trgovine

Komentar: Atributi razreda mogu se navesti unutar samog razreda ili na vezi asocijacije (npr. adresaIsporuke : Adresa). Metoda potvrdiNarudzbu pokreće plaćanje, a metode posaljiNarudzbu i isporuciNarudzbu mijenjaju status Narudzbe. Razred Placanje apstraktan je te je nužno odabrati jednu od konkretnih implementacija plaćanja.

Metoda stvoriNarudzbu() razreda Kupac predstavlja odgovornost u domenskom modelu, međutim u konkretnoj implementaciji (npr. prema arhitekturi MVC), ta odgovornost može prijeći u upravljački sloj (Controller).

Posebno odustajanje od narudžbe nije potrebno modelirati jer narudžba u tom slučaju jednostavno neće biti pohranjena.

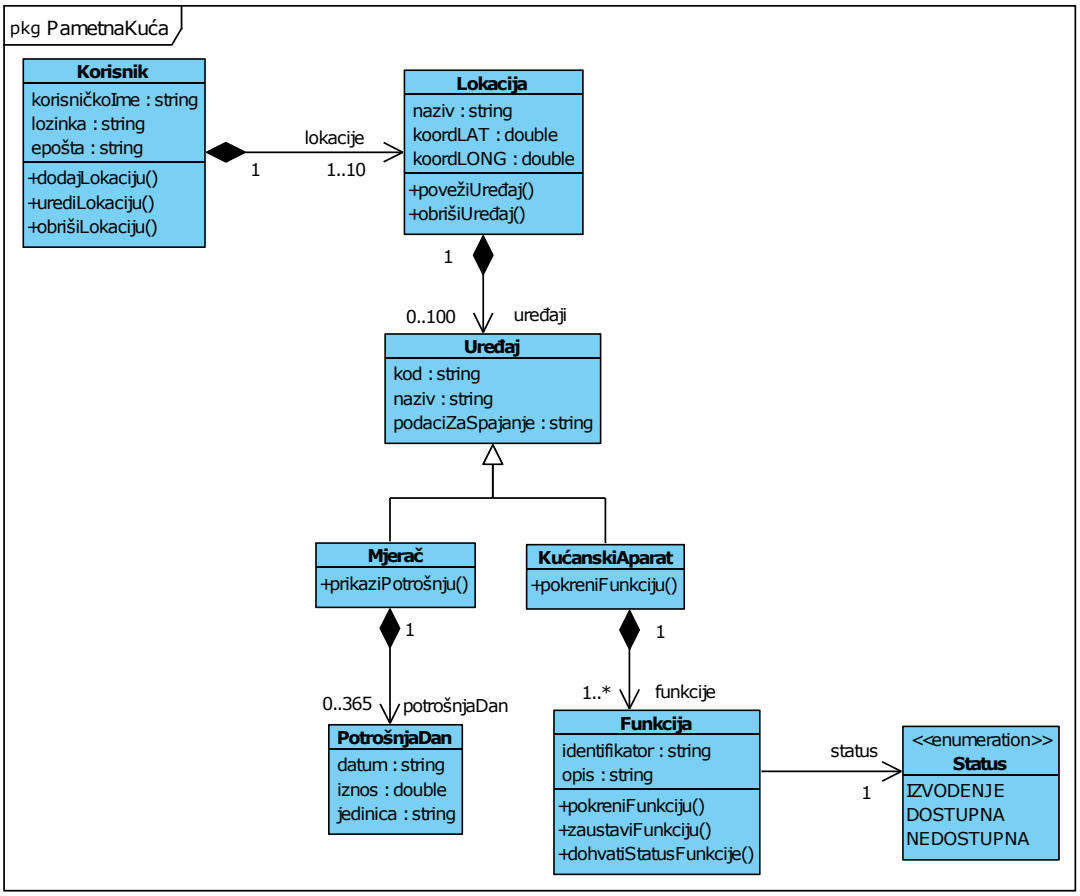
- **Zadatak 12.5 Portal za oglašavanje**



Slika 12.5: Konceptualni dijagram razreda programske potpore portala za oglašavanje

Komentar: Metode razreda Oglašivač i Administrator predstavljaju odgovornosti u domenskom modelu, međutim u konkretnoj implementaciji (npr. prema arhitekturi MVC) ta odgovornost može prijeći u upravljački sloj (Controller). Rješenje koje ne sadržava enumeraciju KategorijeOglasa također bi bilo prihvatljivo, no s obzirom na to da o vrsti oglasa ovisi hoće li oglašivač morati platiti članarinu, s implementacijske je strane elegantnije rješenje u kojem enumeracija postoji. Kategorije članarine istaknute su samo kao enumeracija KategorijaČlanarine jer razred Članarina ne mijenja atribute ni odgovornosti ovisno o tipu članarine te u tom slučaju nije nužno uvoditi nasljeđivanje.

- **Zadatak 12.6 Pametna Kuća**



Slika 12.6: Konceptualni dijagram razreda mobilne aplikacije „Pametna Kuća”

13. UML dijagrami stanja

13.1 Zadaci

■ **Zadatak 13.1 — Tablični kalkulator.** Modelirajte stanje ćelije u tabličnom kalkulatoru pomoću UML dijagrama stanja.

Ćelija u tabličnom kalkulatoru nefokusirana je sve dok se na nju ne klikne mišem ili pozicionira putem tipkovnice. Fokusiranjem ćelije podeblja se njezin okvir. Kada je ćelija fokusirana, moguć je unos podataka, koji počinje unosom bilo kojeg znaka različitog od „Enter", „Tab" ili „Esc" i traje sve dok se ne unese neki od tih znakova ili dok se ne pomakne fokus na neku drugu ćeliju klikom miša ili pozicioniranjem putem tipkovnice. Svaki novi uneseni znak dodaje se na kraj niza novog unosa. Pri unosu podataka u ćeliju, na zaslonu se prikazuju novo uneseni podaci, ali se zapamćeni sadržaj ćelije ne mijenja. Sadržaj ćelije osvježava se tek nakon što se unesu znakovi „Enter" ili „Tab" ili se fokus pomakne na neku drugu ćeliju klikom miša ili putem tipkovnice. Ako se unese znak „Esc" sadržaj ćelije ostaje nepromijenjen tj. novi se unos odbacuje, ali ona i dalje ostaje fokusirana. ■

■ **Zadatak 13.2 — Oglas na portalu za oglašavanje.** Modelirajte stanja oglasa na portalu za oglašavanje iz zadatka 10.6. i pri tome uzmite u obzir sljedeće napomene.

Oglas je nakon objave aktivan. Oglašivač ga može po želji deaktivirati i reaktivirati, a može ga i obrisati. Dok je oglas aktivan, broje se njegovi pregledi. Kada se oglas deaktivira, više se ne prikazuje na portalu i vidljiv je samo oglašivaču koji ga je stvorio i koji ga može reaktivirati. Pri reaktivaciji oglasa, broj pregleda resetira se na nulu. Kada se oglas stvori ili reaktivira, prva 24 sata kraj njega stoji status „Novo!". Ako je oglas u kategoriji koja se plaća, a oglašivaču je istekla članarina, on će se automatski deaktivirati. Oglašivač može svoj oglas obrisati te se on tada trajno uklanja iz sustava. ■

■ **Zadatak 13.3 — Izvođenje poslova na procesoru.** Dijagramom stanja modelirajte izvođenje poslova na dvojezgrenom procesoru. Zasebno prikažite: a) stanje posla na procesoru i b) stanje procesora.

Pretpostavite da svaki posao ima zabilježenu količinu priručne memorije koju je zauzeo i približnu količinu vremena potrebnog za izvođenje. Posao se na početku nalazi u priručnoj memoriji u stanju čekanja i ostaje u tom stanju sve dok se jedna procesorska jezgra ne oslobodi. Kada se jedna jezgra na procesoru oslobodi, posao prelazi u stanje razmatranja. U tom stanju,

onaj posao koji zauzima najviše priručne memorije odabire se za izvršavanje. Svi se ostali poslovi vraćaju u stanje čekanja. Svaki posao pri izvršavanju na procesoru mora proći kroz tri stanja: 1) predobradu, u kojoj se optimira, 2) izvođenje i 3) zapis rezultata (rezultati se zapisuju u glavnu memoriju te se postavlja signalna zastavica da je jezgra prazna).

Pretpostavite da jedna jezgra izvodi samo jedan posao u nekom trenutku te da se proces razmatranja, tj. odabira posla za izvršavanje, izvodi na jezgri koja se oslobodila. Također pretpostavite da se, ako dođe do gašenja procesora, svi poslovi koji su trenutno u izvršavanju prekidaju i brišu. ■

■ **Zadatak 13.4 — Korisničko sučelje e-trgovine.** Modelirajte stanja korisničkog sučelja pri stvaranju narudžbe u e-trgovini iz zadatka 10.5. za registriranog i prijavljenog korisnika.

Nakon prijave u sustav korisniku se prikazuje katalog, a košarica je prazna. Korisnik započinje kupnju dodavanjem proizvoda iz kataloga u košaricu. U svakom trenutku korisnik može pregledati sadržaj košarice te ubačene proizvode izbaciti ili promijeniti njihovu količinu.

Pri pregledu košarice korisnik može odabrati stvaranje narudžbe ako je gotov s ubacivanjem svih proizvoda ili može odabrati nastavak kupnje čime se vraća na katalog i može dodavati još proizvoda. Ako odabere stvaranje narudžbe, prikazuje mu se forma za unos podataka za plaćanje. Nakon potvrde unosa podataka za plaćanje omogućuje se unos podataka za dostavu, s time da se prvo dohvaćaju podaci o korisnikovoj adresi iz sustava, a zatim se ostavlja mogućnost da ih on ispravi po želji. Kada korisnik potvrdi podatke za dostavu, narudžba je stvorena te se korisniku ispisuje obavijest o uspješnoj kupovini potvrda, košarica se prazni i on se vraća na pregled kataloga.

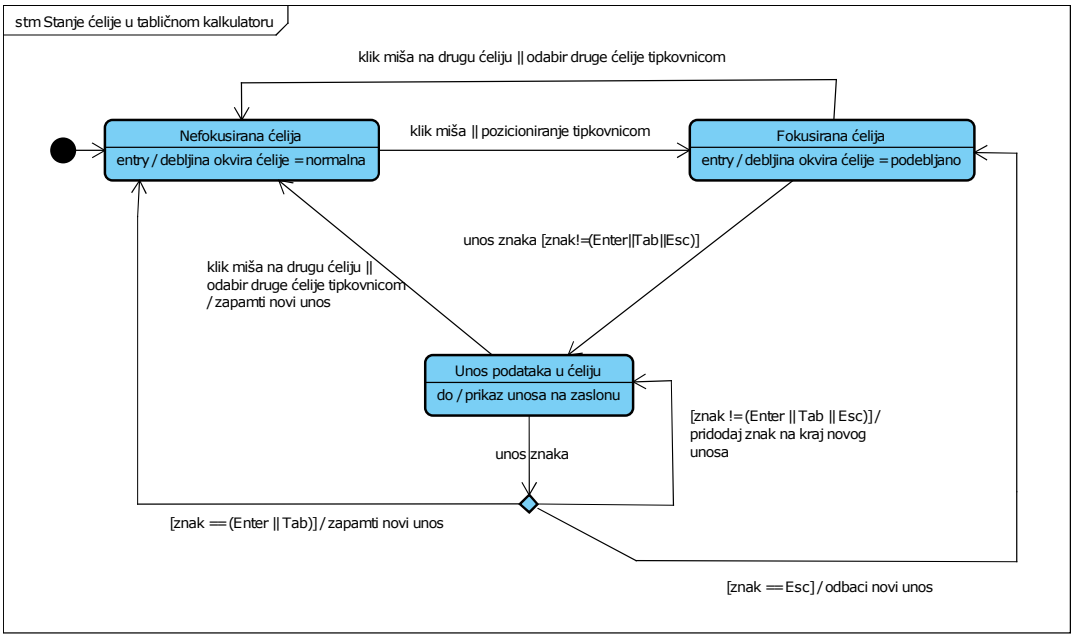
U svakom trenutku prije potvrde podataka za dostavu, korisnik može odustati od narudžbe, čime se vraća na katalog, a svi proizvodi i dalje ostaju u košarici. Korisnik može odabrati i opciju pražnjenja košarice, čime se iz košarice uklanjaju svi proizvodi. Ako napusti e-trgovinu (zatvori preglednik ili se odjavi) u bilo kojem trenutku prije potvrde podataka za dostavu, podaci o artiklima dodanim u košaricu pamte se te se pri ponovnoj prijavi u aplikaciju korisniku prikazuje stranica kataloga s očuvanim sadržajem košarice. ■

■ **Zadatak 13.5 — Pametna Kuća – stanja kućanskog aparata.** Modelirajte stanja kućanskog aparata u mobilnoj aplikaciji „Pametna Kuća” iz zadatka 10.7. Osim osnovnog teksta, uzmite u obzir i sljedeći opis.

Svaki kućanski aparat nudi mogućnost pokretanja jedne funkcije ili više njih. Nakon dodavanja aparata u aplikaciju on se smatra ispravnim, tj. ima status „ISPRAVAN”. Pri pokretanju funkcije aparata korisnik postavlja vrijeme trajanja izvođenja funkcije. Funkciju je moguće pokrenuti samo ako je njezin status „DOSTUPNA”. Dok aparat izvršava neku od funkcija, prikazuje se vrijeme do završetka trenutno aktivne funkcije. Za to vrijeme korisnik može zadati i pokretanje novih funkcija, koje će započeti nakon što trenutno aktivna funkcija završi. Nakon što završi sve zadane funkcije, aparat se vraća u stanje mirovanja i prikazuje se naziv zadnje izvršene funkcije. Ako dođe do zatajenja aparata, korisnika se obavještava o zatajenju te se status aparata mijenja u „KVAR” i ostaje takav sve do popravka. Dok je aparat u kvaru, nije moguće pokretati nove funkcije. Nakon popravka aparat nastavlja s radom ondje gdje je stao prije zatajenja. Brisanjem aparata iz aplikacije uklanjaju se svi podaci o aparatu i njegovim funkcijama. ■

13.2 Rješenja

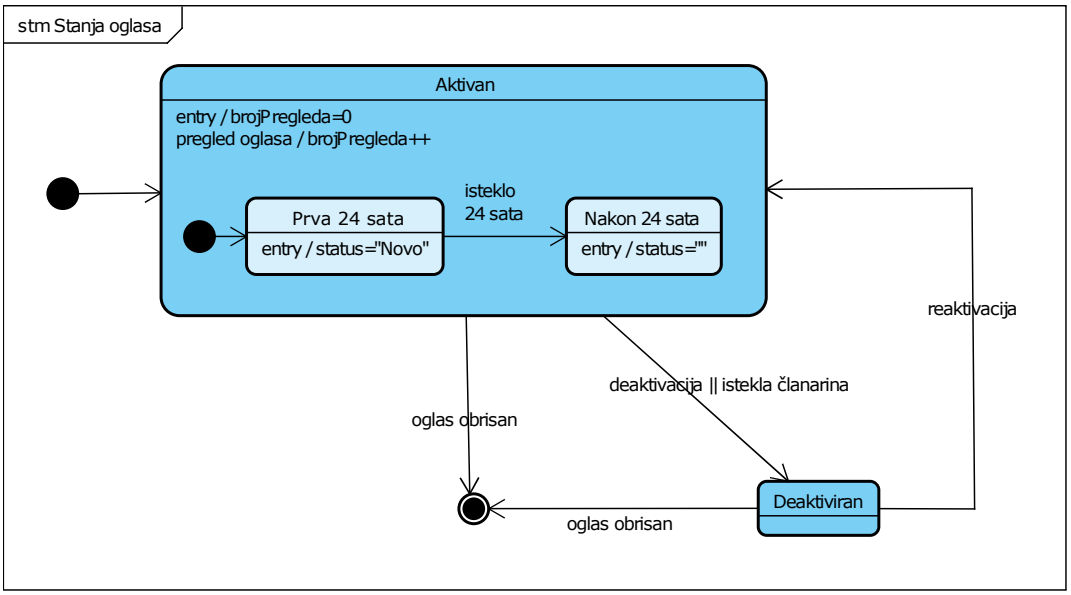
- **Zadatak 13.1 Tablični kalkulator**



Slika 13.1: Dijagram stanja ćelije u tabličnom kalkulatoru

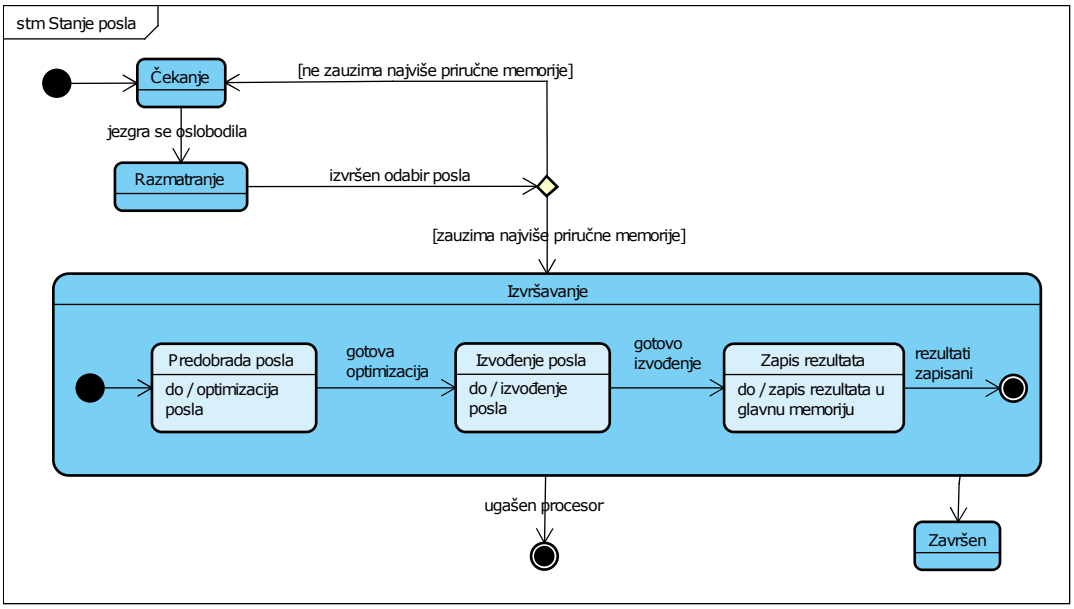
Komentar: Na prijelazima koji su vezani za unos znaka potrebno je razlikovati događaj (unos znaka) od provjere uvjeta koji je znak unesen (npr. je li unesen znak „Esc”). To je zato što svi moderni radni okviri koji služe za razvoj korisničkog sučelja imaju metode za osluškivanje (engl. *listener*) unosa znaka (engl. *key*) na tipkovnici (bilo kojeg znaka), što predstavlja događaj, a zatim razvojni programer piše kôd u kojem se provjerava koji je znak unesen.

- **Zadatak 13.2 Oglas na portalu za oglašavanje**

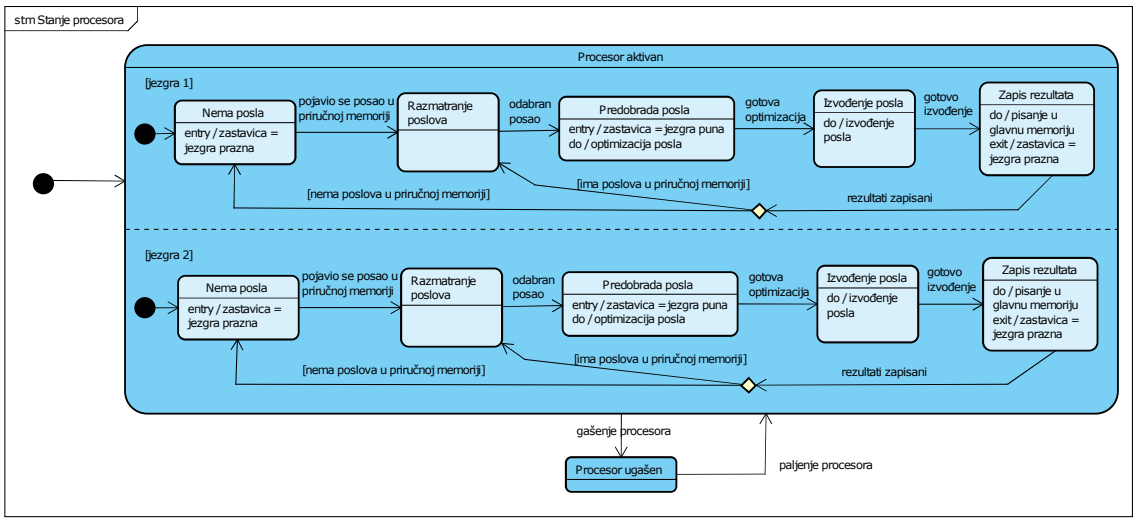


Slika 13.2: Dijagram stanja oglasa na portalu za oglašavanje

- **Zadatak 13.3 Izvođenje poslova na procesoru**



Slika 13.3: Dijagram stanja izvođenja posla

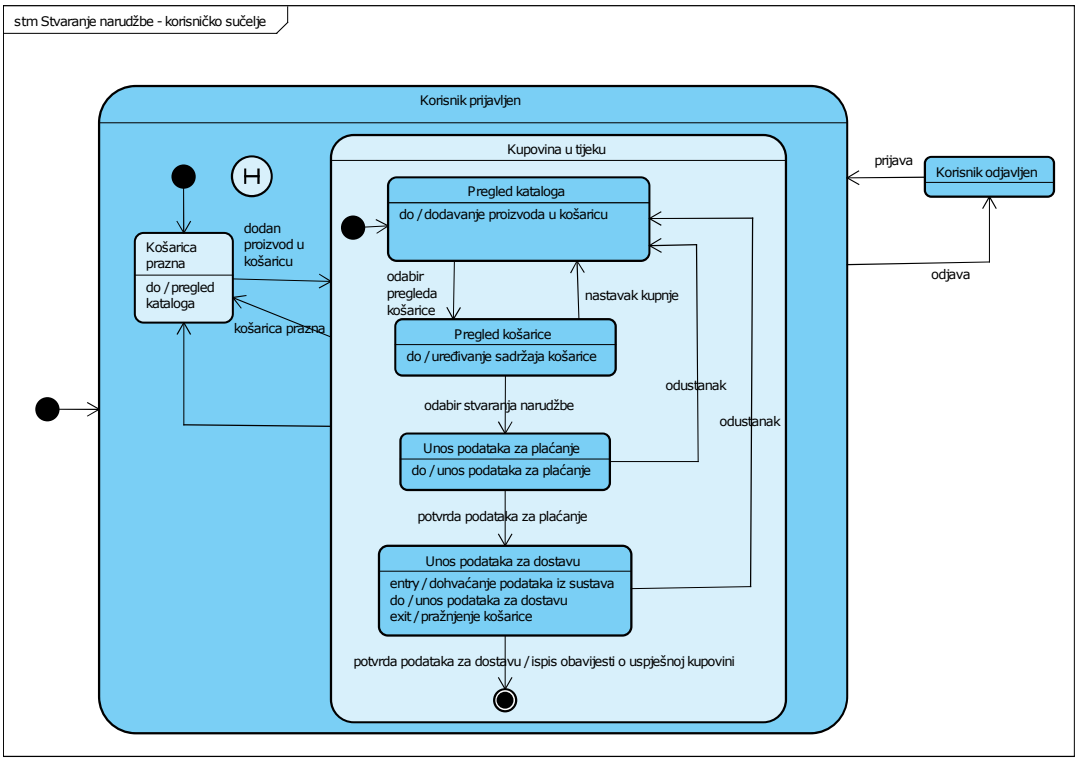


Slika 13.4: Dijagram stanja procesora

Komentar: Na dijagramu stanja posla potrebno je razlikovati stanje u kojem je posao uspješno završen i u kojem je izvršavanje prekinuto te se prema naputku u zadatku posao briše čime nestaje instanca tog posla (čvor završnog stanja). Na prijelazu između stanja „Izvršavanje" i „Završen" nije naveden događaj (okidač) zato što je „Izvršavanje" složeno stanje unutar kojeg je eksplicitno naznačen završetak složenog stanja (nakon što su rezultati zapisani) te se podrazumijeva da se taj prijelaz automatski aktivira kada složeno stanje završi.

Na dijagramu stanja procesora postoje dvije jezgre koje u paraleli prolaze kroz isti skup stanja, što je prikazano dvjema horizontalnim regijama. Na tom dijagramu gašenje procesora ne vodi u završno stanje zato što instanca procesora gašenjem ne nestaje te paljenjem ponovno počinje ciklus obrade poslova.

- **Zadatak 13.4 Korisničko sučelje e-trgovine**

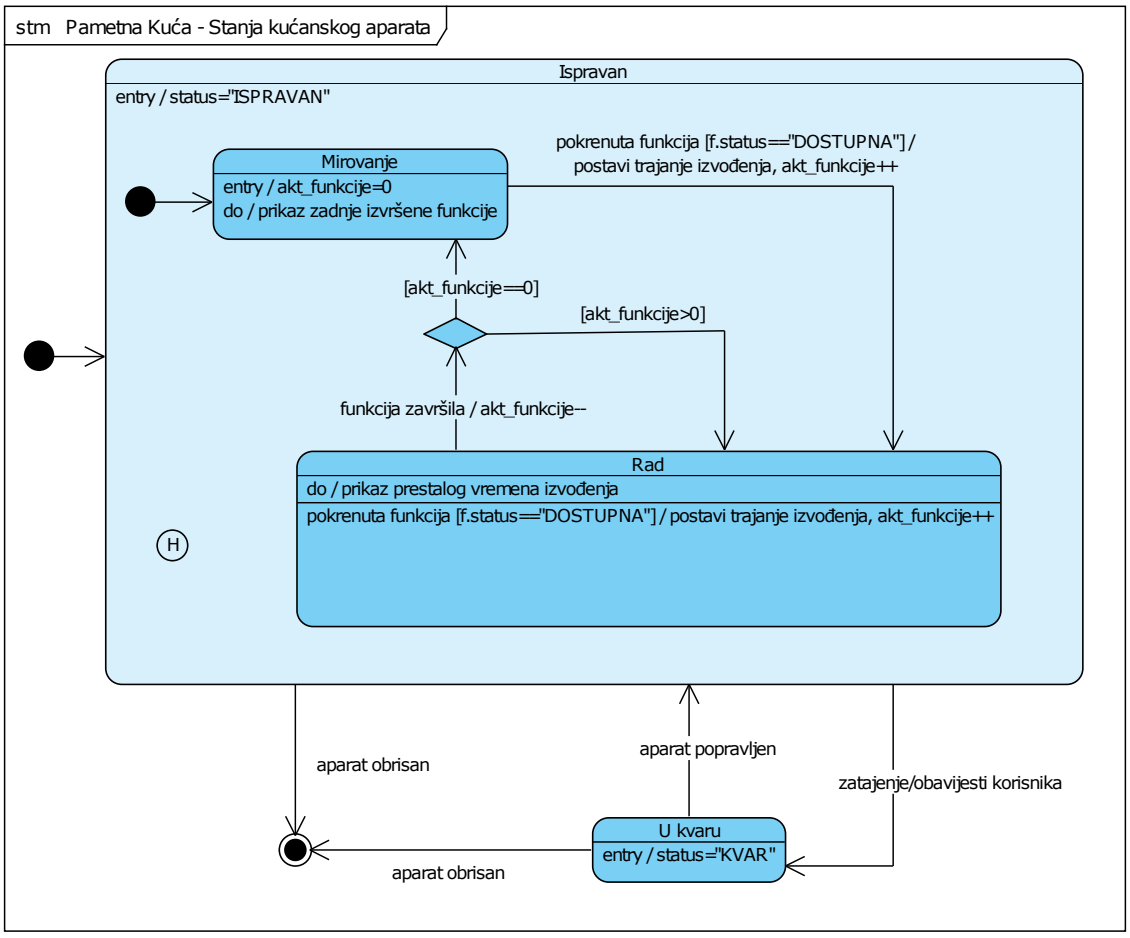


Slika 13.5: Dijagram stanja korisničkog sučelja e-trgovine

Komentar: Budući da postoji pamćenje stanja košarice za korisnike koji su prijavljeni, u stanje „Korisnik prijavljen" uvode se dva podstanja vezana za stanje košarice: „Košarica prazna" i „Kupovina u tijeku" te pseudostanje plitke povijesti (engl. *shallow history*). Kada ne bi postojalo podstanje „Kupovina u tijeku", nego bi stanja sadržana u njemu bila na istoj razini kao i pseudostanje plitke povijesti, onda bi se nakon ponovne prijave korisnik vraćao na zadnji korak u kojem je bio prije nego što se odjavio (npr. unos podataka za dostavu), a u zadatku je eksplicitno zadano da se vraća na pregled kataloga.

Na prijelazu između stanja „Kupovina u tijeku" i „Košarica prazna" nije naveden događaj (okidač) zato što je „Kupovina u tijeku" složeno stanje unutar kojeg je eksplicitno naznačen završetak složenog stanja (potvrda podataka za dostavu) te se podrazumijeva da se taj prijelaz automatski aktivira kada složeno stanje završi.

- **Zadatak 13.5 Pametna Kuća – stanja kućanskog aparata**



Slika 13.6: Dijagram stanja kućanskog aparata u mobilnoj aplikaciji „Pametna Kuća”

Komentar: Važno je još jednom napomenuti da se na ovom dijagramu modeliraju stanja kućanskog aparata unutar mobilne aplikacije, tj. stanja objekta koji predstavlja stvarni kućanski aparat, a ne stanja stvarnog fizičkog aparata.

14. UML dijagrami aktivnosti

14.1 Zadaci

■ **Zadatak 14.1 — Prodaja autobusnih karata.** Modelirajte dijagramom aktivnosti postupak prodaje autobusnih karata iz zadatka 10.2. Osim osnovnog opisa uzmite u obzir i sljedeće.

Blagajnik u aplikaciju najprije unosi informacije o putovanju (npr. podatke o relaciji i broju putnika) za koje će prodati kartu. Aplikacija zatim ispisuje cijenu, a blagajnik o tome obavještava putnika, koji odabire način plaćanja. Potrebno je modelirati oba načina plaćanja: gotovinom i karticom. Pri gotovinskom plaćanju putnik predaje novac blagajniku, a on u aplikaciji potvrđuje da je karta plaćena te zatim aplikacija pohranjuje podatke i ispisuje kartu.

Pri kartičnom plaćanju, blagajnik u aplikaciji pokreće naredbu naplate karticom, a putnik zatim provlači karticu i unosi PIN. Za provedbu naplate aplikacija se spaja na bankovni poslužitelj. Nakon uspješnog završetka transakcije aplikacija pohranjuje podatke te se ispisuje karta. Radi jednostavnosti, pretpostavite da putnik uvijek unosi točan PIN i da transakcija uvijek uspijeva. Također, zanemarite mogućnost da putnik odustaje od kupnje karte zbog neprihvatljive cijene. ■

■ **Zadatak 14.2 — Portal za oglašavanje – objava novog oglasa.** Modelirajte aktivnost objave novog oglasa iz zadatka 10.6. Osim osnovnog opisa uzmite u obzir i sljedeće informacije.

Pretpostavite da je korisnik već prijavljen i da se aktivnost pokreće odabirom opcije za objavu novog oglasa, nakon čega aplikacija prikazuje obrazac za unos oglasa. Korisnik popunjava podatke za oglas i predaje oglas. Sustav tada provjerava kategoriju oglasa. Ako se radi o plaćenoj kategoriji, sustav još provjerava i status članstva. Ako je korisnik već platio članarinu, oglas se pohranjuje. U suprotnom, prikazuje se obrazac za plaćanje. Korisnik unosi podatke o plaćanju te se sustav spaja na servis za plaćanje i pokušava izvršiti transakciju, a za to vrijeme korisniku na ekranu prikazuje da je transakcija u tijeku. Po završetku transakcije oglas se sprema, a potvrda o kupnji šalje se u PDF formatu na adresu e-pošte. Pretpostavite da je transakcija uvijek uspješna. ■

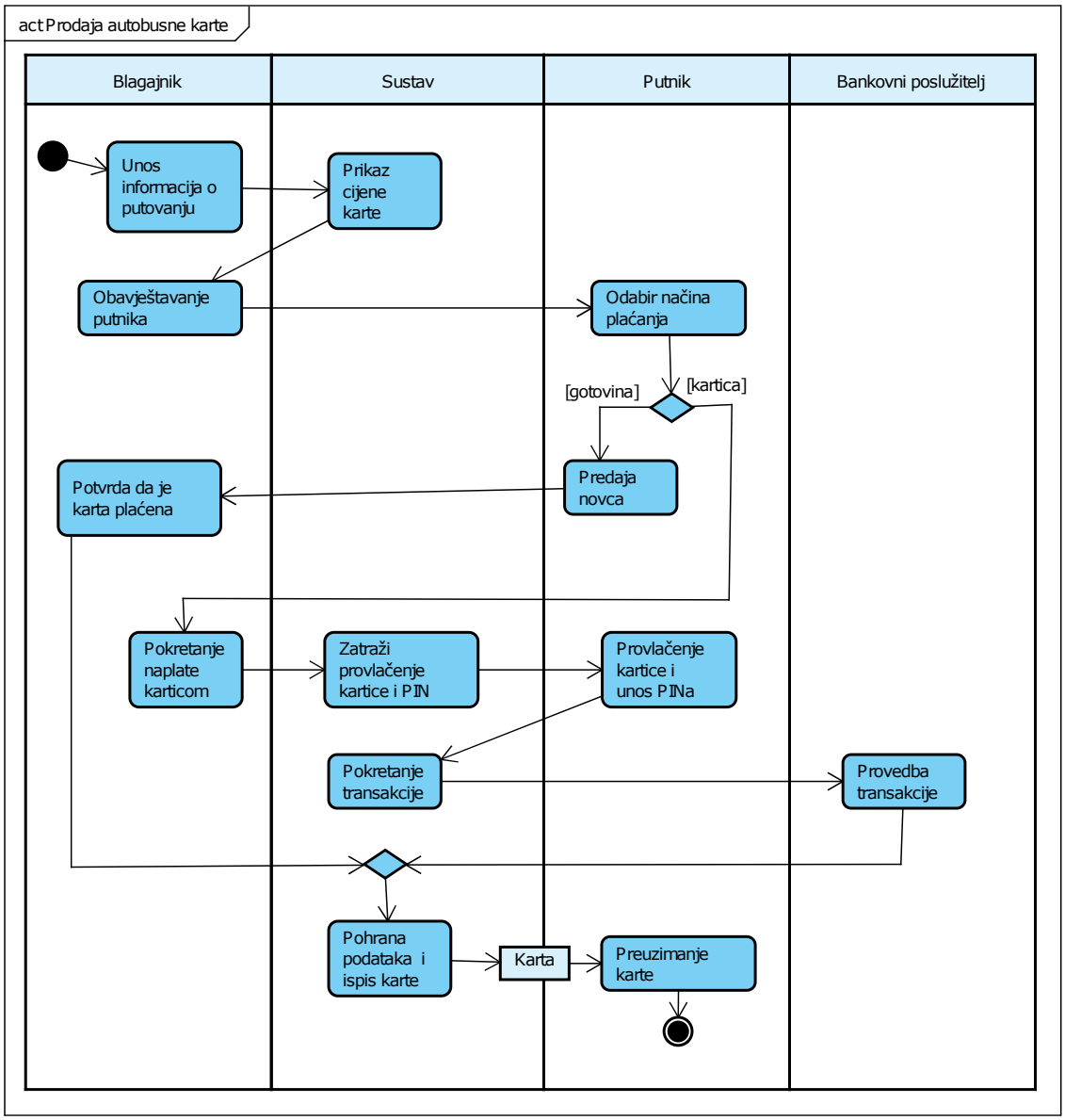
■ **Zadatak 14.3 — Pametna Kuća – pokretanje funkcije kućanskog aparata.** Modelirajte aktivnost pokretanja funkcije pametnog kućanskog aparata iz zadatka 10.7. Osim osnovnog teksta uzmite u obzir i sljedeće informacije.

Da bi pokrenuo neku funkciju na aparatu, korisnik najprije treba odabrati željeni kućanski aparat, a zatim željenu funkciju te unijeti trajanje njezina izvođenja. Nakon toga aplikacija se spaja

na kućanski aparat i provjerava status funkcije. Ako je status funkcije „DOSTUPNA”, aplikacija aparatu šalje naredbu za pokretanje funkcije te istodobno zapisuje podatke o tom događaju na poslužitelj BackendServer slanjem datoteke u JSON formatu. Nakon što je funkcija uspješno pokrenuta na aparatu i podaci zapisani na poslužitelju, aplikacija obavještava korisnika o uspješnom pokretanju. Ako je status funkcije „NEDOSTUPNA”, aplikacija obavještava korisnika da se funkcija ne može pokrenuti. ■

14.2 Rješenja

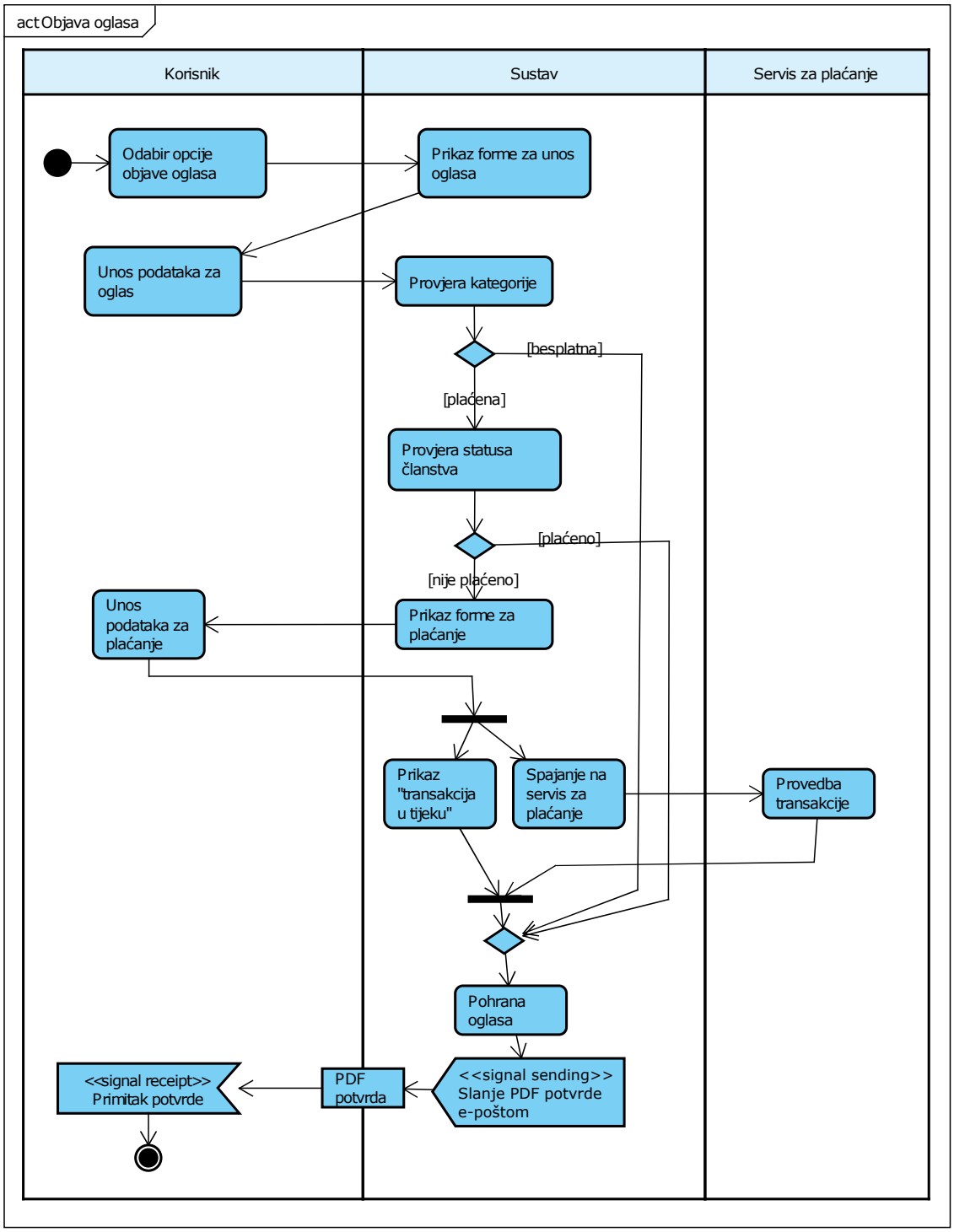
- **Zadatak 14.1 Prodaja autobusnih karata**



Slika 14.1: Dijagram aktivnosti prodaje autobusne karte

Komentar: Na dijagramu je modeliran objektni čvor Karta da bi se istaknulo da se konkretni objekt prosljeđuje putniku, ali je potpuno ispravno rješenje u kojem se taj čvor izostavlja.

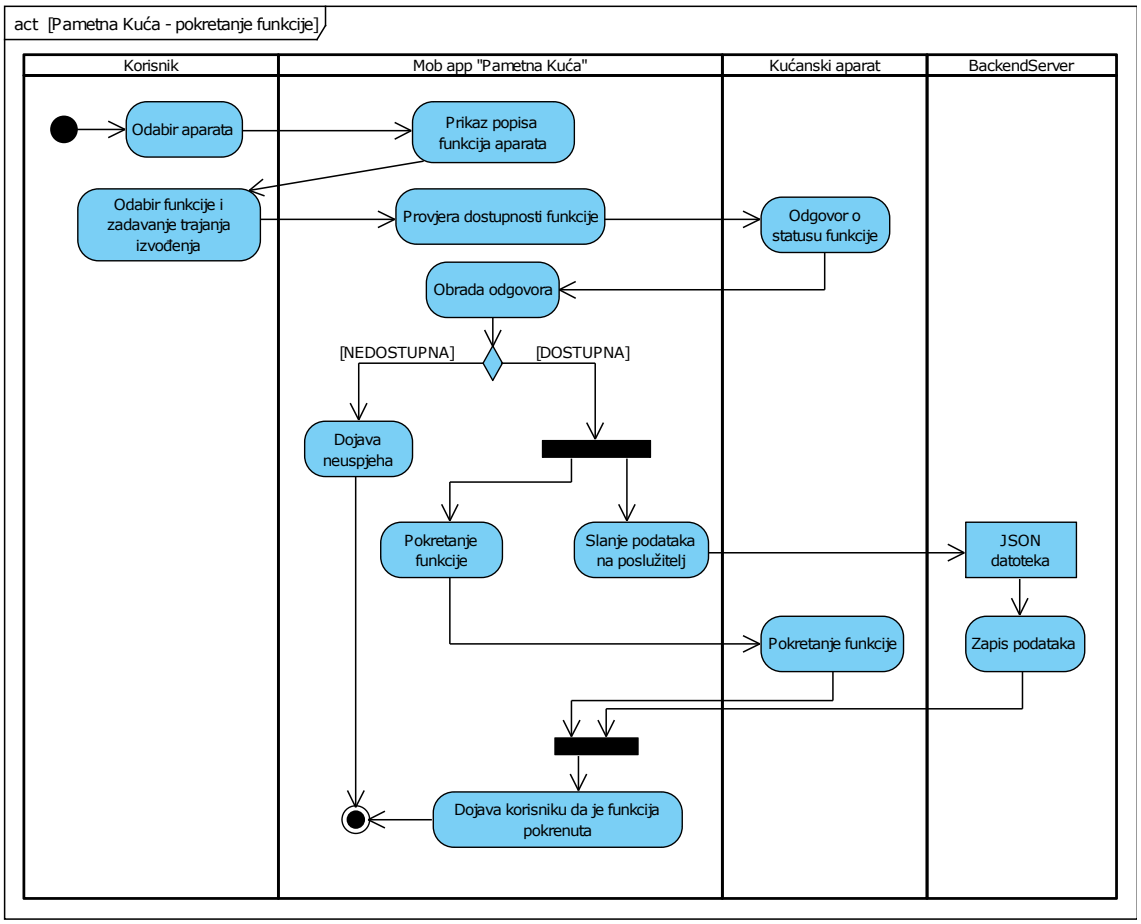
- **Zadatak 14.2 Portal za oglašavanje – objava novog oglasa**



Slika 14.2: Dijagram aktivnosti objave novog oglasa na portalu za oglašavanje

Komentar: Umjesto čvorova signala za slanje i primanje potvrde e-poštom moguće je koristiti obične čvorove akcije.

- **Zadatak 14.3 Pametna Kuća – pokretanje funkcije kućanskog aparata**



Slika 14.3: Dijagram aktivnosti pokretanja funkcije kućanskog aparata u mobilnoj aplikaciji „Pametna Kuća”

15. UML dijagrami komponenti

15.1 Zadaci

■ **Zadatak 15.1 — MVC.** Modelirajte arhitekturni obrazac MVC (engl. Model-View-Controller).

Komponenta Model sadržava sve podatke i poslovnu logiku, komponenta View je zadužena za prikaz podataka i interakciju s korisnikom, a komponenta Controller je posrednik između komponenti Model i View. Komponenta View poziva komponentu Controller putem sučelja IEvent. Komponenta Controller osvježava podatke korištenjem sučelja IUpdate komponente Model. Nakon svake promjene podataka komponenta Model obavještava komponentu View tako što poziva metode sučelja INotify koje ostvaruje komponenta View. Osvježene podatke iz Modela komponenta View dobiva putem sučelja IData. ■

■ **Zadatak 15.2 — Portal za oglašavanje.** Modelirajte *web*-aplikaciju portala za oglašavanje iz zadatka 10.6. uz pretpostavku sljedećeg opisa strukture aplikacije.

Web-aplikacija sastoji se od četiri komponente: Upravljači, Modeli, JSPPogledi i DTO. Upravljači pružaju REST_API sučelje na koje vanjski *web*-preglednik može slati zahtjeve i putem kojeg šalju JSPPoglede kao odgovor. Upravljači se također povezuju s vanjskom uslugom EPlaćanje putem sučelja EP_API. Upravljači koriste komponentu DTO za pristup SQL bazi podataka koja ostvaruje sučelje JDBC. DTO upisuje podatke iz baze podataka u komponentu Modeli, a Upravljači koriste Modele za manipulaciju podacima. ■

■ **Zadatak 15.3 — Pametna Kuća.** Modelirajte komponente mobilne aplikacije „Pametna Kuća" iz zadatka 10.7. uz pretpostavku sljedećeg opisa strukture aplikacije.

Mobilna aplikacija strukturirana je prema obrascu MVVM (engl. *Model-View-ViewModel*). Komponenta View odgovorna je za prikaz korisničkog sučelja i sadržava komponente Activity i Fragment. Ona od komponente ViewModel dohvaća sve podatke potrebne za prikaz.

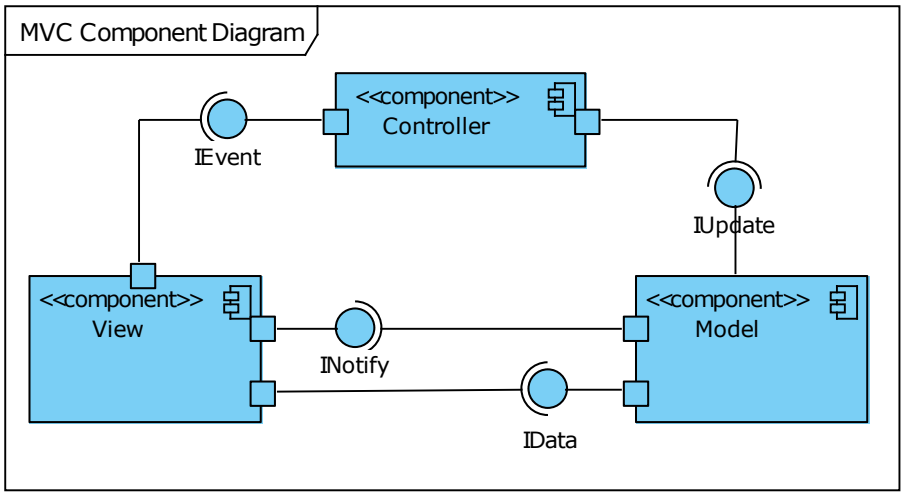
Komponenta ViewModel odgovorna je za komunikaciju s pametnim uređajem, dohvaćanje podataka za povezivanje s uređajem pri prvom povezivanju od registracijske usluge SmartDeviceRegService te za upravljanje svim ostalim podacima o korisniku, uređajima itd. Za komunikaciju s pametnim uređajem, komponenta ViewModel koristi sučelje „CTRL API", koje implementira svaki pametni uređaj. Podatke od usluge SmartDeviceRegService dohvaća preko sučelja „REG API". Nadalje, komponenta ViewModel upravlja svim ostalim podacima u aplikaciji korištenjem

komponente Repository te na temelju podataka koje od nje dobiva osvježava View.

Komponenta Repository dohvaća podatke spajanjem na sučelje „DATA API" poslužitelja BackendServer i pohranjuje te podatke u komponenti Model, koja sadržava komponente DAO i Entity. ■

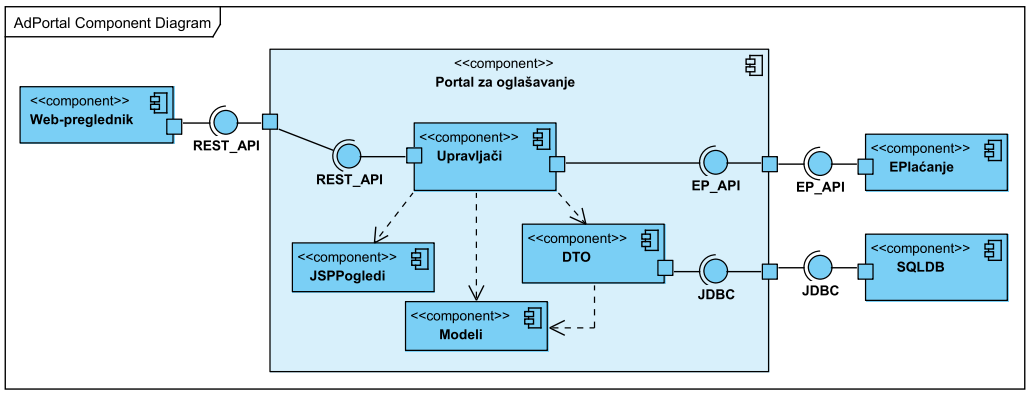
15.2 Rješenja

- **Zadatak 15.1 MVC**



Slika 15.1: Dijagram komponenti arhitekturnog obrasca MVC

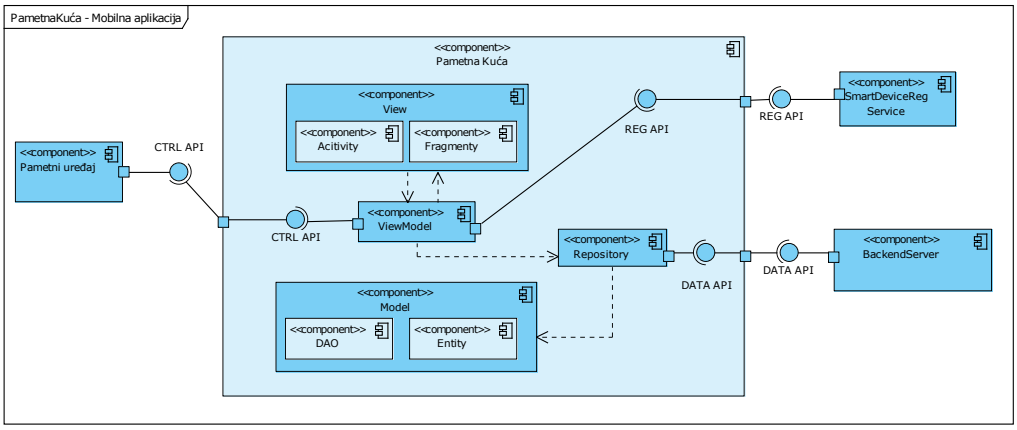
- **Zadatak 15.2 Portal za oglašavanje**



Slika 15.2: Dijagram komponenti portala za oglašavanje

Komentar: Gledano iz perspektive komponenti *Web*-preglednik, EPlaćanje i SQLDB, unutarnje komponente *web*-aplikacije i njihova sučelja skriveni su te sva interakcija s njima mora ići putem priključaka (engl. *ports*). Zato komponenta Portal za oglašavanje mora imati vanjska sučelja REST_API, EP_API i JDBC, koja se onda putem priključaka delegiraju na unutarnja sučelja komponenti. Za komponente za koje nije eksplicitno definirano sučelje, kao npr. kada Upravljači koriste komponentu DTO, komponente se povezuju vezom ovisnosti.

- **Zadatak 15.3 Pametna Kuća**



Slika 15.3: Dijagram komponenti mobilne aplikacije „Pametna Kuća”

Komentar: Mobilna aplikacija prikuplja podatke od vanjskih uređaja, poslužitelja i usluga, kao što su na primjer pametni uređaji ili usluga SmartDeviceRegService. Međutim, o dijelovima sustava koji su vanjski u odnosu na mobilnu aplikaciju nisu dane nikakve dodatne informacije, osim o sučeljima na koja se mobilna aplikacija može spojiti da bi dobila podatke. U skladu s time, vanjske uređaje, poslužitelje i usluge predstavljamo jednostavnim komponentama za koje jedino definiramo ostvarena sučelja koja koristi mobilna aplikacija.

16. UML dijagrami razmještaja

16.1 Zadaci

■ **Zadatak 16.1 — Klijent – poslužitelj.** Modelirajte sustav u kojem klijent komunicira s poslužiteljem.

Na klijentskoj je strani PC računalo s operacijskim sustavom Linux Debian na kojem je pokrenut *web*-preglednik Mozilla Firefox. Klijent se protokolom HTTP spaja na poslužitelj. Na poslužitelju je instaliran operacijski sustav Linux CentOS. Poslužiteljska aplikacija pod nazivom WebAPP pokrenuta je unutar poslužiteljskog programa Apache HTTP.

Nacrtajte jedan specifikacijski dijagram razmještaja (engl. *Specification-level Deployment Diagram*) i jedan dijagram razmještaja instanci (engl. *Instance-level Deployment Diagram*) s dvije instance klijenta i jednom instancom poslužitelja. Nazive instanci odaberite proizvoljno. ■

■ **Zadatak 16.2 — Trorazinska arhitektura.** Modelirajte specifikacijskim dijagramom razmještaja (engl. *Specification-level Deployment Diagram*) trorazinsku arhitekturu: klijent – poslužitelj – baza podataka.

Na klijentskoj strani koristi se PC računalo na kojem je pokrenuta aplikacija ClientAPP. Poslužiteljska aplikacija ServerAPP nalazi se na računalu s operacijskim sustavom RHEL. Baza podataka implementirana je kao usluga u oblaku Heroku Postgres. Klijent komunicira s poslužiteljem protokolom SSH, a poslužitelj komunicira s bazom podataka protokolom HTTPS. ■

■ **Zadatak 16.3 — Vremenska prognoza.** Modelirajte specifikacijskim dijagramom razmještaja (engl. *Specification-level Deployment Diagram*) informacijski sustav za davanje vremenske prognoze.

Klijent se putem mobilne aplikacije MeteoAPP.apk za Android OS spaja na *web*-poslužitelj MeteoServer. Poslužiteljska *web*-aplikacija MeteoWeb.war pokrenuta je pomoću poslužiteljskog programa Apache Tomcat na računalu na kojem je instaliran operacijski sustav WinServer 2019. *Web*-poslužitelj dobavlja podatke od vanjskog servisa EUMeteo. Ako se servis EUMeteo nedostupan, *web*-poslužitelj dobavlja informacija od servisa USMeteo. Za komunikaciju između klijenta i poslužitelja te poslužitelja i servisa koristi se protokol HTTP. ■

■ **Zadatak 16.4 — E-trgovina.** Modelirajte sustav e-trgovine iz zadatka 10.5. uz sljedeće dodatne informacije.

Web-aplikacija sastoji se od dva dijela: korisničkog sučelja (engl. *front-end*) i poslužiteljske aplikacije u pozadini (engl. *back-end*). Korisničko je sučelje implementirano u radnom okviru Angular i pokrenuto na platformi Heroku. Poslužiteljska je aplikacija implementirana korištenjem Node.js na platformi Firebase, a pokrenuta je unutar spremnika (engl. *container*) Docker. Baza podataka, u koju poslužiteljska aplikacija sprema sve podatke, također se nalazi na platformi Firebase i izvodi se korištenjem usluge Firestore. Baza podataka sadržava sheme Korisnici, Proizvodi i Narudžbe. Klijent se putem *web*-preglednika spaja na korisničko sučelje (*front-end*), putem kojeg komunicira s poslužiteljskom aplikacijom (*back-end*). Za komunikaciju se koristi protokol HTTPS.

Nacrtajte specifikacijski dijagram razmještaja (engl. *Specification-level Deployment Diagram*). Nacrtajte i jedan dijagram razmještaja instanci (engl. *Instance-level Deployment Diagram*) na kojem ćete prikazati pristup jednog klijenta s PC računala, jednog klijenta s mobilnog uređaja i dvije instance spremnika Docker koje se koriste radi ujednačavanja opterećenja. ■

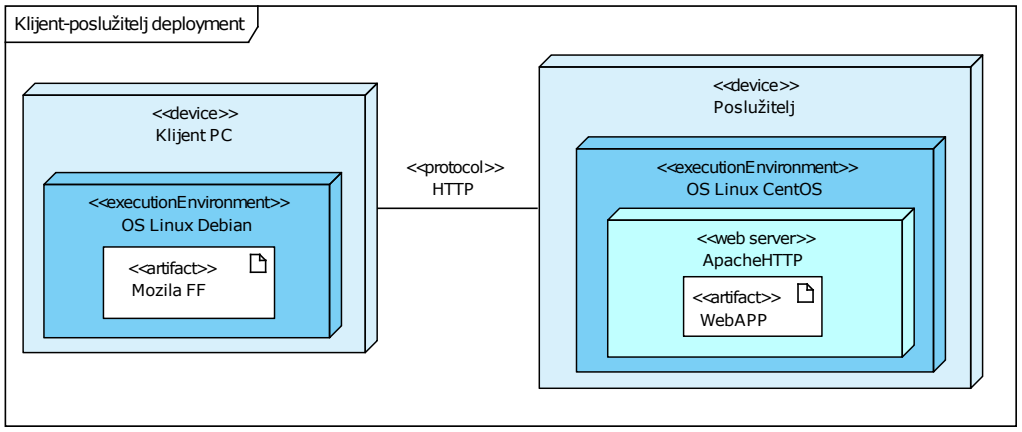
■ **Zadatak 16.5 — Pametna Kuća.** Modelirajte cijeli sustav vezan za rad mobilne aplikacije „Pametna Kuća" iz zadatka 10.7.

Mobilna aplikacija PametnaKucaAPP.apk podržana je isključivo na operacijskom sustavu Android v9 ili novijem. Osim aplikacije, na pametnom su telefonu pohranjene i postavke za spajanje na pametne uređaje u datoteci Settings.cfg.

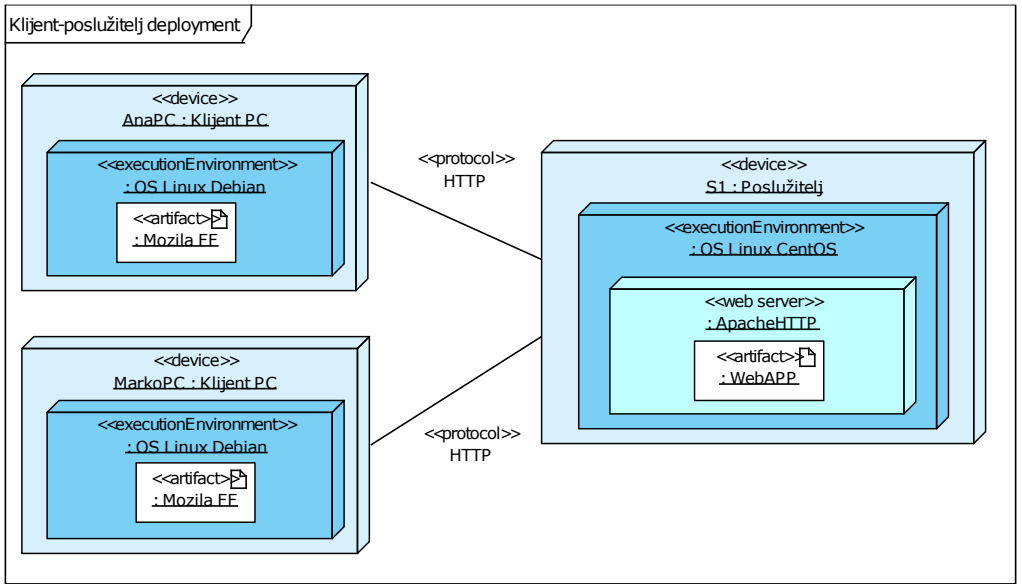
Poslužitelj BackendServer pokrenut je u oblaku Azure korištenjem dvaju Docker spremnika: na jednom je spremniku pokrenuta .NET poslužiteljska aplikacija Server.exe, a na drugom baza podataka MongoDB. Mobilna aplikacija komunicira s poslužiteljem Backend protokolom HTTPS, a s pametnim uređajima protokolom TLS. Podatke za prvo spajanje s uređajem mobilna aplikacija dobiva od usluge SmartDeviceRegService na koju se također spaja protokolom HTTPS. ■

16.2 Rješenja

- **Zadatak 16.1 Klijent – poslužitelj**



Slika 16.1: Specifikacijski dijagram razmještaja arhitekture klijent – poslužitelj

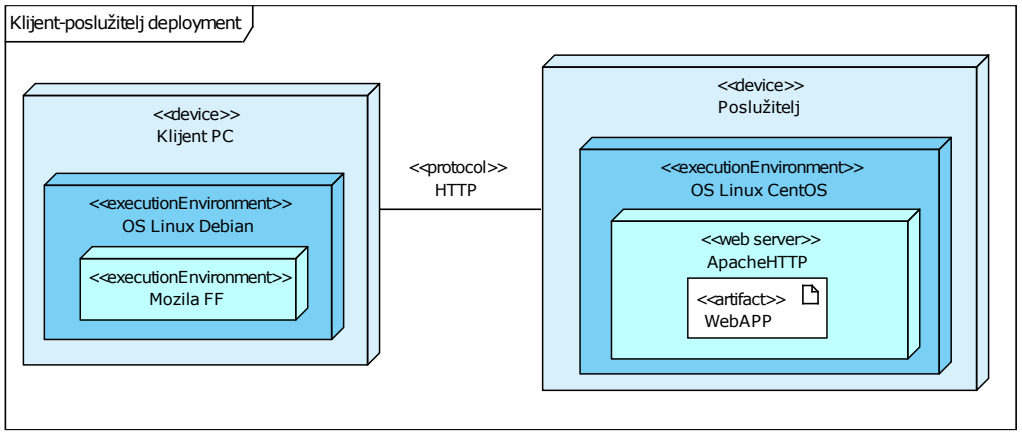


Slika 16.2: Dijagram razmještaja instanci arhitekture klijent – poslužitelj

Komentar: Na dijagramima razmještaja nužno je koristiti stereotipe za opis svih čvorova i komponenti. Okolina izvođenja (operacijski sustav, poslužiteljski programi kao npr. Tomcat, Apache PHP i sl.) uvijek se prikazuje kao čvor. Moguće je koristiti generički stereotip «execution environment» ili konkretnije specificirati npr. «server», «service» itd. Generički stereotip za sve komponente (aplikacije koje se izvode na čvorovima) je «artifact», ali moguće je koristiti i neke specifičnije kao npr. «document», «executable», «file», «library», «script», «source» itd.

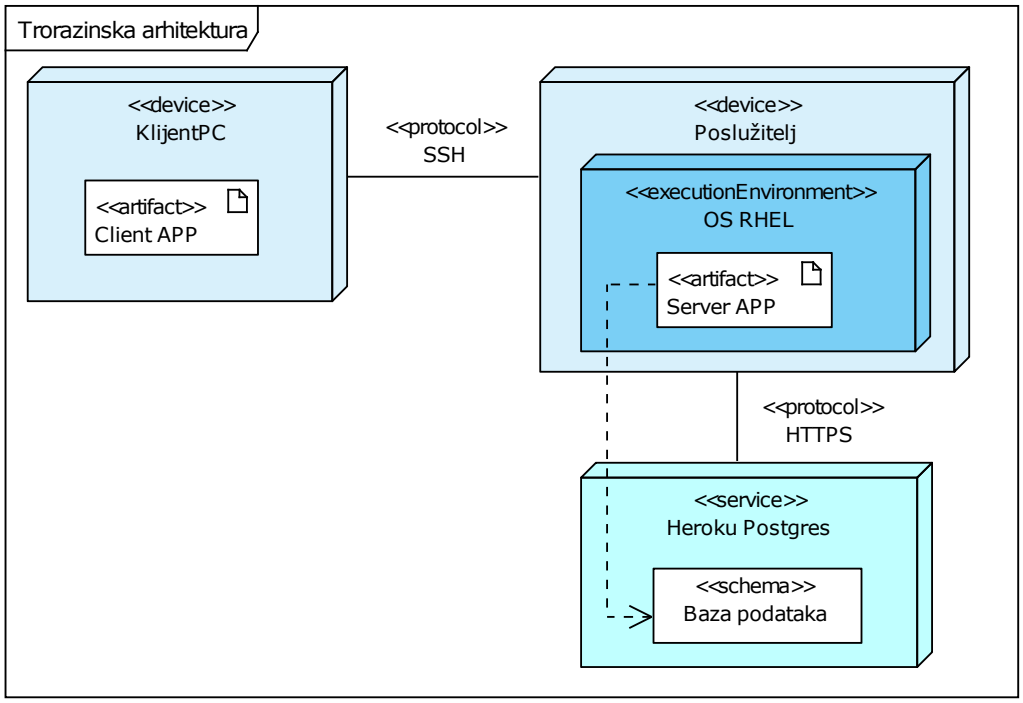
Na dijagramu instanci treba uočiti sintaksu „NazivInstance: Tip”. Moguće je imati više instanci istog tipa, npr. „MarkoPC” i „AnaPC” primjeri su dviju instanci istog tipa čvora „Klijent PC”. Međutim, nije nužno uvijek pisati naziv instance. Ako naziv nije poznat (ili nije važan), sintaksa je oblika „:Tip”: npr. „:LinuxDebian” označava da je okolina izvođenja tipa „Linux Debian”, ali nema neki poseban naziv.

Konačno, *web*-preglednik moguće je modelirati kao artefakt, ali i kao okolinu izvođenja, kao što je prikazano na slici 16.3. To je zato što su moderni *web*-preglednici, za razliku od nekadašnjih, sposobni sami izvesti programski kôd u nekim jezicima, kao npr. JavaScript, i iz te ih je perspektive ispravno smatrati izvršnom okolinom. U praksi se mogu pronaći oba pristupa, ovisno o tome što se želi istaknuti. Na primjer, ako *web*-preglednik samo prikazuje HTML stranice koje dobiva od poslužitelja, onda se može smatrati artefaktom, a ako sam izvodi dio *web*-aplikacije (napisane npr. u Reactu ili Angularu), onda ga je ispravnije smatrati okolinom izvođenja, a izvršni kôd (napisan u npr. JavaScriptu ili TypeScriptu) koji se dobiva od poslužitelja i izvodi u pregledniku može se modelirati kao artefakt na klijentskoj strani.



Slika 16.3: Alternativno rješenje za specifikacijski dijagram razmještaja iz zadatka 16.1

- **Zadatak 16.2 Trorazinska arhitektura**

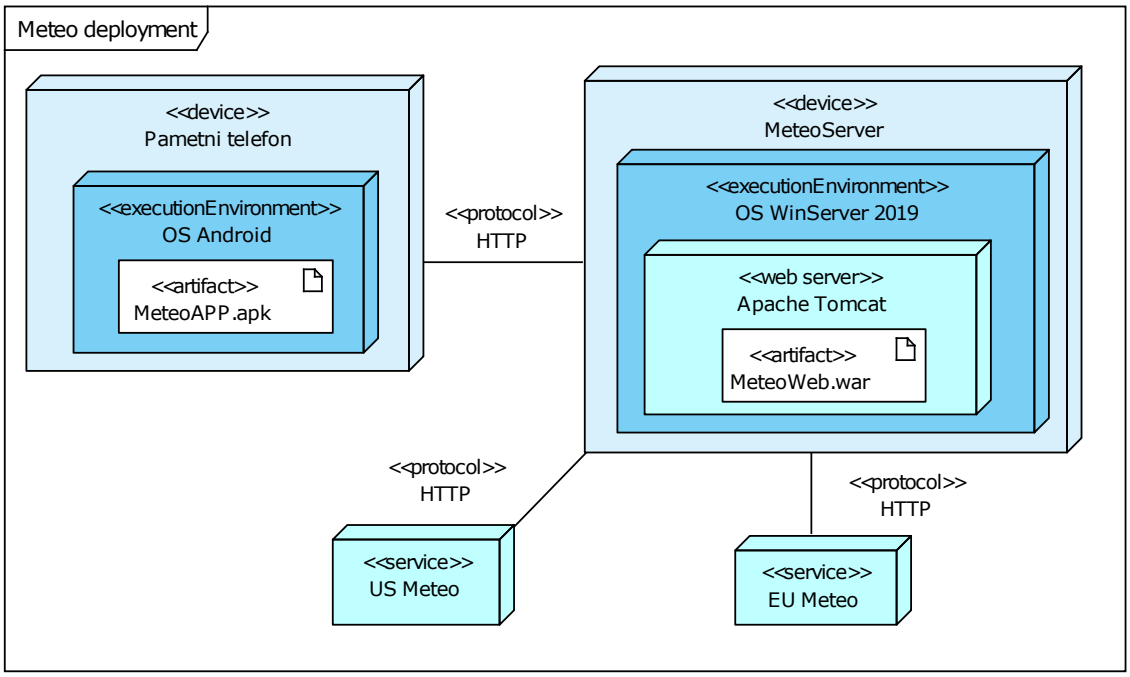


Slika 16.4: Specifikacijski dijagram razmještaja za trorazinsku arhitekturu

Komentar: Osim na fizičke čvorove («device»), dijelovi sustava mogu se pustiti u pogon korištenjem usluge oblaka, kao npr. Heroku Postgres za poslužitelj baze podataka. U tom se slučaju najčešće koristi stereotip «service» ili «cloud service» da bi se naznačila razlika u odnosu na fizičke čvorove. Također, kada je riječ o korištenju usluga u oblaku, vrlo često nisu dostupne informacije o detaljima okoline izvođenja kao što je operacijski sustav i sl. pa onda to nije niti potrebno modelirati.

Na ovom je dijagramu nadalje potrebno uočiti da, za razliku od prethodnog primjera, nije naznačen operacijski sustav na klijentskom računalu. Glavni je razlog to što takva informacija nije dana u tekstu zadatka, a može se tumačiti i da OS na klijentskoj strani nije bitan, tj. klijentska aplikacija u ovom slučaju radi neovisno o OS-u. Treba uočiti i ovisnost između artefakata ServerAPP i Baze podataka.

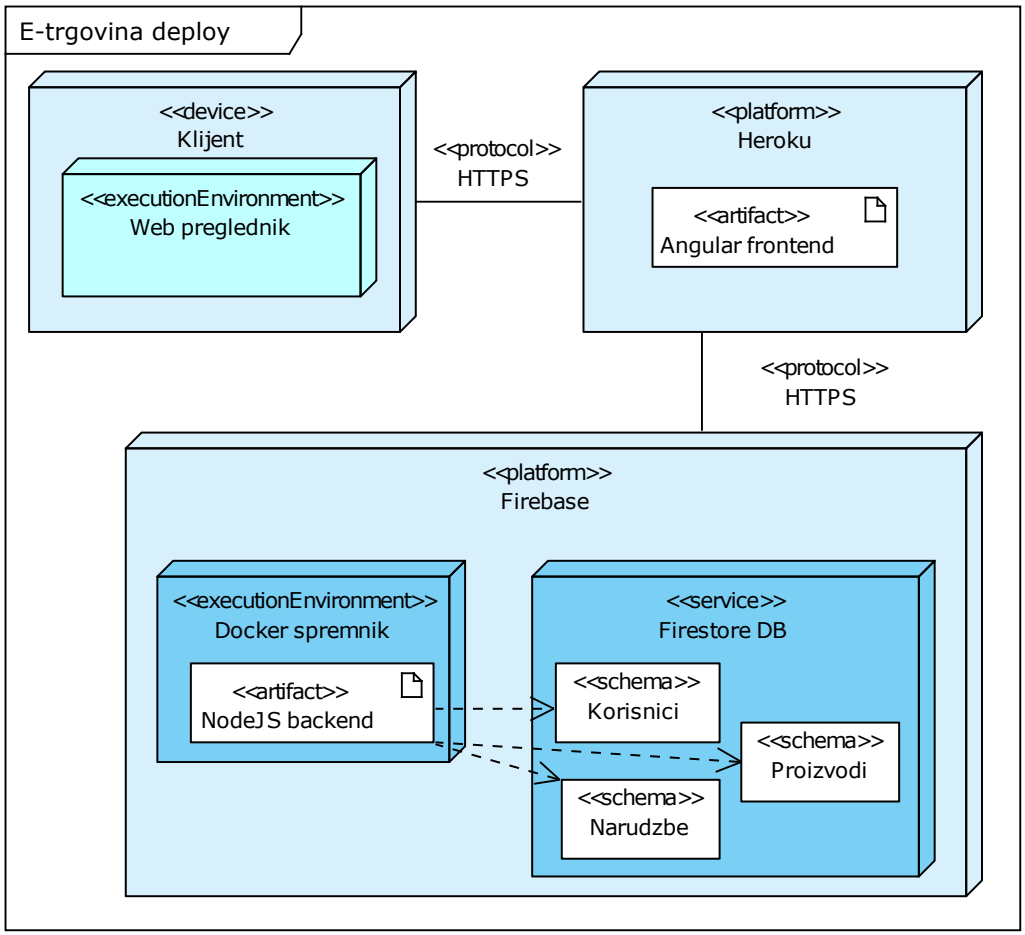
- **Zadatak 16.3 Vremenska prognoza**



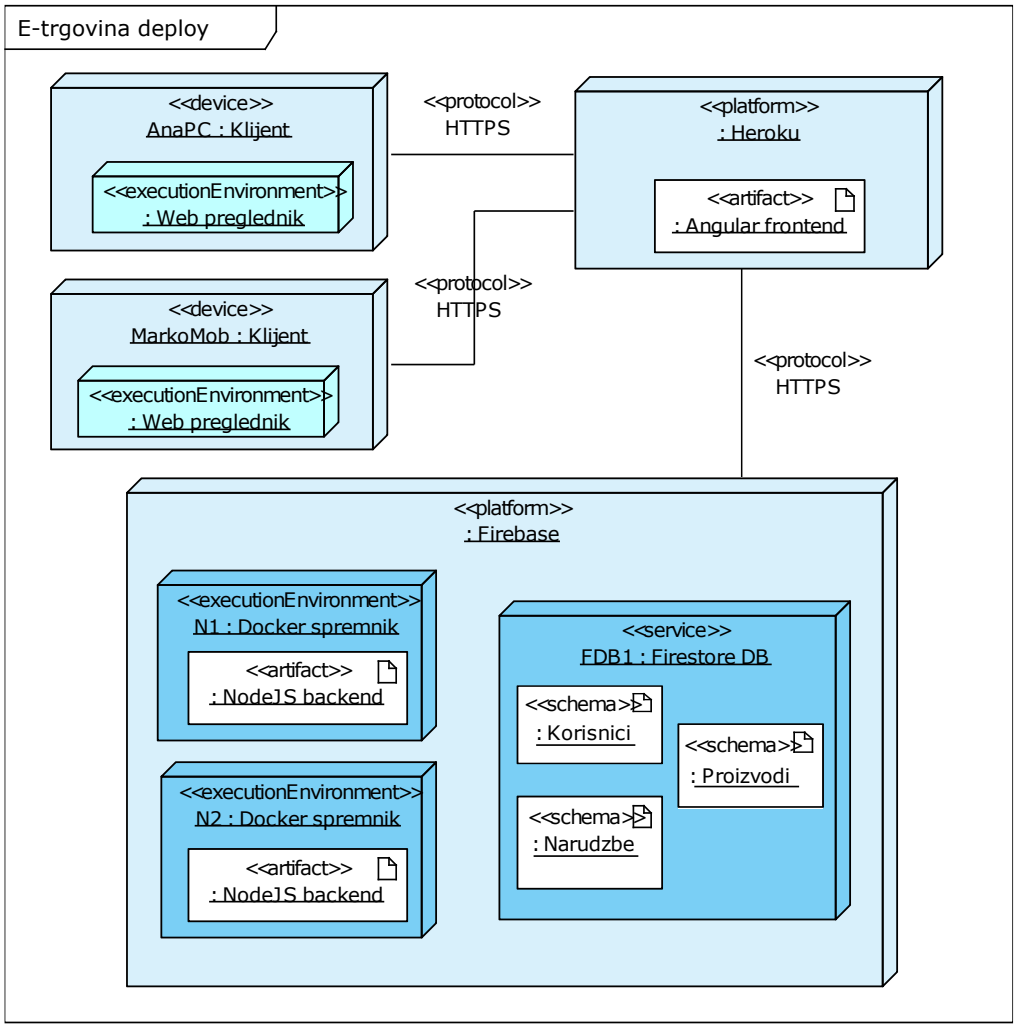
Slika 16.5: Specifikacijski dijagram razmještaja za informacijski sustav za davanje vremenske prognoze

Komentar: O vanjskim servisima na koje se spaja MeteoServer nemamo nikakvih dodatnih informacija pa ih prikazujemo kao jednostavne čvorove sa stereotipom «service».

- **Zadatak 16.4 E-trgovina**

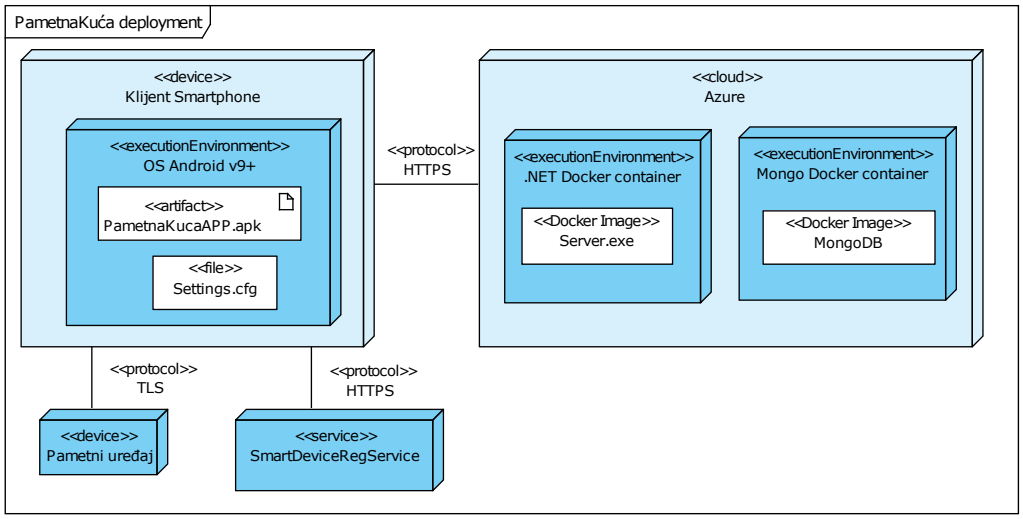


Slika 16.6: Specifikacijski dijagram razmještaja za E-trgovinu



Slika 16.7: Dijagram razmještaja instanci za E-trgovinu

- **Zadatak 16.5 Pametna Kuća**



Slika 16.8: Specifikacijski dijagram razmještaja za aplikaciju „Pametna Kuća”

Komentar: Za konfiguracijsku datoteku Settings.cfg korišten je stereotip «file» da bi se pobliže označilo da se radi o datoteci, ali ispravno je koristiti i generički stereotip «artifact». Za poslužiteljsku aplikaciju i bazu podataka Mongo korišteni su stereotipi «Docker Image» jer se standardno sve aplikacije na Docker spremnicima pokreću u obliku unaprijed pripremljenih slika (engl. *image*) koje objedinjavaju datoteke s izvršnim kodom i sve potrebne konfiguracijske datoteke, knjižnice i sl. No, ovdje bi također bilo ispravno koristiti generički stereotip «executionEnvironment». Konačno, s obzirom na to da za pametne uređaje i uslugu SmartDeviceRegService nisu poznate nikakve dodatne informacije o unutarnjoj strukturi, oni se modeliraju kao jednostavni čvorovi.

Literatura

[1] Ambler, S. 2002. *Agile Modeling: Effective Practices for eXtreme Programming and the Unified Process*, 1st ed. Wiley, Hoboken, NJ, USA.

[2] Ambler, S.; Lines, M. 2019. *Choose Your WoW!: A Disciplined Agile Delivery Handbook for Optimizing Your Way of Working (WoW)*, 1st ed. Independently published, ISBN-10: 1790447844, ISBN-13: 978-1790447848.

[3] Arlow, J.; Neustadt, I. 2005. *UML 2 And The Unified Process: Practical Object-Oriented Analysis And Design*, 2nd ed. Addison-Wesley Professional, Boston, MA, USA.

[4] Bittner, K.; Spence, I. 2002. *Use Case Modeling*, 1st ed. Addison-Wesley Professional, Boston, MA, USA.

[5] Blaha, M.; Rumbaugh, J. 2004. *Object-Oriented Modeling and Design with UML*, 2nd ed. Pearson Education, London, UK.

[6] Bruegge, B.; Dutoit, A. H. 2010. *Object-Oriented Software Engineering: Using UML, Patterns, and Java*, 3rd ed. Prentice Hall, Upper Saddle River, NJ, USA.

[7] Booch, G.; Rumbaugh, J.; Jacobson I. 2005. *The Unified Modeling Language User Guide*, 2nd ed. Addison-Wesley Professional, Boston, MA, USA.

[8] Booch, G.; Maksimchuk, R. A.; Engle, M. W. et. al. 2007. *Object-Oriented Analysis and Design with Applications*, 3rd ed. Addison-Wesley Professional, Boston, MA, USA.

[9] Bourque, P.; Fairley, R. E., eds. 2014. *Guide to the Software Engineering Body of Knowledge*, Version 3.0, IEEE Computer Society, https://www.computer.org/education/bodies-of-knowledge/software-engineering pristupljeno u listopadu 2023.

[10] Daoust, N. 2012. *UML Requirements Modeling For Business Analysts*, 1st ed. Technics Publications, LLC, Linda Vista, Sedona, AZ, USA.

[11] Dennis, A.; Wixom, B.; Tegarden, D. 2020. *Systems Analysis and Design: An Object-Oriented Approach with UML*, 6th ed. Wiley, Hoboken, NJ, USA.

[12] Fowler, M. 2003. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*, 3rd ed. Addison-Wesley Professional, Boston, MA, USA.

[13] Graessle, P; Coad, P; Sivdon, I. 2005. *UML 2.0 in Action: A project-based tutorial*, 1st ed. Packt Publishing, Birmingham, UK.

[14] Hay, D. C. 2011. *UML and Data Modeling: A Reconciliation*, Technics Publications LLC, Linda Vista, Sedona, AZ, USA.

[15] Jacobson, I. 1992. O*bject Oriented Software Engineering: A Use Case Driven Approach*, 1st ed. Addison-Wesley, Boston, MA, USA.

[16] Jacobson, I.; Booch, G.; Rumbaugh, J. 1998. *The Unified Software Development Process*, 1st. ed. Addison-Wesley Professional, Boston, MA, USA.

[17] Jović, A.; Horvat, M.; Grudenić, I. 2014. *UML-dijagrami: zbirka primjera i riješenih zadataka*, sveučilišni priručnik, Graphis d.o.o., Zagreb.

[18] Kohn, M. 2009. *Succeeding with Agile: Software Development Using Scrum*, 1st ed. Addison-Wesley Professional, Boston, MA, USA.

[19] Kroll, P.; Kruchten, P. 2003. *The Rational Unified Process Made Easy: A Practitioner's Guide to the RUP*, 1st. ed. Addison-Wesley Professional, Boston, MA, USA.

[20] Kruchten, P. 1995. *Architectural Blueprints — The "4+1" View Model of Software Architecture*, IEEE Software, vol. 12, no. 6, pp. 42-50.

[21] Kruchten, P. 2003. *The Rational Unified Process: An Introduction*, 3rd. ed. Addison-Wesley Professional, Boston, MA, USA.

[22] Larman, C. 2004. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development*, 3rd ed. Pearson Education, London, UK.

[23] Lethbridge, T. C.; Laganière, R. 2004. *Object-Oriented Software Engineering: Practical Software Development Using UML and Java*, 2nd ed. McGraw-Hill Publishing Company, London, UK.

[24] Liskov, B. 1987. *Keynote address - data abstraction and hierarchy*, SIGPLAN Not. 23, 5, 17–34., DOI: 10.1145/62139.62141

[25] Manger, R. 2016. *Softversko inženjerstvo*, sveučilišni udžbenik, Element d.o.o., Zagreb

[26] Martin, R. C. 2002. *Agile Software Development: Principles, Patterns and Practices*, Prentice Hall, Upper Saddle River, NJ, USA.

[27] Mellor, S.; Balcer, M. 2002. *Executable UML: A Foundation for Model-Driven Architecture*, 1st ed. Addison-Wesley Professional, Boston, MA, USA.

[28] Object Management Group 2017. *About the Unified Modeling Language Specification Version 2.5.1*, https://www.omg.org/spec/UML/2.5.1/About-UML pristupljeno u listopadu 2023.

[29] O'Docherty, M. 2005. *Object-Oriented Analysis and Design: Understanding System Development with UML 2.0*, 1st ed. Wiley, Hoboken, NJ, USA.

[30] Pilone, D.; Pitman, N. 2005. *UML 2.0 in a Nutshell*, 2nd ed. O'Reilly Media, Sebastopol, CA, USA.

[31] Podeswa, H. 2005. *UML for the IT Business Analyst: A Practical Guide to Object-Oriented Requirements Gathering*, 1st ed. Thomson Course Technology PTR, Boston, MA, USA.

[32] Pressman, R. 2009. *Software Engineering: A Practitioner's Approach* (7th Edition), McGraw-Hill Science/Engineering/Math, New York, NY, USA.

[33] Rosenberg, D.; Scott, K. 2001. *Applying Use Case Driven Object Modeling With UML: An Annotated E-Commerce Example*, 1st ed. Addison-Wesley Professional, Boston, MA, USA.

[34] Rosenberg, D. 2007. *Use Case Driven Object Modeling with UML: Theory and Practice*, 1st ed. Springer, Cham, Switzerland.

[35] Rubin, H. 2012. *Essential Scrum: A Practical Guide to the Most Popular Agile Process*, 1st ed. Addison-Wesley Professional, Boston, MA, USA.

[36] Rumpe, B. 2016. *Modeling with UML: Language, Concepts, Methods*, 1st ed. Springer, Cham, Switzerland.

[37] Rumpe, B. 2017. *Agile Modeling with UML: Code Generation, Testing, Refactoring*, 1st ed., Springer, Cham, Switzerland.

[38] Schwaber, K.; Sutherland, J. 2020. *The Scrum Guide: The Definitive Guide to Scrum: The Rules of the Game*, https://scrumguides.org/docs/scrumguide/v2020/2020-Scrum-Guide-US.pdf#zoom=100 pristupljeno u listopadu 2023.

[39] Seidl, M.; Scholz, M.; Huemer, C. et. al. 2015. *UML @ Classroom: An Introduction to Object-Oriented Modeling*, Springer, Cham, Switzerland.

[40] Sommerville I. 2016. *Software Engineering*, 10th ed. Person Education Limited, Harlow, UK.

Kazalo

P

R

S

U

V

Z

Č

Ž