

Angelina Njeguš

OBRASCI PROJEKTOVANJA SOFTVERA

Beograd, 2023.

UNIVERZITET SINGIDUNUM

TEHNIČKI FAKULTET

Angelina Njeguš

OBRASCI PROJEKTOVANJA SOFTVERA

Software Design Patterns

Prvo izdanje

Beograd, 2023.

OBRASCI PROJEKTOVANJA SOFTVERA - *Software Design Patterns*

Autor:

prof. dr Angelina Njeguš

Redovni profesor Univerziteta Singidunum

Recenzenti:

prof. dr Mladen Veinović

Redovni profesor Univerziteta Singidunum

prof. dr Marko Tanasković

Redovni profesor Univerziteta Singidunum

prof. dr Boško Nikolić

Redovni profesor Elektrotehničkog fakulteta, Univerziteta u Beogradu

Izdavač:

Univerzitet Singidunum

Danijelova 32

www.singidunum.ac.rs

Za izdavača:

prof. dr Milovan Stanišić

Priprema za štampu:

Miloš Višnjić

Dizajn korica:

Aleksandar Mihajlović, MA

Godina izdanja:

2023

Tiraž:

1000

Štampa:

Caligraph, Beograd

ISBN: 978-86-7912-796-9

Copyright © 2023 Univerzitet Singidunum

Izdavač zadržava sva prava.

Reprodukcija pojedinih delova ili celine ove publikacije nije dozvoljena.

*Ovu knjigu posvećujem svome bratu Novaku Njegušu, koji me i dalje bodri i ispunjava
bratskom ljubavlju iako nije više među nama.*

Sadržaj

PREDGOVOR	1
1. UVOD U OBRASCE RAZVOJA SOFTVERA	5
1.1. RAZUMEVANJE OSNOVNIH RAZLIKA IZMEĐU OBRAZACA ARHITEKTURE I OBRAZACA PROJEKTOVANJA SOFTVERA	6
1.2. ŠTA JE ARHITEKTURA SOFTVERA?	7
<i>Zašto je arhitektura bitna?</i>	8
<i>Koja je razlika između stila i obrazaca arhitekture softvera?</i>	8
1.3. ŠTA SU OBRASCI PROJEKTOVANJA SOFTVERA?	9
2. OSNOVNI POJMOVI I RAD SA OBRASCIMA PROJEKTOVANJA SOFTVERA U IBM RSA ALATU	15
2.1. KATEGORIJE OBRAZACA PROJEKTOVANJA I NJIHOV PRIKAZ U ALATU IBM RSA	15
<i>Vežba 2-1. Dodavanje Pattern Explorer pogleda u IBM RSA alatu</i>	17
2.2. OSNOVNI ELEMENTI OBRAZACA PROJEKTOVANJA	18
2.2.1. Opis osnovnih elemenata obrasca na primeru „State pattern“ i postupak realizacije u alatu	18
<i>Vežba 2-2. Primeniti State obrazac nad klasom Player i izgenerisati izvorni kôd</i>	21
2.2.2. Postupak uključivanja obrasca „Singleton pattern“ u projekat	40
<i>Vežba 2-3. Primeniti Singleton obrazac nad klasom CheckMyLuck kroz korake razvoja projekta od specifikacije zahteva do izvršavanja koda</i>	42
3. KREACIONI OBRASCI PROJEKTOVANJA	61
3.1. FACTORY METHOD	62
<i>Vežba 3-1. Primeniti Factory Method obrazac nad klasom order Pizza (poručivanje pica)</i>	66
<i>Vežba 3-2. Modelovati aplikaciju za logistiku primenom Factory Method obrasca</i>	68
<i>Vežba 3-3. Razviti aplikaciju za igranje igara primenom Factory Method obrasca</i>	74
3.2. ABSTRACT FACTORY	77
<i>Vežba 3-4. Razviti aplikaciju za rad sa različitim sistemima za upravljanje bazom podataka</i>	82
3.3. BUILDER	86
<i>Vežba 3-5. Razviti jednostavnu aplikaciju za kreiranje Happy Meal obroka</i>	88

3.4. PROTOTYPE	92
<i>Vežba 3-6. Primeniti osnovni tip implementacije Prototype obrasca za kloniranje klase Zaposleni</i>	96
<i>Vežba 3-7. Primeniti implementaciju Prototype obrasca tipa PrototypeRegistry na aplikaciju koja klonira vozila</i>	98
3.5. OBJECT POOL	101
<i>Vežba 3-8. Primeniti ObjectPool obrazac na Taxi aplikaciju</i>	102
4. OBRASCI STRUKTURALNOG PROJEKTOVANJA	111
4.1. ADAPTER	112
<i>Vežba 4-1. Primeniti Adapter obrazac</i>	115
4.2. DECORATOR	122
<i>Vežba 4-2. Primeniti obrazac nad aplikacijom za opis i cenu sladoleda sa dodacima</i>	125
4.3. FACADE	128
<i>Vežba 4-3. Primeniti odgovarajući obrazac</i>	131
4.4. BRIDGE	134
<i>Vežba 4-4. Primeniti odgovarajući obrazac na sledeći model Banke</i>	137
<i>Vežba 4-5. Primeniti odgovarajući obrazac projektovanja na aplikaciju za slanje različitih vrsta poruka</i>	140
4.5. COMPOSITE	141
<i>Vežba 4-6. Primeniti Composite obrazac na deo aplikacije za prikaz putanje fajla</i>	143
<i>Vežba 4-7. Primeniti Composite obrazac za crtanje geometrijskih oblika na crtežu</i>	144
4.6. FLYWEIGHT	146
<i>Vežba 4-8. Primeniti Flyweight obrazac na igricu AngryBirds sa funkcionalnostima iscrtavanja i bojenja AngryBirds objekata</i>	151
4.7. PROXY	154
<i>Vežba 4-9. Primeniti Proxy obrazac na autorizaciju plaćanja kreditnom karticom</i>	158
5. OBRASCI PROJEKTOVANJA PONAŠANJA	163
5.1. ITERATOR	164
<i>Vežba 5-1. Razviti aplikaciju koja prikazuje nazive tema prolaskom kroz kolekcije tema</i>	166
5.2. COMMAND	171
<i>Vežba 5-2. Primeniti Command obrazac na sistem poručivanja hrane u restoranu</i>	174
5.3. MEDIATOR	179
<i>Vežba 5-3. Primeniti Mediator obrazac na aplikaciju za upravljanje sletanjima aviona</i>	181

5.4. MEMENTO	184
<i>Vežba 5-4. Primeniti Memento obrazac na scenario čuvanja sadržaja Artikla</i>	187
5.5. STRATEGY	191
<i>Vežba 5-5. Implementirati transakciju Plaćanja koja koristi dve strategije: plaćanje putem kreditne kartice ili putem PayPal sistema</i>	192
<i>Vežba 5-6. Primeniti odgovarajući obrazac na aplikaciju za upravljanje robotima</i>	194
5.6. TEMPLATE METHOD	198
<i>Vežba 5-7. Primeniti Template method obrazac na aplikaciju za igranje igrice</i>	201
5.7. VISITOR	203
<i>Vežba 5-8. Kreirati aplikaciju jedne online prodavnice koja svojim potrošačima želi da omogući pregled svojih poručenih proizvoda i ukupnu cenu poručene robe</i>	204
5.8. CHAIN OF RESPONSIBILITY	207
<i>Vežba 5-9. Primeniti obrazac Chain of Responsibility na sistem za odobravanje odsustva zaposlenih</i>	209
5.9. OBSERVER	214
<i>Vežba 5-10. Primeniti Observer obrazac na aplikaciju za slanje obaveštenja klijentima o rasprodajama koje su u toku</i>	218
<i>Vežba 5-11. Primeniti Observer obrazac za registrovanje korisnika na sportske vesti</i>	223
LITERATURA	226
PRILOZI	229
PRILOG1: PITANJA ZA TESTIRANJE ZNANJA	230
PRILOG2: REŠENJA ZADATAKA	238
Zadatak 2-1. Primeniti State obrazac nad klasom MobileAlert	238
Zadatak 3-1. Kreirati aplikaciju za generisanje računa za električnu energiju u zavisnosti od plana potrošnje primenom Factory Method obrasca	240
Zadatak 3-2. Primeniti odgovarajući obrazac projektovanja na aplikaciju koja se bavi sklapanjem normalnih i gejming računara	243
Zadatak 3-3. Primeniti odgovarajući obrazac projektovanja na aplikaciju koja klonira objekte tipa Osobe (Person) na Univerzitetu	244
Zadatak 4-1. Primeniti Adapter obrazac za prikaz brzine kola u MPH i km/h	247
Zadatak 4-2. Primeniti odgovarajući obrazac na aplikaciju za rentiranje kola	249
Zadatak 4-3. Primeniti obrazac nad aplikacijom za dizajniranje željenih kola	252
Zadatak 4-4. Primeniti odgovarajući obrazac na aplikaciju koja iscrtava obojene geometrijske oblike	254

Zadatak 4-5. Primeniti Composite obrazac na aplikaciju za opis zaposlenih	256
Zadatak 4-6. Primeniti Flyweight obrazac na Counter Strike igricu	259
Zadatak 5-1. Kreirati aplikaciju koja će omogućiti klijentu da lista nazive pesama i njihovih autora, redosledom koji on odabere	262
Zadatak 5-2. Primeniti Command obrazac na komandu butovanje (Bootng) računara	265
Zadatak 5-3. Primeniti odgovarajući obrazac na sistem za daljinsko upravljanje svetlom i ventilacijom u kući	266
Zadatak 5-4. Primeniti Mediator obrazac na aplikaciju za četovanje	269
Zadatak 5-5. Primeniti Memento obrazac na sledeći scenario	272
Zadatak 5-6. Kreirati aplikaciju koja omogućava korisniku da se poveže sa svojim prijateljima putem SM (Social Media) platformi i to preko Instagram-a, Facebook-a i LinkedIn-a	275
Zadatak 5-7. Primeniti Template method obrazac na različite načine poručivanja robe, a koji prate istu proceduru poručivanja	277
Zadatak 5-8. Primeniti obrazac na aplikaciju za servisiranje vozila. Svako vozilo (Car, Van, Motorbike) je posećeno od strane inspektora (VehicleInspector) koji vrši inspekciju (VehicleInspection) i izračunava servisnu naknadu za svako vozilo i ukupne troškove servisiranja	279
Zadatak 5-9. Primeniti obrazac Chain of Responsibility na aplikaciju za obradu i filtriranje pristiglih emailova klijenata	283
Zadatak 5-10. Kreirati aplikaciju za banku koja će redovno obaveštavati svoje klijente (BankClients) i društvene medije (SocialMedia) o promeni kamate (interest) na pozajmljena sredstva	286

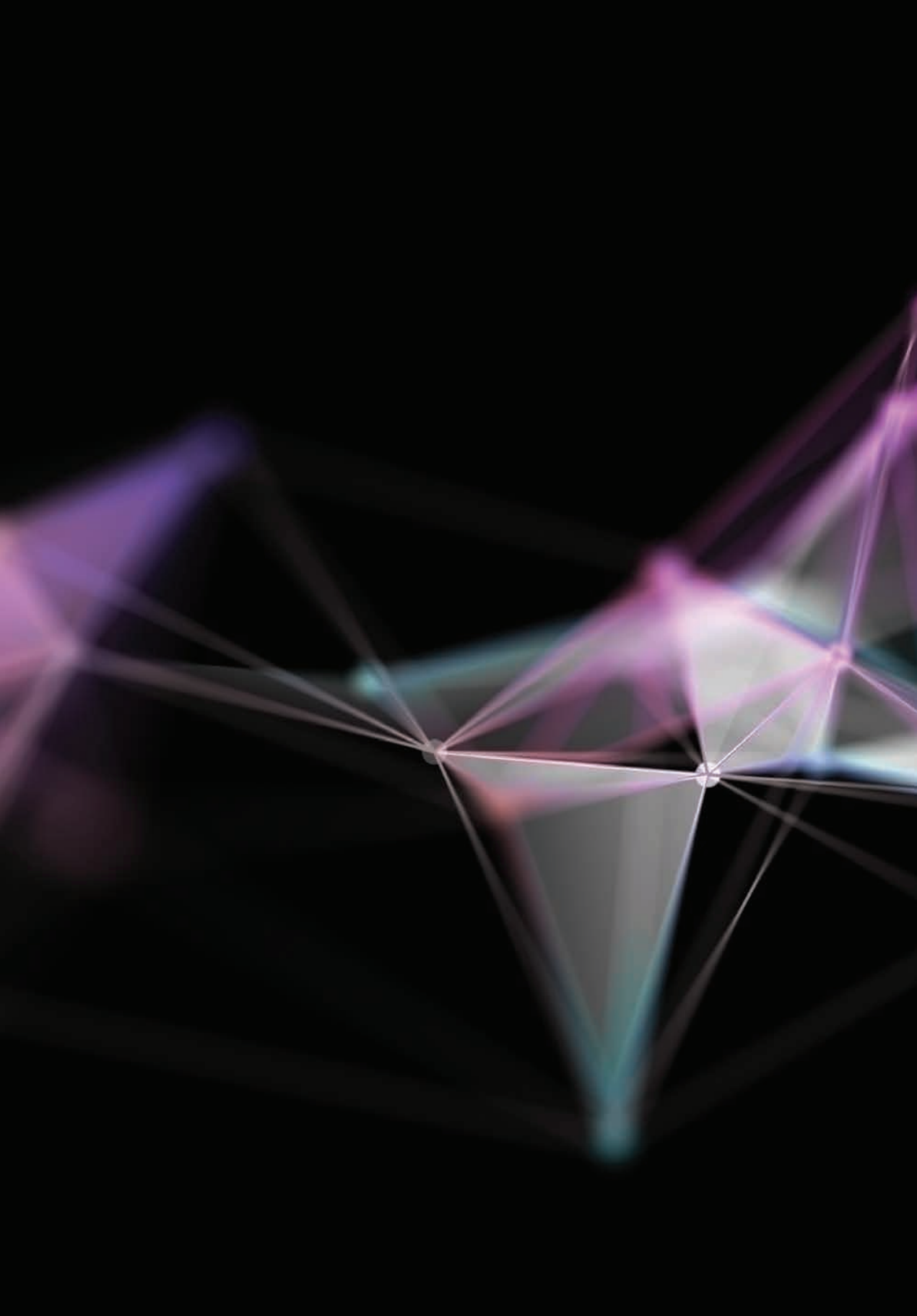
Predgovor

Ovaj udžbenik je namenjen studentima Tehničkog fakulteta i Fakulteta za informatiku i računarstvo Univerziteta Singidunum, ali i svima onima koji žele da steknu znanja iz obrazaca projektovanja softvera (*Design Patterns*). Specifičnost ovog udžbenika se ogleda u tome što se koristi snaga okruženja za razvoj softvera „IBM® Rational® Software Architect“ (IBM RSA) koje pruža širok spektar alata za specifikaciju, analizu, projektovanje i implementaciju softvera. Kako je u ovom udžbeniku fokus na obrascima projektovanja softvera, pomoću ovog alata, na jednostavan način se sagledava primena obrazaca na dijagramu klasa tokom projektovanja softvera, a prilikom generisanja programskog koda i osnovna struktura obrazaca, što znatno olakšava prilagođavanje koda specifičnostima aplikacije, a samim tim se ubrzava savladavanje gradiva i primena obrazaca projektovanja.

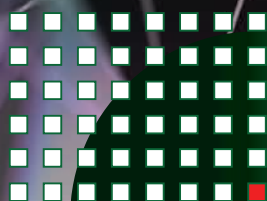
Do danas, razvijeno je dosta obrazaca projektovanja, namenjenih bilo različitim vrstama aplikacija (npr. *Big Data design patterns*, *Machine Learning design patterns*) ili različitim stilovima razvoja aplikacije (npr. *SOA design patterns*, *Microservices design patterns*). U ovom udžbeniku razmatraju se obrasci objektno-orijentisanog razvoja softvera, koji predstavljaju osnovu za bilo kakvu dalju nadogradnju.

U uvodnom delu udžbenika, sagledavanjem celokupnog životnog ciklusa razvoja softvera ukazuje se na položaj primene obrazaca projektovanja softvera. Takođe se ističe razlika između obrazaca arhitekture softvera (*Architectural Patterns*) i obrazaca projektovanja (*Design Patterns*) softvera. U drugom delu se objašnjava primena obrazaca pomoću IBM *Rational Software Architect* (RSA) alata, gde je akcenat na funkcionalnostima samog alata, radi lakšeg praćenja objašnjenja obrazaca. U narednim poglavljima, koja su podeljena prema kategorijama obrazaca, daju se objašnjenja obrazaca uz prikaz praktičnih vežbi i zadataka za testiranje poznavanja obrazaca. U prilogu udžbenika su dostupna pitanja za testiranje znanja i rešenja zadataka.

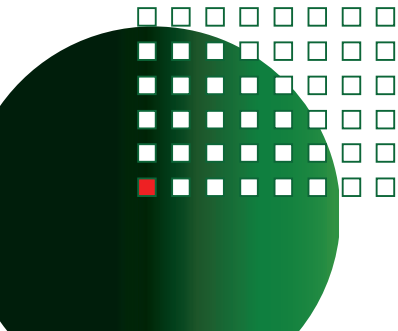
Za bilo kakva dodatna pitanja ili komentare vezano za ovaj udžbenik možete pisati autoru na anjugus@singidunum.ac.rs.



POGLAVLJE 1



UVOD U OBRASCE RAZVOJA SOFTVERA



1. UVOD U OBRASCE RAZVOJA SOFTVERA

Dobili ste projekat, koji ste dogovorili za dobru cenu, ali ste obećali i vremenski rok za isporuku delova rešenja. S obzirom na složenost aplikacije, a i krajnji rok, angažovali ste odgovarajući broj članova tima gde je svaki stručan u svom domenu. Okupili ste tim da se dogovorite o načinu rada. O čemu ćete se tačno dogovarati?

Prvi dogovor koji se želi postići sa timom, to je dogovor oko **metodologije razvoja softvera**. Ukoliko se odabere *Scrum* okvir (*framework*), organizacija tima, razgovori sa korisnicima, pa i sam razvoj sistema, imaće svoj specifičan tok orijentisan oko sprintova (Njeguš & Milanov, 2011). Ukoliko se odabere dizajnersko razmišljanje (*Design Thinking*) tada se ne upušta u razvoj, sve dok se ne kreira takav prototip rešenja, koji reflektuje realne potrebe, probleme i izazove sa kojima se susreće korisnik. Metodologije razvoja softvera predstavljaju dogovor ne samo oko organizacije tima, već i o dinamici rada i isporuke verzija rešenja (Milanov & Njeguš, 2012). Danas se uglavnom koriste agilne metodologije razvoja softvera, mada su i dalje kod nekih kompanija zastupljeni tradicionalni pravci, zbog jasno definisanih koraka u koje su ugrađene dobre prakse razvoja sistema.

Nakon izabrane metodologije razvoja softvera, naredni korak je **specifikacija korisničkih zahteva**. Izabrana metodologija će uticati na način specifikiranja korisničkih zahteva. U svakom slučaju, u ovom koraku bitno je da se sa korisnicima dogovori tačan spisak najbitnijih, tržišno isplativih (*Minimum Marketable Feature*, MMF), funkcionalnosti rešenja. Kod objektno-orijentisanog razvoja softvera u ovom koraku će se kreirati dijagram slučajeva upotrebe (*Use Case*, UC), na kome će se jasno videti dogovorene ključne funkcionalnosti aplikacije, kao i spoljni akteri sa kojima će sistem komunicirati. Međutim, naziv slučajeva upotrebe ne objašnjava dovoljno o načinu njegove realizacije, te se pribegava kreiranju i dijagrama aktivnosti (*Activity Diagram*, AD), koji algoritamski opisuje konkretne aktivnosti određenog slučaja upotrebe (UC). Mogu se koristiti i druge notacije za specifikiranje zahteva korisnika, npr. BPMN (*Business Process Model and Notation*) koji se može koristiti za jedan detaljan prikaz aktivnosti procesa u okviru slučaja upotrebe. Ukoliko se tim opredelio za neku od agilnih metodologija razvoja, npr. Scrum, specifikacija počinje sa kreiranjem korisničkih priča (*user stories*), koje će biti sortirane po prioritetu, u dokumentu koji se naziva *product backlog*. Treba imati u vidu, ukoliko je zahtev korisnika da se razvije sistem koji, između ostalog, realno preslikava način trenutnog poslovanja, tada je neophodno **snimiti postojeće stanje sistema**. Za snimak postojećeg stanja mogu se koristiti razne tehnike. Rezultat ove faze su modeli procesa i podataka, kreiranih bilo dijagramima toka podataka (*Data Flow Diagram*, DFD), IDEF (*Integration DEFinition*) standardnim notacijama za modelovanje procesa i podataka, BPMN notacijom ili dr. (Njeguš, 2021)

Nakon ugovorenih zahteva pristupa se **modelovanju aplikacije kroz faze analize i projektovanja softvera**. Ukoliko je u pitanju sistem sa složenijim funkcionalnostima, modeli će biti prikazani sa različitim nivoom detalja (npr. konceptualni, logički ili fizički model), a mogu obuhvatiti i različite perspektive sistema (npr. dijagrami stanja, komponenti i dr.).

Svakako **prvi korak**, pre kreiranja bilo kog modela jeste **dogovor oko arhitekture softvera prema kome će se ceo tim voditi**. Nakon toga sledi faza analize sistema, gde se kreiraju logički modeli, odnosno fokus je na tome šta sistem treba da radi, a ne i kako (npr. u ovom koraku se ne razmišlja o tipovima podataka atributa ili polja, niti o punim nazivima metoda i sl.). Ukoliko je generalno postignut dogovor oko logičkih modela, prelazi se u fazu projektovanja softvera, gde fizički modeli reflektuju programski kod aplikacije. U ovom koraku, poznavanje i primena odgovarajućih **obrazaca projektovanja** (*Design Patterns*) mogu ubrzati i olakšati generisanje kvalitetnog i efikasnog koda, koji je robustan i adaptibilan na buduće zahteve korisnika.

U zavisnosti od nivoa modelovanja, zavisice i vreme **kodiranja aplikacije**. Što su modeli preciznije definisani, to je vizuelno jasniji kod, a i manje će vremena biti potrebno u doterivanju generisanog koda. Razvijene i tesitrane funkcionalnosti od strane developera predstavljaju ulaz u **fazu testiranja**. Tester primenjuju razne nivo testiranja (npr. jedinično, integraciono ili sistemsko testiranje), kao i metode testiranja (npr. funkcionalno, regresiono, statičko, bezbednostno, testiranje upotrebljivosti i dr.). Primarni cilj testiranja jeste da aplikacija rešava potrebe klijenata koje su dogovorene u početnim fazama projekta. Kada su ispravljene sve kritične greške kreće se sa **fazom uvođenja rešenja i puštanjem aplikacije u produkciono okruženje**. Kada klijent počne da koristi razvijeno rešenje, tada se često pojave realni problemi koje je neophodno rešavati u hodu, a koji su sastavni deo **faze održavanja** aplikacije.

Svi ovi, prethodno opisani koraci razvoja softvera predstavljaju neke vrste obrazaca ili šablona proisteklih iz dosadašnjih najboljih praksi u razvoju kvalitetnog i efektivnog rešenja. U ovom udžbeniku, primarni akcenat je na obrascima projektovanja (*design patterns*), koji su sastavni deo faze projektovanja softvera.

1.1. RAZUMEVANJE OSNOVNIH RAZLIKA IZMEĐU OBRAZACA ARHITEKTURE I OBRAZACA PROJEKTOVANJA SOFTVERA

U kojoj fazi razvoja softvera se razmatra arhitektura softvera, a u kojoj obrasci projektovanja (*design patterns*)?

Bilo koju vrstu aplikacija da razvijate, generalno ćete razmatrati:

1. **Zahteve korisnika** na osnovu kojih ćete napraviti bilo specifikaciju korisničkih zahteva ili prototipe aplikacije ili dijagrame slučajeva upotrebe (*use case*) i njima prateće dijagrame aktivnosti ili ćete koristeći neki model procesa, kreirati npr. BPMN (*Business Process Model and Notation*) dijagrame ili ćete kreirati listu korisničkih priča tzv. *Product Backlog* i sl. Način na koji ćete mapirati zahteve korisnika zavisi od načina organizacije tima (zainteresovanih strana, tzv. *stakeholders*), odnosno od zajedničkog dogovora koju ćete metodologiju razvoja softvera koristiti (npr. *Scrum*, *Kanban*, *Design Thinking*, *DevOps* i sl.).

2. **Arhitekturu softvera** – Nakon dogovora sa korisnikom šta će se razvijati, u kom vremenskom periodu, sa kojim resursima i dr., u narednom koraku tim se dogovara o načinu realizacije, organizacije i o samoj strukturi aplikacije, odnosno o arhitekturi softvera, koja predstavlja osnovu za razvoj aplikacije.
3. **Razvoj softvera** – U zavisnosti od veličine aplikacije, najčešće će se prvo kreirati i testirati logički modeli aplikacije, koji se rade u **fazi analize softvera** (*software analysis*), a zatim kreirati fizički modeli, koji se rade u **fazi projektovanja softvera** (*software design*). Ovi modeli se prevode u izvorni kod koji se dalje testira i verifikuje sa zainteresovanim stranama.
4. **Testiranje softvera** – Bilo da se softver testira manuelno ili automatski neophodno je primeniti razne nivoe (npr. jedinično, integraciono ili sistemsko testiranje) i metode testiranja (npr. funkcionalno, regresiono, statičko, bezbednostno, testiranje upotrebljivosti i dr.).
5. **Uvođenje softvera** (*Deployment*) – kada se softver koristi od strane korisnika, kaže se da je u produkciji. Generalno faza uvođenja softvera može podrazumevati i obuku korisnika i nabavku dodatne opreme i druga dodatna angažovanja neophodna za neosmetanu upotrebu softvera od strane korisnika. Krajnji cilj ipak jeste zadovoljan klijent i softver koji reflektuje strategiju organizacije, a u koraku je sa savremenim trendovima u razvoju informacionih tehnologija (IT).

Očigledno je da će se o arhitekturi softvera i mogućim stilovima i obrascima dogovarati u fazi pre početka razvoja softvera. Dok će se obrasci projektovanja (*design patterns*) razmatrati u fazi projektovanja softvera (*software design*).

1.2. ŠTA JE ARHITEKTURA SOFTVERA?

Arhitektura softvera je dogovor o načinu razvoja i organizovanja softverskih komponenti. Neka detaljnija objašnjenja i definicije date su u nastavku:

- Arhitektura softvera podrazumeva skup strateških odluka o **strukturi softverskog sistema**, koji je prikazan kao **kolekcija komponenata** koji ispunjavaju željene **funkcionalnosti sistema**, optimizujući pritom kvalitet, performanse, bezbednost i upravljivost celokupnog sistema.
- Arhitektura se fokusira na organizovanje komponenata kako bi se podržale specifične funkcionalnosti sistema.
- Arhitektura softvera obuhvata niz značajnih odluka o organizaciji softverskog sistema koji se tiču funkcionalnosti, upotrebljivosti, otpornosti, performansi, ponovne upotrebljivosti, razumljivosti, ekonomskih i tehnoloških ograničenja i kompromise (Kruchten et al, 1995).
- Arhitektura softvera je struktura ili struktuiranje sistema koji obuhvata softverske elemente, njihove javno vidljive osobine i veze između njih (Bass et al, 2021).

Zašto je arhitektura bitna?

Softver mora da bude izgrađen na čvrstim temeljima. Rizici loše arhitekture su nestabilnost softvera, nemogućnost da podrži postojeće ili buduće poslovne zahteve ili ga je teško uvesti ili upravljati u proizvodnom okruženju.

Obrasci (šabloni, uzorci) arhitekture softvera (*Architectural patterns*) predstavljaju skup principa koji nude apstraktni okvir sistema koji olakšavaju podelu i ponovnu upotrebu pružajući pritom rešenja za učestale probleme. Svaki od stilova arhitekture rešava određene probleme i odnosi se na određene karakteristike sistema, performansi, procese, distribuciju i dr.

Koja je razlika između stila i obrazaca arhitekture softvera?

Iako se u široj literaturi poistovećuju stilovi arhitekture (*Architectural styles*) i obrasci arhitekture (*Architectural patterns*) softvera, ipak postoji neznatna razlika. Naime stil arhitekture je na višem nivou apstrakcije, dok obrazac arhitekture prikazuje način implementacije stila arhitekture. Određeni stilovi arhitekture po kategorijama su prikazani u tabeli 1. Na primer, ukoliko je u aplikaciji fokus na komunikaciji distribuiranih komponentata, predlaže se servisno-orijentisana arhitektura ili mikroservisna ili magistrala poruka (*message bus*). Ukoliko je fokus na uvođenju (*deployment*) komponentata sistema na čvorove na mreži (npr. veb server, aplikativni server, server baze podataka), onda je odgovarajući stil, na primer, višeslojna arhitektura i sl.

Tabela 1. Pojedini stilovi arhitekture prema kategorijama.

Kategorija	Primeri stilova arhitekture softvera
Struktuiranost (<i>Structure</i>)	Cevi i filteri (<i>Pipes and Filters</i>), Komponentno zasnovani (<i>Component-based</i>), Objektno-orijentisani, Slojeviti (<i>Layered</i>) stilovi
Distribuiranost (<i>Distributed</i>)	Servisno-orijentisani, <i>Proxy</i> , <i>Peer to Peer</i> , <i>Broker</i> , <i>Cloud Computing</i> stilovi
Adaptivnost (<i>Adaptive</i>)	<i>Microkernel</i> , <i>Domain-specific</i> , <i>Master and Slave</i> stilovi
Razmena poruka (<i>Messaging</i>)	<i>Publish-Subscribe</i> , <i>Message Bus</i> stilovi
Deljenja memorija (<i>Shared Memory</i>)	<i>Blackboard</i> , <i>Database-centric</i> , <i>Role-based</i> stilovi
Konkurentnost (<i>Concurrency</i>)	<i>Reactor</i> , <i>Event Driven</i> , <i>Active Object</i> stilovi

Stil arhitekture je generalno dogovor po kom principu će se razvijati aplikacija, odnosno kako će se organizovati izvorni kod. Neki od postojećih stilova arhitekture su:

- Monolitna (*Monolithic*)
- Slojevita (*Layered*)
- Komponentna (*Component-based*)
- Cevi i filteri (*Pipes and filters*)
- Vođena događajima (*Event-driven*)
- Objava-Pretplata (*Publish-subscribe*)
- Servisno-orijentisana i dr.

Stilovi arhitekture softvera se mogu implementirati na različite načine, sa specifičnim okvirima, politikama, praksama, tehničkim okruženjem i dr.

Obrasci su rešenja za ponavljajuće probleme. **Obrasci arhitekture** softvera utiču na organizaciju i raspoređivanje programskog koda, a nekad i na čitavu aplikaciju bilo horizontalno (npr. kako struktuirati kod unutar sloja) ili vertikalno (npr. kako se zahtev obrađuje između slojeva aplikacije). Primeri obrazaca arhitekture su:

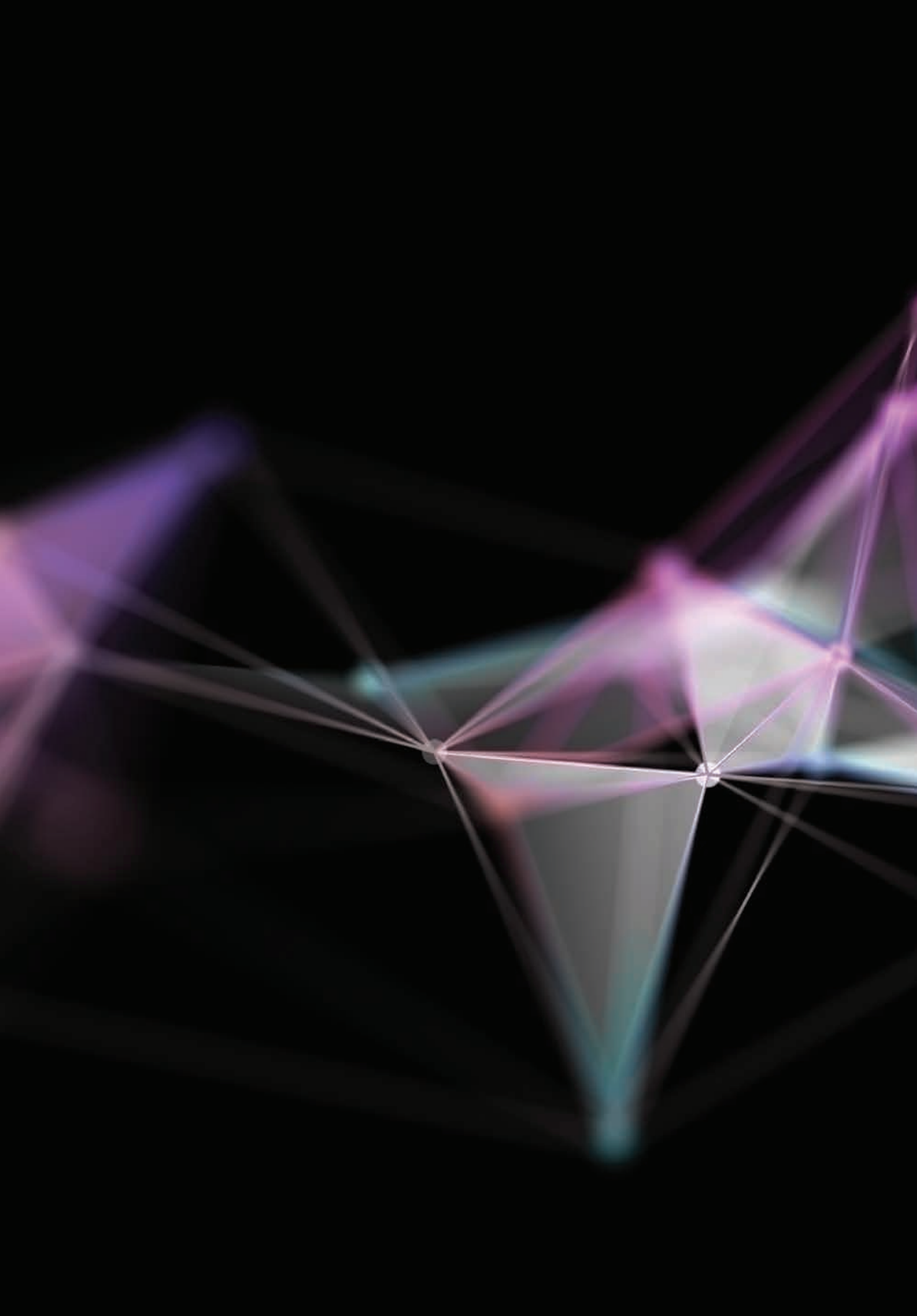
- *Slojevita (Layered)*, npr. troslojna (*Three tier*)
- *Model-View-Controller (MVC)*
- *Master-slave*
- *Pipe-filter*
- *Broker*
- *Peer-to-peer*
- *Event-bus*
- Mikroservisna i dr.

1.3. ŠTA SU OBRASCI PROJEKTOVANJA SOFTVERA?

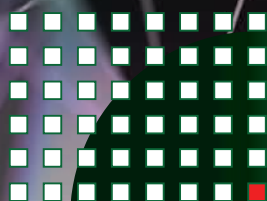
Do sada ste uvideli da pre nego što se upustite u razvoj aplikacije, sa svojim razvojnim timom se dogovarate o arhitekturi softvera, oko koje ćete svi razvijati aplikaciju. U zavisnosti od tipa aplikacije koju razvijate, odabracete odgovarajući obrazac arhitekture. Nakon toga, krećete sa razvojem aplikacije. Tokom razvoja nailazićete na razne dileme, probleme i druga pitanja, npr. uočili ste da vam je dijagram klasa mnogo složen i da ima tendenciju rasta ne samo u obimu klasa već i njihovih međusobnih veza, posebno kako budu pristizali novi zahtevi korisnika. Posle određenog vremena, uočavate i da se iste strukture ponavljaju na vašim dijagramima i druga pitanja. Olakšavajuća okolnost je da su generacije programera pre vas, rešavajući ista pitanja, kreirali opšta rešenja za učestale probleme, odnosno tzv. šablone ili obrasce projektovanja (*design patterns*).

Obrasci projektovanja su primeri najbolje prakse programiranja koji pomažu da kôd bude čitljiv, ponovo upotrebljiv, da se bolje izvršava, da bude lak za održavanje i dr. (Sataloff et al., 2005) Ovi obrasci postaju posebno popularni izlaskom knjige „*Design Patterns: Elements of Reusable Object-Oriented Software*“, četvorice autora *Erich Gamma, Richard Helm, Ralph Johanson i Jphm Vlissides*, 1995 godine.

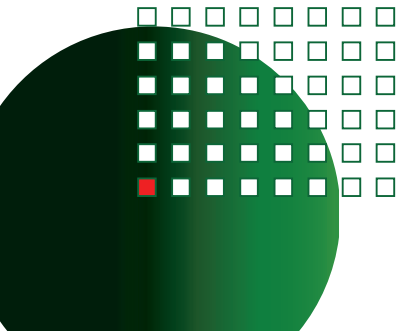
Zaokružujući svoje bogato iskustvo u objektno-orijentisanom projektovanju softvera, ovi autori su objavili 23 obrazaca projektovanja, koji nikada do tada nisu bili publikovani. Obrasci projektovanja su u osnovi “inženjerski priručnik” za projektovanje softverskih sistema.



POGLAVLJE 2



**OSNOVNI POJMOVI I RAD SA
OBRASCIMA PROJEKTOVANJA
SOFTVERA U IBM RSA
ALATU**



2. OSNOVNI POJMOVI I RAD SA OBRASCIMA PROJEKTOVANJA SOFTVERA U IBM RSA ALATU

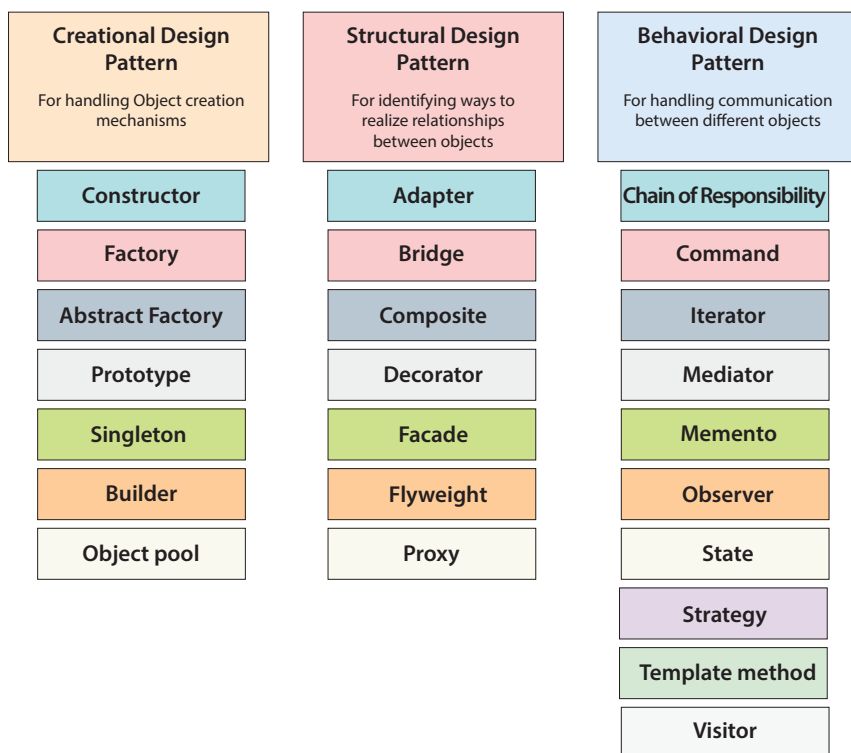
2.1. KATEGORIJE OBRAZACA PROJEKTOVANJA I NJIHOV PRIKAZ U ALATU IBM RSA

U zavisnosti od problema koji rešavaju, obrasci projektovanja se mogu svrstati u tri kategorije (Slika 1) (Bin Uzayr, 2023):

- ♦ **Kreacioni obrasci** ili obrasci kreiranja objekata (*Creational patterns*) – rešavaju probleme kreiranja ili instanciranja objekata, na način koji je prilagođen određenoj primeni. Posebno su važni u situacijama u kojima bi uobičajen pristup kreiranja objekata doveo do povećanja kompleksnosti aplikacije. Neki od obrazaca koji spadaju u ovu grupu su:
 - Singleton
 - Factory
 - Abstract Factory
 - Builder
 - Prototype
 - Object pool
- ♦ **Strukturalni obrasci** (*Structural patterns*) - rešavaju problem sklapanja ili kombinovanja klasa na način da se postigne bolja fleksibilnost i efikasnost. Neki od obrazaca koji se bave boljom organizacijom struktura klasa su:
 - Adapter
 - Bridge
 - Composite
 - Decorator
 - Facade
 - Flyweight
 - Proxy
- ♦ **Obrasci ponašanja** (*Behavioral patterns*) – bave se algoritmima, interakcijama i raspoređivanjem odgovornosti između klasa, odnosno akcenat je na interakciji, pre nego na struktuiranju klasa. Interakcija između klasa bi trebala da bude takva da klase mogu lako da komuniciraju, ali da i dalje ostane labava povezanost između njih. Neki od obrazaca su:
 - Iterator
 - Observer
 - State

- Mediator
- Memento
- Strategy
- Template Method
- Visitor
- Command
- Chain of Responsibility

Slika 1. Lista obrazaca projektovanja prema kategorijama.



(modifikovano sa izvora: <https://www.telerik.com/blogs/design-patterns-in-javascript>)

2.2. OSNOVNI ELEMENTI OBRAZACA PROJEKTOVANJA

Da bi razumeli suštinu obrazaca projektovanja, način na koji rešavaju probleme i njegove osnovne elemente, započecemo jednim primerom iz realnog života.

Ukoliko se kaže reč „kuhinja“, na šta vas asocira? Verovatno ste pomislili na hranu, frižider, šporet i sl. S obzirom da svako zna šta je kuhinja i koji su njeni osnovni elementi, da li to znači da su sve kuhinje identične? Koncept kuhinje se može implementirati na hiljade različitih načina. Kuhinja je samo obrazac, ideja koja omogućava ljudima da dele zajedničku viziju bez ograničenja kako će implementirati tu viziju.

Da bi adekvatno razumeli i primenili obrazac, svaki obrazac opisuje problem koji rešava, resurse koji su mu potrebni za rešavanje problema, pravila kako treba koristiti te resurse i smernice gde i kada treba koristiti obrazac. U tabeli 2., opisani su osnovni elementi na primeru „kuhinja“ kao obrasca.

Tabela 2. Primer opisa elemenata obrasca na primeru „kuhinja“.

Elementi obrasca	Opis elemenata obrasca za kuhinju
Problem koji rešava	Potrebno nam je mesto gde ćemo čuvati i pripremati hranu.
Resursi koji su potrebni za rešavanje problema	Možemo koristiti uređaje, hranu, posuđe i dr.
Pravila kako treba koristiti resurse za rešavanje problema	Frižider skladišti hranu, rerna služi za prženje ili podgrevanje hrane i dr.
Smernice gde treba koristiti obrazac	Ovaj obrazac dobro funkcioniše kada se koristi u kući, dok nije preporuka da se koristi van kuće (npr. kod kampovanja).

Radi boljeg razumevanja, u narednim potpoglavljima opisaće se osnovni elementi i načini primene jednostavnijih obrazaca kao što su *State*, *Singleton* i *Observer*.

2.2.1. Opis osnovnih elemenata obrasca na primeru „*State pattern*“ i postupak realizacije u alatu

Da li ste nekada pisali metode u kojima treba da proveravate mnogo uslova koristeći iskaze *if-else* ili *switch*? Ne možete da odredite kako metod radi dok prvo ne proverite uslove različitih vrednosti atributa u objektu. Kada proveravate vrednosti atributa u uslovu, vi ustvari proveravate stanje objekta.

Kada implementacija metoda zavisi od promena u vrednostima atributa (promene stanja), kôd može postati veoma složen kako za pisanje tako i za održavanje. *State design pattern* (Obrazac Stanja) nudi rešenje koje olakšava inicijalno pisanje koda i umanjuje troškove njegovog održavanja.

Ukoliko, na primer, razvijate aplikaciju za upravljanje zalihama proizvoda. U zavisnosti od stanja klase Proizvod, realizovaće se određene metode. Proizvod može biti u stanjima: primljen u magacin (*Received*), na zalihama (*InStock*), održavan (*Held*) i prodat (*Sold*) (Slika 3). Da bi se izbeglo mnoštvo *if - then* iskaza radi ispitivanja stanja u kome se objekat trenutno nalazi, rešenje je da se primeni *State pattern*.



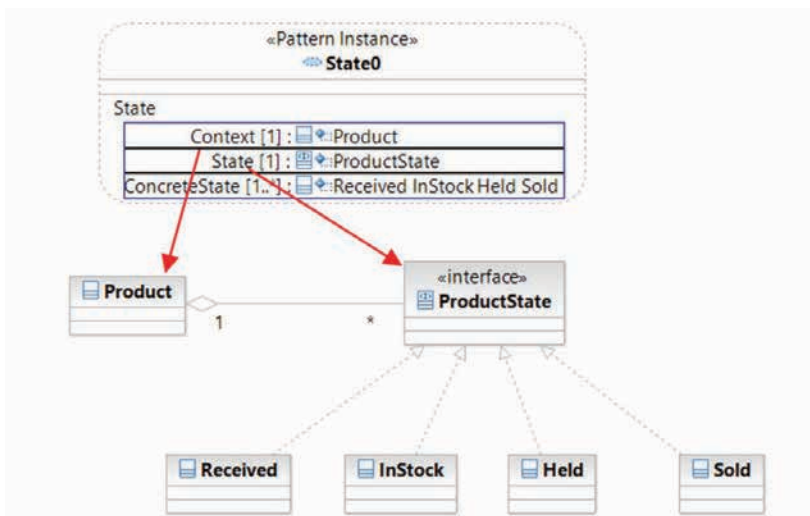
Prema opisu osnovnih resursa neophodnih za primenu ovog obrasca, prikazanih u Tabeli 3, pored klase Proizvod koja obezbeđuje kontekst (*Context*), kreiraće se i klasa tipa interfejs StanjeProizvoda (*ProductState*) i onoliko konkretnih klasa koliko ima mogućih stanja.

Tabela 3. Opis osnovnih elemenata *State* obrasca nad klasom Proizvod.

Elementi obrasca	Opis elemenata obrasca <i>State</i> za klasu Proizvod
Problem	Ponašanje objekta Proizvod treba da se promeni u zavisnosti od toga u kom se stanju nalazi.
Resursi	Resursi koji rešavaju ovaj problem su: • Klasa Kontekst (<i>Context</i>) [1]: Product (Proizvod) koji održava referencu na objekat konkretnog stanja koje se koristi za definisanje trenutnog stanja objekta. On delegira ponašanje specifično za stanje različitim objektima stanja preko interfejsa <i>ProductState</i>. • Interfejs Stanje (<i>State</i>) [1]: ProductState (Stanje proizvoda) je obično realizovan kao interfejs koji definiše metode kojima svako stanje mora da rukuje. • Konkretna stanja (<i>ConcreteState</i>) [1..*]: Klase <i>Received</i>, <i>InStock</i>, <i>Held</i> i <i>Sold</i> koje sadrže ponašanje specifično za stanje, odnosno obezbeđuju implementaciju za metode definisane u <i>ProductState</i>.
Pravila	<i>ProductState</i> je generalizovana ili interfejs klasa koja definiše individualna stanja proizvoda kao i njihove implementacije za svaku metodu. Klasa <i>Product</i> prihvata pojedinačne zahteve od korisnika i svaki delegira odgovarajućoj instanci podklase <i>ProductState</i> , koja predstavlja trenutno stanje proizvoda. Instanca <i>ProductState</i> izvršava implementaciju i vraća kontrolu klasi <i>Product</i> .
Smernice	Ovaj obrazac ne koristiti tamo gde ne postoji značajan broj stanja ili ukoliko se ponašanje ne razlikuje značano od stanja do stanja.

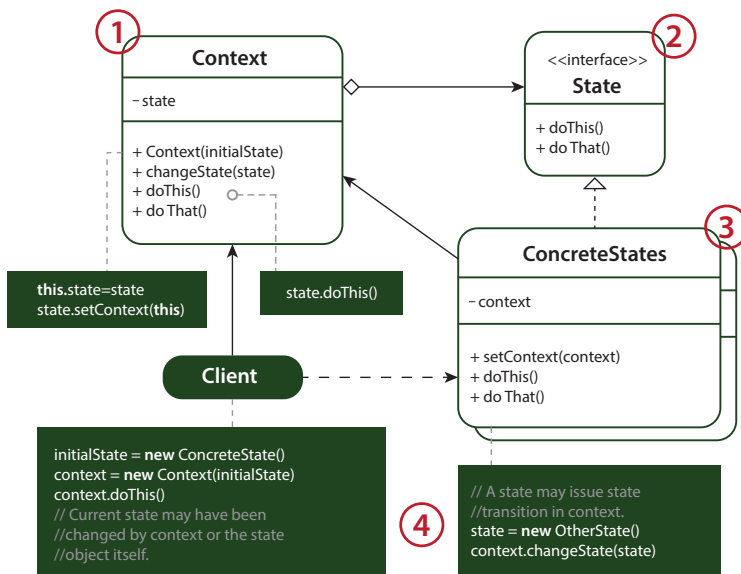
Konceptualni dijagram klasa sa primenjenim obrascem *State* na primeru klase *Proizvod* je prikazan na slici 3.

Slika 3. Dijagram klasa za *State* obrazac na primeru klase *Proizvod*.



Na osnovu ovog primera, može se izvesti osnovna struktura *State* obrasca, prikazane na slici 4. Ovaj obrazac spada u kategoriju obrazaca ponašanja.

Slika 4. Osnovna struktura *State* obrasca (Shvets, 2019).



Da bi se još bolje objasnio *State* obrazac, na jednom jednostavnom primeru prikazaće se programski kod pisan u Javi.

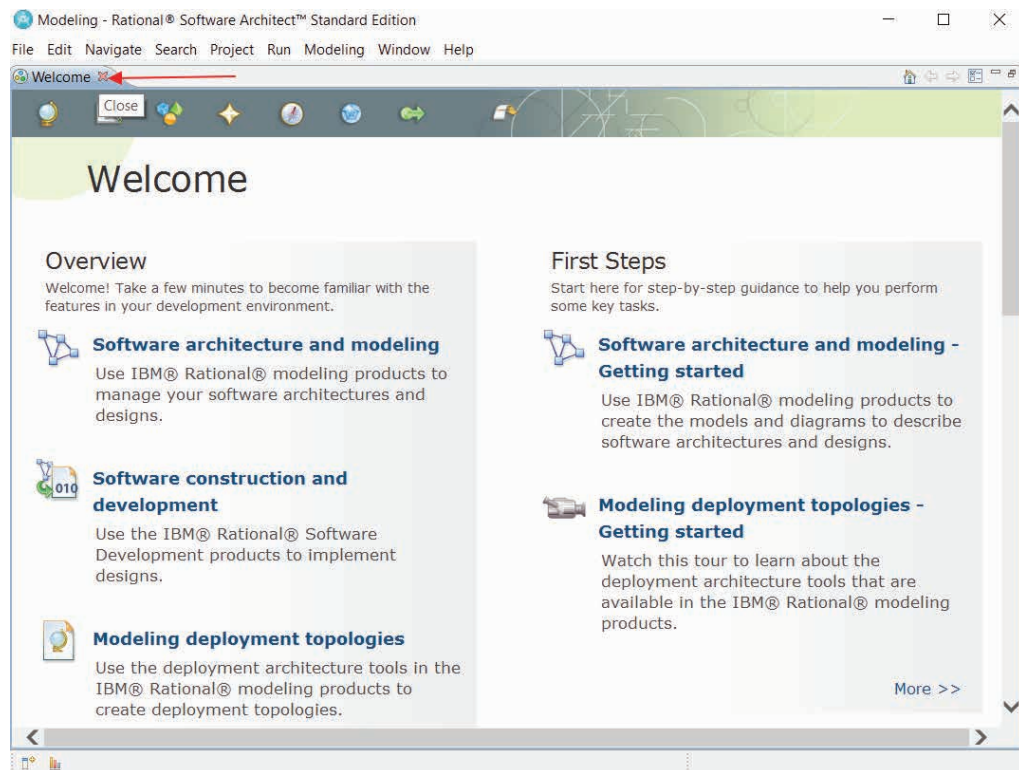
Vežba 2-2. Primeniti *State* obrazac nad klasom *Player* i izgenerisati izvorni kôd.

Igrač (*Player*) može da bude u dva stanja *StartState* i *StopState*. Dok je u stanju *StartState*, ispisati „*Player is in start state*“, a dok je u stanju *StopState*, prikazati „*Player is in stop state*“. Kreirati logički dijagram klasa, primeniti *State* obrazac, izgenerisati Java programski kod i prikazati rezultat.

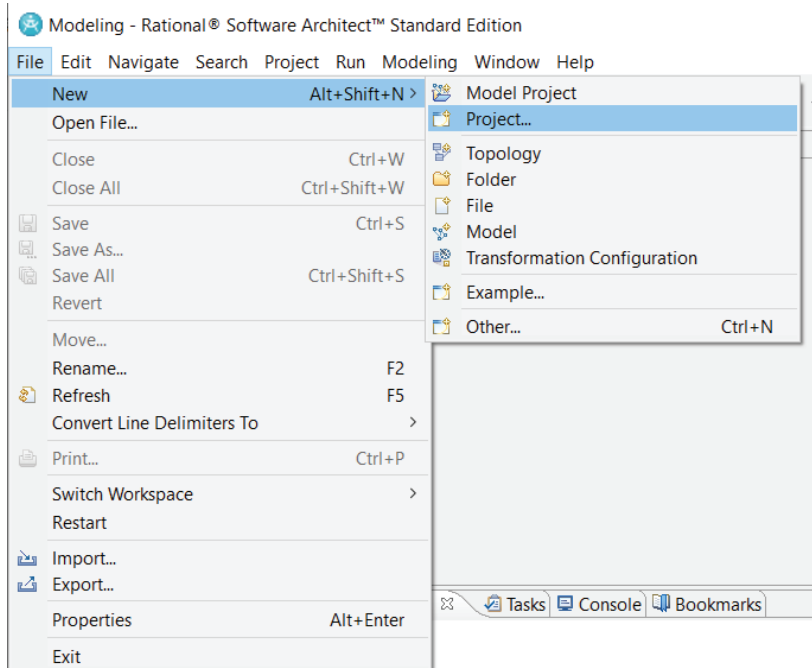
Koraci izrade ove vežbe u IBM RSA alatu su:

Korak 1. Kreirati projekat.

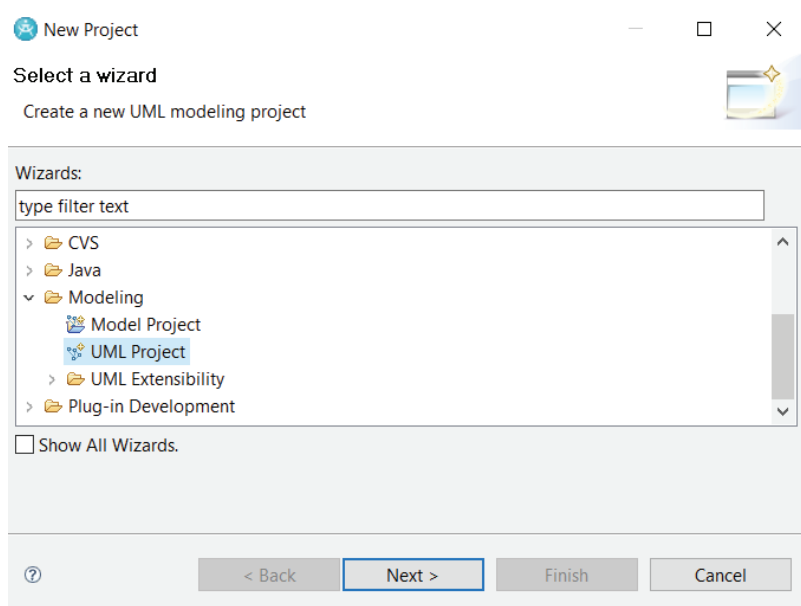
- ♦ Ukoliko prvi put pokrećete RSA alat, pojaviće vam se *Welcome* ekran. Potrebno je da ga zatvorite, klikom na , kako bi vam se pojavilo radno okruženje.



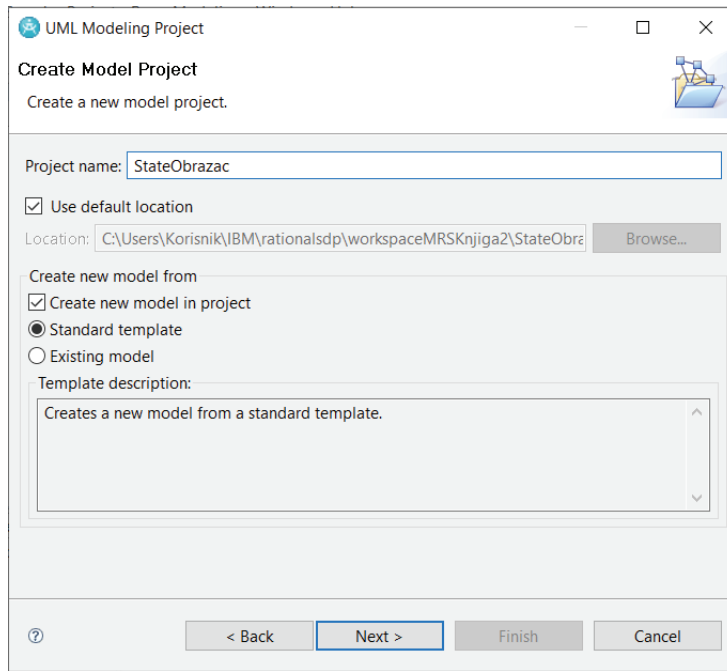
- ◆ Kliknite na **File > New > Project**:



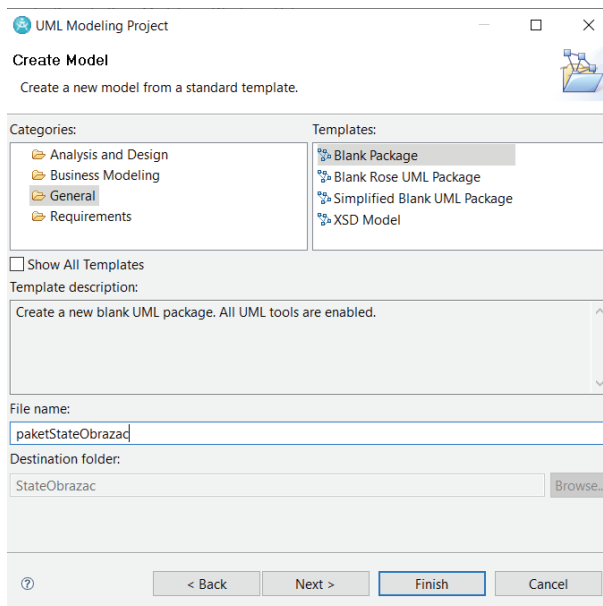
- ◆ Proširite **Modeling** i kliknite na **UML Project**, zatim na **Next**:



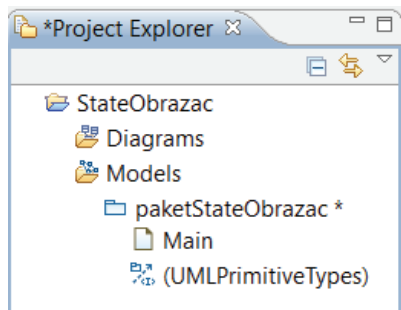
- Imenujte projekat, npr. **Project name: StateObrazac** i kliknite na **Next**.



- Imenujte paket, npr. **File name: paketStateObrazac** i kliknite na **Next**.

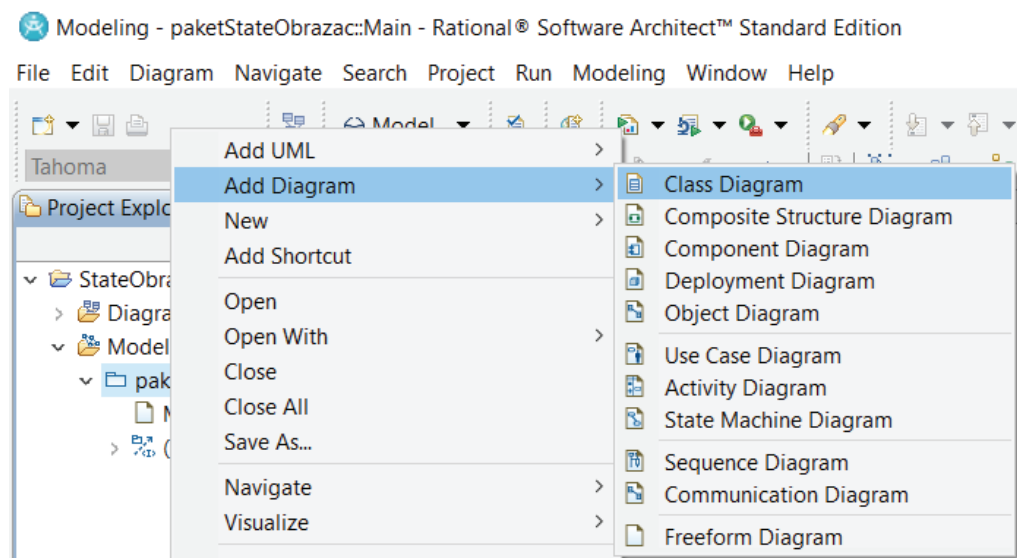


- ♦ Projekat je kreiran. Možete ga proširiti, kako bi videli sve elemente projekta.

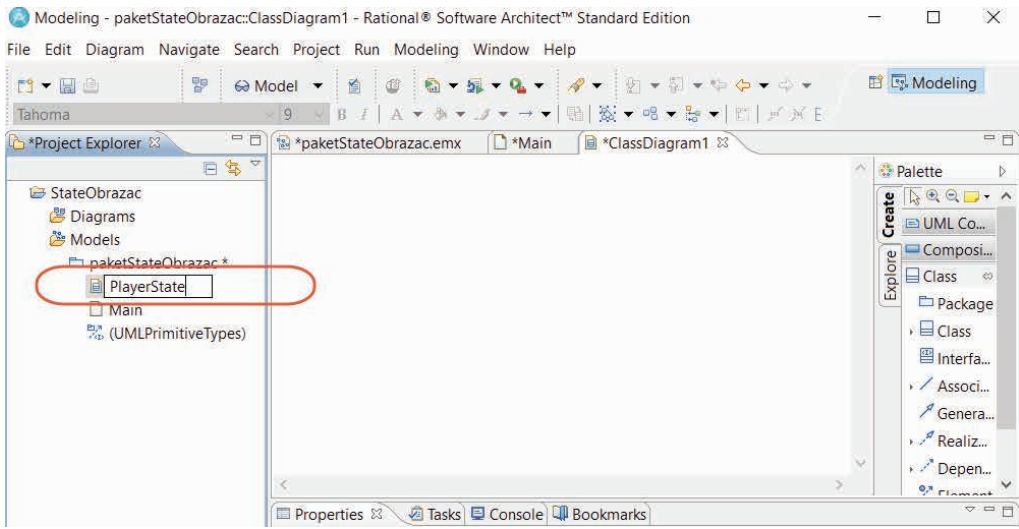


Korak 2. Kreirati dijagram klasa.

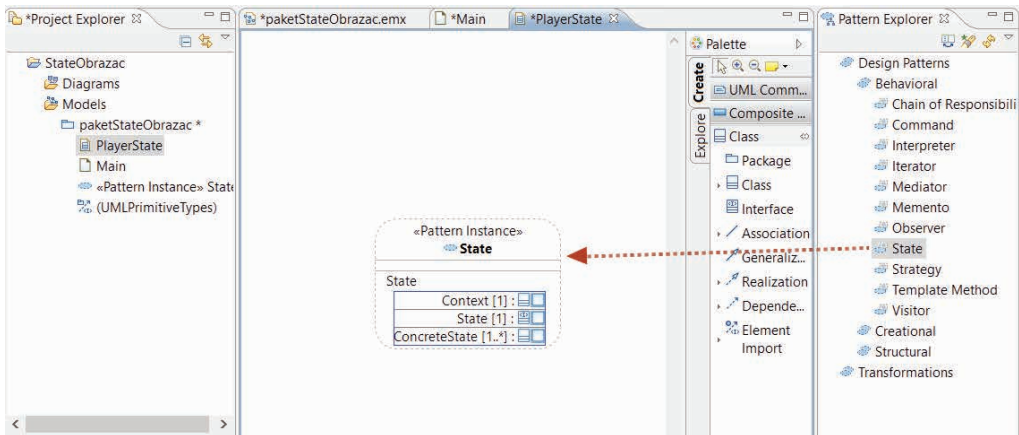
- ♦ Desnim klikom na paket paketStateObrazac selektujete **Add Diagram > Class Diagram**.



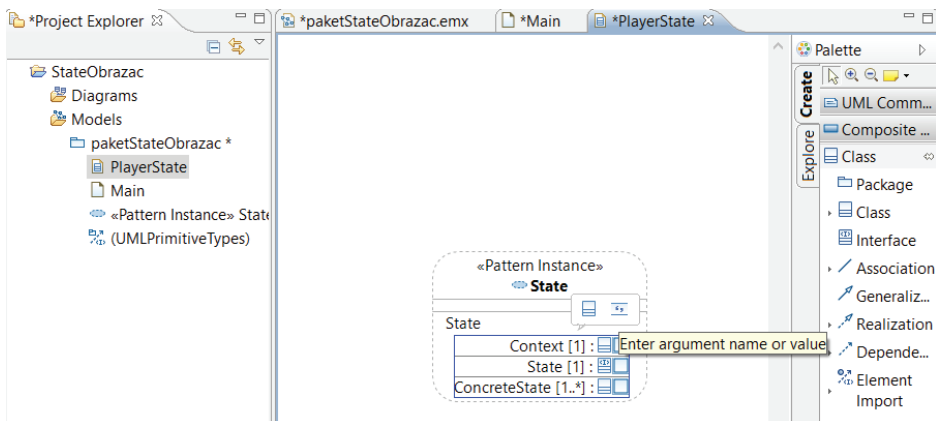
- Imenujte dijagram klasa kao npr. *PlayerState*



- Sa prozora *Pattern Explorer* (krajnji desni ekran) prevucite *State pattern* na radnu površinu:

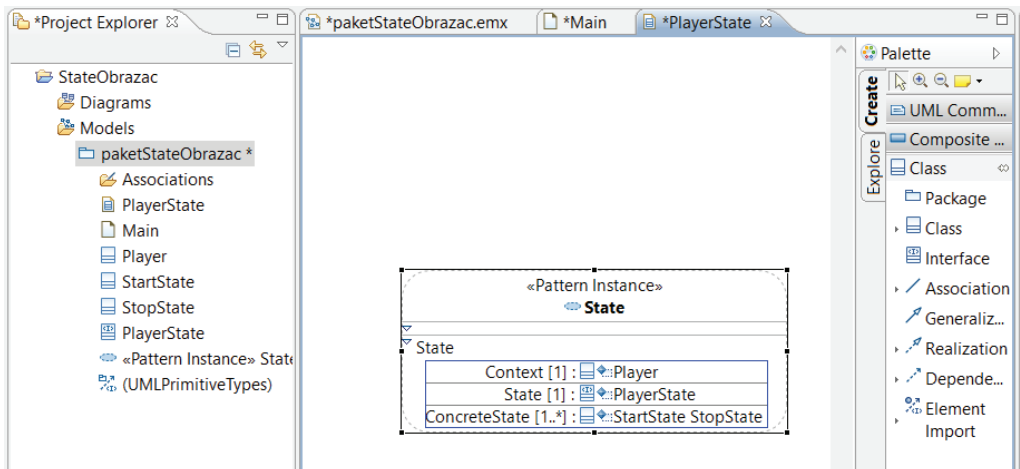


- S obzirom da vam je ovo prvi dijagram i da nemate prethodno kreiranih klasa, nove klase ćete dodati tako što ćete u okviru *State* obrasca, pored parametra *Context*, kliknuti na prazan kvadrat kako bi vam se pojavile prečice za unos naziva klasa. Sa prečice selektujete drugu ikonicu za unos naziva ili vrednosti argumenta (*Enter argument name or value*):

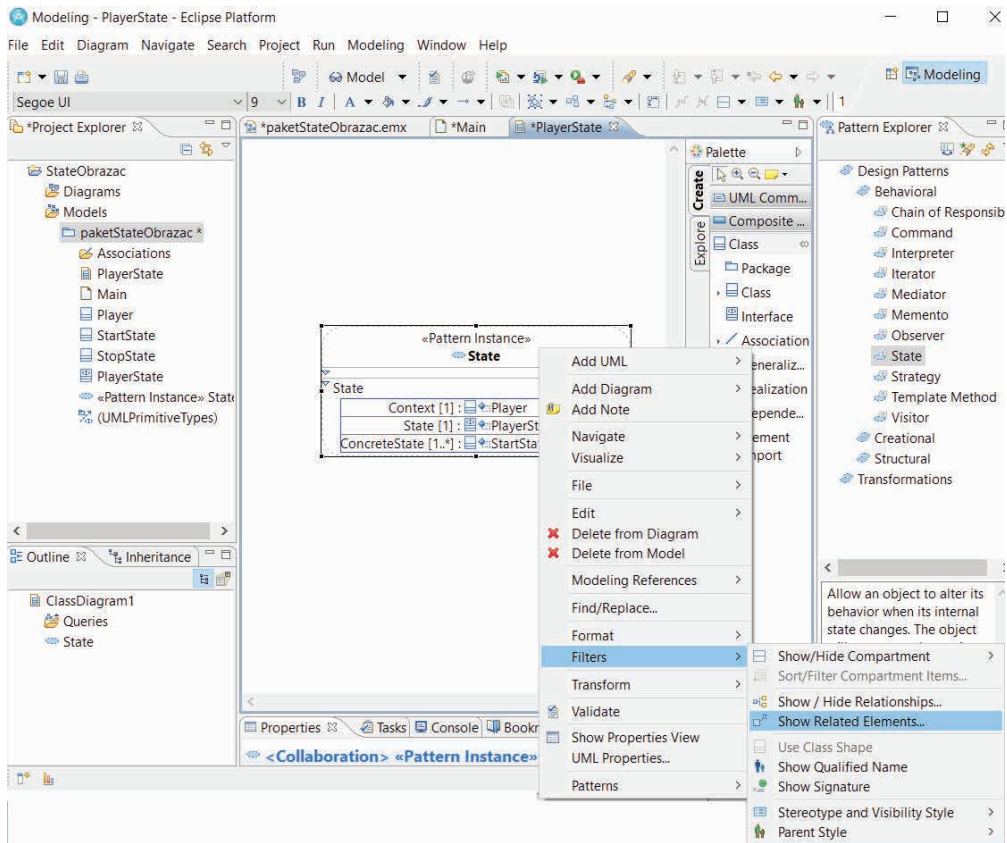


Ukoliko imate prethodno kreiranih klasa, onda ih samo prevučete sa *Project Explorer* prozora, ka prečicama za unos vrednosti parametara obrasca.

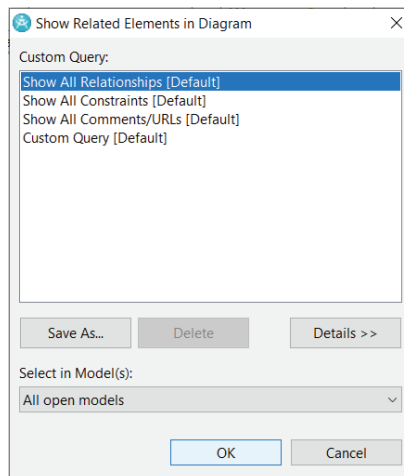
- ♦ Unesite nazive klasa i za ostale parametre:
 - *Context* klasa: **Player**
 - *State*: **PlayerState**
 - *ConcreteState*: **StartState**, zatim ponovo kliknite na prečicu unosa i unesite **StopState**.



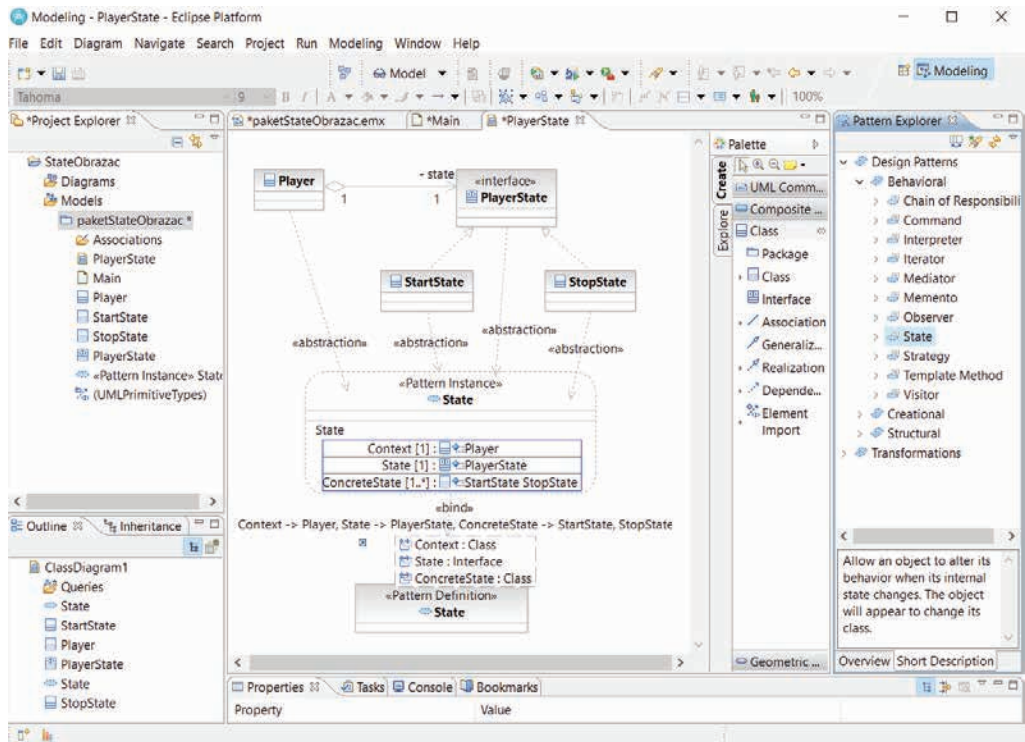
- ♦ Prikažite veze između klasa, kako bi imali kompletan dijagram klasa, tako što ćete kliknuti na *State* obrazac kako bi ga selektovali, zatim desnim klikom na njega selektujete **Filters > Show Related Elements**:



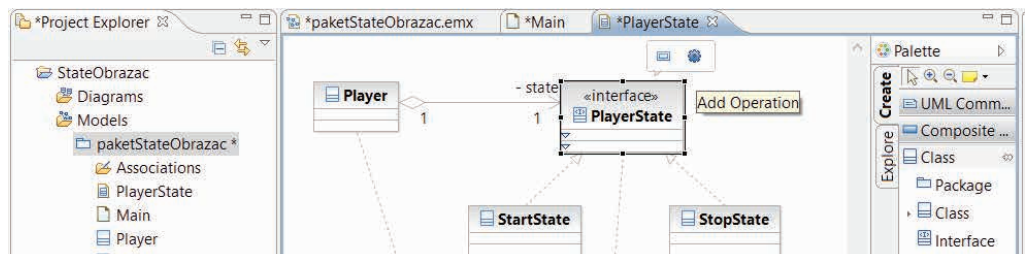
- ◆ Na novom prozoru selektujte **Show All Relationships (Default)** i kliknite na **OK**.



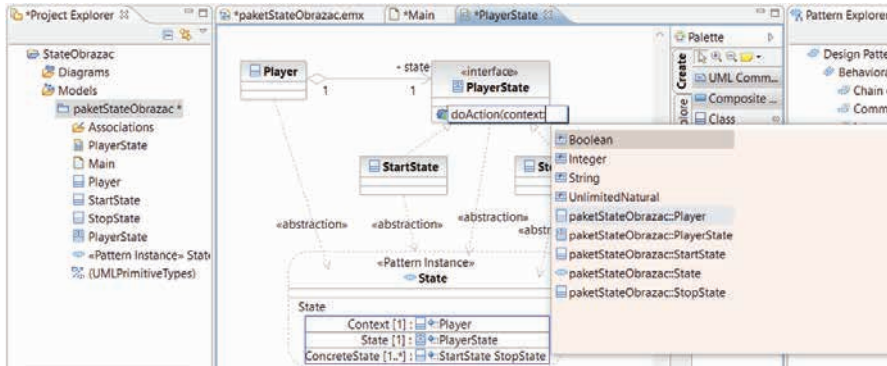
- ♦ Možete **preraspodeliti klase** kako bi vam bilo preglednije:



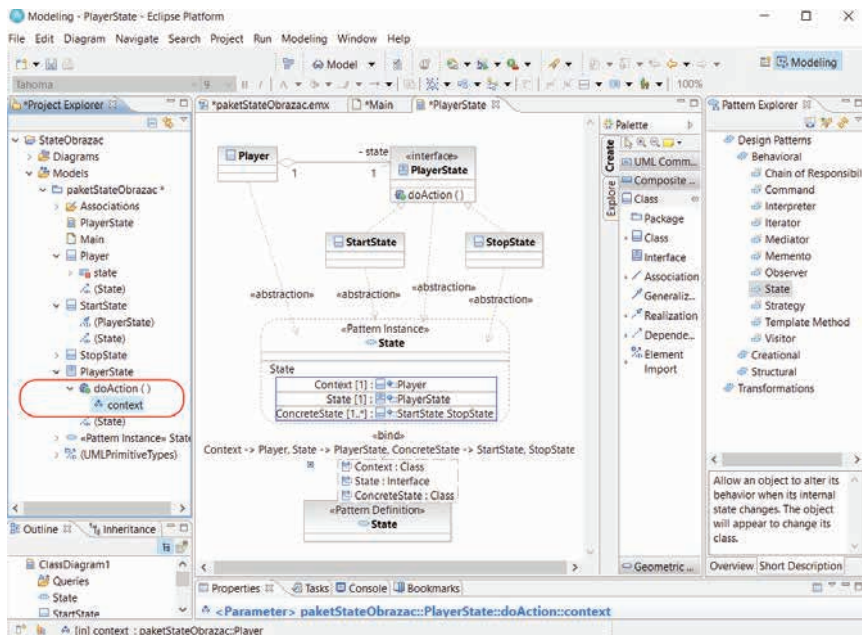
- ♦ Vaš dijagram klasa za sada nema ni atribute ni metode, pa ih je potrebno uneti. Predite mišem preko interfejs klase *PlayerState* kako bi vam se pojavile prečice za unos atributa i metoda. Kliknite na drugu opciju (zupčanik) radi unosa metode (*Add Operations*):



- Unesite metod *doAction*, koja ima jedan parametar *context* tipa klase *Player*, tj. ***doAction(context:Player)***. Kada unesete dvotačku iza parametra pojaviće vam se dostupni tipovi podataka, te selektujte *Player* iz paketa *paketStateObrazac*:



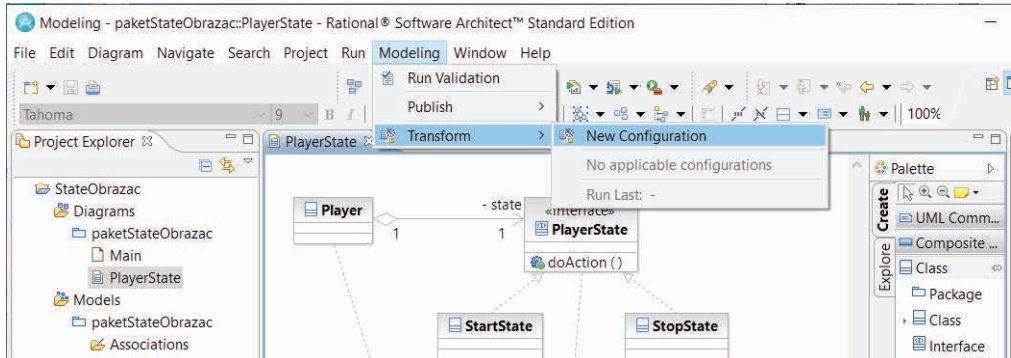
- Kada završite sa unosom metode, njeni parametri neće biti vidljivi na dijagramu klasa, ali se vide kada u okviru *Project Explorer*-a, proširite interfejs *PlayerState*, a zatim i metodu *doAction*:



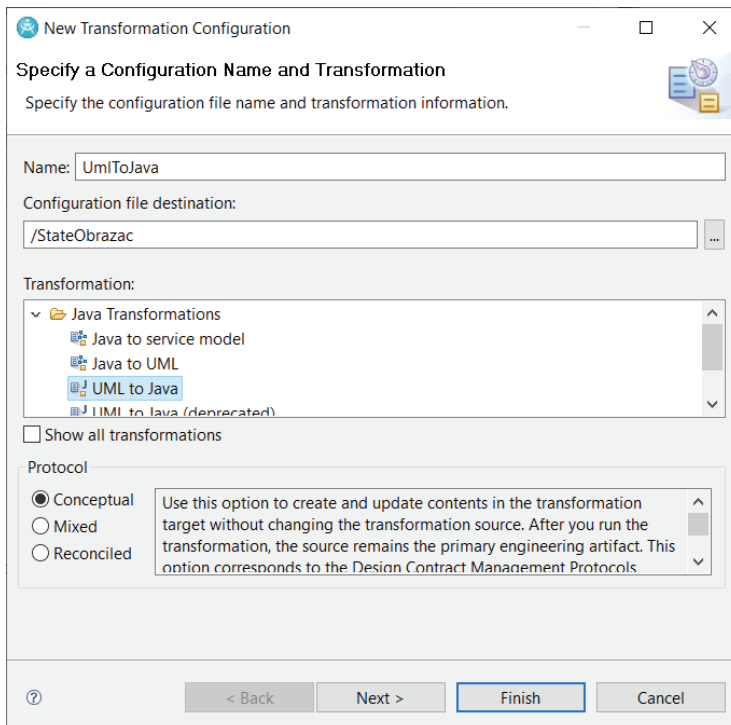
- Dijagram klasa je kreiran i sada možete generisati programski kod.

Korak 3. Generisati kôd u Javi i prilagoditi ga zadatku.

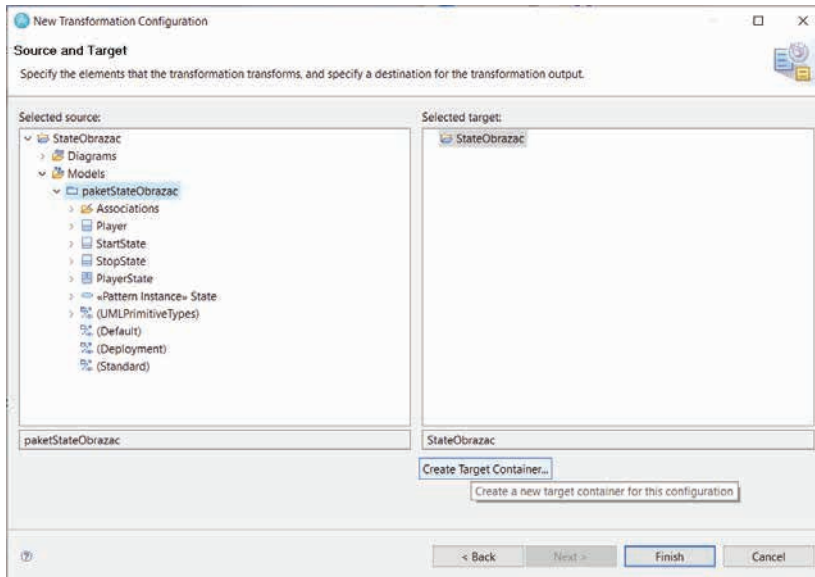
- ♦ Da bi generisali programski kod na osnovu dijagrama klasa, kliknite u meniju na **Modeling > Transform > New Configuration**:



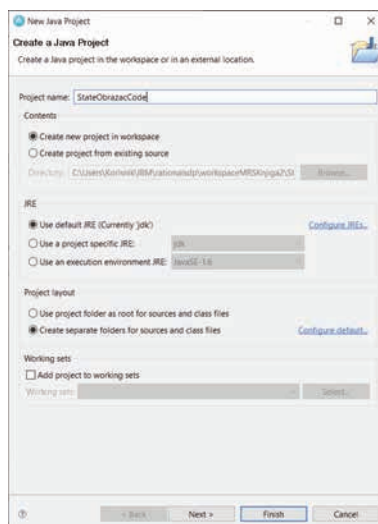
- ♦ Unesite naziv ove konfiguracije, na primer **Name: UmItoJava**. Pod **Transformation** selektujte **UML to Java** i kliknite na **Finish**.



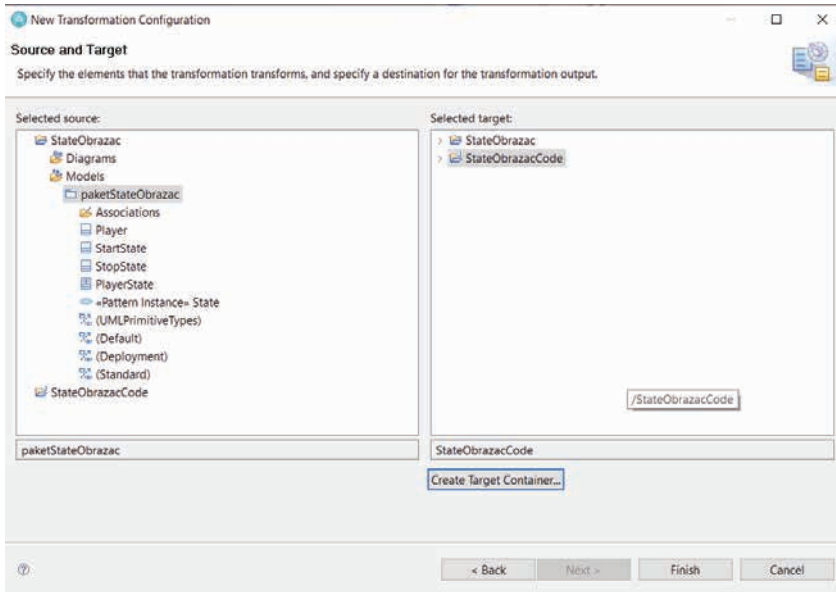
- ♦ Sa leve strane prozora **Selected source**, selektovaćete sve klase iz paketa **paket-StateObrazac**. U slučaju da želite samo za određene klase da generišete kod, onda ćete selektovati samo te klase, a ne ceo paket. Zatim, sa desne strane (ispod prozora **Selected target**) kliknite na komandu **Create Target Container**, kako bi ste kreirali target Java projekat gde će se nalaziti programski kod.



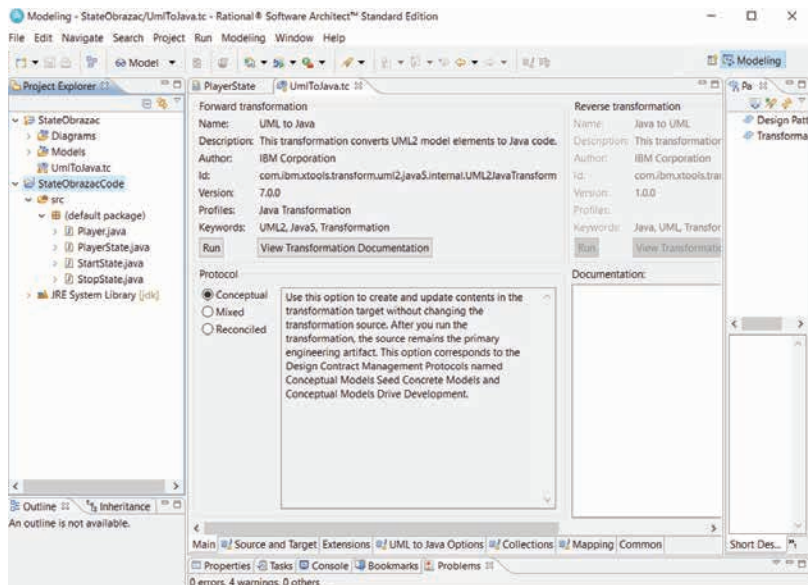
- ♦ Na novom prozoru samo unesite naziv Java projekta, npr. **Project name: State-ObrazacCode** i kliknite na **Finish**:



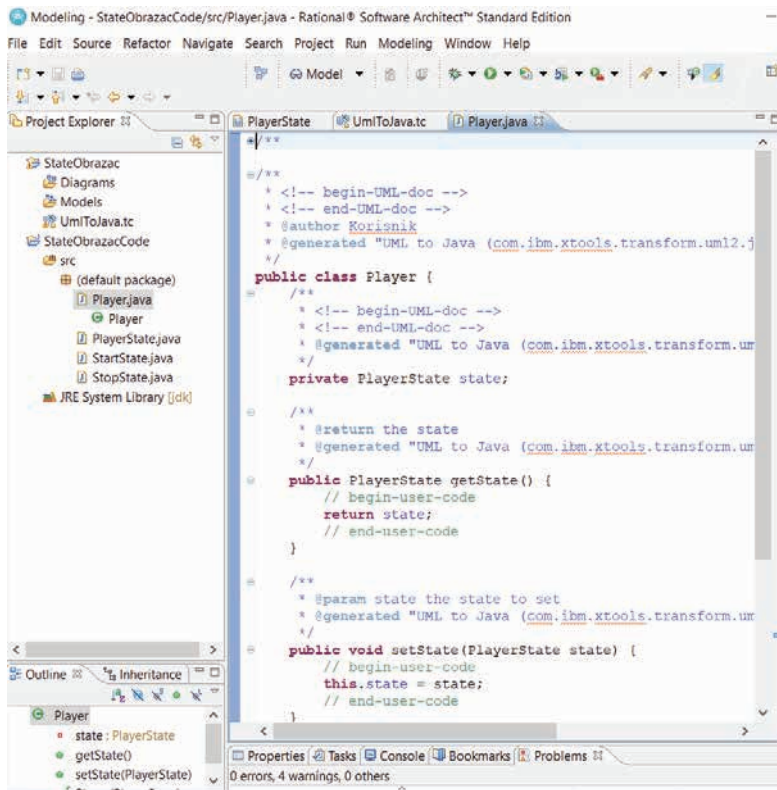
- ♦ Proverite još jednom da li su vam sa leve strane prozora *Selected source* selektovane klase za koje želite da generišete kod, a sa desne strane prozora *Selected target*, da li je selektovan budući projekat gde će biti generisani kod. Ukoliko je sve u redu, kliknite na **Finish**.



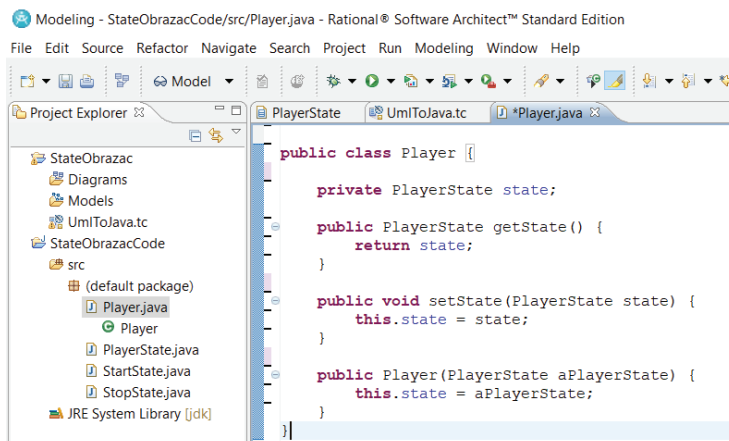
- ♦ Na novom prozoru kliknite na komandu **Run**:



- ♦ Pronađite novi projekat sa leve trane ekrana, u okviru prozora *Project Explorer*. Proširite klase i pogledajte šta je sve izgenerisano po difoltu za *State* obrazac:



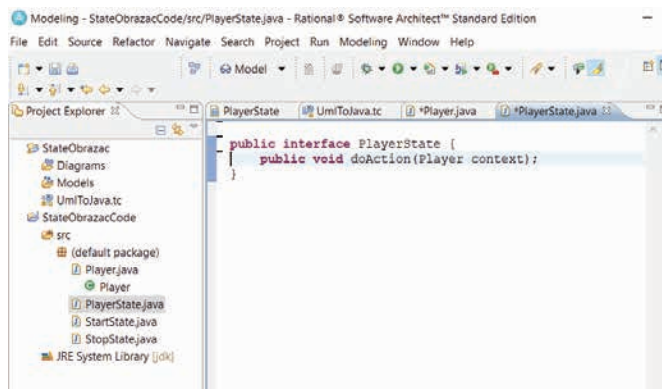
- ♦ Možete obrisati komentare, kako bi videli samo kod:



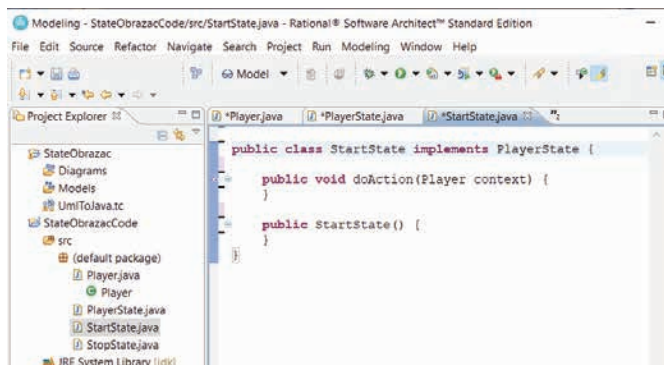
- ♦ Vidimo da za klasu *Player.java* nije izgenerisan neparametarski konstruktor, pa ga možete dodati:

```
public class Player {  
  
    private PlayerState state;  
  
    public PlayerState getState() {  
        return state;  
    }  
  
    public void setState(PlayerState state) {  
        this.state = state;  
    }  
  
    public Player(PlayerState aPlayerState) {  
        this.state = aPlayerState;  
    }  
  
    public Player() {  
    }  
}
```

- ♦ Izgenerisani interfejs *PlayerState.java*:



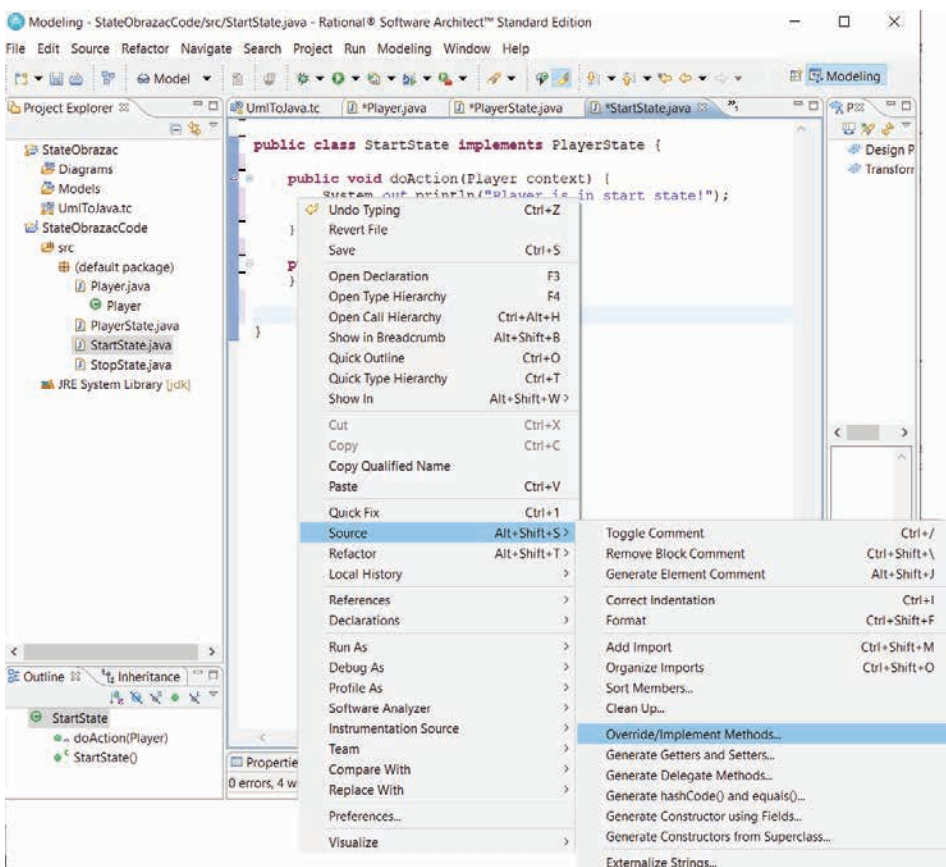
- ♦ Izgenerisana klasa *StartState.java*:



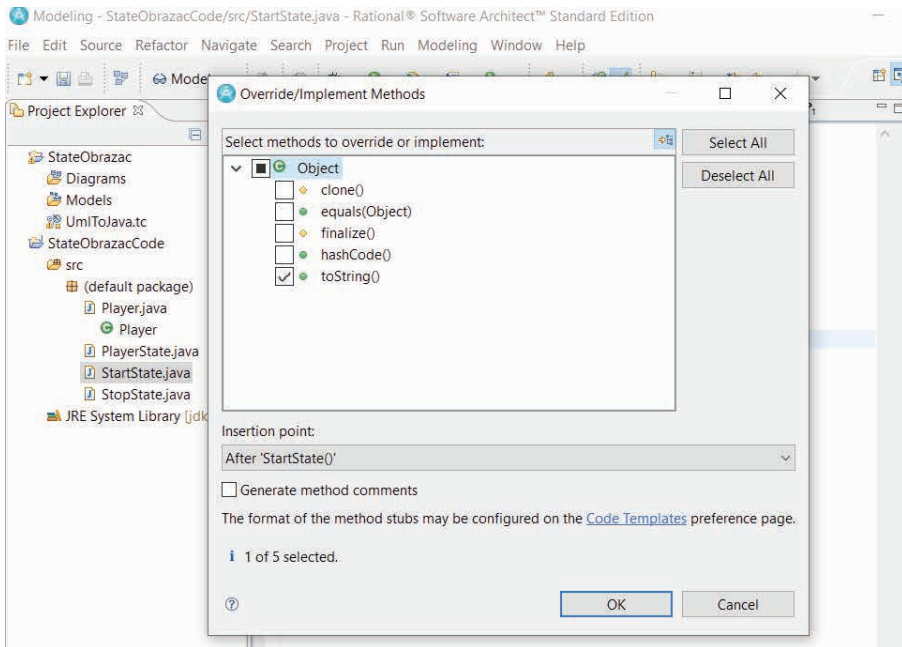
- S obzirom da su metode klase *StartState.java* prazne, potrebno ih je dopuniti. Metoda *doAction()* treba samo da ispiše „*Player is in start state!*“ i da postavi ovo novo stanje:

```
public void doAction(Player context) {
    System.out.println("Player is in start state!");
    context.setState(this);
}
```

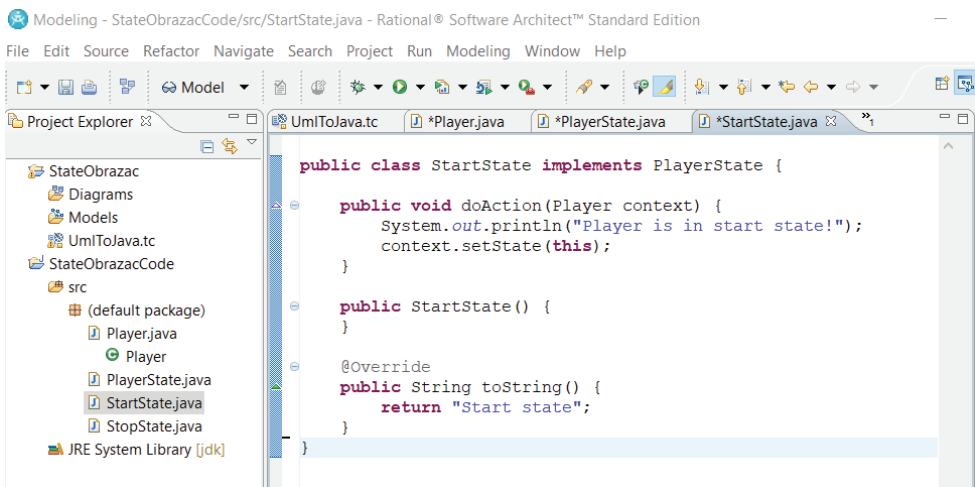
- Zatim ćete implementirati *toString()* metodu. Automatsko dodavanje metode je:
 - Desni klik na przanu površinu, gde želite da dodate metod, selektujte **Source** > **Override/Implement Methods ...**



- Sa novog prozora, čekirajte *toString()* i kliknite na **OK**.
- Umesto difoltnog ispisivanja metode *toString()*, unesite *return "Start state"*:



- ♦ Sačuvajte izmene, klikom u meniju na **File > Save All**.
- ♦ Isti postupak će te primeniti i na **StopState().java**, s tim što će se ispisati „*Player is in stop state*“, a **toString()** metoda ispisuje **Stop state**:

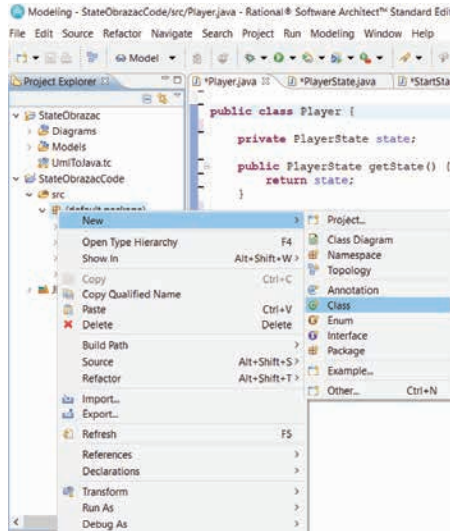


- ♦ S obzirom da ste prilagodili izgenerisani kod, sada možete kreirati glavnu klasu i izvršiti kod.

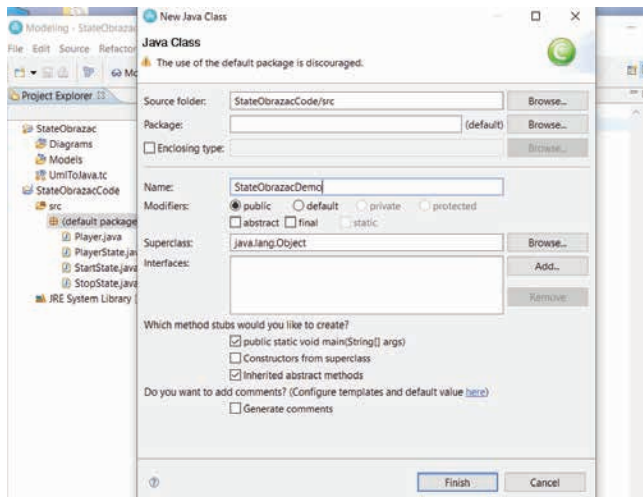
Korak 4. Kreirati glavnu klasu i izvršiti kod.

◆ Kreiranje glavne klase:

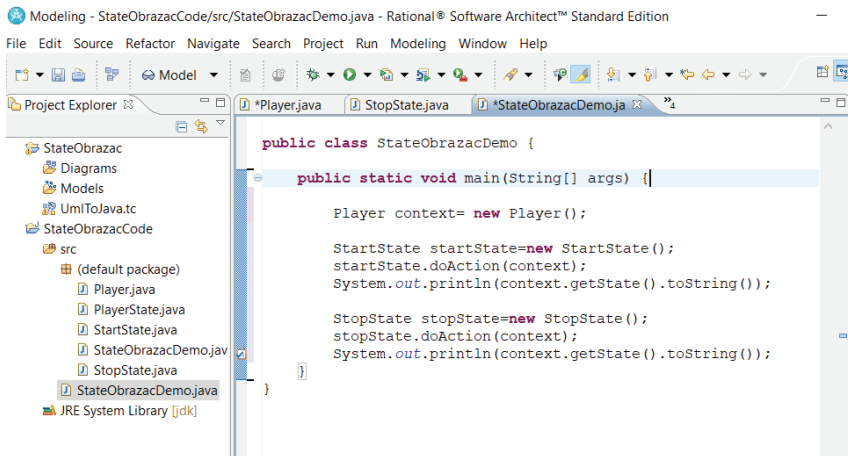
- Desni klik na paket ovog Java projekta (na prozoru *Project Explorer*), selektovati **New > Class**:



- Na novom prozoru unesite naziv klase, npr. **Name: StateObrazacDemo** i čekirajte **public static void main(String[] args)** i kliknite na **Finish**.

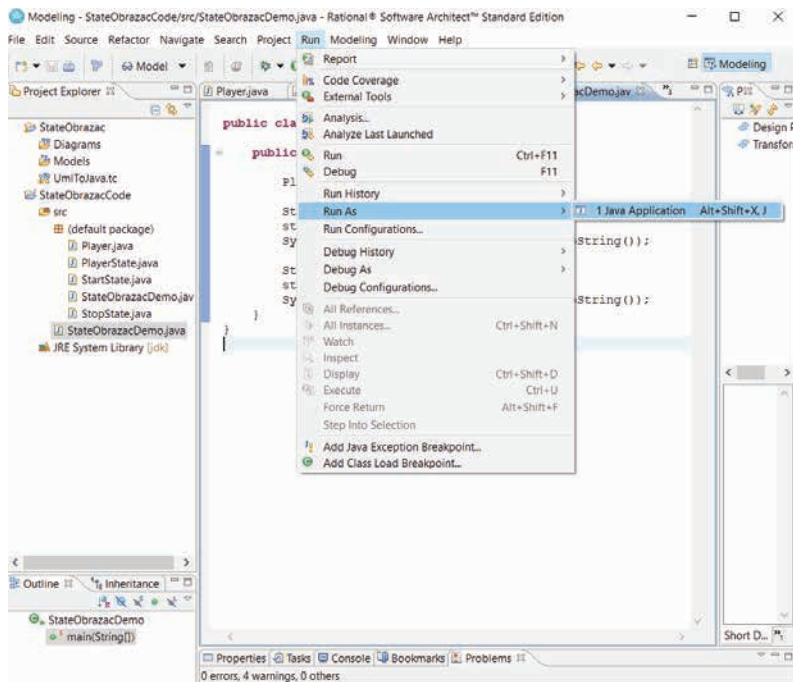


- Istancirajte klasu **Player** i njegova oba stanja, a zatim pozovite **doAction()** i **toString()** metode, kao što je prikazano na slici dole:

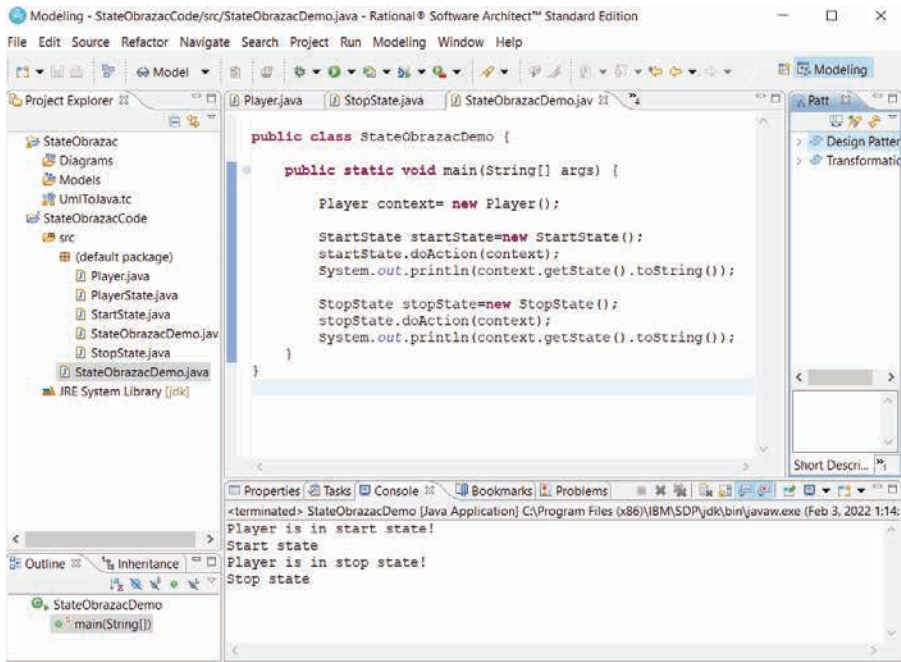


♦ Izvršite kod:

- Kôd možete izvršiti na nekoliko načina. Na narednim slikama je prikazan samo jedan od. U meniju kliknite na **Run**, zatim na **Run As > Java Application**:



- Rezultat će biti ispisan na konzoli (*Console*):



Pokušajte da primenite *State* obrazac na sledećem zadatku i proverite rešenje na kraju knjige:

Zadatak 2-1. Primeniti *State* obrazac nad klasom *MobileAlert*.

Mobilni telefon može biti u dva stanja: stanje vibracije (*Vibration*) i stanje tišine (*Silent*). Kada je telefon u stanju vibracije, samo ispisati "*Vibration mode*", a kada je u stanju tišine ispisati "*Silent mode*". Prikazati dijagram klasa sa osnovnim elementima *State* obrasca i relacijama između njih, izgenerisati kod, dopuniti ga i izvršiti.

2.2.2. Postupak uključivanja obrasca „*Singleton*“ u projekat

Najjednostavniji obrazac koji se često kombinuje sa drugim obrascima je *Singleton*. Single ukazuje na jedan objekat – jednom objektu se može globalno pristupiti od strane više klijenata. *Singleton* zabranjuje kreiranje više objekata unutar jedne klase, odnosno osigurava da se instanca klase kreira samo jedanput i da ta klasa omogućava pristup samo tom jednom objektu. Kada god pokušamo da kreiramo novi objekat iste klase, biće nam vraćena originalna instanca.

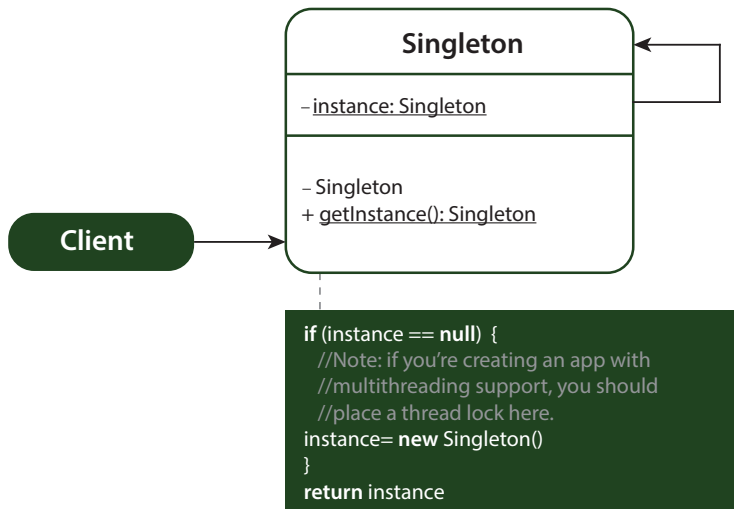
Situacija gde bi se *Singleton* mogao upotrebiti je ukoliko se klasa koristi za održavanje brojača, tj. brojanja poseta npr. nekom sajtu ili usled postojanja samo jednog primerka klase, npr. jednog konfiguracionog objekta ili jednog deljenog resursa (npr. veze sa bazom) kome pristupaju i ažuriraju mnogi delovi sistema. Svaki objekat koji zahteva klasu koja implementira *Singleton* obrazac biće joj data referenca na isti objekat. *Singleton* spada u kategoriju kreacionih obrazaca (Tabela 4).

Tabela 4. Opis osnovnih elemenata *Singleton* obrasca.

Elementi	Opis elemenata obrasca <i>Singleton</i>
Problem	Koristi se za rešavanje problema kreiranja samo jedne instance određene klase. Osnovna ideja je da klasa ima samo jedan jedinstveni objekat koji se može deliti i koristiti širom aplikacije. <i>Singleton</i> se često koristi u situacijama kada je potrebno da postoji samo jedna instanca klase koja kontroliše pristup zajedničkim resursima, kao što su baza podataka, sistemski fajlovi ili mrežni resursi. Na taj način se osigurava da se resursi pravilno upravljaju i da se sprečava višestruko kreiranje istog objekta.
Resursi	<i>Singleton klasa</i>: klijentima obezbeđuje pristup jedinstvenoj instanci.
Pravila	- Difoltni konstruktor <i>Singleton()</i> mora biti <i>private</i> kako bi se sprečilo da klijenti kreiraju objekte klase korišćenjem <i>new</i> operatora. - Metod <i>getInstance()</i> mora biti <i>static</i>, kako bi ga pozivali klijenti bez potrebe za instanciranjem klase. Ovaj metod poziva privatni konstruktor kako bi kreirao objekat. Svi naredni pozivi ovoj metodi vraćaju keširani objekat.
Smernice	Ovaj obrazac treba koristiti kada mora postojati tačno jedan objekat klase, koji je pristupačan klijentima preko svima poznate tačke pristupa.

Osnovna struktura *Singleton* obrasca je data na narednoj slici 5.

Slika 5. Osnovna struktura *Singleton* obrasca (Shvets, 2019).



Struktura *Singleton* klase su:

- **instance**: Privatno statičko polje koje služi za čuvanje instance *Singleton* klase.
- **Singleton()**: Difoltni konstruktor je privatni kako bi se sprečilo da drugi objekti koriste konstruktor za instanciranje upotrebom *new* operatora.
- **Public static** metod **getInstance()**: s obzirom da je difoltni konstruktor *private*, njemu se pristupa putem ove statičke javne metode. Ovaj metod poziva privatni konstruktor za kreiranje jednog objekta i čuva ga unutar statičkog polja *instance*. Svi naredni pozivi ovoj metodi, vratiće keširan objekat.

Postoje razni načini implementacije *Singleton* obrasca. Ovde će se predstaviti samo dva: *eager* i *lazy* inicijalizacija.

- ♦ Kod **eager** inicijalizacije *Singleton* objekat se inicijalizuje u vreme kreiranja klase. Ovo je najlakši način za kreiranje *Singleton* klase, međutim nedostatak je taj što se istanca kreira iako je klijentska aplikacija možda i ne koristi. Takođe, ukoliko se inicijalizuju veći objekti, to može povećati vreme pokretanja.

```

public final class Singleton {
    private static final Singleton INSTANCE = new Singleton();
    private Singleton() {
    }
    public static Singleton getInstance() {
        return INSTANCE;
    }
}
    
```

- ♦ Kod **lazy inicijalizacije** klasa će kreirati Singleton objekat samo kada druga klasa to zahteva. Nakon toga, isti objekat će biti vraćen za buduće zahteve. U ovom slučaju, se ne inicijalizuje INSTANCE polje unapred. Umesto toga, mi ga inicijalizujemo samo pri prvom pozivu metode *getInstance()*. Postoji nekoliko nedostataka kada koristimo ovaj pristup u višenitnom okruženju.

```
public final class LazySingleton {
    private static LazySingleton INSTANCE;
    protected LazySingleton() {
        //Called only once by the getInstance() method.
    }
    public static LazySingleton getInstance() {
        if (INSTANCE == null) {
            INSTANCE = new LazySingleton();
        }
        return INSTANCE;
    }
}
```

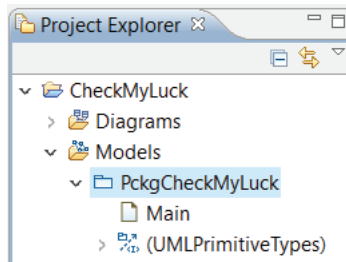
Vežba 2-3. Primeniti *Singleton* obrazac nad klasom *CheckMyLuck* kroz korake razvoja projekta od specifikacije zahteva do izvršavanja koda.

Kreirati jednostavnu aplikaciju za igre na sreću. Aplikacija, nazvana *CheckMyLuck*, koja će na osnovu akcije korisnika (klika na tastaturi) izgenerisati slučajni broj. Ukoliko je taj broj manji od 0,5 prikazaće se poruka „Winner“, a u suprotnom „Looser“. Modelovati aplikaciju koristeći dvoslojnu arhitekturu, primeniti *Singleton* obrazac, generisati kod u Javi i prikazati rezultate rada aplikacije.

Korak 1. Kreirati projekat.

Koraci izrade ove vežbe u IBM RSA alatu su:

- ♦ *File > New > Project* (ili *Other*)
- ♦ Proširiti *Modeling > UML Project > Next*
- ♦ Imenovati projekat: *Project name: CheckMyLuck > Next*
- ♦ Imenovati paket: *File name: PckgCheckMyLuck > Finish*

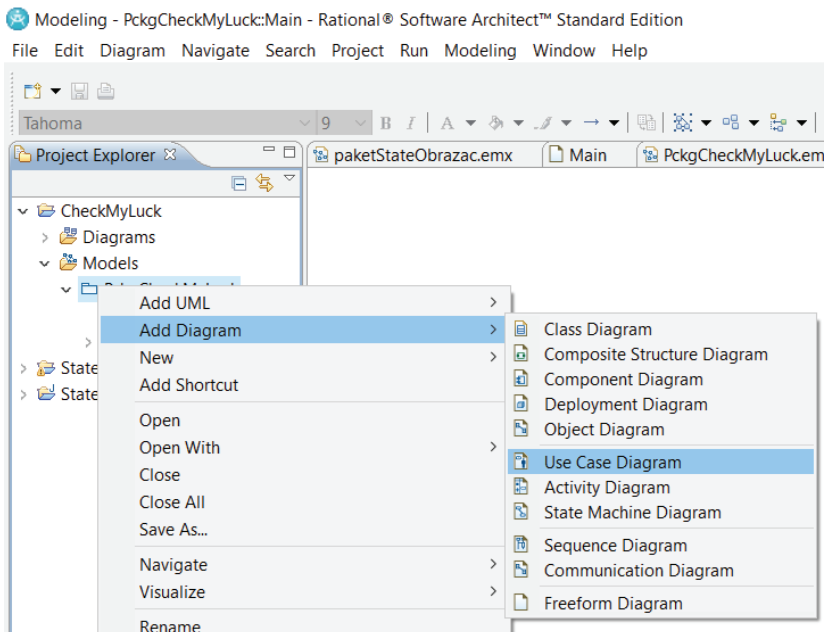


Korak 2. Kreirati dijagram slučaja upotrebe (Use case diagram).

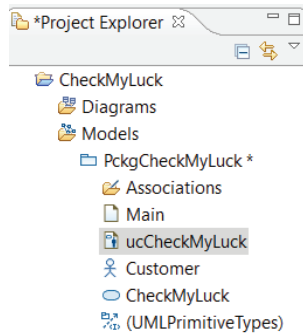
Radi simulacije jednog kompletnog projekta od ideje, preko modela, dizajna, kodiranja do izvršavanja, kreiraćemo nekoliko jednostavnih UML dijagrama. U fazi dogovora sa korisnicima, najlakše je vizuelno predstaviti ključne module ili funkcionalnosti aplikacije koristeći dijagram slučaja upotrebe (*use case diagram*). S obzirom na jednostavnost ove vežbe, naš use case dijagram će imati samo jedan slučaj upotrebe (*use case*) koji ćemo nazvati, kao i sam projekat, *CheckMyLuck* sa kojim će korisnik biti u dvosmernoj komunikaciji (*association*).

Koraci kreiranja *use case* dijagrama su:

- ◆ **Desni klik na paket „PkgCheckMyLuck“, klik na *Add Diagram > Use Case Diagram***

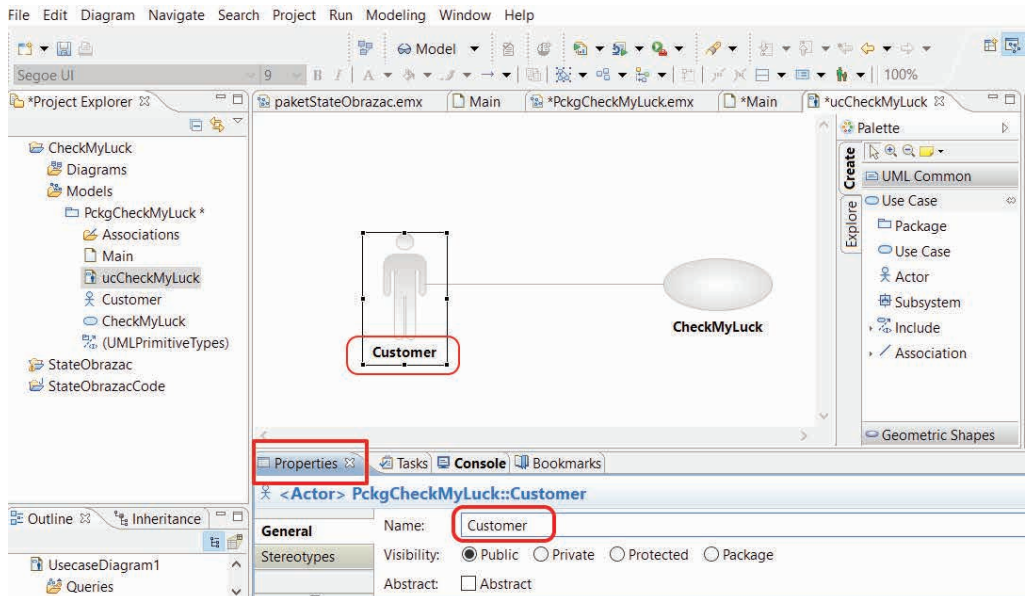


- ♦ Imenujte *use case* dijagram kao npr. **ucCheckMyLuck** (uc u nazivu dijagrama je od *use case*).



- ♦ Sa palete (*Palette*), koja se nalazi sa desne strane, prebacite **jednog aktera (Actor)**, **jedan use case** i povežite ih **relacijom asocijacije (Association)**.
- ♦ Imenujte elemente, dvoklikom na njihove difoltne nazive ili u okviru *Properties* prozora koji se nalazi ispod dijagrama (na dnu ekrana), kao što je prikazano na donjem dijagramu.

Dijagram 1. *Use case* dijagram za aplikaciju igre na sreću "CheckMyLuck".



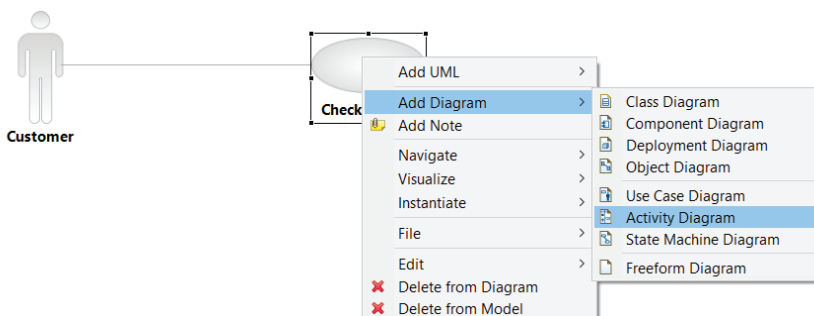
Napomene: Najčešće zabune i greške kod kreiranja use case dijagrama

- ♦ Postoji razlika između use case dijagrama i običnog use case-a:
 - Use case dijagram sadrži elemente kao što su akter, use case, relacije i dr.
 - Jedan use case podrazumeva slučaj upotrebe koji predstavlja jednu funkcionalnost aplikacije, koja obično sadrži više od dve aktivnosti unutar nje. Aktivnosti use case-a će biti opisane na dijagramu aktivnosti
- ♦ Use case nije aktivnost! Jedan use case sadrži više aktivnosti unutar sebe. Zamislite aktivnost kao jedan klik na aplikaciji. Pogrešno je staviti npr. use case "Dodavanje studenta", drugi use case "Brisanje studenta" i sl. Ovo je obično samo jedan use case koji možete nazvati "Održavanje informacija o studentima", dok su aktivnosti unutar tog use case-a brisanje, dodavanje, izmena i sl. Te aktivnosti će biti vidljive tek na dijagramu aktivnosti. Međutim, ukoliko use case "Brisanje studenta" zahteva više aktivnosti korisnika, tj. zamislite da vam se otvara novi prozor gde će korisnik unositi ili selektovati neke podatke, izvršavati neke komande i dr. u tom slučaju "Brisanje studenta" može biti use case jer u sebi sadrži više aktivnosti kako bi se realizovao.
- ♦ Naziv use case-a mora biti glagol! S obzirom da use case predstavlja neku funkcionalnost aplikacije, njen naziv treba to i da reflektuje. Ukoliko use case imenujete sa imenicom, npr. "Student", iz ovog naziva se ne vidi tačno šta taj use case treba da radi, s toga je neophodno upotrebiti glagol, npr. "Održavanje informacija o studentima". Namena use case-a treba da bude jasna svim članovima razvojnog tima.

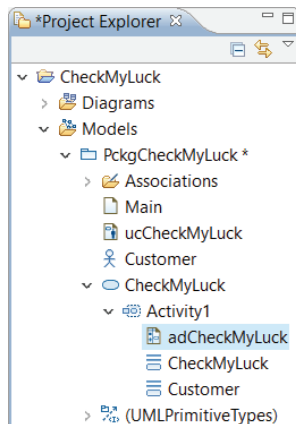
Korak 3. Kreirati dijagram aktivnosti (Activity diagram).

Da bi sagledali algoritam rada use case-a *CheckMyLuck*, kreiraćemo dijagram aktivnosti za taj use case:

- ♦ **Desni klik na use case *CheckMyLuck* > Add Diagram > Activity Diagram**

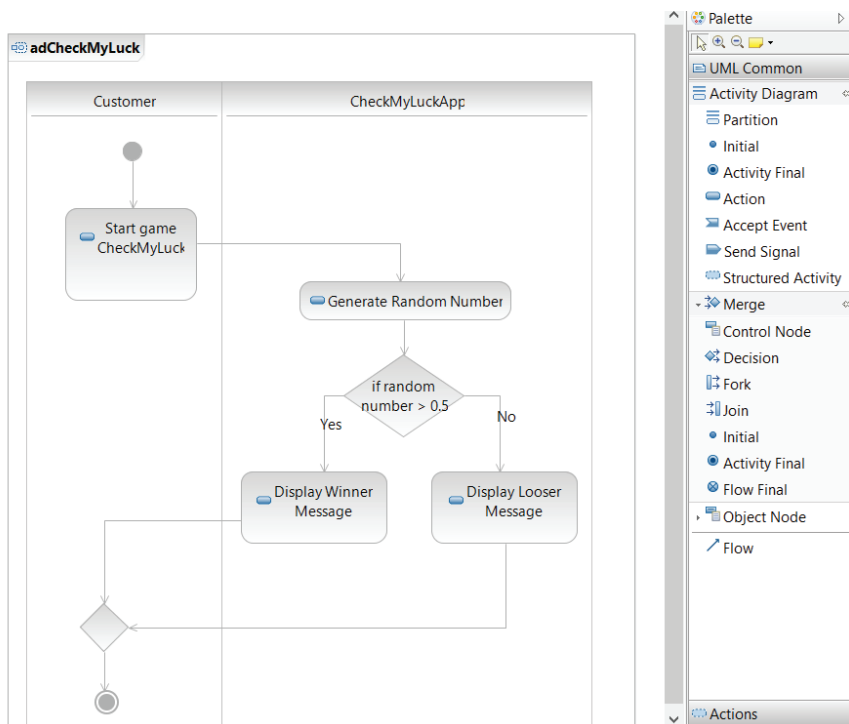


- ♦ Imenovati dijagram kao **adCheckMyLuck** (*ad* je od *activity diagram*)



- ♦ Prevući dve particije (*Partition*) i pomoću elemenata *Initial*, *Action*, *Flow*, *Decision*, *Merge* i *Activity Final*, kreirati dijagram aktivnosti kao što je prikazano na sledećem dijagramu:

Dijagram 2. Dijagram aktivnosti za use case „CheckMyLuck“



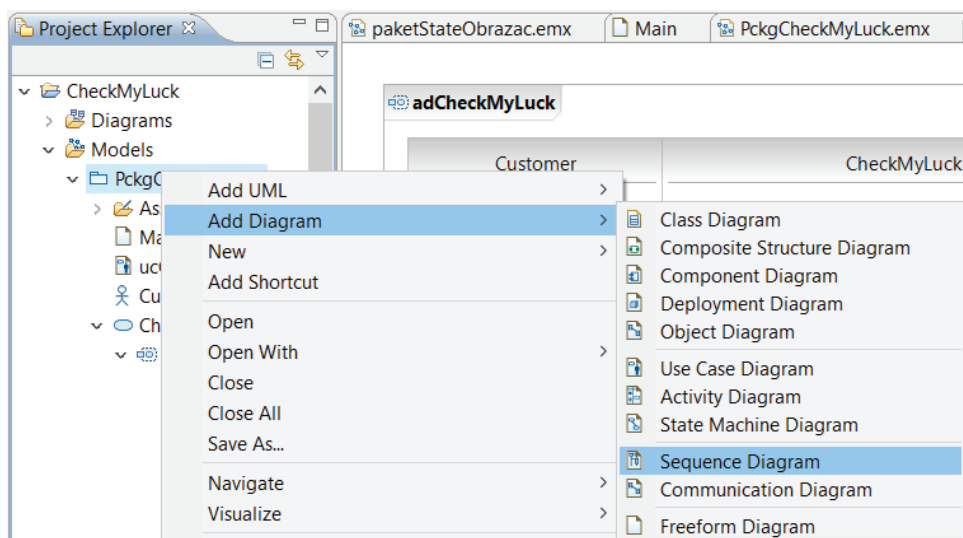
Korak 4. Kreirati dijagram sekvenci (*Sequence diagram*).

Prethodni dijagrami se uglavnom kreiraju u dogovoru sa korisnikom za koga se razvija aplikacija, kako bi korisnik mogao da sugerise korake rada aplikacije. Nakon dogovora sa korisnikom, **prelazi se u fazu analize softvera gde se sa razvojnim timom dogovara o arhitekturi softvera i uglavnom kreiraju dijagrami sekvenci i dijagram klasa, koji vizuelizuju sam kôd aplikacije.**

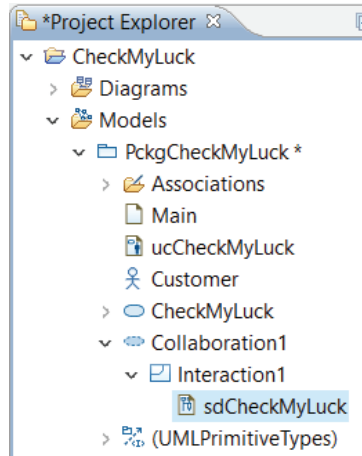
U zavisnosti od vrste aplikacije koju razvijate, zavisice i arhitektura softvera. Na primer, za razvoj igrica (*game developmet*) uglavnom se koristi *Entity Component System* arhitektura, za interaktivne veb aplikacije, uglavnom se koristi standardna *Model View Controller* (MVC) arhitektura itd. S obzirom na jednostavnost ove aplikacije, koristiće se dvoslojna arhitektura (*two-tier architecture*), gde će poslovna logika biti na klijentskom delu aplikacije. Međutim, da bi uočili razliku u kreiranju dijagrama sekvenci, simuliraćemo i MVC arhitekturu ove aplikacije.

Koraci kreiranja dijagrama sekvenci su:

- ♦ Desni klik na paket „*PckgCheckMyLuck*“, klik na *Add Diagram > Sequence Diagram*:

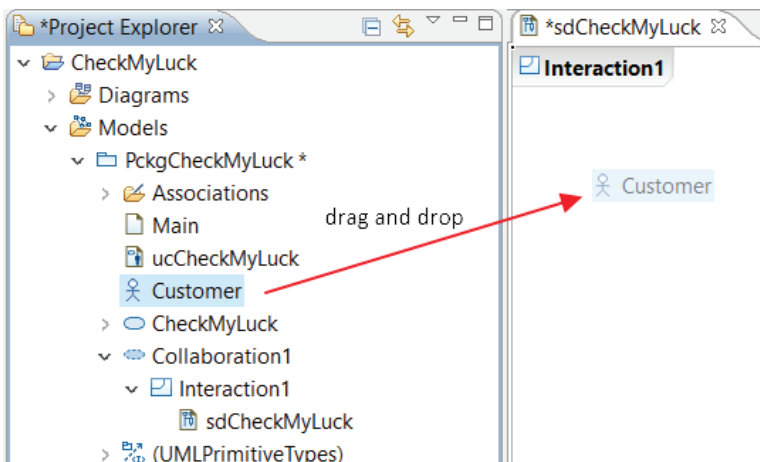


- ♦ Možete imenovati dijagram, npr: *sdCheckMyLuck*:

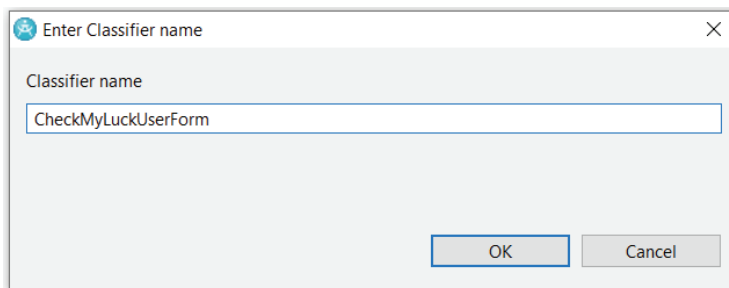
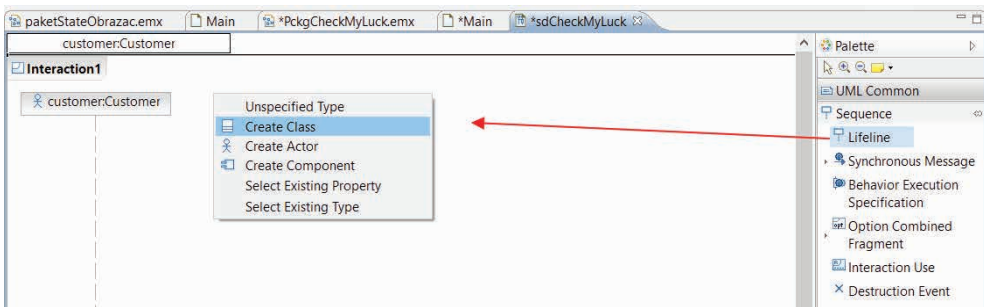


Primer primene *Model-View-Controller* arhitekture:

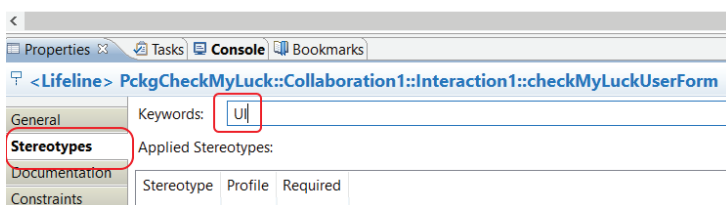
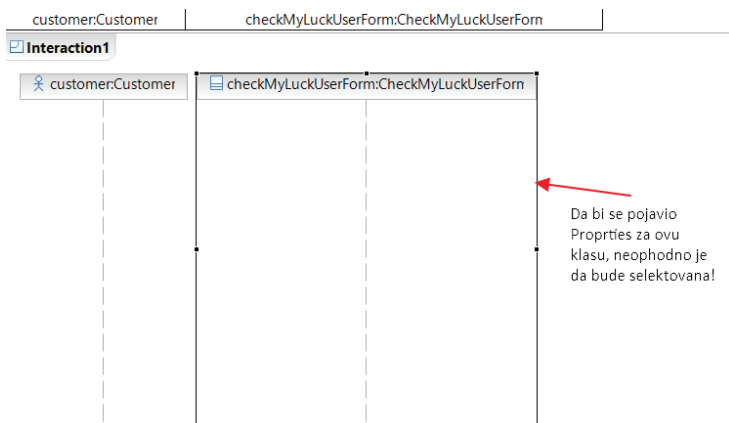
- ♦ Ukoliko ste se **opredelili za MVC arhitekturu** softvera, dijagram sekvenci će se sastojati od tri vrsta klasa: **klase tipa korisnički interfejs** (*User Interface, UI*) - *View*, **klase tipa kontroler** - *Controller* i **klase tipa tabele** (*Table*) - *Model*.
- ♦ Počecemo kreiranje dijagrama sekvenci sa akterom *Customer* i zamislimo da je to stvarno osoba koja preko interfejsa komunicira sa aplikacijom koja je instalirana na aparatu za igranje. S obzirom da smo aktera već kreirali na *use case* dijagramu, dovoljno je da ga samo prevučemo na radnu površinu:



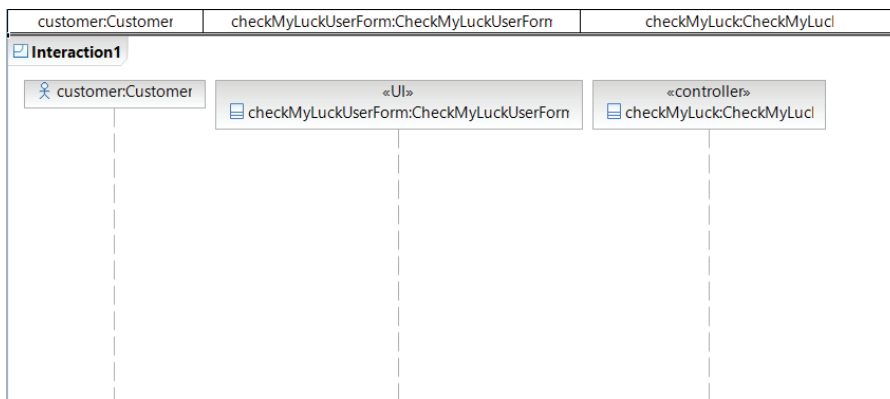
- Gledajući *use case* dijagram, akter *Customer* je prvi *lifeline*, relacija asocijacije je klasa tipa interfejs preko koga on komunicira sa aplikacijom i *use case* je klasa tipa *controller* koja upravlja celokupnim sistemom i na kojoj se nalazi poslovna logika aplikacije. Stoga, **prvi lifeline** je **Customer** tipa akter, **drugi lifeline** je klasa koju ćemo imenovati kao **CheckMyLuckUserForm** (korisnička forma) i **treći lifeline** je klasa tipa *controller* koju ćemo imenovati kao **CheckMyLuckApp** (aplikacija ili sistem). Svaki novi *lifeline* dodajemo tako što sa palete kliknemo na *Lifeline* i prevučemo na dijagram, kliknemo na *Create Class*, imenujemo i dodamo stereotip po potrebi.



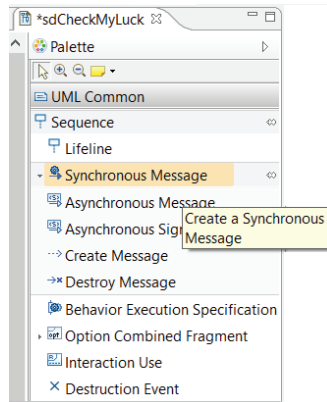
- Stereotipi (*Stereotypes*) se dodaju kako bi bliže objasnili namenu klase. Da bi dodali stereotip kod klase *CheckMyLuckUserForm*, selektujte taj *lifeline* i ukucajte *UI* (*User Interface*) pod *Stereotypes*:



- ♦ Finalni izgled dodatih *lifelines* je prikazan na donjoj slici. **Lifeline** predstavlja **životni vek objekta koji se instancira na određenoj klasi**. Prednost dijagrama sekvenci je taj što hronološki može da se prati interakcija između klasa.



- ♦ Kada ste završili sa dodavanjem *lifelines* i to posmatrajući *use case* dijagram, sada na osnovu dijagrama aktivnosti dodajete poruke interakcija između klasa. Prva aktivnost, na dijagramu aktivnosti, je *StartGame* koju započinje *Customer*. Aktivnost se na dijagramu sekvenci prikazuje kao poruka (*message*). Potrebno je selektovati odgovarajuću vrstu poruke i prevući je od korisnika (*Customer*) do korisničke forme (*CheckMyLuckUserForm*).
- ♦ Ovde se postavlja pitanje koja je to vrsta poruke, da li je sinhrona (mora da čeka da primaoc poruke odgovori), asinhrona (ne mora da se čeka da primaoc poruke obradi), *Create Message* (slična *new* operatoru), *Destroy Message* (sličan *delete* naredbi).

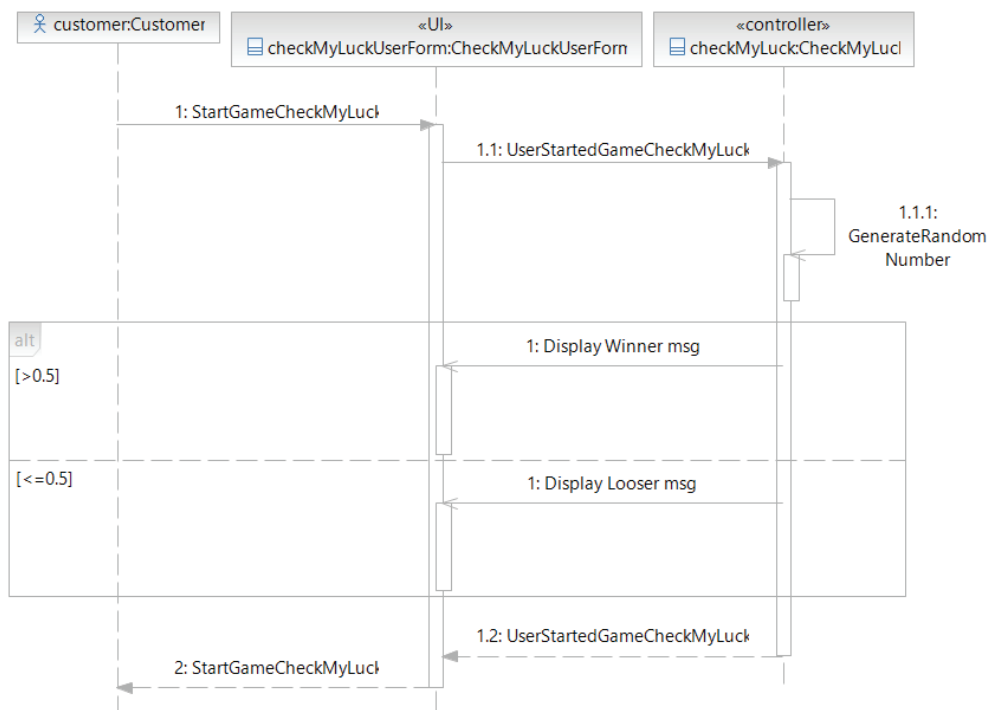


- ♦ S obzirom da korisnik mora da čeka da sistem obradi poruku (npr. generiše slučajni broj, prikaže poruku i dr.) selektovaćemo sinhronu poruku (***Synchronous Message***) koju ćemo imenovati, za početak, isto kao što ste imenovali tu aktivnost na dijagramu aktivnosti – ***StartGameCheckMyLuck*** (Dijagram 3). Sinhrona poruka će automatski prikazati i povratnu poruku, koja nema nikakvu ulogu, sem da nam naznači, kada korisnik može da započne slanje nove poruke.
- ♦ Ukoliko korisnička forma (obično pisana u HTMLu sa *Script* komandama) zna šta treba da uradi sa tom porukom, tj. ukoliko ste deo poslovne logike aplikacije prebacili na *front-end* deo aplikacije, onda se ta poruka neće prosleđivati *back-end* delu aplikacije već će, u ovom primeru, slučajni (*random*) broj generisati na korisničkoj formi. Međutim, ukoliko ste poslovnu logiku ipak implementirali na *back-end* delu onda će *front-end* odnosno klijentska strana aplikacije samo proslediti tu poruku pozadinskoj aplikaciji. **Dok je aktivna instanca (pravougaoni stubić na lifeline-u klase)**, odatle ćemo vući opet **sinhronu poruku ka kontroleru**. Poruku možete imenovati kao npr. ***UserStartedGameCheckMyLuck*** (Dijagram 3).
- ♦ Kontroler generiše slučajni broj, koji se prikazuje kao asinhrona poruka na *lifeline-u* klase *CheckMyLuckApp*. Takođe *if-then-else* deo se proverava na kontroleru, tako da ćete prevući Alternativni fragment (*Alternative combined fragment*) preko svih *lifeline*-ova, ali nakon što je generisan slučajni broj (Dijagram 3).

- ♦ Kao što je i rečeno u zadatku, a vidljivo sa algoritma prikazanog na dijagramu aktivnosti, ukoliko je broj veći od 0.5, vući asinhronu poruku od kontrolera do korisničke forme i imenovati „*Display Winner msg*“, u suprotnom „*Display Looser msg*“ (Dijagram 3).

Dijagram 3. Dijagram sekvenci za use case „*CheckMyLuck*“ kreiran prema MVC arhitekturi.

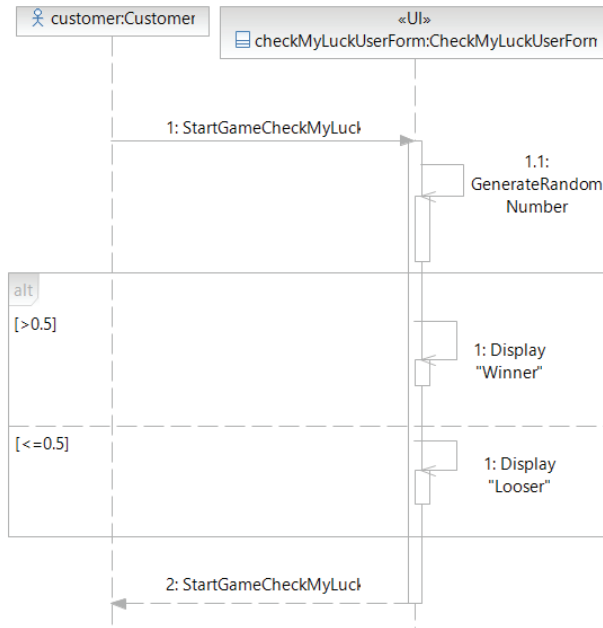
Interaction1



Dvoslojna arhitektura aplikacije

- ♦ S obzirom da je u zadatku rečeno da će se za ovu aplikaciju koristiti dvoslojna arhitektura softvera, onda izbacujemo kontrolera, jer se celokupna poslovna logika nalazi na klijentskoj strani aplikacije. U tom slučaju dijagram sekvenci bi izgledao kao na sledećem dijagramu.

Dijagram 4. Dijagram sekvenci *use case*-a *CheckMyLuck* kreiran prema dvoslojnoj arhitekturi.



Napomena:

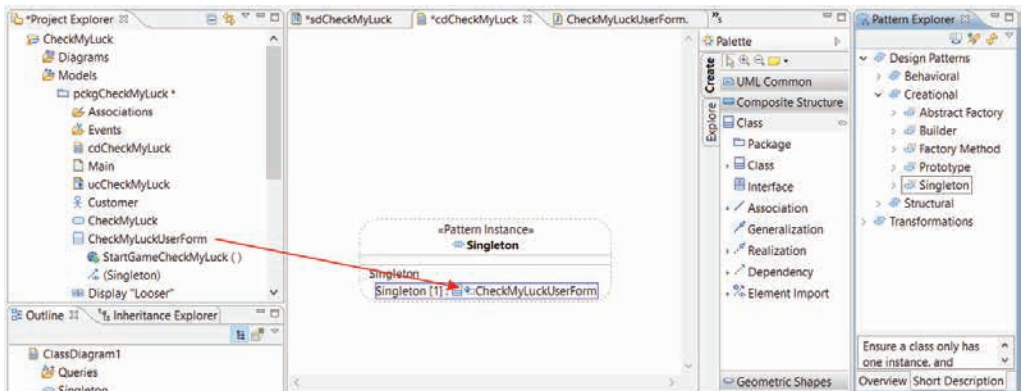
Ukoliko se poruke bilo "Winner" ili "Looser" prikazuju samo na korisničkoj formi tj. na interfejsu aplikacije, onda poruke ne idu do *lifeline*-a *Customer*, jer u ovom slučaju smo zamislili da je to akter kao osoba, a ne klasa. Međutim, ukoliko ste osmislili da se te poruke prenose glasovno ili na neki drugi način direktno akteru *Customer*, u tom slučaju poruka ide do *lifeline*-a *Customer*.

Korak 5. Kreirati dijagram klasa (Class diagram).

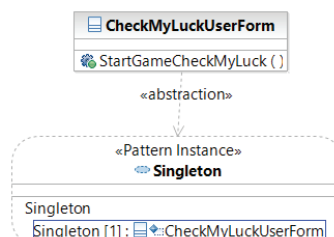
Tokom faze analize softvera uglavnom se kreira logički dijagram klasa, dok se u fazi projektovanja softvera, kreira fizički dijagram klasa. Kod kreiranja logičkog modela, uglavnom razmišljamo o tome šta aplikacija treba da radi, a ne i kako će biti implementirana. Dok kod kreiranja fizičkih modela, nakon što smo se usaglasili šta treba da radi aplikacija i dali joj početni okvir, razmišljamo i kako će biti implementirana, odnosno u fazi projektovanja (*design phase*) primenjujemo obrasce projektovanja (*design patterns*).

Koraci primene *Singleton* obrasca na dijagramu klasa su:

- ♦ Prevući *Singleton* obrazac sa *Pattern Explorer* pogleda.
- ♦ S obzirom da je klasa *CheckMyLuckUserForm*, već kreirana na dijagramu sekvenci, prevući ćemo je sa *Project Explorer*-a na polje obrasca za povezivanje (*binding box*), kao na donjoj slici:



- ♦ Da bi prikazali sve veze između klase i obrasca, potrebno je selektovati sve elemente ukoliko ih ima više (desni klik na praznu površinu, *Select > All*). Desni klik na bilo koji od selektovanih elemenata, odabrati *Filters > Show Related Elements*. Na novom prozoru, selektovati *Show All Relationships > OK*.



- ♦ Po potrebi dopuniti dijagram klasa sa atributima i metodama i njihovim tipovima podataka i povratnim vrednostima.

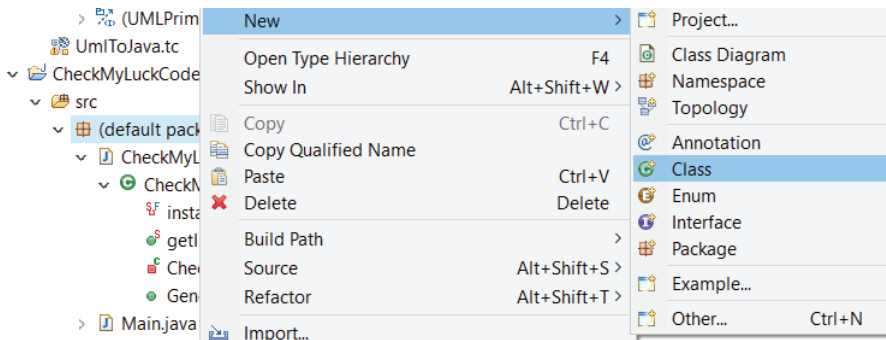
Korak 6. Izgenerisati kod u Javi, dopuniti ga i prikazati rezultate.

Koraci generisanja koda su:

- ♦ U maniju kliknite na *Modeling > Transform > New Configuration*.
- ♦ Unesite naziv ove konfiguracije, na primer *Name: UmlToJava*. Pod *Transformation* selektujte "UML to Java" i kliknite na *Finish*.
- ♦ Sa leve strane prozora **Selected source**, selektovaćete paket *pckgCheck-MyLuck*, a sa desne strane (ispod prozora *Selected target*) kliknite na komandu *Create Target Container*, kako bi ste kreirali target Java projekat gde će se nalaziti programski kod.
- ♦ Na novom prozoru samo unesite naziv Java projekta, npr. **Project name: CheckMyLuckCode** > *Finish*.
- ♦ Ukoliko je selektovano ono što želite, kliknite ponovo na *Finish*.
- ♦ Na novom prozoru kliknite na komandu *Run*.
- ♦ Pronađite novi projekat sa leve strane ekrana, tj. u okviru prozora **Project Explorer**. Proširite klase i pogledajte šta je sve izgenerisano po difoltu za *Singleton* obrazac.
- ♦ Dopunite kôd, kao na primer:

```
public class CheckMyLuckUserForm {
    public String GenerateRandomNumber() {
        if (Math.random()>0.5) {
            return "Winner";
        }
        return "Looser";
    }
    private static final CheckMyLuckUserForm instance = new CheckMyLuckUserForm();
    private CheckMyLuckUserForm() {
    }
    public static CheckMyLuckUserForm getInstance() {
        return instance;
    }
}
```

- ♦ Dodajte glavnu klasu, desnim klikom na difoltni paket, kliknuti na *New > Class*:



- ♦ Na novom prozoru unesite naziv klase, npr. *Main* i čekirajte da vam se izgeneriše automatski *main()* metoda.
- ♦ Unesite kôd kao na primer:

```
import java.io.IOException;
```

```
public class Main {
```

```
    public static void main (String[] args) throws IOException {
        while (true) {
            System.in.read();
            System.out.println(CheckMyLuckUserForm.getInstance().
                GenerateRandomNumber());
        }
    }
}
```

- ♦ Klikom na prečicu **Run** izvršite kôd i kliknite nekoliko puta po tastaturi da vidite kakve ste sreće:

Winner

Winner

Winner

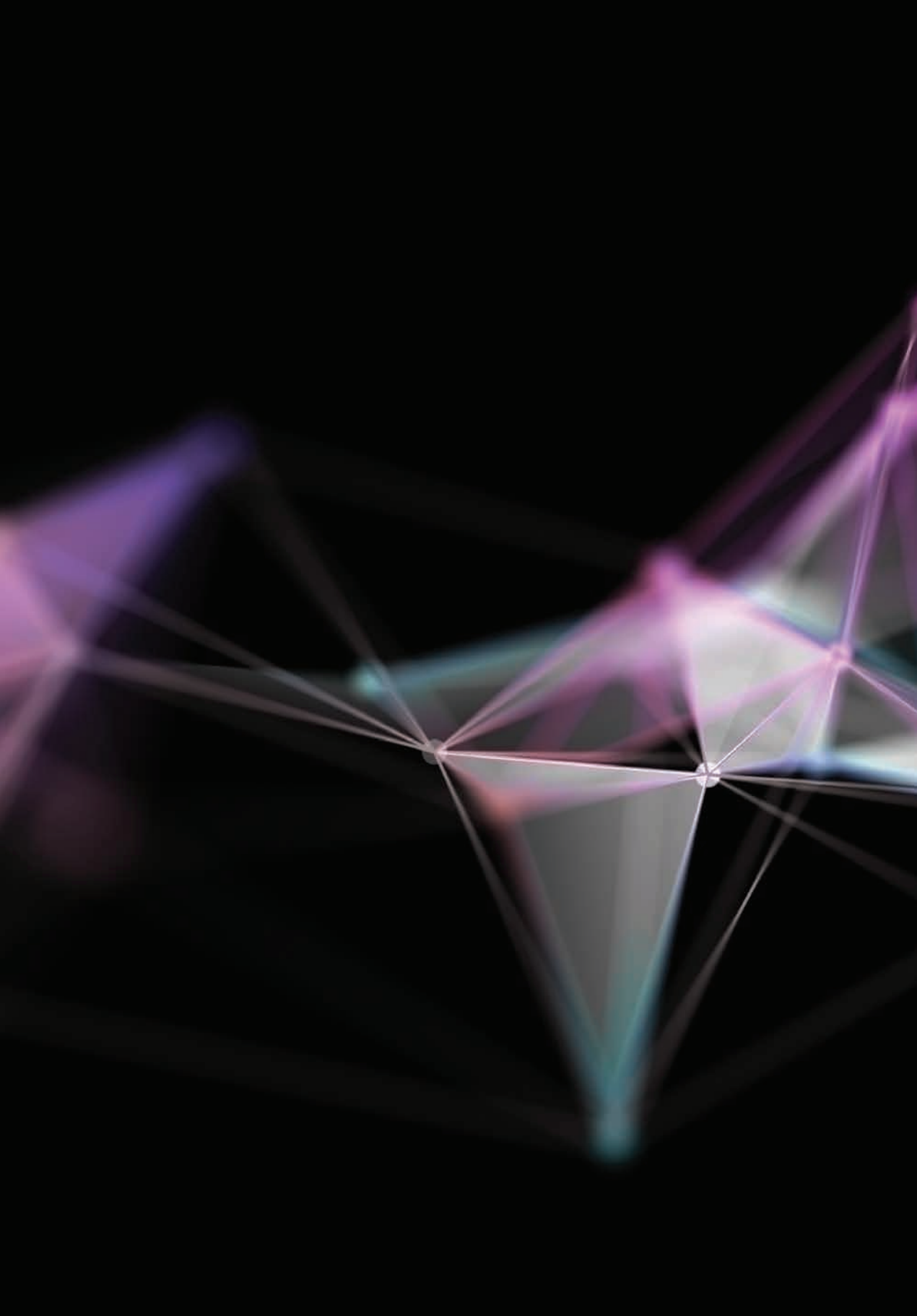
Looser

Looser

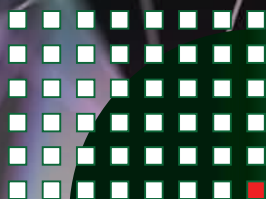
Looser

Winner

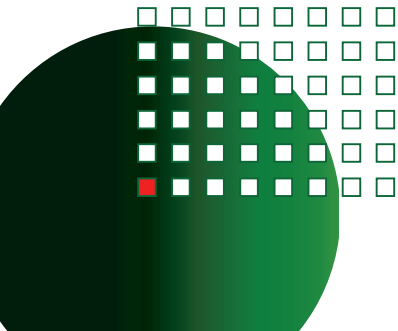
Winner



POGLAVLJE 3



**KREACIONI OBRASCI
PROJEKTOVANJA**



3. KREACIONI OBRASCI PROJEKTOVANJA

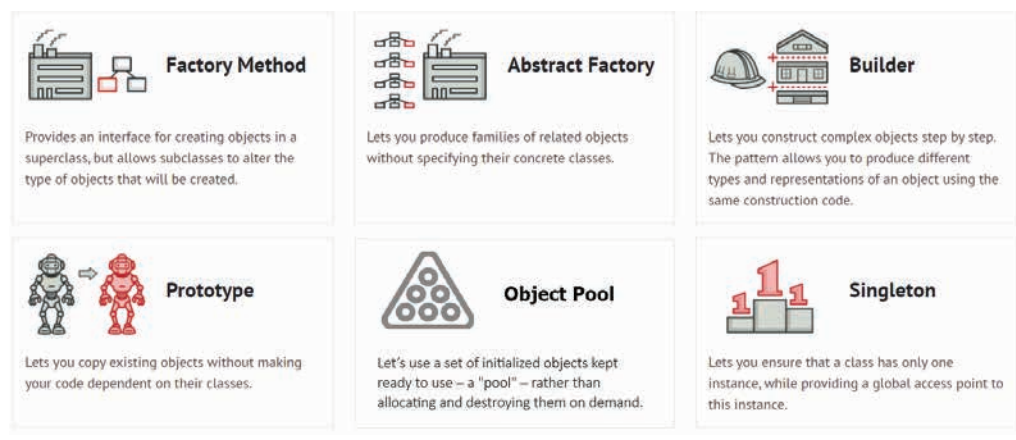
Kreacioni obrasci se bave kreiranjem objekata, na način prilagođen određenoj primeni. Posebno su važni u situacijama u kojima bi uobičajen pristup kreiranja objekata doveo do povećanja kompleksnosti projekta.

Neki od obrazaca koji spadaju u ovu grupu su:

- ♦ **Factory Method** – omogućava da klasa delegira odgovornost za kreiranje odgovarajućeg objekta. Kao i u realnom životu, kada nešto poručujete, vas i ne zanima da znate svaki detalj kako fabrika kreira taj proizvod. Tako da se logika kreiranja proizvoda izmešta iz klijentskog koda u novu klasu koja obično u svom nazivu ima *Factory* i u okviru te klase sadrži *create* metodu.
- ♦ **Abstract Factory** – obezbeđuje interfejs za kreiranje familije logički ili funkcionalno povezanih fabrika bez specificiranja konkretnih klasa familije objekata.
- ♦ **Builder** - obrazac koji daje smernice za određivanje klase objekta kreiranog po njegovom sadržaju ili kontekstu.
- ♦ **Prototype** - omogućava da klase budu kopirane ili klonirane iz instance prototipa, a ne da se kreiraju nove instance.
- ♦ **Singleton** - daje smernice višestrukim objektima za deljenje zajedničkog objekta bez potrebe da znaju da li on već postoji.
- ♦ **Object Pool** - daje smernice za ponovno korišćenje objekata umesto kreiranja novih objekata.

U ovom udžbeniku razmatraće se sedam kreacionih obrasca i to: *Factory Method*, *Abstract Factory*, *Builder*, *Prototype*, *ObjectPool* i *Singleton*. Kako je *Singleton* obrazac već bio opisan u prethodnom poglavlju, u nastavku će se objasniti rad ostalih obrazaca (Slika 6).

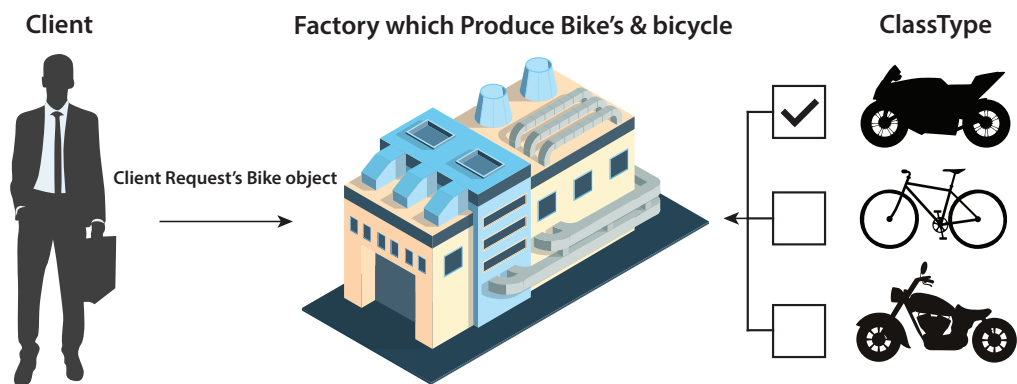
Slika 6. Kreacioni obrasci projektovanja.



3.1. FACTORY METHOD

Obrazac fabričke metode (*Factory Method*) funkcioniše po sličnom principu kao što u realnom životu funkcionišu fabrike (Mehra, 2021). Naime, u realnosti fabrike služe za proizvodnju ili kreiranje nekog proizvoda. Klijent koji poručuje proizvode preko neke, na primer, onlajn prodavnice, ne mora da zna, a često ga i ne zanima, na koji način su ti proizvodi proizvedeni (Slika 7). Odnosno, logika kreiranja proizvoda se izmešta iz klijentskog koda i stavlja u novu klasu Fabrika (*Factory*) u okviru koje se nalazi metoda *create()*, tzv. *factory method* (fabrički metod).

Slika 7. Poređenje obrasca Factory method sa realnim primerom u praksi.



Često postoji potreba za kreiranjem nekog objekta, a da u trenutku njegovog kreiranja nije poznat konkretan tip tog objekta. Na primer, pretpostavimo da postoji apstraktna klasa *Document*, čije su potklase *TextDocument* i *DrawingDocument* (Hadzi-Puric, n.d.). Takođe, pretpostavićemo da postoji klasa *Application*, sa potklasama *TextApp* i *DrawingApp*. Klasa *Application* sadrži zajednički kod za manipulaciju sa dokumentima, a koji je nezavisan od tipa dokumenta, kao što su otvaranje dokumenata, zatvaranje, snimanje, prikazivanje na ekranu i sl. Međutim, klasa *Application* je odgovorna i za kreiranje novih dokumenata, a ta operacija zavisi od tipa dokumenta, kako bi klasa *Application* znala koji tip aplikacije (*TextApp* ili *DrawingApp*) da pokrene. Kada postoji ovakav problem, tada je rešenje primena obrasca *Factory method*.

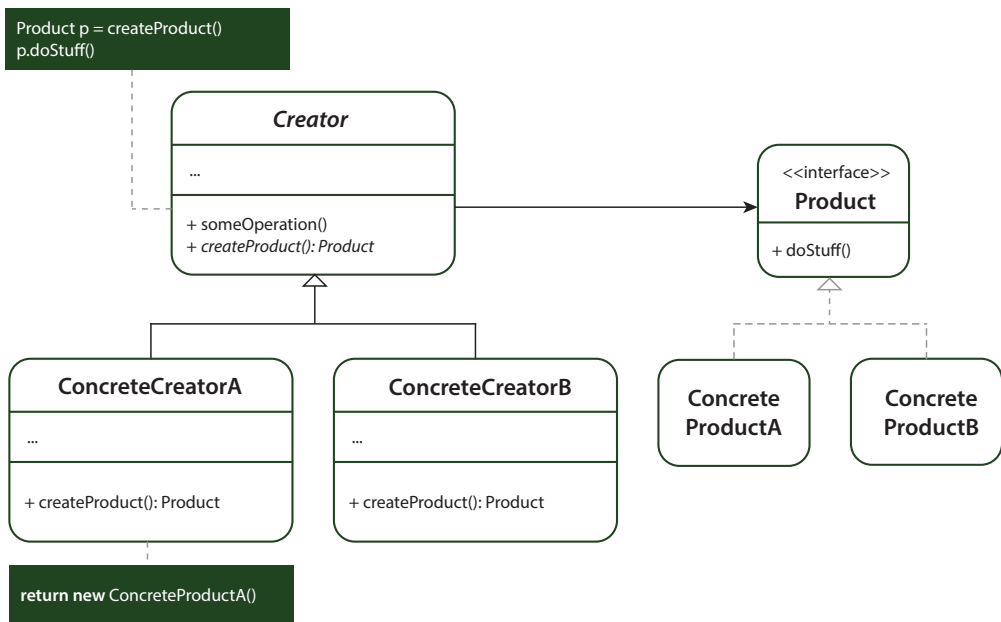
Takođe, ukoliko se u budućnosti ova aplikacija bude proširila (npr. dodavanjem *PaintDocument* i *PaintApp* klasama) ili zamenila nekim drugim (npr. umesto *TextDocument* uvodi se *WordDocument*), tada je rešenje primena *Factory method* obrasca.

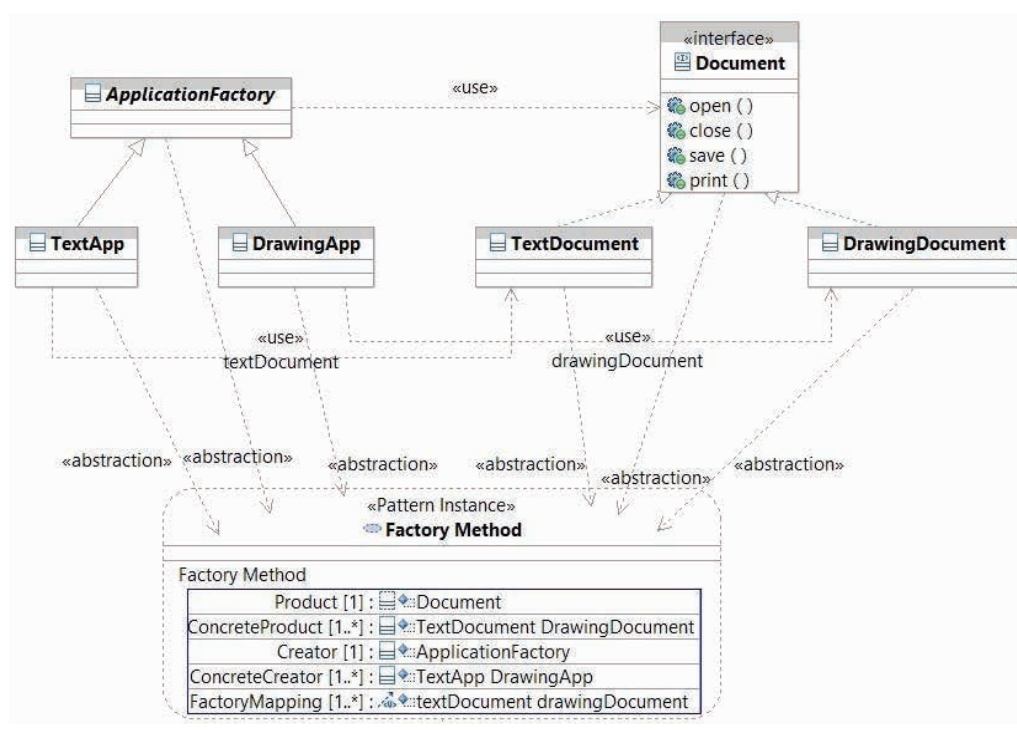
Primena *Factory method* obrasca usloviće da se operacije kreiranja različitih objekata tipa *Document*, izmeste iz *Application* klase i prebace u novu klasu, koju možemo nazvati npr. *ApplicationFactory*, u okviru koje se nalazi apstraktna metoda *create()*, a koja će biti konkretno realizovana u potklasama *TextApp* i *DrawingApp*. Klasa *Application* će sada samo instancirati *ApplicationFactory* objekat, kojoj će delegirati instanciranje konkretnog objekta (određeni tip dokumenta), pozivajući njegovu metodu *create()*.

Sve promene u kodu (npr. dodavanje novih tipova dokumenata) se događaju (kreiraju) u fabričkoj klasi (*ApplicationFactory*), bez menjanja klijentskog koda aplikacije. Ukoliko više klijenata želi da instancira isti set klasa, onda se upotrebom *Factory* objekta, eliminiše redundantni kôd. Aplikaciju je lakše modifikovati i održavati, jer je potrebno samo promeniti kôd u fabričkoj klasi, pre nego kod svakog klijenta.

Prateći osnovnu strukturu *Factory Method* obrasca (Slika 8), na slici 9., prikazuje se primena ovog obrasca na primeru aplikacije za kreiranje dokumenata.

Slika 8. Osnovna struktura *Factory Method* obrasca (Shvets, 2019).



Slika 9. Primena *Factory Method* obrasca na aplikaciju za kreiranje dokumenata.

Na osnovu opisanih problema kada je poželjno koristiti *Factory Method* obrazac i navedene osnovne strukture, u tabeli 5., daje se opis osnovnih elemenata ovog obrasca (Ivanković & Lacmanović, n.d.).

Tabela 5. Opis osnovnih elemenata *Factory Method* obrasca.

Elementi	Opis elemenata obrasca <i>Singleton</i>
Problem	• Potreba za kreiranjem nekog objekta, a da u trenutku njegovog kreiranja nije poznat konkretan tip tog objekta. • Potreba za eliminisanjem da se u kôd ugrađuju klase koje su specifične za pojedine aplikacije. • Potreba za budućim proširivanjem ili izmenama klasa.
Resursi	• <i>Product</i> [1]: definiše interfejs objekta koj pravi <i>Factory Method</i>. • <i>ConcreteProduct</i> [1..*]: implementira interfejs <i>Product</i>. • <i>Creator</i> [1]: deklarise <i>Factory Method</i> (<i>createProduct():Product</i>) koji vraća objekat tipa <i>Product</i>. Kreator može da pozove <i>factory method</i> radi pravljenja objekta tipa <i>Product</i> (<i>Product p=createProduct();</i>). • <i>ConcreteCreator</i> [1..*]: može da nadjača <i>Factory Method</i> da bi vratio instancu konkretnog proizvoda (<i>ConcreteProduct</i>).
Pravila	• <i>Factory metod</i> kaže da se samo definiše interfejs ili apstraktna klasa za kreiranje objekta, ali neka potklase odluče koji tip klase će instancirati. Drugim rečima, potklase su odgovorne za kreiranje instance klase. • <i>Factory Method</i> obrazac je takođe poznat i kao virtuelni konstruktor.
Smernice	Ovaj obrazac treba koristiti kada: • Klasa ne može da zna klasu objekta koji treba da napravi. • Klasa namerno hoće da njene potklase odrede objekte koje prave.

Prednosti ovog obrasca su:

- ♦ Omogućava potklasama da odaberu tip objekta koji će kreirati
- ♦ Promoviše labavo povezivanje eliminišući potrebu za vezivanjem aplikacije sa specifičnim klasama u kodu. To znači, da kôd komunicira samo sa rezultatima interfejsa ili apstraktne klase.

Vežba 3-1. Primeniti *Factory Method* obrazac nad klasom *orderPizza* (poručivanje pica).

Kreirati aplikaciju za poručivanje pica. Picerija trenutno nudi dve vrste pica: *Cheese-Pizza*, *GreekPizza* i *PepperoniPizza*. Picerija ima u planu da dopuni svoju ponudu sa još nekim vrstama pica. Ukoliko se pokaže da se neke pice ne prodaju najbolje, picerija će prestati da nudi tu vrstu pica.

Korak 1. Analiza problema sa postojećim kodom za poručivanje pica.

Koji su problemi u postojećem kodu:

```
Pizza orderPizza() {
    Pizza pizza = new Pizza();
    pizza.prepare();
    pizza.bake();
    pizza.cut();
    pizza.box();
    return pizza;
}
```

Radi fleksibilnosti, bilo bi bolje da je *Pizza* apstraktna klasa ili interfejs, ali je onda problem što ne možemo direktno instancirati jednu od njih.

Problem u prethodnom kodu je taj što korisnici poručuju više od jedne vrste pica. Dopunjen kod bi onda izgledao:

```
Pizza orderPizza(String type) {
    Pizza pizza;
    if (type.equals("cheese")) {
        pizza = new CheesePizza();
    } else if (type.equals("greek")) {
        pizza = new GreekPizza();
    } else if (type.equals("pepperoni")) {
        pizza = new PepperoniPizza();
    }
    pizza.prepare();
    pizza.bake();
    pizza.cut();
    pizza.box();
    return pizza;
}
```

Ukoliko se u porudžbini kreira pica, usled promene menija, morali bi smo stalno da menjamo kôd. Na primer, ukoliko je picerija zaključila da im ne ide dobro prodaja *GreekPizza*, oni je više neće nuditi. Takođe, usled potreba klijenata, meni će biti dopunjen sa još dve vrste pica i to *VeggiePizza* i *ClamPizza*. Treba imati na umu da se priprema (*prepare*, *bake*, *cut* i *box*) pica nije menjala godinama, tako da te operacije ostaju iste.

```

Pizza orderPizza(String type) {
    Pizza pizza;
    if (type.equals("cheese")) {
        pizza = new CheesePizza();
    } else if (type.equals("greek")) {
        pizza = new GreekPizza(); // nije više u ponudi
    } else if (type.equals("pepperoni")) {
        pizza = new PepperoniPizza();
    } else if (type.equals("veggie")) {
        pizza = new VeggiePizza();
    } else if (type.equals("clam")) {
        pizza = new ClamPizza();
    }
    pizza.prepare();
    pizza.bake();
    pizza.cut();
    pizza.box();
    return pizza;
}

```

Korak 2. Primena *factory* metode kao rešenje problema.

Da bi rešili problem izmene menija u klijentskoj aplikaciji, razdvojićemo *PizzaStore* (picerija) od *PizzaFactory* (fabrike koja kreira traženu picu). Pica će biti apstraktna klasa ili interfejs za ostale tipove pica koje se nude u meniju.

```

public class PizzaFactory {
    public Pizza createPizza(String type) {
        Pizza pizza = null;
        if (type.equals("cheese")) {
            pizza = new CheesePizza();
        } else if (type.equals("pepperoni")) {
            pizza = new PepperoniPizza();
        } else if (type.equals("veggie")) {
            pizza = new VeggiePizza();
        } else if (type.equals("clam")) {
            pizza = new ClamPizza();
        }
        return pizza;
    }
}

public class PizzaStore {
    PizzaFactory factory;
    public PizzaStore(PizzaFactory factory) {
        this.factory = factory;
    }

    public Pizza orderPizza(String type) {
        Pizza pizza;

```

```

        pizza = factory.createPizza(type);
        pizza.prepare();
        pizza.bake();
        pizza.cut();
        pizza.box();
        return pizza;
    }
    // ostale metode...
}

```

Vežba 3-2. Modelovati aplikaciju za logistiku primenom *Factory Method* obrasca.

Opis problema trenutne aplikacije za logistiku:

Zamislite da kreirate aplikaciju za upravljanje logistikom. Prva verzija vaše aplikacije može da prevozi samo kamionima, tako da najveći deo koda živi unutar klase Kamioni. Nakon nekog vremena, vaša aplikacija postaje prilično popularna. Svakog dana dobijate na desetine zahteva od kompanija za pomorski prevoz da u aplikaciju uključite morsku logistiku.

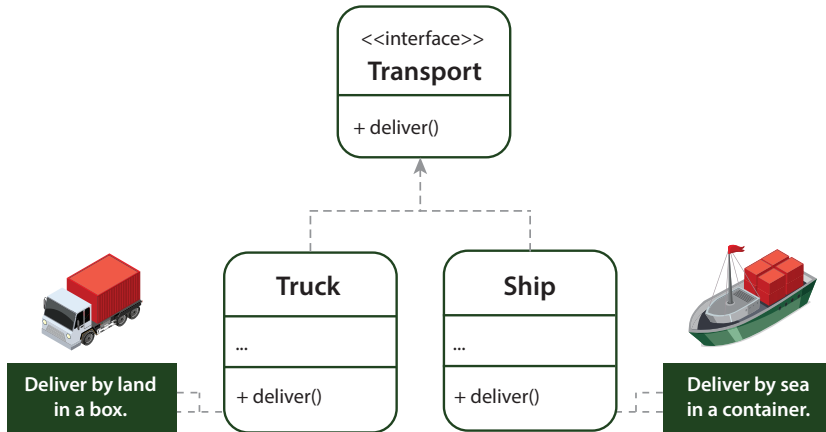
Dodavanje brodova u aplikaciju zahtevalo bi promene celokupnog koda. Štaviše, ako kasnije odlučite da dodate drugu vrstu prevoza u aplikaciju, verovatno ćete morati da izvršite sve ove promene ponovo. Kao rezultat toga, na kraju ćete dobiti prilično gadan kod, prepun uslova koji menjaju ponašanje aplikacije u zavisnosti od klase transportnih objekata.



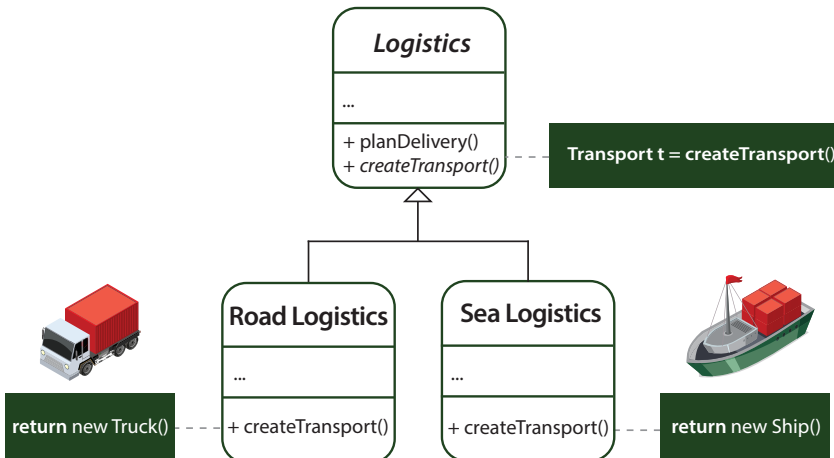
Dodavanje nove klase u program nije tako jednostavno ako je ostatak koda već povezan sa postojećim klasama!

Rešenje problema:

Obrazac *Factory Method* predlaže da zamenite direktne pozive za kreiranje objekta (koristeći *new* operator) sa pozivima na specijalni *factory* metod. Objekti su i dalje kreirani preko *new* operatora, ali se pozivaju iz *factory* metode *create()*. Objekti koji se vraćaju od strane *factory* metode se često nazivaju „proizvodima“. Na primer, klase *Truck* (Kamion) i *Ship* (Brod) treba da implementiraju interfejs *Transport*, koji deklariše metodu koja se zove *deliver()*. Svaka klasa drugačije implementira ovu metodu: kamioni isporučuju teret kopnom, a brodovi isporučuju teret morskim putem (Slika 10).

Slika 10. Primer interfejsa tipa *Product* i njegove potklase (Shvets, 2019).

Factory metoda u klasi *RoadLogistics* vraća objekte *truck*, dok *factory* metoda u klasi *SeaLogistics* vraća brodove (*Ship*) (Slika 11).

Slika 11. Apstraktna *factory* klasa *Logistics* sa *factory* metodom *createTransport()* i njene konkretne implementacije (Shvets, 2019).

Kôd koji koristi *factory* metodu (često se naziva klijentski kôd) ne vidi razliku između konkretnih proizvoda vraćenih od strane različitih potklasa. Klijent, u ovom primeru, sve proizvode tretira kao apstraktni *Transport*. Klijent zna da svi objekti *Transport* treba da imaju metod *deliver()*, ali kako to funkcioniše nije bitno za klijenta.

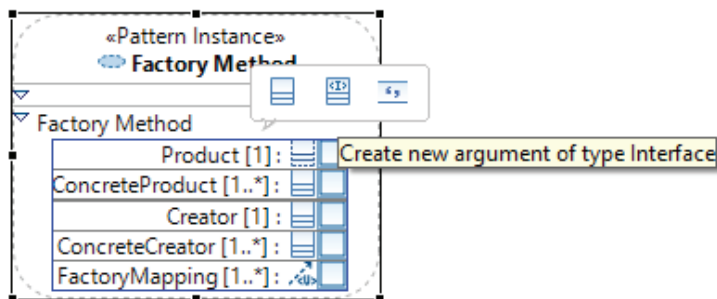
Nakon razumevanja problema i njegovog rešenja, u nastavku se prikazuju koraci rada na ovoj vežbi.

Korak 1. Kreirati projekat i u okviru njega dijagram klasa i prevući obrazac *Factory Method*.

Kreirati projekat pod nazivom *FactoryLogistics*. U okviru istoimenog paketa ovog projekta **kreirati dijagram klasa** u okviru koga će se prevući obrazac *Factory Method*.

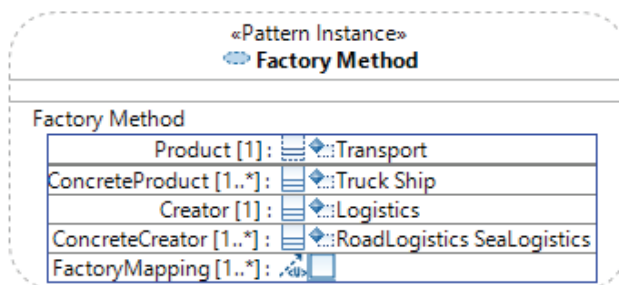
Uneti nazive klasa u obrazac, tako što ćete prevući kursor miša preko *Product* u okviru koga će se prikazati prečica za selekciju klase ili interfejsa ili unosa novih argumenata (Slika 12). Kliknuti na prečicu unosa novih argumenata i uneti klasu *Transport*.

Slika 12. Prečice za unos naziva klasa u elemente obrasca.



Nastaviti sa unosom klasa, kao što je prikazano na slici 13.

Slika 13. Unos klasa u osnovne elemente *Factory Method* obrasca.

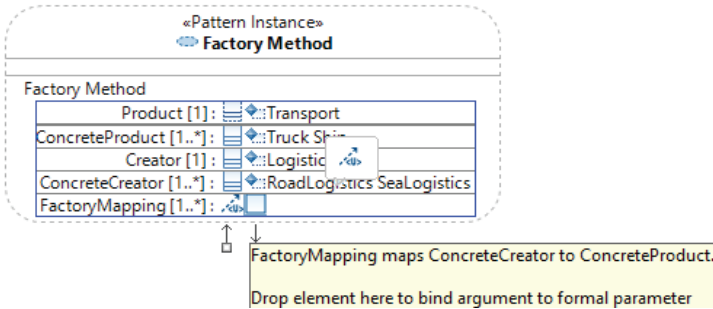


Korak 2. U okviru obrasca *Factory Method* izvršiti mapiranje.

Potrebno izvršiti mapiranje konkretnog kreatora sa konkretnim proizvodom. U ovom primeru *RoadLogistics* kreira *Truck*, dok *SeaLogistics* kreira *Ship*. Da bi se izvršilo mapiranje, potrebno je mišem preći preko dela *FactoryMapping*, kako bi se pojavila prečica za mapiranje.

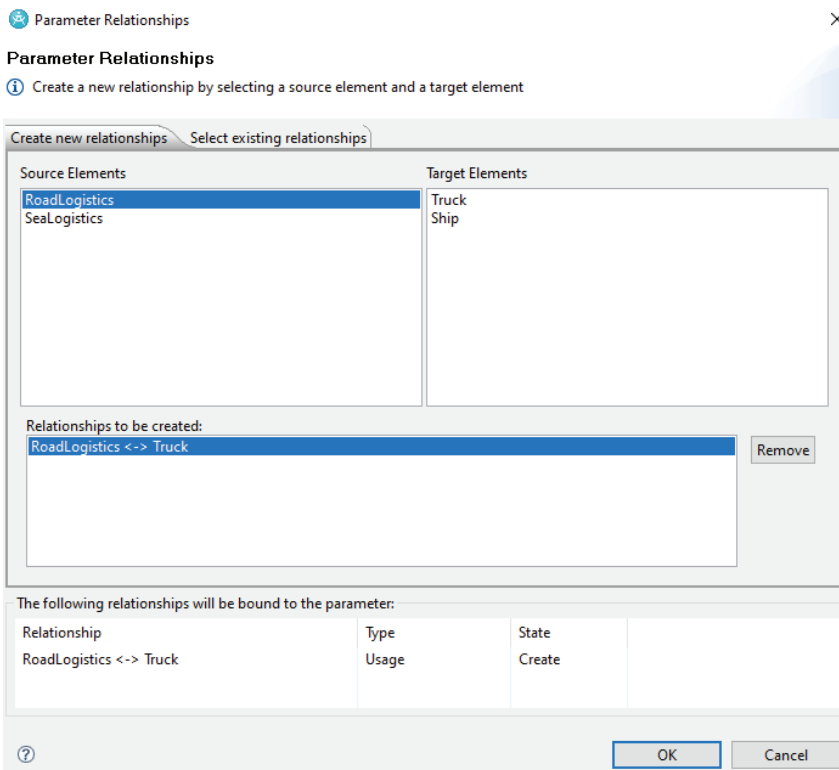
Kliknuti na ikonu:  (Slika 14).

Slika 14. Prečica za izvršavanje mapiranja (*FactoryMapping*).



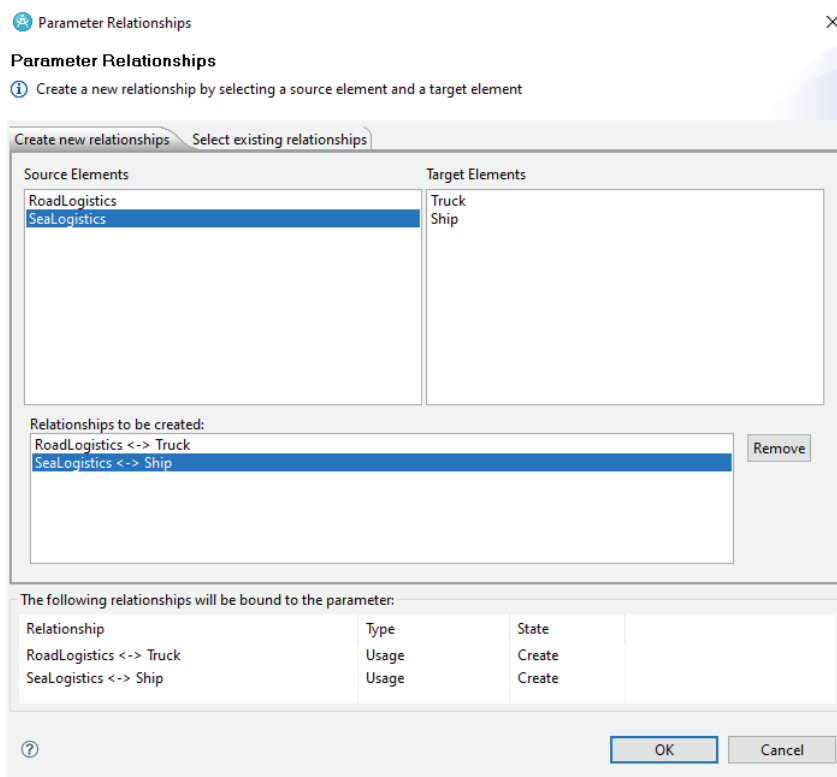
Klikom na prečicu pojaviće se prozor *Parameter Relationships* (Slika 15), u okviru koga je potrebno izvršiti mapiranje. Selektujte *RoadLogistics*, kao izvorni (*source*) element, a sa desne strane selektujte *Truck* kao target element. Pod *Relationships to be created* se vidi da *RoadLogistics* koristi odnosno kreira instancu *Truck*.

Slika 15. Prozor za mapiranje parametara.



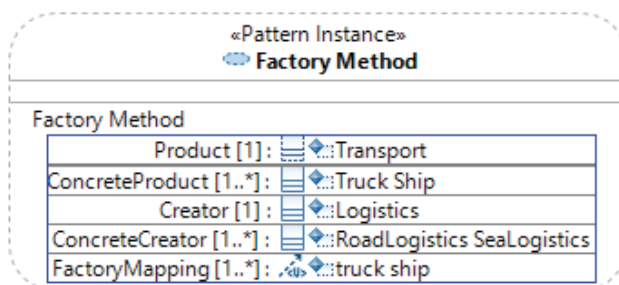
Ponoviti postupak i za *SeaLogistics* koji koristi *Ship* i kliknuti OK (Slika 16).

Slika 16. Finalni rezultat mapiranja kreatora proizvoda.



Finalni izgled obrasca je prikazan na slici 17.

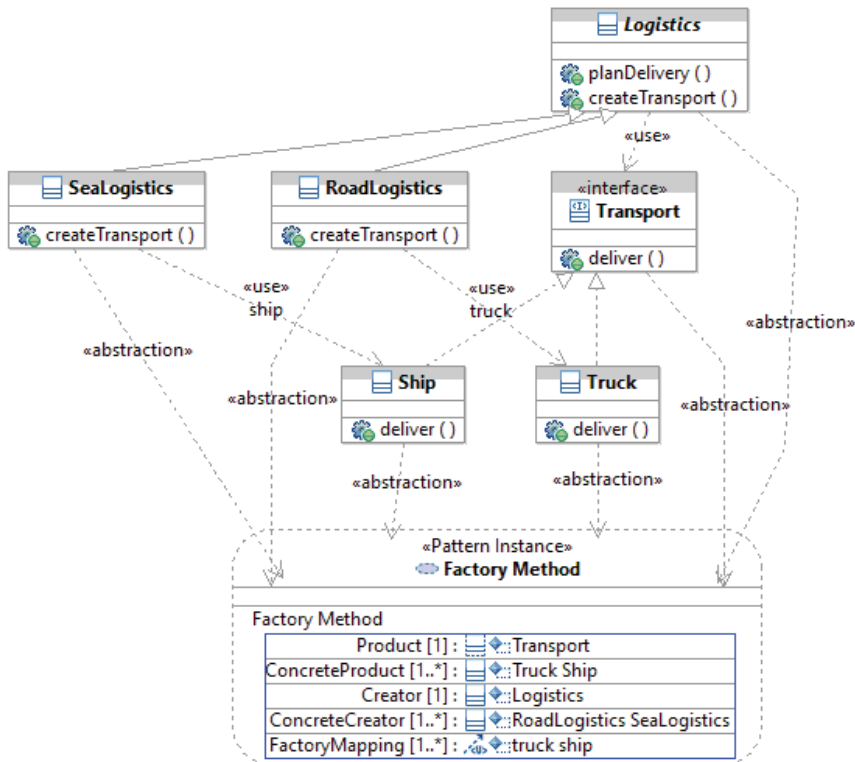
Slika 17. *Factory Method* sa popunjenim klasama i izvršenim mapiranjem.



Korak 3. Izvršiti povezivanje (bind) obrasca sa klasama.

Povezati (Bind) obrazac sa klasama, tako što ćete desnim klikom na selektovani obrazac, odabrati *Filters > Show Related Elements > Show All Relationships > OK*. Nad kreiranim dijagramom klasa, možete uneti metode, kao što je prikazano na slici 18.

Slika 18. Dijagram klasa sa uvedenim *Factory Method* obrascem.



Vežba 3-3. Razviti aplikaciju za igranje igara primenom *Factory Method* obrasca.

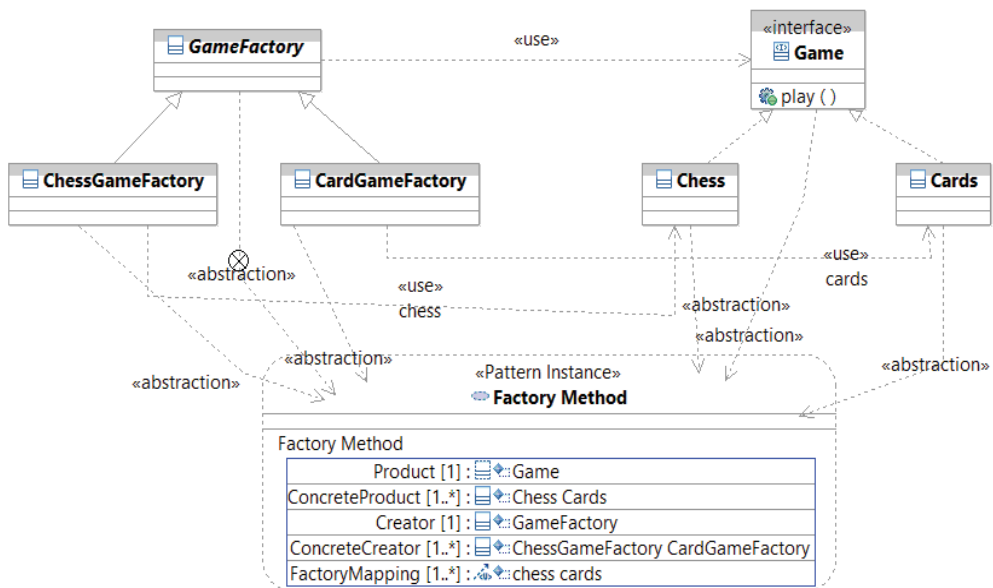
Kreirati aplikaciju koja će omogućiti korisniku igranje sledećih igara: šah (*Chess*) i karte (*Cards*). Treba imati u vidu proširenje postojeće aplikacije sa novim vrstama igara. Modelovati osnovnu strukturu ove aplikacije uz primenu odgovarajućeg obrasca projektovanja i izgenerisati Java kod.

Koraci rešenja ove vežbe su:

Korak 1. Modelovanje aplikacije za igranje igara primenom *Factory Method* obrasca.

S obzirom na buduće proširenje aplikacije sa novim igrama primeniće se *Factory Method* obrazac koji omogućava labavo povezivanje klasa, a samim tim i lakše modifikovanje i održavanje aplikacije. Dijagram klasa sa primenom ovog obrasca bi izgledao kao na slici 19.

Slika 19. Primena *Factory Method* obrasca na dijagram klasa aplikacije za igranje igara.



Korak 2. Generisati i dopuniti izvršni kod i prikazati rezultat rada.

Nakon generisanja izvršnog koda na osnovu početnog modela, kôd ćemo dopuniti sa osnovnim metodama radi demonstracije rada ove aplikacije.

```
public interface Game {
    public void play();
}

public class Chess implements Game {
    public void play() {
        System.out.println("Start Chess Game!");
    }
    public Chess() {
    }
}

public class Cards implements Game {
    public void play() {
        System.out.println("Start Cards Game!");
    }
    public Cards() {
    }
}

public abstract class GameFactory {
    public abstract Game createGame();
}

public class ChessGameFactory extends GameFactory {
    public Game createGame() {
        return new Chess();
    }
}

public class CardGameFactory extends GameFactory {
    public Game createGame() {
        return new Cards();
    }
}

public class FactoryGameMain {
    public static void main(String[] args) {
        System.out.println("*** Factory Method pattern demo ***\n");
        //Creating a chessGameFactory
        GameFactory chessGameFactory = new ChessGameFactory();
        //Creating a chess using the Factory Method
        Game achess = chessGameFactory.createGame();
        achess.play();
        //Creating a cardGameFactory
        GameFactory cardGameFactory=new CardGameFactory();
        Game aCard = cardGameFactory.createGame();
        aCard.play();
    }
}
```

Izvršavanjem aplikacije dobija se sledeći rezultat:

```
*** Factory Method pattern demo ***
```

```
Start Chess Game!
```

```
Start Cards Game!
```

Zadatak 3-1. Kreirati aplikaciju za generisanje računa za električnu energiju u zavisnosti od plana potrošnje primenom Factory Method obrasca.

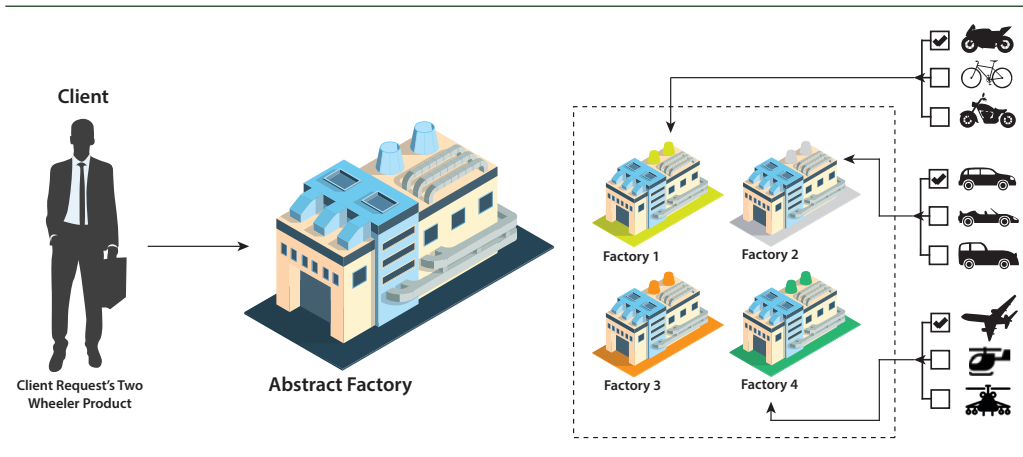
U zavisnosti od tipa plana potrošnje (*Plan*) koji može biti plan domaćinstva (*DomesticPlan*) ili plan preduzeća (*CommercialPlan*) izračunava se posebna tarifa (metod: **getRate():void**). Klasa *Plan* na osnovu određene tarife i jedinice potrošnje električne energije generiše račun (metod: **calculateBill(int units):void**). Primeniti odgovarajući obrazac, prikazati dijagram klasa, generisati i dopuniti kod i prikazati izlazne rezultate ove aplikacije.

3.2. ABSTRACT FACTORY

Kod *Factory Method* obrasca, za svaki novi proizvod nam je potrebna po jedna fabrika. *Abstract Factory* je proširenje *Factory Method* obrasca. Dok se *Factory Method* koristi za kreiranje samo jednog proizvoda, *Abstract Factory* se bavi kreiranjem porodica povezanih ili zavisnih proizvoda (Slika 20).

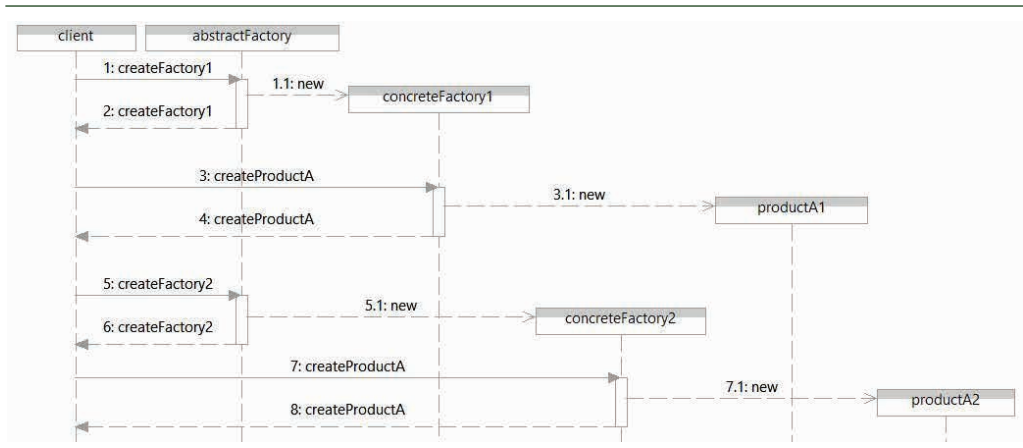
Abstract factory enkapsulira grupu pojedinačnih fabrika koji imaju zajedničke teme bez specificiranja njihovih konkretnih klasa. Ovaj obrazac definiše interfejs ili apstraktnu klasu za kreiranje familije povezanih (ili zavisnih) objekata, ali bez specificiranja njihovih konkretnih potklasa, što znači da apstraktna fabrika omogućava klasi da vrati fabrike klasa.

Slika 20. Poređenje *Abstract Factory* obrasca sa realnim primerom.



Radi boljeg razumevanja ovog obrasca, na slici 21 se prikazuje dijagram sekvenci.

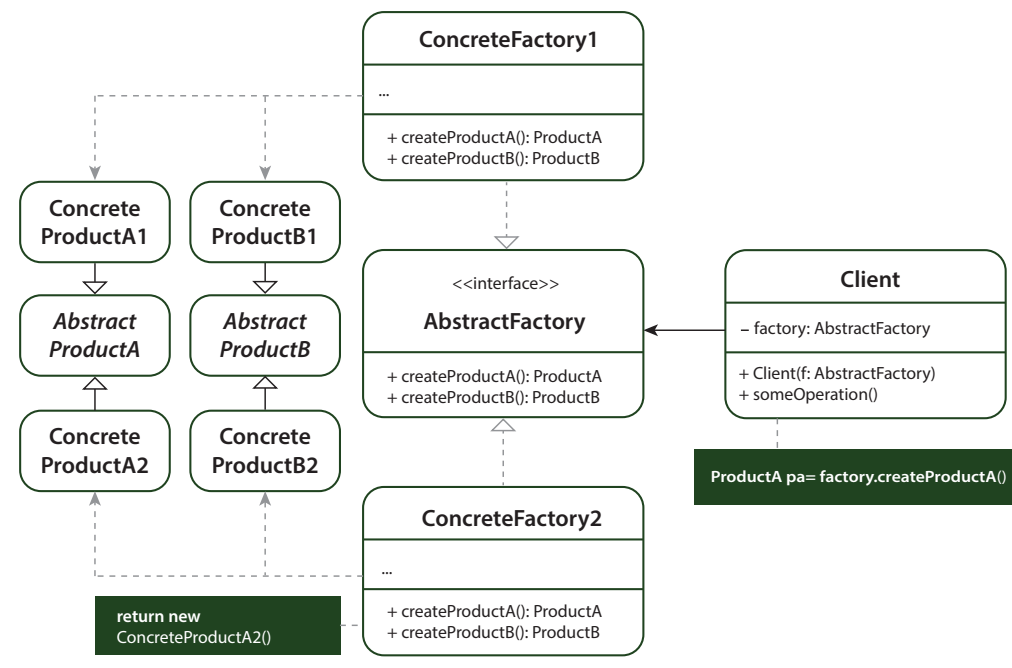
Slika 21. Dijagram sekvenci *Abstract Factory* obrasca.



Klasa *Client* kojoj trebaju objekti proizvoda A (*ProductA*), ne instancira direktno klase *ProductA1* i *ProductA2*. Umesto toga, *Client* se obraća interfejsu *Abstract Factory* za kreiranje objekata, što klijenta čini nezavisnim od toga kako su objekti kreirani. Klasa *ConcreteFactory1* implementira *AbstractFactory* interfejs, instancirajući klasu *ProductA1*.

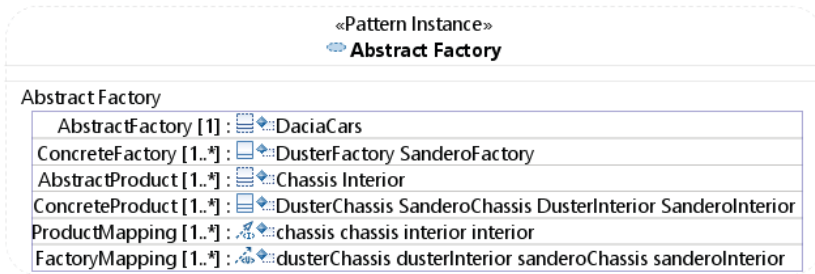
Osnovna struktura *Abstract Factory* obrasca je prikazana na slici 22.

Slika 22. Osnovna struktura *Abstract Factory* obrasca (Shvets, 2019).

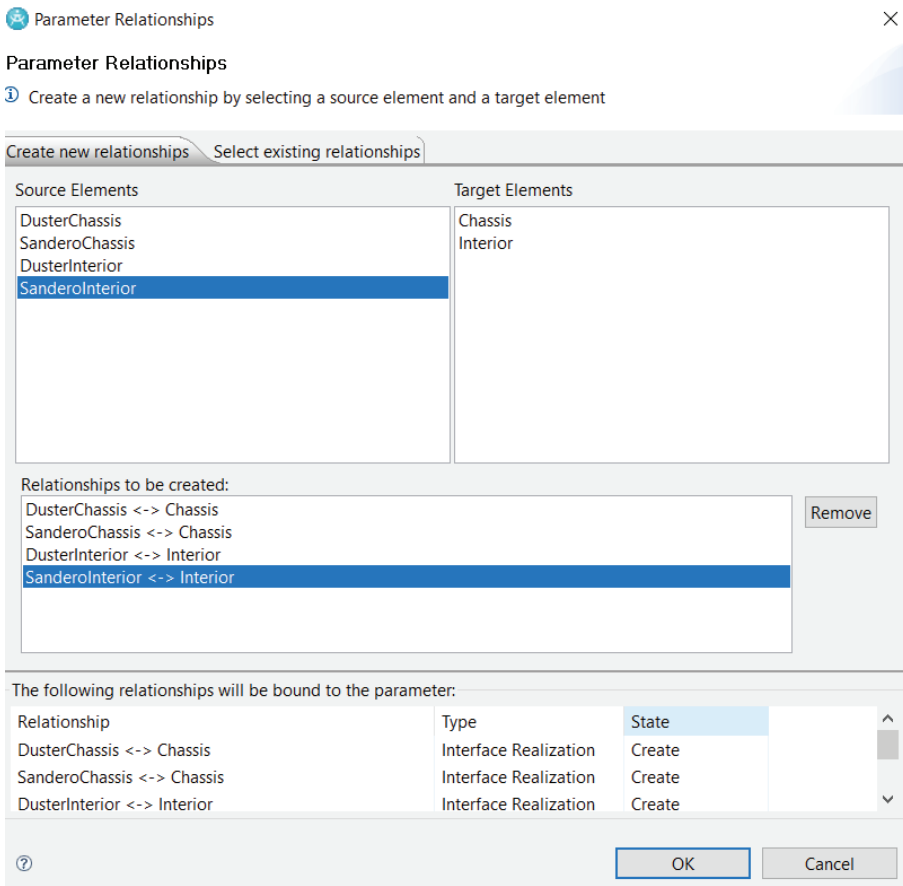


Da bi bolje objasnili ovu osnovnu strukturu koristićemo primer proizvodnje automobila. Na primer, potrebno je razviti aplikaciju za proizvodnju *Dacia* automobila (*DaciaCars*), konkretno *Sandero* i *Duster* modela. Kompanija *Dacia* ima fabriku za proizvodnju *Sandera* i fabriku za proizvodnju *Dustera*. Proizvode koje aplikacija treba instancira za ova dva modela su: šasija (*Chassis*) i interijer (*Interior*).

Koristeći *Abstract Factory* obrazac, kao što je prikazano na slici 23, apstraktna fabrika je *DaciaCars*, a konkretne fabrike su fabrika za proizvodnju *Dustera* (*DusterFactory*) i fabrika za proizvodnju *Sandera* (*SanderoFactory*). U ovom primeru apstraktnih proizvoda ima ukupno dva i to: šasija (*Chassis*) i interijer (*Interior*). Konkretni proizvodi za Šasiju su šasija za Duster automobil (*DusterChassis*) i šasija za Sandero automobil (*SanderoChassis*). Takođe, konkretni proizvodi za Interijer su interijer Dustera (*DusterInterior*) i interijer Sandera (*SanderoInterior*).

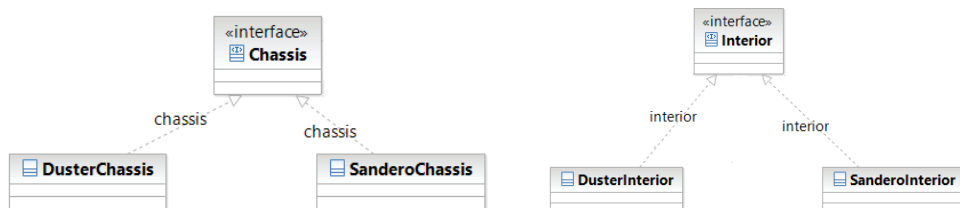
Slika 23. *Abstract Factory* obrazac za aplikaciju za proizvodnju automobila.

Mapiranje proizvoda (*ProductMapping*), za ovaj primer, prikazan je na slici 24. *DusterChassis* i *SanderoChassis* su konkretni proizvodi, apstraktnog proizvoda *Chassis*, dok su *DusterInterior* i *SanderoInterior* konkretni proizvodi apstraktnog proizvoda *Interior*.

Slika 24. Mapiranje proizvoda kod *Abstract Factory* obrasca.

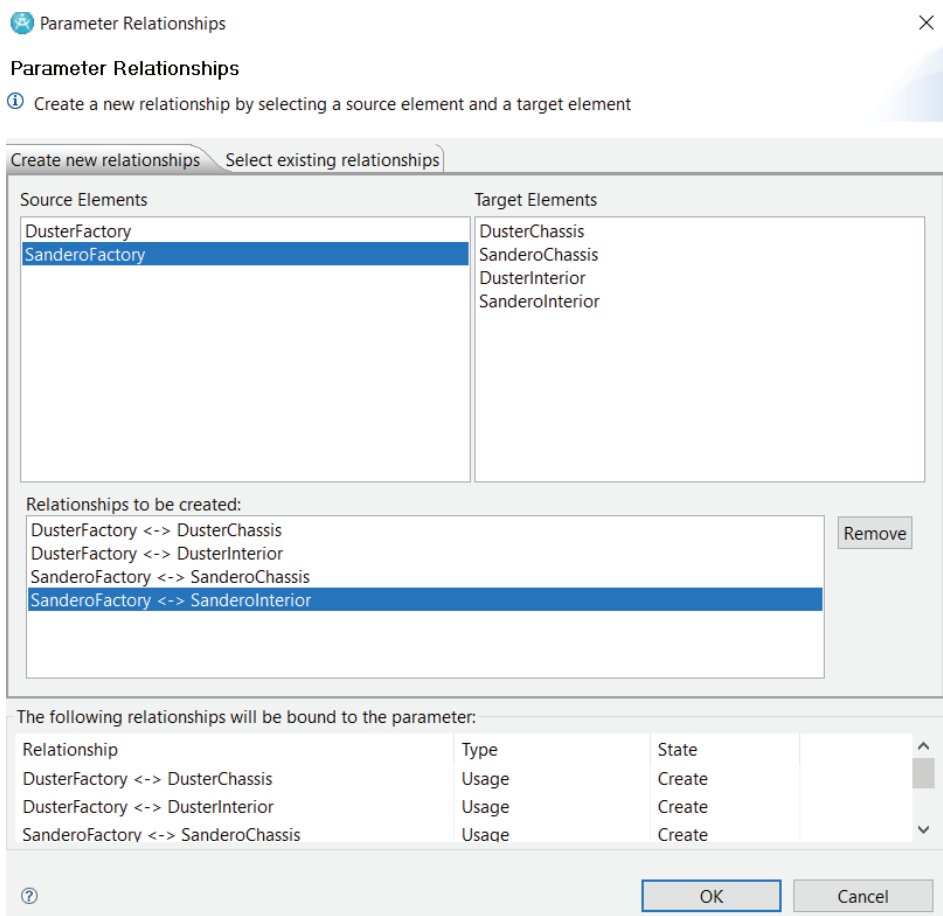
Rezultat ovog mapiranja bi na dijagramu klasa izgledao:

Slika 25. Rezultat mapiranja proizvoda prikazanog na dijagramu klasa.



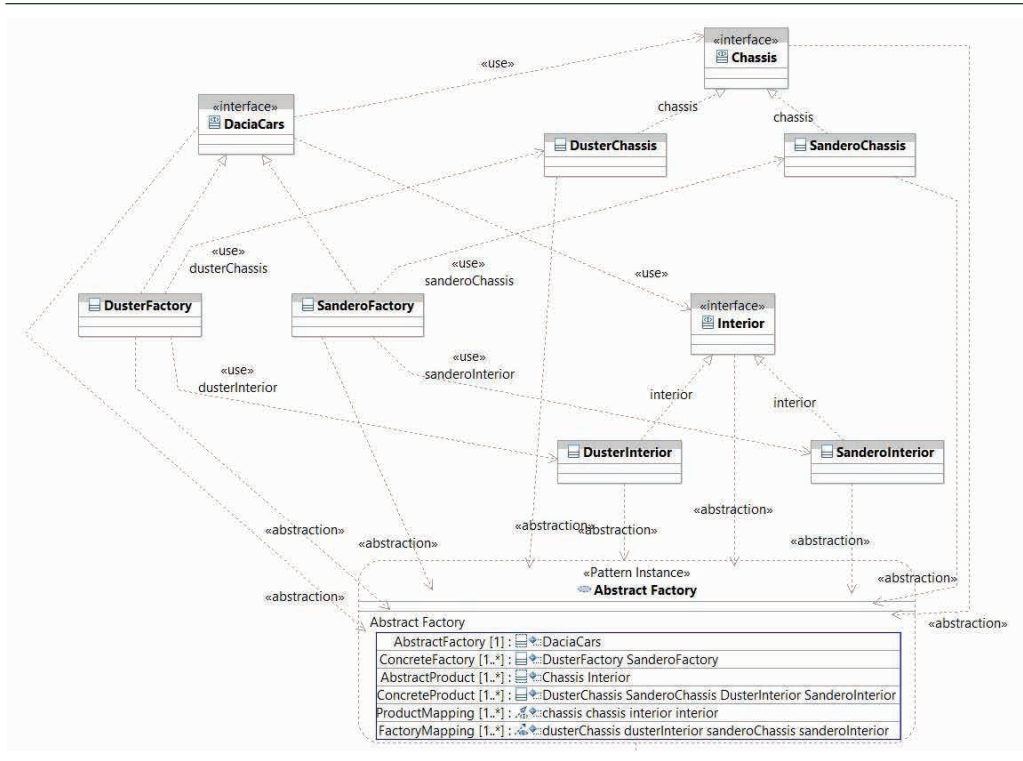
Mapiranje fabrike (*FactoryMapping*) pokazuje da *DusterFactory* instancira *DusterChassis* i *DusterInterior*, a *SanderoFactory* koristi *SanderoChassis* i *SanderoInterior* klase konkretnih proizvoda (Slika 26).

Slika 26. Mapiranje fabrika (*FactoryMapping*) kod *Abstract Factory* obrasca.



Nakon povezivanja (*bind*) obrasca sa klasama, celokupan dijagram klasa bi izgledao kao na slici 27.

Slika 27. Dijagram klasa aplikacije za proizvodnju automobila primenom *Abstract Factory* obrasca.



Osnovni elementi *Abstract Factory* obrasca su opisani u sledećoj tabeli.

Tabela 6. Opis osnovnih elemenata *Abstract Factory* obrasca (modifikovano sa Hadzi-Puric, n.d.).

Elementi	Opis elemenata obrasca <i>Abstract Factory</i>
Problem	- Potrebno je implementirati određene porodice klasa sa zajedničkim funkcionalnostima. - Sistem treba da bude nezavisan od načina kreiranja i predstavljanja proizvoda. - Sistem treba da bude konfigurisan jednom od više familija proizvoda.
Resursi	<i>AbstractFactory</i> [1]: deklariše interfejs koji definiše operacije za kreiranje apstraktnih objekata proizvoda. <i>ConcreteFactory</i> [1..*]: implementira interfejs <i>AbstractFactory</i> i pruža konkretne implementacije operacija za kreiranje konkretne familije proizvoda. <i>AbstractProduct</i> [1..*]: deklariše interfejs koji predstavlja apstraktni proizvod. Svi konkretni proizvodi pripadaju određenoj apstraktnoj familiji proizvoda. <i>ConcreteProduct</i> [1..*]: definiše konkretan objekat proizvoda koji će biti kreiran odgovarajućom konkrentom fabrikom. Implementira interfejs apstraktnog proizvoda. <i>Client</i> klasa [1]: koristi interfejse koje pružaju <i>AbstractFactory</i> i <i>AbstractProduct</i> za interakciju sa proizvodima. Klijent ne zavisi od konkretne implementacije proizvoda ili fabrika, već koristi apstraktne interfejse.
Pravila	Obezbeđuje interfejs za kreiranje familija logički ili funkcionalno povezanih objekata bez specificiranja konkretnih klasa familije objekata.
Smernice	Ovaj obrazac treba koristiti kada: - potrebno forsirati da proizvodi iz familije budu korišćeni zajedno. - potrebno ponuditi klasnu biblioteku proizvoda otkrivajući samo interfejse, ne implementacije.

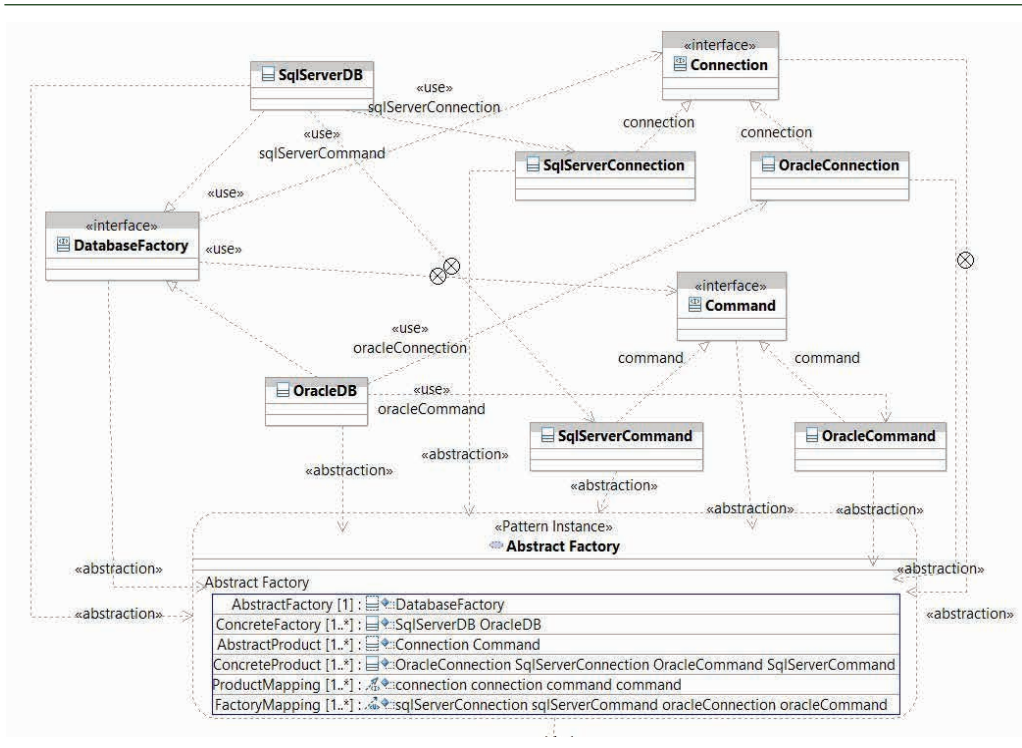
Uz pomoć *AbstractFactory* obrasca, omogućava se kreiranje familije objekata proizvoda koji su međusobno povezani i koriste se zajedno. Ovaj obrazac obezbeđuje visok nivo fleksibilnosti, omogućavajući promenu konkretne fabrike i proizvoda bez uticaja na klijentski kod.

Vežba 3-4. Razviti aplikaciju za rad sa različitim sistemima za upravljanje bazom podataka.

Razviti aplikaciju koja treba da podržava dve različite baze podataka i to SQL Server i Oracle. Dakle, kad god treba da upišete podatke u bazu podataka, potrebno je da kreirate vezu specifičnu za SQL Server (*SQLServerConnection*) ili vezu specifičnu za Oracle (*OracleConnection*). Takođe, kad god je potrebno da koristite komande ove dve specifične baze, Oracle će koristiti *OracleCommand*, a SQL Server će koristiti *SQLServerCommand*. Primeniti odgovarajući obrazac, prikazati dijagram klasa, izgenerisati kod i prikazati rezultate rada konkretnog obrasca.

Koraci rešenja ove vežbe su:

Korak 1. Kreirati dijagram klasa i primeniti *Abstract Factory* obrazac.



Korak 2. Izvršiti i dopuniti kod sa jednostavnom metodom i prikazati rezultate.

```
public interface Connection {
    void Action();
}

public class OracleConnection implements Connection {
    public void Action() {
        System.out.println("Connect to Oracle Database!");
    }
    public OracleConnection() {
    }
}

public class SqlServerConnection implements Connection {
    public void Action() {
        System.out.println("Connect to SQL SERVER Database!");
    }
}
```

```
        public SqlServerConnection() {  
        }  
    }  
}
```

```
public interface Command {  
    void CommandAction();  
}
```

```
public class OracleCommand implements Command {  
    public void CommandAction() {  
        System.out.println("Apply Oracle commands!");  
    }  
    public OracleCommand() {  
    }  
}
```

```
public class SQLServerCommand implements Command {  
    public void CommandAction() {  
        System.out.println("Apply SQL SERVER commands!");  
    }  
    public SQLServerCommand() {  
    }  
}
```

```
public interface DatabaseFactory {  
    public Connection createConnection();  
    public Command createCommand();  
}
```

```
public class OracleDB implements DatabaseFactory {  
  
    public Connection createConnection() {  
        return new OracleConnection();  
    }  
    public Command createCommand() {  
        return new OracleCommand();  
    }  
}
```

```
public class SqlServerDB implements DatabaseFactory {  
  
    public Connection createConnection() {  
        return new SqlServerConnection();  
    }  
    public Command createCommand() {  
        return new SQLServerCommand();  
    }  
}
```

```
public class DatabaseMain {  
    public static void main(String[] args) {  
        DatabaseFactory myDB;  
        Connection myConnection;  
        Command myCommand;  
        System.out.println("*** Abstract Factory Pattern Demo  
***\n");  
        myDB = new SqlServerDB();  
        myConnection = myDB.createConnection();  
    }  
}
```

```
        myConnection.Action();
        myCommand = myDB.createCommand();
        myCommand.CommandAction();
        System.out.println("*****");
        myDB = new OracleDB();
        myConnection = myDB.createConnection();
        myConnection.Action();
        myCommand = myDB.createCommand();
        myCommand.CommandAction();
    }
}
```

Rezultat izvršavanja:

*****Abstract Factory Pattern Demo*****

Connect to SQL SERVER Database!

Apply SQL SERVER commands!

Connect to Oracle Database!

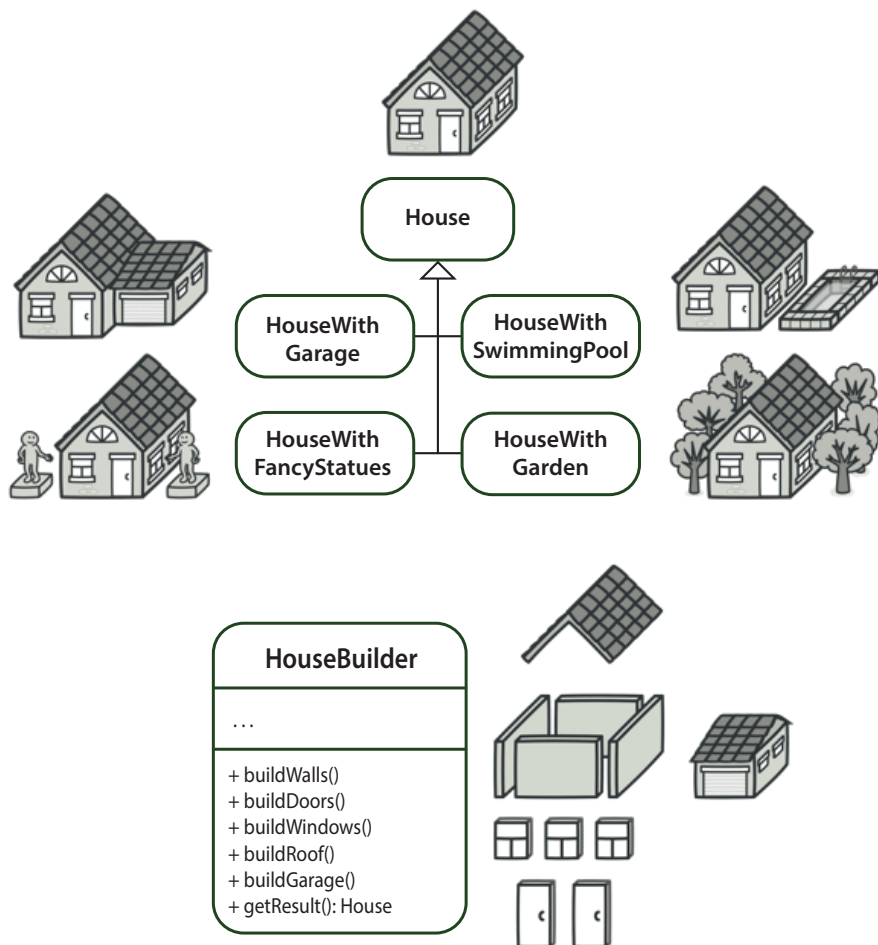
Apply Oracle commands!

3.3. BUILDER

Kreacioni obrazac *Builder* je koristan kada je potrebno da se kreiraju složeni objekti sklapanjem njegovih manjih delova tj. objekata, koristeći određeni algoritam ili proceduru. Ovaj obrazac omogućava da se proizvedu različiti tipovi i reprezentacije jednog objekta koristeći isti konstrukcioni kod.

Obrazac *Builder* omogućava da se korak po korak konstruišu složeni objekti. Na primer, postoje različite implementacije kuća (Slika 28). Da biste sagradili bilo koji tip kuće, potrebno je da izgradite četiri zida i pod, postavite vrata i prozore, napravite krov itd. Ukoliko želite veću kuću, sa dvorištem i drugim dodatnim elementima, opet ćete koristiti iste konstrukcije objekte. Obrazac *Builder* predlaže da izvučete kôd konstrukcije objekta iz njegove vlastite klase i premestite ga u odvojeni objekat nazvanog „graditelj” (*Builder*) (Slika 28).

Slika 28. Poređenje *Builder* obrasca sa realnim primerom (Shvets, 2019).

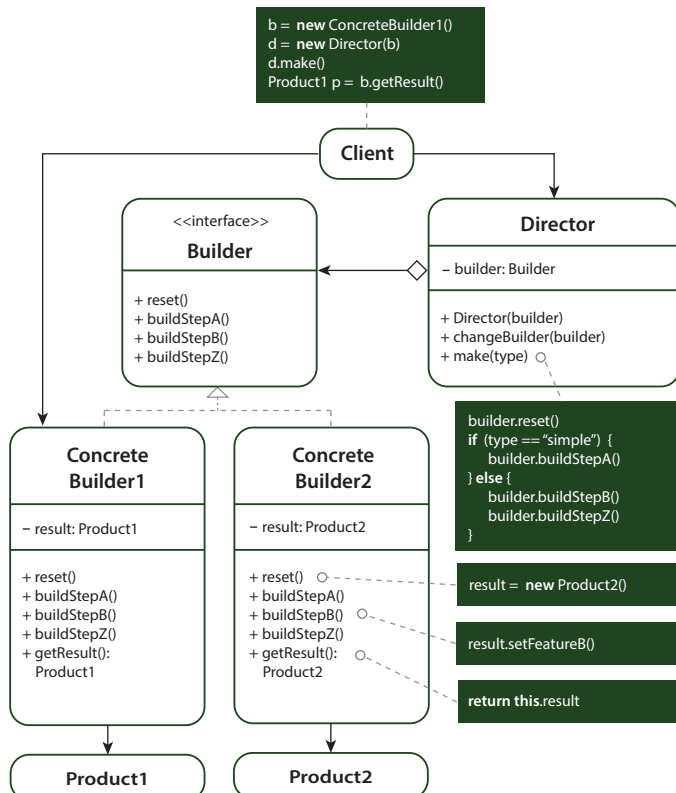


Obrazac *Builder* se može koristiti tamo gde je potrebno da gradite slične vrste složenih objekata, korak po korak. Razdvaja se procedura konstruisanja kompozitnih objekata od njihove reprezentacije, tako je moguće jednom procedurom kreirati više različitih reprezentacija (Joshi, 2018).

Osnovna struktura ovog obrasca je prikazana na slici 29. Proizvod (*Product*) je složeni objekat koji želite da kreirate. *ConcreteBuilder* konstruiše i sklapa delove proizvoda primenom apstraktnog interfejsa *Builder*. Objekti *ConcreteBuilder* grade interne reprezentacije proizvoda i definišu proces kreiranja ili mehanizme sklapanja. Interfejs *Builder* obezbeđuje metode za dobijanje objekta. Direktor (*Director*) je odgovoran za kreiranje konačnog objekta koristeći interfejs *Builder*, odnosno Direktor klasa definiše redosled po kojem će se pozivati koraci konstrukcije, na način da se mogu kreirati ili ponovo upotrebiti specifične konfiguracije proizvoda. Drugim rečima, direktor koristi interfejs *Builder* i kontroliše korake i redosled za pravljenje finalnog proizvoda.

Klasa *Director* ne kreira i ne sklapa direktno objekte klase *Product1* i *Product2*. Umesto toga, klasa *Director* se obraća *Builder* interfejsu za izgradnju (kreiranje i sklapanje) delova kompleksnog objekta, čime je *Director* nezavisan od toga koja je konkretna klasa instancirana. Klasa *ConcreteBuilder1* implementira interfejs *Builder* kreirajući i sklapajući objekte klase *Product1* (Slika 29).

Slika 29. Osnovna struktura *Builder* obrasca projektovanja (Shvets, 2019).



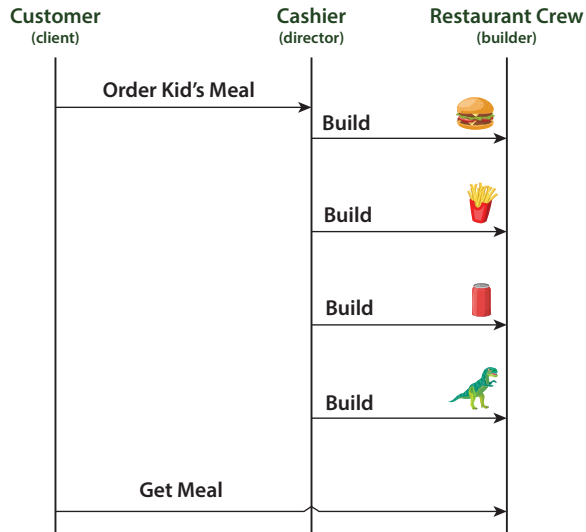
Opis osnovnih elemenata *Builder* obrasca su navedena u narednoj tabeli.

Tabela 7. Opis osnovnih elemenata obrasca *Builder*.

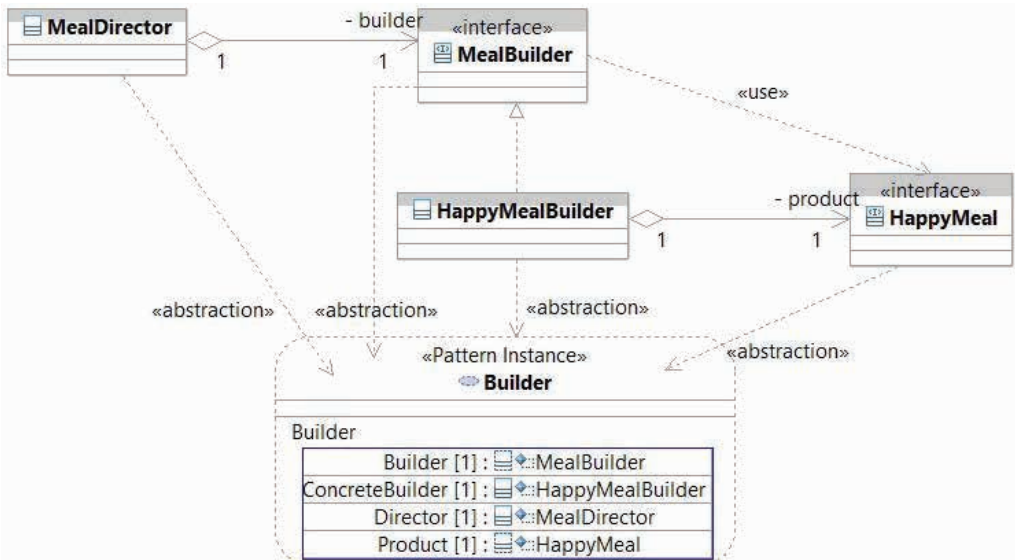
Elementi	Opis elemenata obrasca <i>Builder</i>
Problem	Potrebno je kreirati složene objekte sklapanjem njegovih manjih delova tj. objekata, koristeći iste konstrukcione objekte.
Resursi	<i>Builder</i> [1]: deklariše korake konstrukcije proizvoda koji su zajednički za sve tipove graditelja. <i>ConcreteBuilder</i> [1..*]: obezbeđuje različite implementacije konstrukcionih koraka. Konkretni graditelji mogu proizvesti proizvode koji ne prate opšti interfejs. <i>Director</i> [1]: definiše redosled pozivanja konstrukcionih koraka, tako da se mogu kreirati i ponovo upotrebiti specifične konfiguracije proizvoda. <i>Product</i> [1..*]: rezultujući objekti. Proizvodi, konstruisani od strane različitih graditelja, ne moraju da pripadaju istoj hijerarhiji klase ili interfejsa. <i>Client</i> klasa [1]: Mora da pridruži direktora sa jednim od <i>Builder</i> objekata. Obično se ovo uradi samo jedanput, pomoću parametara direktorovog konstruktora. Zatim direktor koristi taj <i>Builder</i> objekat za sve naredne konstrukcije.
Pravila	- Enkapsulirati, odnosno sakriti detalje složenog procesa izgradnje (konstrukcione korake) u posebnu klasu. - Kombinovanjem istih konstrukcionih koraka mogu se proizvesti različiti proizvodi.
Smernice	Ovaj obrazac treba koristiti kada je potrebno graditi složene objekte, korak po korak, kombinovanjem istih konstrukcionih koraka.

Vežba 3-5. Razviti jednostavnu aplikaciju za kreiranje *Happy Meal* obroka.

Simulacija poručivanja *Happy Meal* obroka u McDonalds restoranima, prikazana je koristeći dijagram sekvenci na slici 30. Klijent poručuje *Happy Meal* obrok. Prodavac, u ulazi klase *Director*, prosleđuje zahtev osoblju (*Restaurant Crew*) za kreiranjem obroka, navodeći željene sastojke obroka klijenta. Interfejs *Builder* sadrži zajedničke metode za bilo koju kombinaciju *Happy Meal* obroka, kao što su: kreirajSendvič, kreirajPomfrit, kreirajPiće i kreirajIgračku. Konkretna implementacija će pokazati šta je tačno instancirano, odnosno šta klijentov *Happy Meal* konkretno sadrži. Finalni proizvod je konkretan *Happy Meal* obrok.

Slika 30. Dijagram sekvenci poručivanja *HappyMeal* obroka.

Primenjujući *Builder* obrazac i povezujući ga sa klasama, dobija se sledeći dijagram klasa:

Slika 31. Dijagram klasa na primeru kreiranja *HappyMeal* obroka.

Izvorni kod je malo modifikovan u odnosu na generisani kod dobijenog iz IBM RSA alata.

Klasa *Director*:

```
public class MealDirector {
    private MealBuilder builder;
    public MealBuilder getBuilder() {
        return builder;
    }
    public void setBuilder(MealBuilder builder) {
        this.builder = builder;
    }
    //Director knows how to use the builder and the sequence of steps.
    public void construct(MealBuilder builder) {
        this.builder=builder;
        builder.addSandwich();
        builder.addFrenchFries();
        builder.addDrink();
        builder.addToy();
    }
}
```

Klasa *Builder* tipa interfejs:

```
public interface MealBuilder {
    public void addSandwich();
    public void addFrenchFries();
    public void addDrink();
    public void addToy();
    //the following method is used to retrieve the object that is constructed.
    public HappyMeal getProduct();
}
```

Klasa *Concrete Builder*:

```
public class HappyMealBuilder implements MealBuilder {
    private HappyMeal product;
    private String sandwichName;
    public HappyMealBuilder(String sandwich) {
        product = new HappyMeal();
        this.sandwichName=sandwich;
    }
    @Override
    public void addDrink() {
        product.add("Drink is added!");
    }
    @Override
    public void addFrenchFries() {
        product.add("Fries is added!");
    }
}
```

```

@Override
public void addSandwich() {
    product.add(String.format("Sendwich type is:%s",
this.sendwichName));
}
@Override
public void addToy() {
    product.add("Desired toy is added!");
}
public HappyMeal getProduct() {
    return product;
}
public void setProduct(HappyMeal product) {
    this.product = product;
}
}

```

Klasa *Product* (koja je izmenjena u odnosu na generisanu, odnosno nije interfejs):

```

import java.util.LinkedList;
public class HappyMeal {
    private LinkedList<String> parts;
    public HappyMeal(){
        parts=new LinkedList<String>();
    }
    public void add(String part){
        parts.addLast(part);
    }
    public void showMeal(){
        System.out.println("\nHappyMeal completed as below: ");
        for(String part: parts)
            System.out.println(part);
    }
}

```

Glavna klasa:

```

public class HappyMealMain {
    public static void main(String[] args) {
        System.out.println("**** Builder pattern demo ****");
        MealDirector director = new MealDirector();
        MealBuilder happymeal=new HappyMealBuilder("Cheeseburger");
        director.construct(happymeal);
        HappyMeal hm=happymeal.getProduct();
        hm.showMeal();
    }
}

```

Rezultat izvršavanja programa:

```
*** Builder pattern demo ***
```

```
HappyMeal completed as below:
Sandwich type is: Cheeseburger
Fries is added!
Drink is added!
Desired toy is added!
```

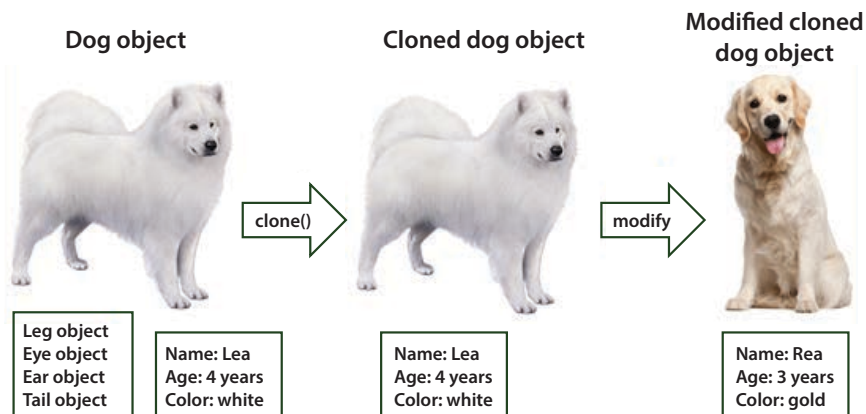
Zadatak 3-2. Primeniti odgovarajući obrazac projektovanja na aplikaciju koja se bavi sklapanjem normalnih i gejming računara.

Primeniti odgovarajući obrazac i modelovati aplikaciju jedne prodavnice *Computer Manufacturer*, koja sklapa dva tipa računara i to: računare normalne konfiguracije (*NormalComputer*) i gejming računare (*GamingComputer*). Ove klase sadrže metode: *addCPU()*, *addMotherboard()*, *addRAM()*, *addGPU()*, *addStorage()* i *getComputer()*. Klasa *Computer* sadrži metode: *ComputerType()* i *printDetails()*. Napraviti instancu odgovarajućeg obrasca projektovanja za gejming računare.

3.4. PROTOTYPE

U realnom životu, prototipovi se koriste za razna vrsta testiranja nekog proizvoda, pre nego što se krene u njegovu proizvodnju. Međutim, u ovom slučaju, prototipovi imaju više pasivnu ulogu. Naime, ukoliko postoji potreba za instanciranjem klase, za koju je već kreiran objekat, tada se neće krenuti sa kreiranjem još jednog objekta, već će se koristiti metoda *clone()*, za kloniranje postojećeg, a zatim će se izvršiti potrebne modifikacije (Slika 32). Dakle, Prototype obrazac se koristi za kreiranje novih objekata (instanci) kloniranjem (kopiranjem) postojećih objekata.

Slika 32. Primer kloniranja objekta „Dog“.

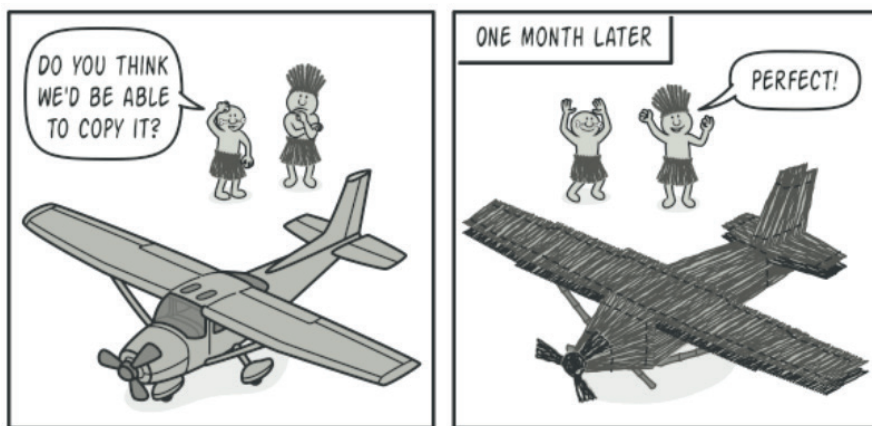


Ukoliko bi koristili klasičan način kreiranja nekog postojećeg objekta, prvo bi kreirali novi objekat iste klase, a zatim bi prošli kroz sva polja originalnog objekta i kopirali njihove vrednosti u novi objekat. Međutim, neka polja objekta mogu biti privatna te nisu vidljiva izvan samog objekta, a ponekad je poznat samo interfejs koji objekat realizuje, a ne i njegova konkretna klasa (Slika 33).

Rešenje za ove probleme je korišćenje *Prototype* obrasca projektovanja. *Prototype* obrazac omogućava kopiranje originalnog objekta u novi objekat, kako bi ga kasnije modifikovali prema našim potrebama. Na primer, ukoliko razvijate igricu sa više nivoa. Svaki nivo je zasnovan na osnovnom nivou, da bi kasnije bio modifikovan. U ovom slučaju možete klonirati osnovni nivo igrice za kreiranje narednog nivoa kako bi izbegli da kreirate nivo od početka.

Suština ovog obrasca je da se kreira objekat željenog tipa, ali ne na način da se kreira nova instanca, već da se kopira instanca konkretne klase. Proces kopiranja (kloniranja) se može ponoviti više puta. Kopije predstavljaju nezavisne objekte čije se vrednosti mogu menjati.

Slika 33. Primer rezultata klasičnog kopiranja postojećeg objekta (Shvets, 2019).



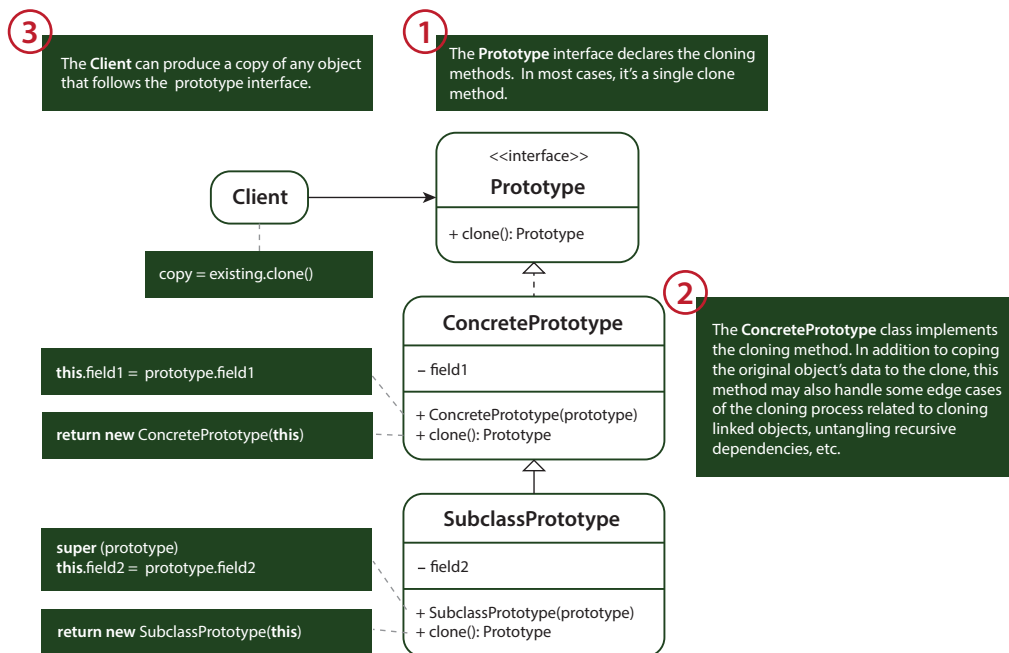
Copying an object "from the outside" isn't always possible.

Glavna prednost *Prototype* obrasca je što se kloniranjem objekata ubrzava instanciranje veoma velikih klasa, koje se dinamički učitavaju i pri tom se čuva znanje o sastavnim delovima velikih struktura podataka koje se mogu klonirati, a da nije potrebno znati potklase od kojih su kreirane (Ivanković & Lacmanović, n.d.). Može se koristiti plitka ili duboka kopija postojećih objekata:

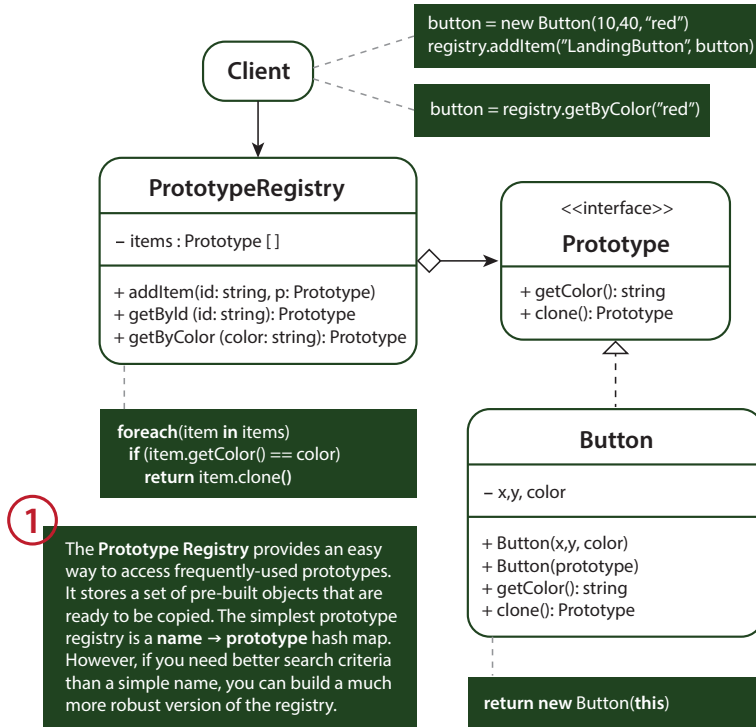
- ♦ U plitkoj kopiji (*Shallow Copy*) kloniramo samo roditeljski objekat, a ne objekte koji ga sadrže.
- ♦ U dubokoj kopiji (*Deep Copy*) kloniramo roditeljski objekat kao i objekte koji ga sadrže.

Osnovna struktura *Prototype* obrasca je prikazana na slici 34. Obrazac deklarira zajednički interfejs *Prototype* koji sadrži metodu *clone()*. Implementacija metode *clone()* je veoma slična u svim klasama. Metod kreira objekat trenutne klase i prenosi sve vrednosti polja starog objekta u novi. Objekat koji podržava kloniranje naziva se prototip.

Slika 34. Osnovna struktura *Prototype* obrasca (struktura osnovne implementacije) (Shvets, 2019).



Ukoliko se često pristupa prototip objektima, implementacija tipa *Prototype registry* obezbeđuje lak pristup tim objektima. Klasa *PrototypeRegistry* skladišti set unapred kreiranih objekata koji su spremni da budu kopirani.

Slika 35. Implementacija tipa *Prototype Registry* (Shvets, 2019).

Opis osnovnih elemenata *Prototype* obrasca su navedena u narednoj tabeli.

Tabela 8. Opis osnovnih elemenata obrasca *Prototype*.

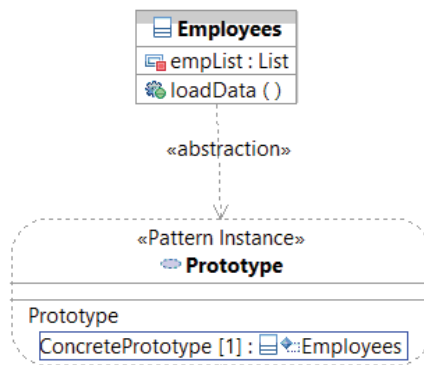
Elementi	Opis elemenata obrasca <i>Prototype</i>
Problem	Instanciranje velikih struktura klasa, koje se dinamički učitavaju, a za koje već postoje kreirani objekti.
Resursi	<i>Klijent (Client)</i> – kreira novi objekat tražeći od prototipa da se klonira ili koristi <i>PrototypeRegistry</i> za pristup instancama prototipa. <i>Prototip (Prototype)</i> – deklarise interfejs za kloniranje objekta. - Konkretni prototip (<i>ConcretePrototype</i>) – implementira operaciju za kloniranje. - Registar prototipa (<i>PrototypeRegistry</i>) – sadrži listu tipova koje je moguće klonirati i njihove ključeve.
Pravila	Interfejs <i>Prototype</i> sadrži metodu <i>clone()</i> koja kreira objekat trenutne klase i prenosi sve vrednosti polja starog objekta u novi. Objekat koji realizuje metodu <i>clone()</i> naziva se konkretni prototip (<i>ConcretePrototype</i>).
Smernice	- Kada želite da kreiranje objekata bude nezavisno od sistema. - Kada je kreiranje objekta dugotrajna i složena operacija. - Kada se klase instanciraju tokom izvršavanja.

Vežba 3-6. Primeniti osnovni tip implementacije Prototype obrasca za kloniranje klase Zaposleni.

Klasa Zaposleni (*Employees*) sadrži listu zaposlenih (*empList*) i metod *loadData()* koji učitava podatke o zaposlenima iz baze podataka. Postoji potreba da se podaci o zaposlenima više puta menjaju.

U ovoj vežbi ne bi bilo dobro da kreirate objekat koristeći ključnu reč *new* i da ponovo punite sve podatke iz baze podataka. Bolji pristup bi bio da klonirate postojeći objekat u novi objekat, a zatim da uradite izmenu nad podacima. Primenom *Prototype* obrasca, dijagram klasa bi izgledao kao na slici 36.

Slika 36. Dijagram klasa za kreiranje prototipa klase *Employees*.



Klasa *Employees* implementira, po difoltu, interfejs *Cloneable*.

```

import java.util.ArrayList;
import java.util.List;

public class Employees implements Cloneable {
    private List<String> empList;
    public void setEmpList(List<String> empList) {
        this.empList = empList;
    }
    public List<String> getEmpList() {
        return empList;
    }
    public Employees(List<String> empList) {
        this.empList = empList;
    }
    public Employees() {
        empList=new ArrayList<String>();
    }
    public void loadData(){
  
```

```

        //read all employees from database and put into the list
        empList.add("Ana Jovanovic");
        empList.add("Novak Djokovic");
        empList.add("David Albahari");
        empList.add("Goran Savic");
    }
    //the clone method is overridden to provide a deep copy of the
    employees list.
    public Object clone() throws CloneNotSupportedException{
        //throw new java.lang.UnsupportedOperationException();
        List<String> temp = new ArrayList<String>();
        for(String s:this.getEmpList()){
            temp.add(s);
        }
        return new Employees(temp);
    }
}

```

Glavna klasa *EmployeeMain*:

```

import java.util.List;
public class EmployeeMain {
    public static void main(String[] args) throws
CloneNotSupportedException {
        Employees emps=new Employees();
        emps.loadData();
        //use the clone method to get the Employee object
        Employees empsNew=(Employees) emps.clone();
        Employees empsDep=(Employees) emps.clone();
        List<String> list=empsNew.getEmpList();
        list.add("Nenad Jovovic");
        list.remove("Novak Djokovic");
        list.remove("Goran Savic");
        List<String> listDep=empsDep.getEmpList();
        listDep.remove("Ana Jovanovic");
        listDep.remove("David Albahari");
        System.out.println("Employees list: "+emps.getEmpList());
        System.out.println("Employees new list: "+list);
        System.out.println("Employees of the new department:"+listDep);
    }
}

```

Rezultat izvršavanja koda:

```

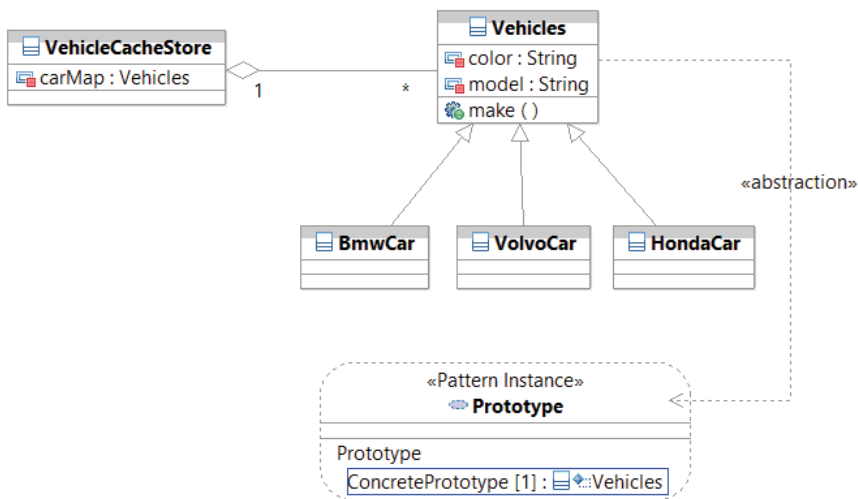
Employees list: [Ana Jovanovic, Novak Djokovic, David Albahari, Goran Savic]
Employees new list: [Ana Jovanovic, David Albahari, Nenad Jovovic]
Employees of the new department: [Novak Djokovic, Goran Savic]

```

Vežba 3-7. Primeniti implementaciju Prototype obrasca tipa *Prototype Registry* na aplikaciju koja klonira vozila.

Klasa *Vehicles* ima atribute boja (*color*) i model i apstraktnu metodu *make()*. Klonirana vozila su *BMWCar*, *HondaCar* i *VolvoCar*. Klasa tipa *PrototypeRegistry* u ovom primeru će biti imenovana kao *VehicleCacheStore*. S obzirom da je prevlačenjem Prototype obrasca, omogućen unos samo jednog elementa *Prototype* klase, u obrazac će biti unešena samo klasa *Vehicle*, a ostale klase koje je nasleđuju će biti dodate ručno (Slika 37).

Slika 37. Dijagram klasa za prototype obrazac *Vehicle*.



```

abstract class Vehicles implements Cloneable {
    private String color = null;
    private String model = null;

    public abstract void make();
    //getters and setters
    public String getColor() {
        return color;
    }
    public void setColor(String color) {
        this.color = color;
    }
    public String getModel() {
        return model;
    }
    public void setModel(String model) {
        this.model = model;
    }
}
  
```

```

    }
    // Clone the objects using Java in build functionality
    public Object clone(){
        Object clone = null;
        try{
            clone = super.clone();
        }
        catch (CloneNotSupportedException e) {
            e.printStackTrace();
        }
        return clone;
    }
    public String toString() {
        return "Vehicle cloned and creates "+this.color+", "+this.model;
    }
}

import java.util.HashMap;
import java.util.Map;
public class VehicleCacheStore {
    // Vehicle Cache for retrieve once copy needed.
    private static Map<String,Vehicles> carMap = new HashMap<String,
    Vehicles>();
    static {
        Vehicles bmwCar=new BmwCar();
        bmwCar.setColor("Red");
        bmwCar.setModel("BMW");
        carMap.put("BMW", bmwCar);
        Vehicles hondaCar=new HondaCar();
        hondaCar.setColor("White");
        hondaCar.setModel("HONDA");
        carMap.put("HONDA", hondaCar);
        Vehicles volvoCar=new VolvoCar();
        volvoCar.setColor("Blue");
        volvoCar.setModel("VOLVO");
        carMap.put("VOLVO",volvoCar);
    }
    //Clone vehicles Objects when required. All the concrete vehicle
    objects override clone() from Vehicles.
    //BmwCar, HondaCar, and VolvoCar create their own copy of the vehicle
    by calling the clone method.
    public static Vehicles getInstance(String carName){
        return (Vehicles) carMap.get(carName).clone();
    }
}

public class BmwCar extends Vehicles {
    public void make(){
        System.out.println("made BMW car");
    }
}

```

```
public class HondaCar extends Vehicles {  
    @Override  
    public void make() {  
        System.out.println("made Honda car");  
    }  
}
```

```
public class VolvoCar extends Vehicles {  
    @Override  
    public void make() {  
        System.out.println("made Volvo car");  
    }  
}
```

```
public class PrototypeClient {  
    // Once make a copy of vehicle then minor modification easy to apply  
    public static void main(String[] args) {  
        //Get BMW car copy  
        Vehicles car1 = VehicleCacheStore.getInstance("BMW");  
        System.out.println(car1);  
        //Get HONDA car copy  
        Vehicles car2 = VehicleCacheStore.getInstance("HONDA");  
        System.out.println(car2);  
        //Get VOLVO car copy  
        Vehicles car3 = VehicleCacheStore.getInstance("VOLVO");  
        System.out.println(car3);  
    }  
}
```

Izlazni rezultat:

Vehicle cloned and creates Red, BMW

Vehicle cloned and creates White, HONDA

Vehicle cloned and creates Blue, VOLVO

Zadatak 3-3. Primeniti odgovarajući obrazac projektovanja na aplikaciju koja klonira objekte tipa Osobe (Person) na Univerzitetu.

Objekti tipa Person na Univerzitetu mogu biti studenti ili zaposleni. Atributi apstraktne klase *Person* su *PersonID*, *Name*, *DoB* (*Date of Birth*). Klasa *Student* sadrži *IndexNo*, a klasa *Zaposleni* (*Employee*) sadrži *EmpID* i *Salary*. Klonirati objekat *Zaposleni*, promeniti iznos plate i prikazati rezultate.

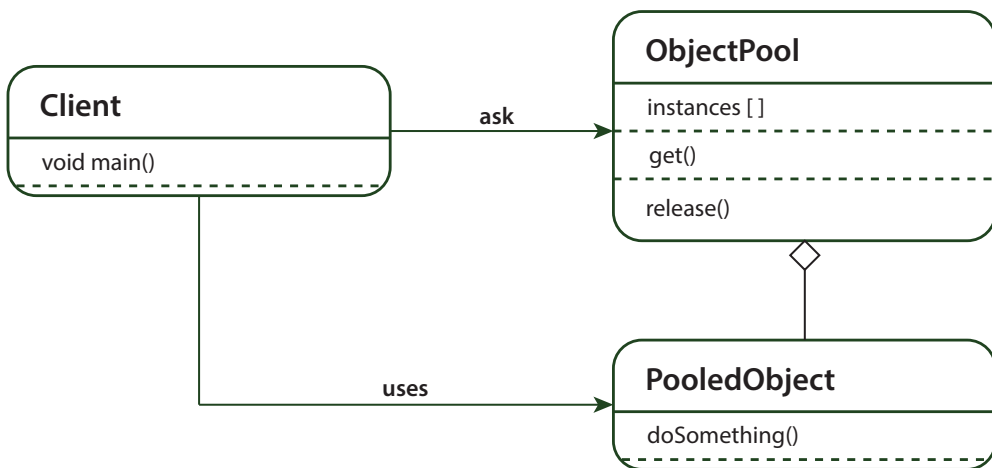
3.5. OBJECT POOL

Object Pool (kontejner objekata) je kreacioni obrazac koji se koristi u situacijama kada je kreiranje objekta složeno i skupo, a potreban je samo za kraći vremenski period. *Object Pool* obezbeđuje keš memoriju za instancirane objekte i prati koji se trenutno koriste, a koji su na raspolaganju. Jednostavno rečeno, *Object Pool* upravlja skupom objekata umesto da se ti objekti po zahtevu kreira i uništavaju (Wengner, 2023).

U osnovi, *ObjectPool* je kontejner koji sadrži određeni broj objekata. Kada se klijent preuzme objekat iz kontejnera, taj objekat više neće biti dostupan drugim klijentima sve dok se ne vrati nazad. Objekti u kontejneru, tj. klasi *ObjectPool* imaju svoj životni ciklus i to kreiranje, validacija i uništavanje.

Osnovna struktura *Object Pool* obrasca je prikazana na slici 38.

Slika 38. Osnovna struktura *Object Pool* obrasca (Java Design Pattern - Object Pool, 2018).



Osnovni elementi obrasca su:

- ♦ *ObjectPool*: Najvažnija klasa jer održava listu dostupnih i zahtevanih objekata.
- ♦ *PooledObject* ili *ReusablePool*: objekat koji je skup ili spor za inicijalizaciju ili ima ograničenu dostupnost, te treba da se čuva u *ObjectPool*-u.
- ♦ *Client*: Klijent klasa koja koristi objekat tipa *PooledObject*.

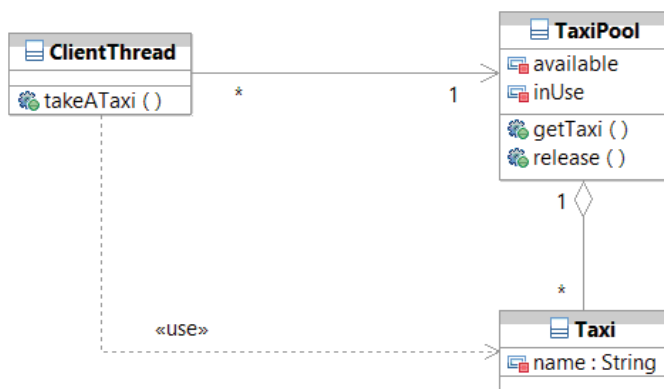
Vežba 3-8. Primeniti *ObjectPool* obrazac na Taxi aplikaciju.

Taxi kompanija ima ograničen broj taxi vozila i odgovorna je za: upravljanje statusom vozila (slobodno ili prevozi putnike), distribuiranje slobodnih vozila za preuzimanje putnika i produženje vremena čekanja u slučaju da su sva vozila zauzeta. Koristeći strukturu *ObjectPool* obrasca, klase ove taxi aplikacije bi bile (*Java Design Pattern - Object Pool*, 2018):

- *Taxi*: klasa koja definiše svojstva i metode taxi objekta.
- *TaxiPool*: ima ulogu taxi kompanije koja izvršava sledeće metode i attribute:
 - Metod *getTaxi()*: ukoliko se čeka dugo na slobodno taxi vozilo, može biti izbačen izuzetak ***TaxiNotFoundException***.
 - Metod *release()*: vraća taxi objekat u *pool*, nakon što je usluga završena.
 - Atribut *available*: čuva listu slobodnih taksija koji čekaju na uslugu.
 - Atribut *inUse*: čuva listu taksija koji su zauzeti.
- *ClientThread* klasa predstavlja klijente koji koriste taksi uslugu i simulira njihove pozive, prevoze i iskrcavanja.

Dijagram klasa za ovu taxi aplikaciju bi bio:

Slika 39. Dijagram klasa za taxi aplikaciju primenom *ObjectPool* obrasca¹.



¹ *ObjectPool* obrazac nije podržan korišćenom verzijom IBM RSA obrasca, te je direktno kreiran dijagram klasa, bez prevlačenja gotovih obrazaca.

Klasa Taxi:

```
public class Taxi {
    private String name;
    //getters and setters
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    //constructor using field
    public Taxi(String name) {
        super();
        this.name = name;
    }
}
```

Klasa TaxiPool:

```
import java.util.*;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.atomic.AtomicBoolean;
import java.util.concurrent.atomic.AtomicInteger;
public class TaxiPool {
    private static final long EXPIRED_TIME_IN_MILLISECOND = 1200; // 1.2s
    private static final int NUMBER_OF_TAXI = 4;
    private final List<Taxi> available = new ArrayList<Taxi>();
    private final List<Taxi> inUse = new ArrayList<Taxi>();
    private final AtomicInteger count = new AtomicInteger(0);
    private final AtomicBoolean waiting = new AtomicBoolean(false);
    public synchronized Taxi getTaxi() throws TaxiNotFoundException {
        if (!available.isEmpty()) {
            Taxi taxi = available.remove(0);
            inUse.add(taxi);
            return taxi;
        }
        if (count.get() == NUMBER_OF_TAXI) {
            this.waitingUntilTaxiAvailable();
            return this.getTaxi();
        }
        Taxi taxi = this.createTaxi();
        inUse.add(taxi);
        return taxi;
    }
    public synchronized void release(Taxi taxi) {
        inUse.remove(taxi);
        available.add(taxi);
        System.out.println(taxi.getName() + " is free");
    }
}
```

```
    }
    private Taxi createTaxi() {
        waiting(200); // The time to create a taxi
        Taxi taxi = new Taxi("Taxi " + count.incrementAndGet());
        System.out.println(taxi.getName() + " is created");
        return taxi;
    }
    private void waitingUntilTaxiAvailable() throws TaxiNotFoundException {
        if (waiting.get()) {
            waiting.set(false);
            throw new TaxiNotFoundException("No taxi available");
        }
        waiting.set(true);
        waiting(EXPIRED_TIME_IN_MILLISECOND);
    }
    private void waiting(long numberOfSecond) {
        try {
            TimeUnit.MILLISECONDS.sleep(numberOfSecond);
        } catch (InterruptedException e) {
            e.printStackTrace();
            Thread.currentThread().interrupt();
        }
    }
}
```

Klasa TaxiNotFoundException:

```
public class TaxiNotFoundException extends Exception {
    private static final long serialVersionUID = -6670953536653728443L;
    public TaxiNotFoundException(String message) {
        System.out.println(message);
    }
}
```

Klasa ClientThread:

```
import java.util.Random;
import java.util.concurrent.TimeUnit;
public class ClientThread implements Runnable {
    private TaxiPool taxiPool;
    public ClientThread(TaxiPool taxiPool) {
        this.taxiPool = taxiPool;
    }
    @Override
    public void run() {
        try {
            takeATaxi();
        } catch (InterruptedException e) {
            // TODO Auto-generated catch block
        }
    }
}
```

```

        e.printStackTrace();
    }
}
private void takeATaxi() throws InterruptedException {
    try {
        System.out.println("New client: " + Thread.currentThread().getName());
        Taxi taxi = taxiPool.getTaxi();
        TimeUnit.MILLISECONDS.sleep(randInt(1000, 1500));
        taxiPool.release(taxi);
        System.out.println("Served the client: " + Thread.currentThread().getName());
    } catch (TaxiNotFoundException e) {
        System.out.println(">>>Rejected the client: " + Thread.currentThread().getName());
    }
}
public static int randInt(int min, int max) {
    return new Random().nextInt((max - min) + 1) + min;
}
}

```

Glavna klasa TaxiApp:

```

public class TaxiApp {
    public static final int NUM_OF_CLIENT = 8;
    public static void main(String[] args) {
        TaxiPool taxiPool = new TaxiPool();
        for (int i = 1; i <= NUM_OF_CLIENT; i++) {
            Runnable client = new ClientThread(taxiPool);
            Thread thread = new Thread(client);
            thread.start();
        }
    }
}

```

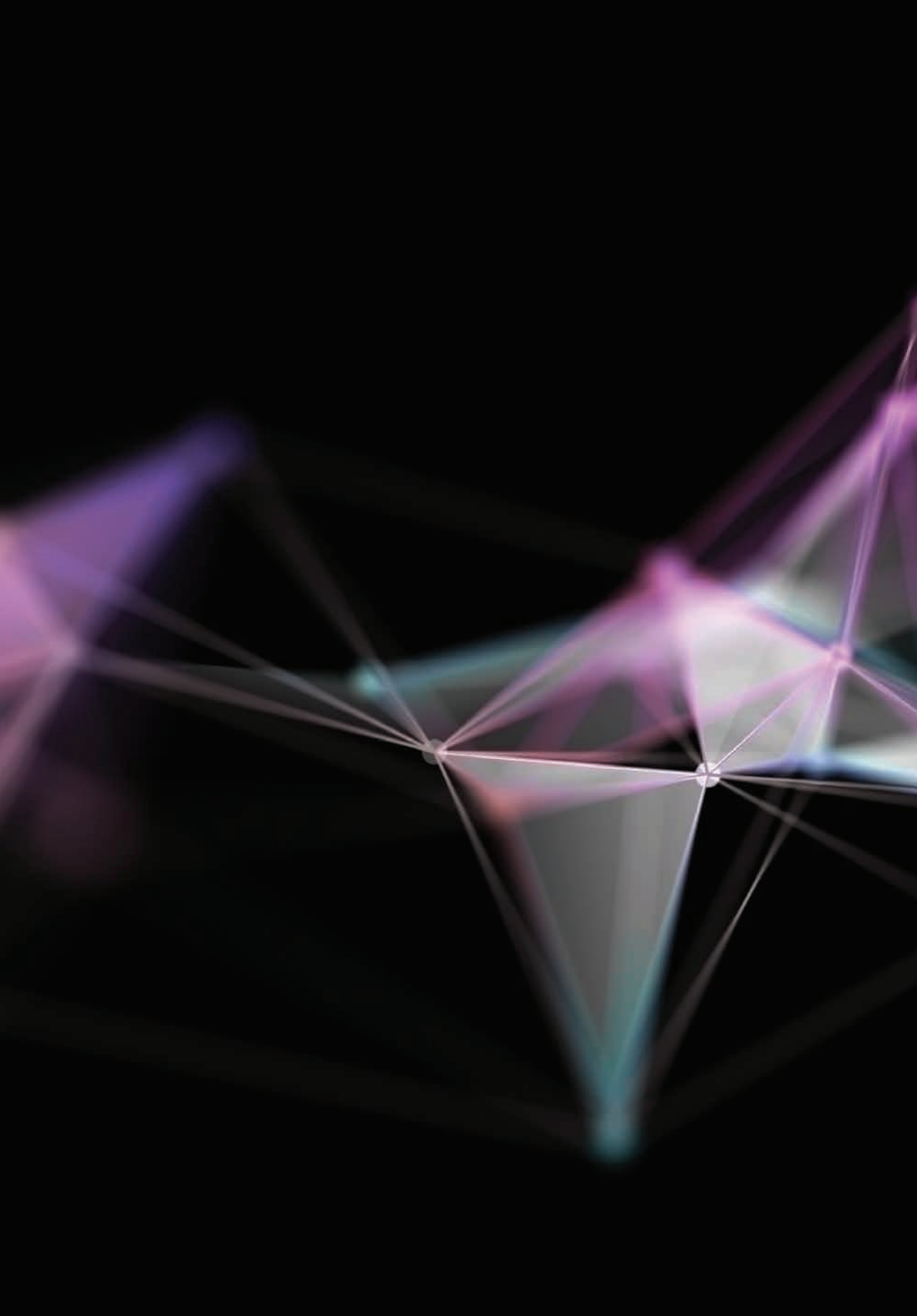
Rezultat:

```

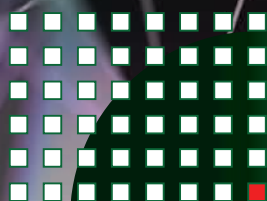
New client: Thread-3
New client: Thread-5
New client: Thread-2
New client: Thread-4
New client: Thread-6
New client: Thread-7
New client: Thread-8
New client: Thread-9
Taxi 1 is created
Taxi 2 is created
Taxi 3 is created
Taxi 4 is created

```

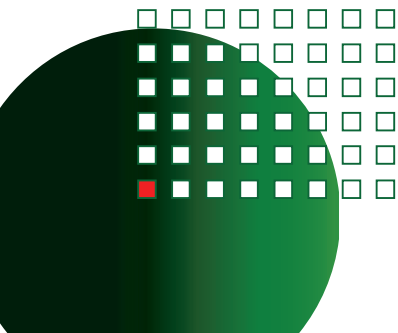
```
No taxi available
Taxi 1 is free
Served the client: Thread-3
Taxi 2 is free
Served the client: Thread-7
Taxi 3 is free
>>>Rejected the client: Thread-6
Taxi 4 is free
Served the client: Thread-8
Served the client: Thread-4
Taxi 2 is free
Served the client: Thread-9
Taxi 1 is free
Served the client: Thread-5
Taxi 3 is free
Served the client: Thread-2
```

POGLAVLJE 4



OBRASCI
STRUKTURALNOG
PROJEKTOVANJA

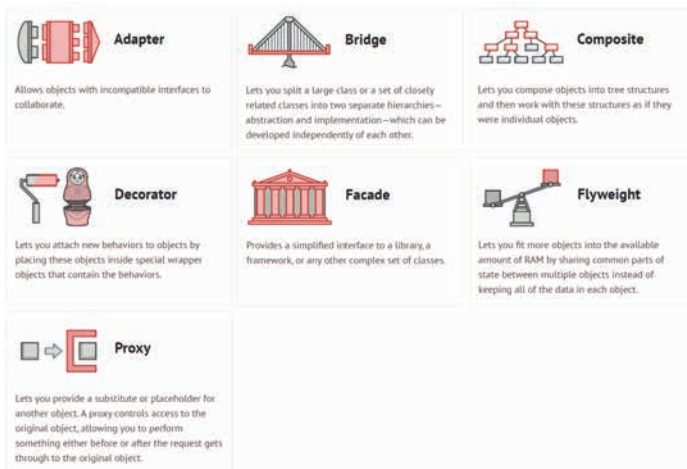


4. OBRASCI STRUKTURALNOG PROJEKTOVANJA

Obrasci strukturalnog projektovanja (*Structural design patterns*) rešavaju problem kombinovanja klasa u fleksibilne i efikasne strukture. Obrasci koji spadaju u ovu grupu su (Slika 40):

- ♦ **Adapter** – omogućava da objekti sa nekompatibilnim interfejsima međusobno sarađuju. Adapter (poznat i kao omotač) omogućava da postojeći objekat radi sa drugim objektom bez modifikacije njihovog izvornog koda.
- ♦ **Decorator** – dinamički pridružuje dodatne odgovornosti (funktionalnosti) objektima smeštajući te objekte u posebne objekte omotača.
- ♦ **Façade** – obezbeđuje pojednostavljen interfejs za set složenih klasa.
- ♦ **Bridge** – omogućava da se razdvoje velike klase ili set tesno povezanih klasa u dve odvojene hijerarhije – apstakcija i implementacije, a koje se mogu razvijati nezavisno jedna od druge.
- ♦ **Composite** – omogućava organizaciju klasa u vidu različitih struktura stabla i rad sa tim strukturama kao da su individualni objekti.
- ♦ **Flyweight** – koristi se u kodu koji zahteva kreiranje velikog broja objekata istovremeno, a gde se oskudeva sa RAM memorijom. Omogućava da se uklopi više objekata u raspoloživu količinu RAM-a, deljenjem zajedničkih delova stanja između više objekata, pre nego da se čuvaju svi podaci u svakom objektu. Objekti na koje je primenjen ovaj obrazac zauzimaju manje memorije, čime oslobađa prostor za druge funkcije koje treba izvršiti., što čini kod efikasnijim, bržim i upotrebljivijim.
- ♦ **Proxy** – obezbeđuje zamenu za drugi objekat. Proxy kontroliše pristup originalnom objektu, omogućavajući da se nešto izvrši pre ili nakon što zahtev dođe do originalnog objekta.

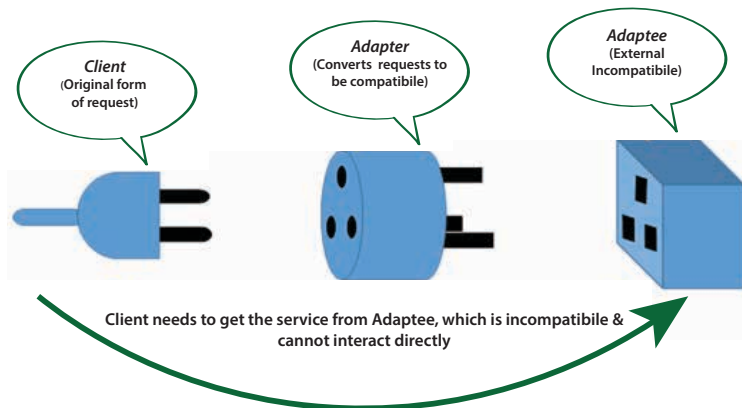
Slika 40. Obrasci strukturalnog projektovanja.



4.1. ADAPTER

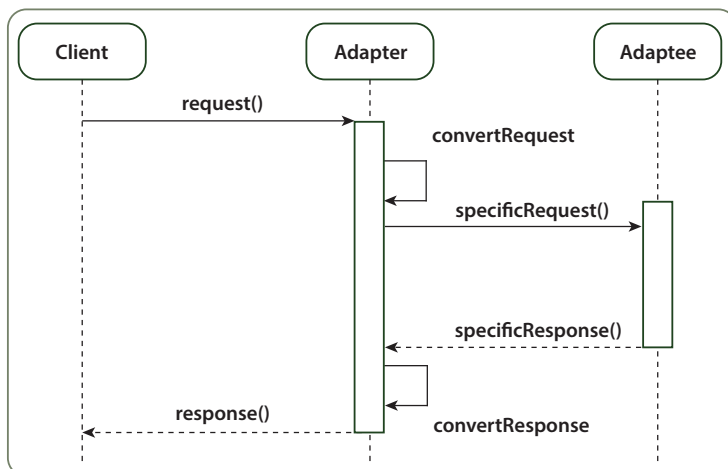
Ovaj obrazac je lako razumeti jer je realni svet pun adaptera (Slika 41). Na primer, imamo *Ethernet* interfejs na jednom kraju i USB na drugom. Pošto su međusobno nespojive, koristimo adapter koji pretvara jedan u drugi. Ovaj primer je prilično analogan objektno orijentisanim adapterima.

Slika 41. Poređenje obrasca Adapter sa realnim primerom u praksi.

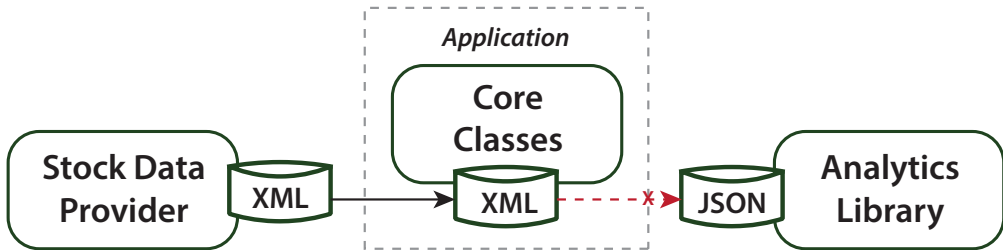


Adapter omogućava da objekti sa nekompatibilnim interfejsima međusobno sarađuju. Adapter (poznat i kao omotač) omogućava da postojeći objekat radi sa drugom objektom bez modifikacije njihovog izvornog koda. Na slici 42, rad Adapter obrasca je objašnjen pomoću dijagrama sekvenci.

Slika 42. Rad adapter obrasca objašnjen na dijagramu sekvenci.

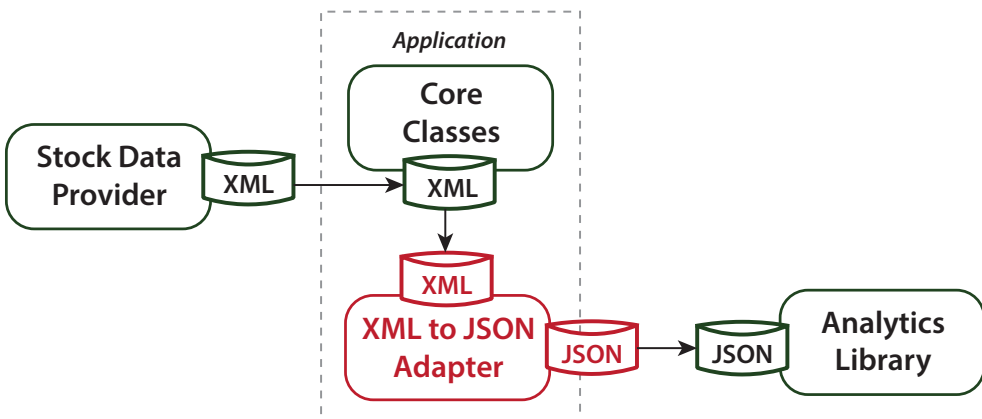


Da bi bolje objasnili rad ovog obrasca, zamislite da pravite aplikaciju za praćenje akcija na berzama. Aplikacija preuzima podatke sa berzi iz različitih izvora u XML formatu, a zatim prikazuje lepe grafikone i dijagrame krajnjem korisniku. U nekom trenutku, odlučili ste da poboljšate aplikaciju tako što ćete integrisati pametnu biblioteku za analitiku od trećeg izvora. Međutim, analitička biblioteka radi samo sa podacima u JSON formatu.



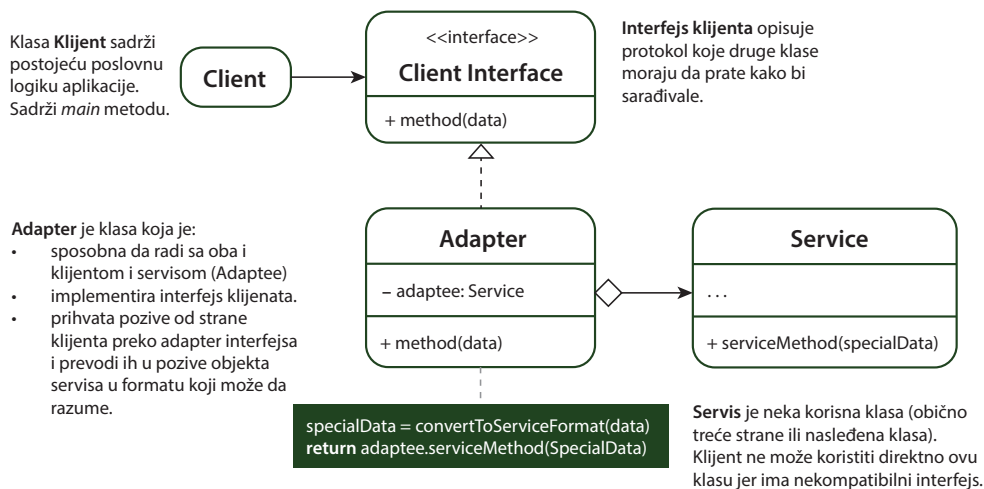
Možda bi vam prva ideja bila da promenite biblioteku tako da radi sa XML-om. Međutim, ovo bi moglo pokvariti neki postojeći kôd koji se oslanja na biblioteku. Što je još gore, možda nećete imati pristup izvornom kodu biblioteke.

Možete da kreirate adapter koji pretvara XML u JSON. Ovo je poseban objekt koji pretvara interfejs jednog objekta tako da drugi objekat može da ga razume. To funkcioniše tako što adapter dobija interfejs, koji je kompatibilan sa jednim od postojećih objekata. Pomoću ovog interfejsa postojeći objekat može bezbedno pozvati metode adaptera. Nakon prihvatanja poziva, adapter prosleđuje zahtev drugom objektu, ali u formatu i redosledu koji drugi objekat očekuje. Moguće je kreirati i dvosmerni adapter koji može pretvoriti pozive u oba smera.



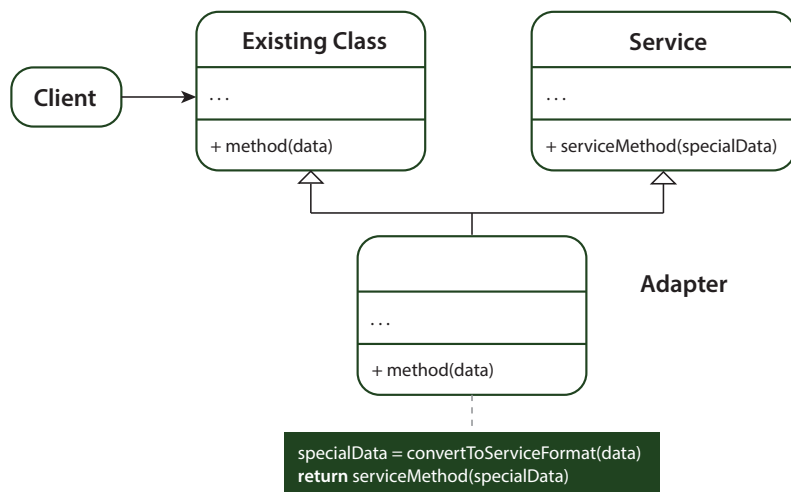
Osnovna struktura jednog načina implementacije ovog obrasca je prikazan na sledećoj slici.

Slika 43. Osnovna struktura jednog načina implementacije Adapter obrasca.



Drugi način implementacije Adapter obrasca koristi nasleđivanje, odnosno adapter nasleđuje interfejsе obe klase u isto vreme (Slika 44). Klasa Adapter ne treba da umotava klase jer nasleđuje ponašanja obe klase. Obratiti pažnju da ovaj pristup može biti implementiran samo sa programskim jezicima koji podržavaju višestruko nasleđivanje klase, kao što je C++.

Slika 44. Osnovna struktura Adapter obrasca sa primenom nasleđivanja (Shvets, 2019).



Osnovne elemente ovog obrasca će se prikazati kroz narednu vežbu.

Vežba 4-1. Primeniti Adapter obrazac.

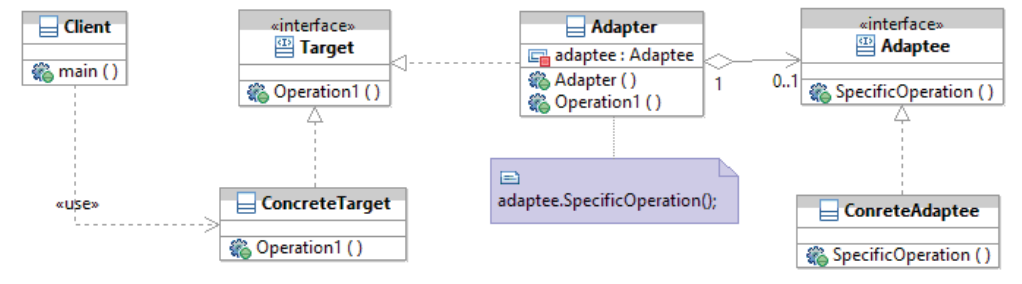
Opis problema:

Aplikacija *AudioPlayer* trenutno reprodukuje audio fajlove u mp3 formatu. Potrebno je proširiti ovu aplikaciju tako da može da reprodukuje i vlc i mp4 formate (*Design Patterns - Adapter Pattern, n.d.*).

Opis rešenja:

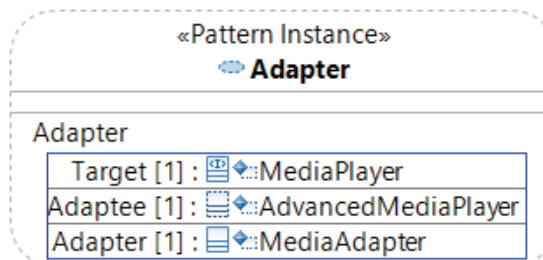
Sagledavajući osnovnu strukturu Adapter obrasca (Slika 45), *ConcreteTarget* u ovom primeru je *AudioPlayer* koji će implementirati interfejs tipa *Target*, koga možemo nazvati **MediaPlayer**. Ovom interfejsu dodaćemo samo jedan metod *play()*. Klasu tipa Adapter ćemo imenovati kao *MediaAdapter*. Interfejs Adaptee ćemo imenovati kao *AdvancedMediaPlayer* koji može da reprodukuje vlc i mp4 formate. Ovaj interfejs realizuju klase *VlcPlayer* i *Mp4Player*.

Slika 45. Detaljnija osnovna struktura Adapter obrasca primenjena.

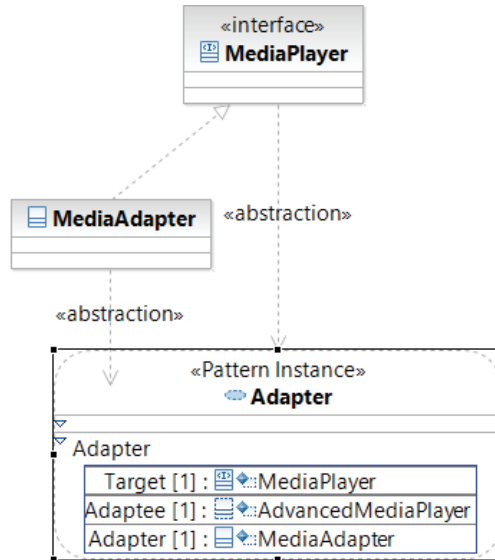


Koraci rada na ovoj vežbi su:

Korak 1. Kreirati projekat, dijagram klasa i prevući i popuniti Adapter obrazac.



Korak 2. Povezati (*bind*-ovati) klase sa obrascem i prikazati dijagram klasa.

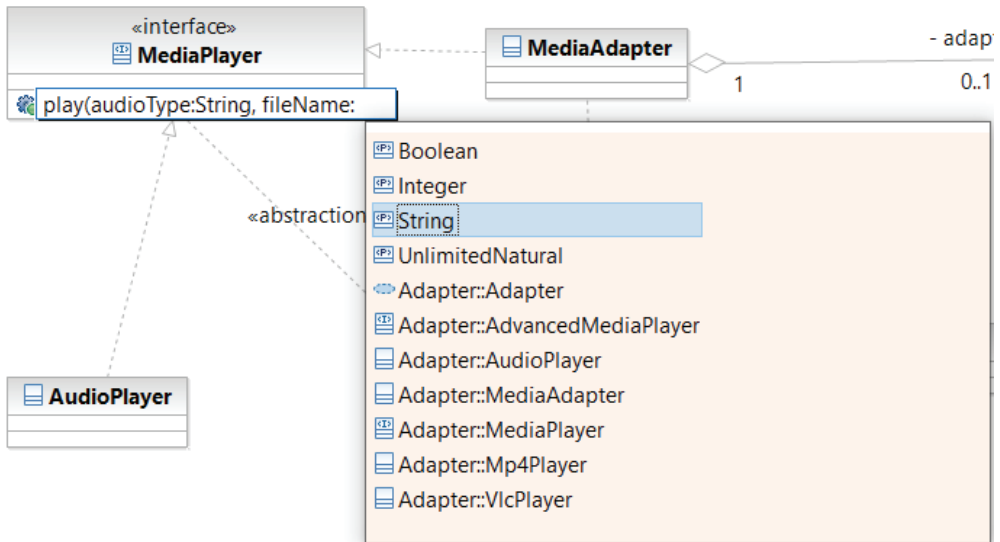


Na generisanom dijagramu klasa, vidi se da nisu prikazane ostale klase, jer nam se daje sloboda da osnovnu strukturu implementiramo na način kako nama odgovara. S toga ćemo preurediti dobijeni dijagram klasa i dodati klase, na sledeći način:

- ♦ S obzirom da je već kreiran interfejs *AdvancedMediaPlayer*, samo ćete ga prevući sa prozora *Project Explorer*, na dijagram.
- ♦ Dodajte klasu *AudioPlayer* i povežite ga sa klasom *MediaPlayer*, relacijom *Realization*.
- ♦ Dodajte i klase *VlcPlayer* i *Mp4Player* i povežite ih sa interfejsom *AdvancedMediaPlayer*.

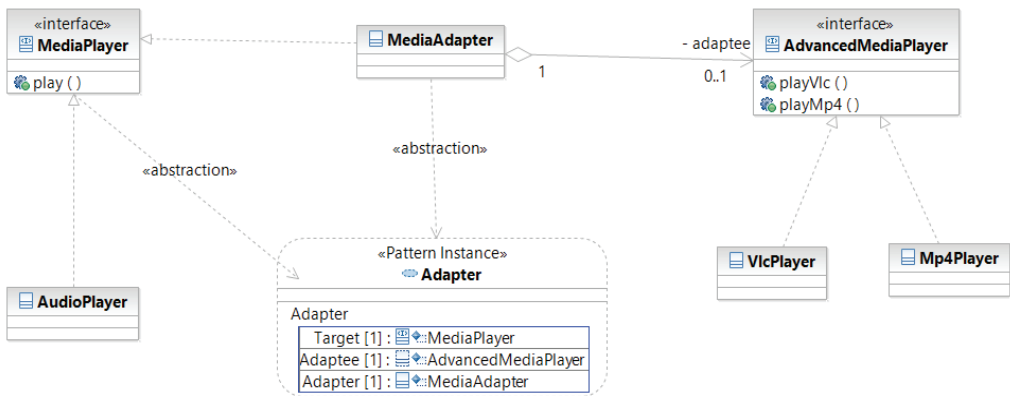
U interfejse ćete uneti odgovarajuće metode i to:

- ♦ u klasi *MediaPlayer*, dodajte metodu *play()* sa parametrima *audioType* tipa *String* i *fileName* tipa *String*. Rezultat aplikacije će prikazati koji format medija i naziv fajla (*fileName*) se trenutno reprodukuje. Parametre dodajete tako što nakon unosa metode otvorite zagradu, ukucate naziv parametra, odmah nakon toga unosite dvotačku, nakon čega će se otvoriti padajući meni kako bi selektovali odgovarajući tip polja – u ovom slučaju to je *String*. Kada završite sa unosom parametara, zatvorite zagradu. Program neće prikazati parametre na dijagramu, već su oni vidljivi kada u okviru *Project Explorer* prozora, proširite klasu u okviru kojeg je dodata metoda, a zatim proširite i tu metodu. Drugi način, da vidite da li ste uspešno dodali parametre jeste u *Properties* prozoru.



- U klasi *AdvancedMediaPlayer*, dodajte metode: *playVlc (fileName:String)* i *playMp4 (fileName:String)*. Parametar *fileName* je dovoljan, jer sam naziv metode govori koji format medija će biti reprodukovano.

Finalni izgled dijagrama klasa je:



Korak 3. Izgenerisati kod, dopuniti i prikazati rezultat rada.

Interfejsi *MediaPlayer* i *AdvancedMediaPlayer* su finalni i u njih nećemo ništa dodavati:

```
public interface MediaPlayer {
    public void play(String audioType, String fileName);
}
public interface AdvancedMediaPlayer {
    public void playVlc(String fileName);
    public void playMp4(String fileName);
}
```

Play() metode kod klasa *VlcPlayer* i *Mp4Player* ćemo dopuniti sa prikazom odgovarajućeg naziva fajla:

```
public class VlcPlayer implements AdvancedMediaPlayer {
    public void playVlc(String fileName) {
        System.out.println("Playing VLC file. Name: "+fileName);
    }
    public void playMp4(String fileName) {
        //do nothing
    }
}
public class Mp4Player implements AdvancedMediaPlayer {
    public void playVlc(String fileName) {
    }
    public void playMp4(String fileName) {
        System.out.println("Playing MP4 file. Name: "+fileName);
    }
}
```

Ukoliko pogledamo klasu *MediaAdapter* videćemo da je izgenerisano polje adaptee, koji je tipa *AdvancedMediaPlayer* kao i njemu prateće *get* i *set* metode. Prikazana je prazna *play* metoda, koju realizuje od interfejsa *MediaPlayer*. Prikazani su konstruktori, parametarski i neparametarski.

```
public class MediaAdapter implements MediaPlayer {
    private AdvancedMediaPlayer adaptee;
    public AdvancedMediaPlayer getAdaptee() {
        return adaptee;
    }
    public void setAdaptee(AdvancedMediaPlayer adaptee) {
        this.adaptee = adaptee;
    }
    public void play(String audioType, String fileName) {
    }
}
```

```

    public MediaPlayer() {
    }
    public MediaPlayer(AdvancedMediaPlayer aAdvancedMediaPlayer) {
        this.adaptee = aAdvancedMediaPlayer;
    }
}

```

Dopunićemo metodu *play()* i dodati parametarski konstruktor kome se prosleđuje *audioType* polje.

- ♦ Metoda *play()* će u zavisnosti od prosleđenog tipa *audioType*, pozivati odgovarajuću metodu *play()*.
- ♦ Parametarski konstruktor će u zavisnosti od *audioType*-a instancirati odgovarajuće klase.

```

public class MediaPlayer implements MediaPlayer {
    private AdvancedMediaPlayer adaptee;
    public AdvancedMediaPlayer getAdaptee() {
        return adaptee;
    }
}

```

```

    public void setAdaptee(AdvancedMediaPlayer adaptee) {
        this.adaptee = adaptee;
    }
    public void play(String audioType, String fileName) {
        if(audioType.equalsIgnoreCase("vlc")){
            adaptee.playVlc(fileName);
        }
        else if (audioType.equalsIgnoreCase("mp4")) {
            adaptee.playMp4(fileName);
        }
    }
}

```

```

    public MediaPlayer() {
    }
    public MediaPlayer(AdvancedMediaPlayer aAdvancedMediaPlayer) {
        this.adaptee = aAdvancedMediaPlayer;
    }
}

```

```

    public MediaPlayer(String audioType){
        if(audioType.equalsIgnoreCase("vlc")){
            adaptee=new VlcPlayer();
        }
        else if (audioType.equalsIgnoreCase("mp4")) {
            adaptee=new Mp4Player();
        }
    }
}

```

Ukoliko pogledamo klasu *AudioPlayer* videćemo da je samo izgenerisana metoda *play*:

```
public class AudioPlayer implements MediaPlayer {
    public void play(String audioType, String fileName) {
    }
}
```

S obzirom da je ovo konkretna target klasa koju će klijent da poziva, u njoj ćemo dodati sve moguće uslove tj. tipove fajla (*audioType*) koji mogu da se zahtevaju od strane klijenta (*main* metode) i proslediti je Adapter klasi tj. *MediaAdapter*. Dopunjena klasa, sada izgleda ovako:

```
public class AudioPlayer implements MediaPlayer {
    MediaAdapter mediaAdapter;
    public MediaAdapter getMediaAdapter() {
        return mediaAdapter;
    }
    public void setMediaAdapter(MediaAdapter mediaAdapter) {
        this.mediaAdapter = mediaAdapter;
    }
    public void play(String audioType, String fileName) {
        if(audioType.equalsIgnoreCase("mp3")){
            System.out.println("Playing mp3 file. Name: "+fileName);
        }
        else if (audioType.equalsIgnoreCase("vlc")|| audioType.
            equalsIgnoreCase("mp4")) {
            mediaAdapter=new MediaAdapter(audioType);
            mediaAdapter.play(audioType, fileName);
        }
        else {
            System.out.println("Invalid media. "+audioType+" format
            not supported!");
        }
    }
}
```

Na kraju ćemo kreirati *main* metodu i izvršiti kod:

```
public class AdapterPatternDemo {
    public static void main(String[] args) {
        AudioPlayer audioPlayer=new AudioPlayer();
        audioPlayer.play("mp3", "Iznad horizonta.mp3");
        audioPlayer.play("mp4", "alone.mp4");
        audioPlayer.play("vlc", "far far away.vlc1");
        audioPlayer.play("avi", "Iznad horizonta.avi");
    }
}
```

Klikom na *Run as > Java Application* izvršavamo kod:

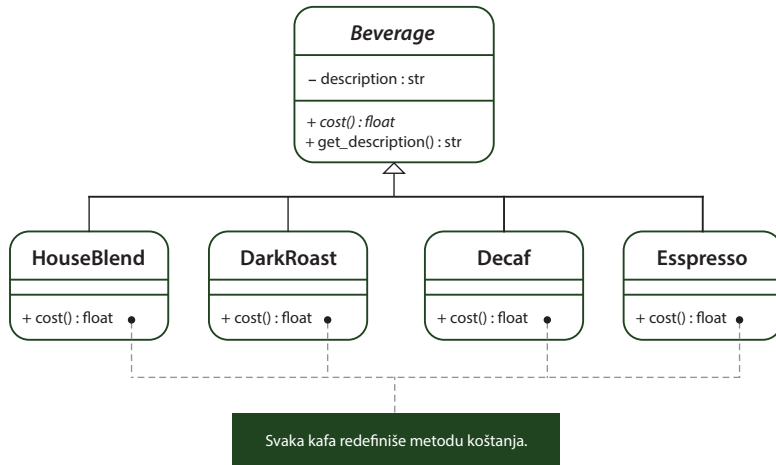
```
Playing mp3 file. Name: Iznad horizonta.mp3  
Playing MP4 file. Name: alone.mp4  
Playing VLC file. Name: far far away.vlc1  
Invalid media. avi format not supported!
```

Zadatak 4-1. Primeniti Adapter obrazac za prikaz brzine kola u MPH i km/h.

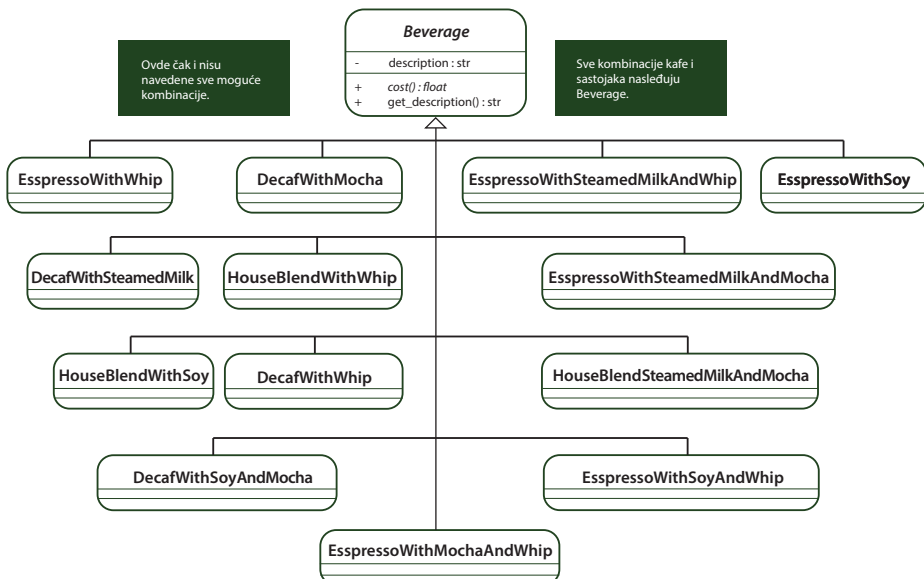
Postojeća aplikacija, koja je razvijena za SAD tržište, vraća najveću brzinu luksuznih kola u miljama na sat (*Miles Per Hour*, MPH). Kako se tržište proširilo, neophodno je istu aplikaciju koristiti i za klijenta u Velikoj Britaniji, koji želi iste rezultate, ali u kilometrima na sat (km/h). Upotrebiti odgovarajući obrazac koji će prikazati maksimalnu brzinu kola *Porsche Spyder* u km/h, ukoliko je poznato da *PorscheSpyder* ide 214 mph. Funkcija `convertMPHtoKMPH` vraća $\text{mph} * 1.60934$.

4.2. DECORATOR

Otvorili ste kafić. U vašem kafiću se služe razna pića, ali za sada ste započeli biznis samo sa kafama. Svaka kafa ima svoj opis i cenu. Neki od tipova kafa koje nudite su: HouseBlend, DarkRoast, Decaf i Espresso. Početni model ove vaše zamisli je:

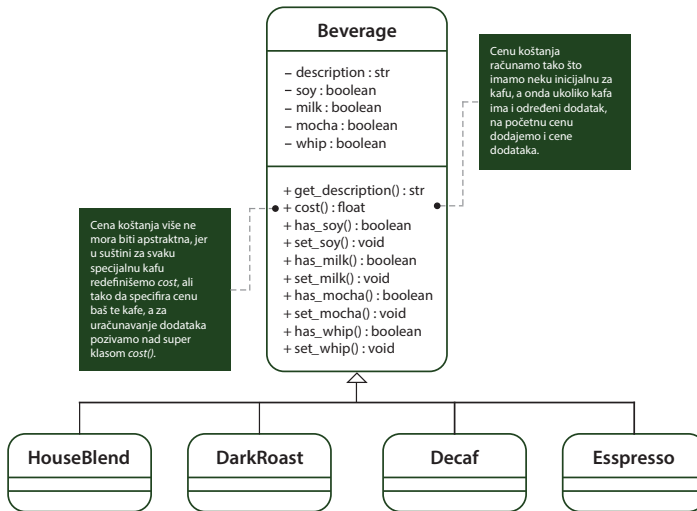


Međutim vaše mušterije ne žele samo kafu, već bi da dodaju i čokoladu, šlag, mleko, sojino mleko i mnoge druge dodatke. Kako bi se za svaki ovaj proizvod (sa dodacima) znala cena, potrebno je uračunati i cenu dodataka i cenu kafe. Jedno od rešenja bi bilo:



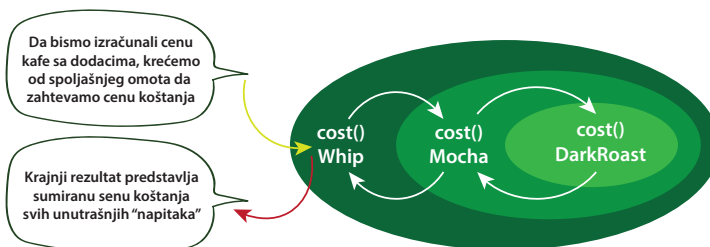
Nasleđivanjem možete rešiti dati problem, ali šta ćete sa ovoliko klasa? Šta ako se doda novi dodatak kafi, na primer kolačić, ili ako se doda nova kafa? Dobija se eksplozija novih klasa. Primenom nasleđivanja se nasleđuje i ponašanje od super klase i to ponašanje se stiče u trenutku kompajliranja aplikacije. Pitate se da li postoji neki drugi način, sa manje klasa.

Možda ste došli na ideju da bi vam bilo jednostavnije da promenite strukturu klase *Beverage* i da joj dodate da ima atribut koji će za dodatke imati *boolean* vrednosti ako su prisutne u kafi. Cena bi se onda mogla računati spram toga koji je dodatak prisutan, pa da se za svaki dodatak uvećava i cena kafe spram cene dodatka. Izgled ovog rešenja bi bilo:



Problemi ovog rešenja bi bili da ukoliko promenite cenu dodatka, morate menjati ceo kod. Ukoliko dodate novi dodatak, za njega se moraju dodati i metode u super klasi. Da li uz sve napitke ide dobro šlag? Šta ako u vaš meni uvrstite ledeni čaj? Šta ako mušterija želi kafu sa duplom čokoladom? Ovo su samo neki od problema ovog rešenja.

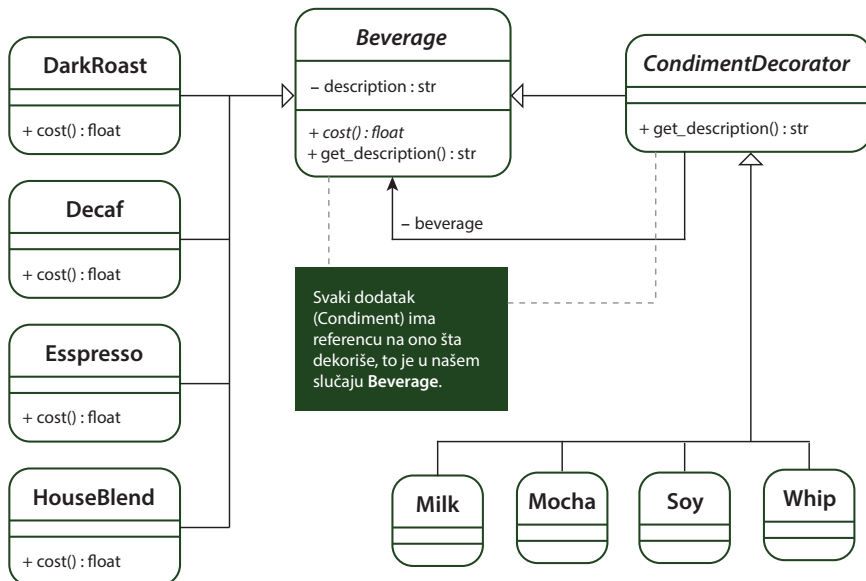
Rešenje je primena Decorator obrasca. Na primer, za mušteriju koja želi da poruči *DarkRoast* sa *Mocha* i *Whip*, uz pomoć dekoratora, dekorisali bi klasu *DarkRoast* prvo sa *Mocha*, pa potom taj dekorisani objekat dekorisali i sa *Whip* dekoratorom.



Započinjemo sa *Beverage* interfejsom i dekorišemo ga sa dodacima u toku izvršavanja:

- ♦ Uzmite *DarkRoast* objekat
- ♦ Dekorišite sa *Mocha* objektom
- ♦ Dekorišite sa *Whip* objektom
- ♦ Pozovite metod *cost()* i prethodno delegiranje će dodati cene dodataka.

Finalni izgleda organizacije klasa primenom ovog obrasca bi bio:

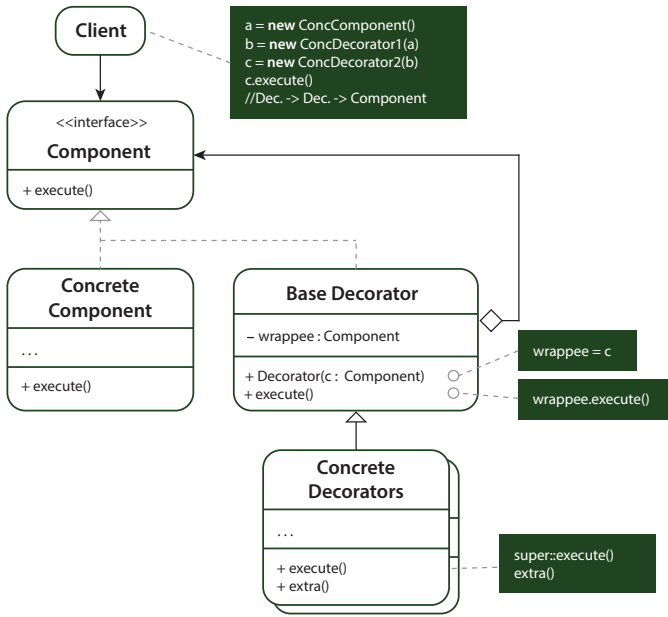


Dekorator (*CondimentDecorator*) ima referencu na objekat koji dekoriše (*beverage*). U njegovim metodama, na primer za *cost()*, se poziva *cost()* nad dekorisanim objektom, a zatim dodaje i cena samog objekta.

U slučaju metode *get_description()* takođe se poziva nad dekorisanim objektom *get_description()*, a zatim se na taj opis dodaje i neki *String* koji opisuje sam dekorator. Specijalni tipovi dekoratora redefinišu neophodne metode.

Osnovna struktura *Decorator* obrasca bi bila:

Slika 46. Osnovna struktura *Decorator* obrasca (Shvets, 2019).

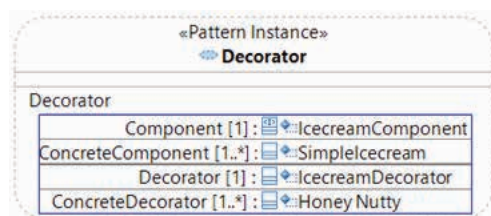


Vežba 4-2. Primeniti obrazac nad aplikacijom za opis i cenu sladoleda sa dodacima.

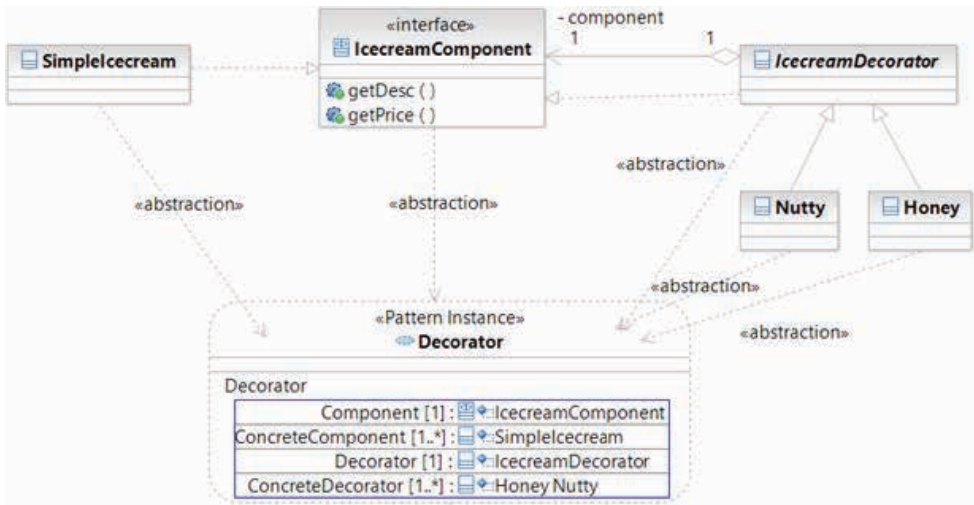
Aplikacija treba da prikaže opis sladoleda sa mogućim dodacima i finalnu cenu. Klijent pored osnovnog sastojka sladoleda (*SimpleIcecream*), može zatražiti dodatke i to med (*Honey*), kao i orašaste dodatke (*Nutty*). Osnovni sastojak sladoleda (*SimpleIcecream*) prikazuje cenu (150 din) i opis. Klase *Honey* i *Nutty* prikazuju dodatak od po 20 din. Izračunati i prikazati ukupnu cenu sladoleda sa dodacima koje je klijent poručio.

Koraci rešenja:

Korak 1. Prevući Decorator obrazac i popuniti ga sa klasama.



Korak 2. Prikazati relacije i dopuniti interfejs sa metodama.



Korak 3. Generisati kod. Obratiti pažnju na tip podataka koji metode vraćaju.

```

public interface IcecreamComponent {
    public String getDesc();
    public double getPrice();
}

public class SimpleIcecream implements IcecreamComponent {
    public String getDesc() {
        return "Osnova sladoleda: 150 din";
    }
    public double getPrice() {
        return 150;
    }
}

//obratiti pažnju na protected polje!
public abstract class IcecreamDecorator implements IcecreamComponent {
    protected IcecreamComponent component;
    public IcecreamComponent getComponent() {
        return component;
    }
    public void setComponent(IcecreamComponent component) {
        this.component = component;
    }
}

```

```

    public IcecreamDecorator(IcecreamComponent aIcecreamComponent) {
        this.component = aIcecreamComponent;
    }
}

public class Nutty extends IcecreamDecorator {
    public Nutty(IcecreamComponent aIcecreamComponent) {
        super(aIcecreamComponent);
    }
    public String getDesc() {
        return component.getDesc()+" , Nutty: 20 din";
    }
    public double getPrice() {
        return component.getPrice()+20;
    }
}

public class Honey extends IcecreamDecorator {
    public Honey(IcecreamComponent aIcecreamComponent) {
        super(aIcecreamComponent);
    }
    public String getDesc() {
        return component.getDesc()+" , Honey: 20";
    }
    public double getPrice() {
        return component.getPrice()+20;
    }
}

```

Korak 4. Kreirati glavnu klasu.

```

public class Client {
    public static void main(String[] args) {
        IcecreamComponent icecream = new SimpleIcecream();
        icecream = new Honey(icecream);
        icecream = new Nutty(icecream);
        System.out.println("Opis "+"\\n"+icecream.getDesc());
        System.out.println("Cena: "+icecream.getPrice());
    }
}

```

Korak 5. Izvršiti kod:

Opis

Osnova sladoleda: 150 din, Honey: 20, Nutty: 20 din

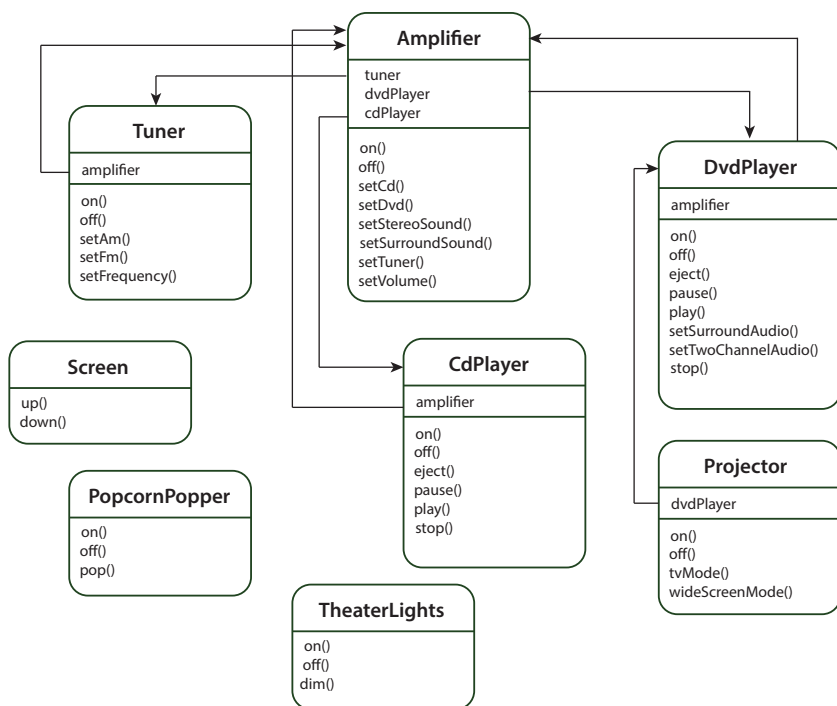
Cena: 190.0

Zadatak 4-2. Primeniti odgovarajući obrazac na aplikaciju za rentiranje kola.

Deo aplikacije za rentiranje kola treba da omogući prikaz osobina rentiranog vozila. Naime korisnik može da zahteva da se na rentirano vozilo postavi dodatna oprema, kao što su sedišta za dete i wifi mobilni ruter. Rentacar agencija nudi dva osnovna tipa vozila: sportska kola i luksuzna kola. Potrebno je prikazati opis rentiranog vozila sa dodatnom opremom (getDesc) i cenu vozila (getPrice).

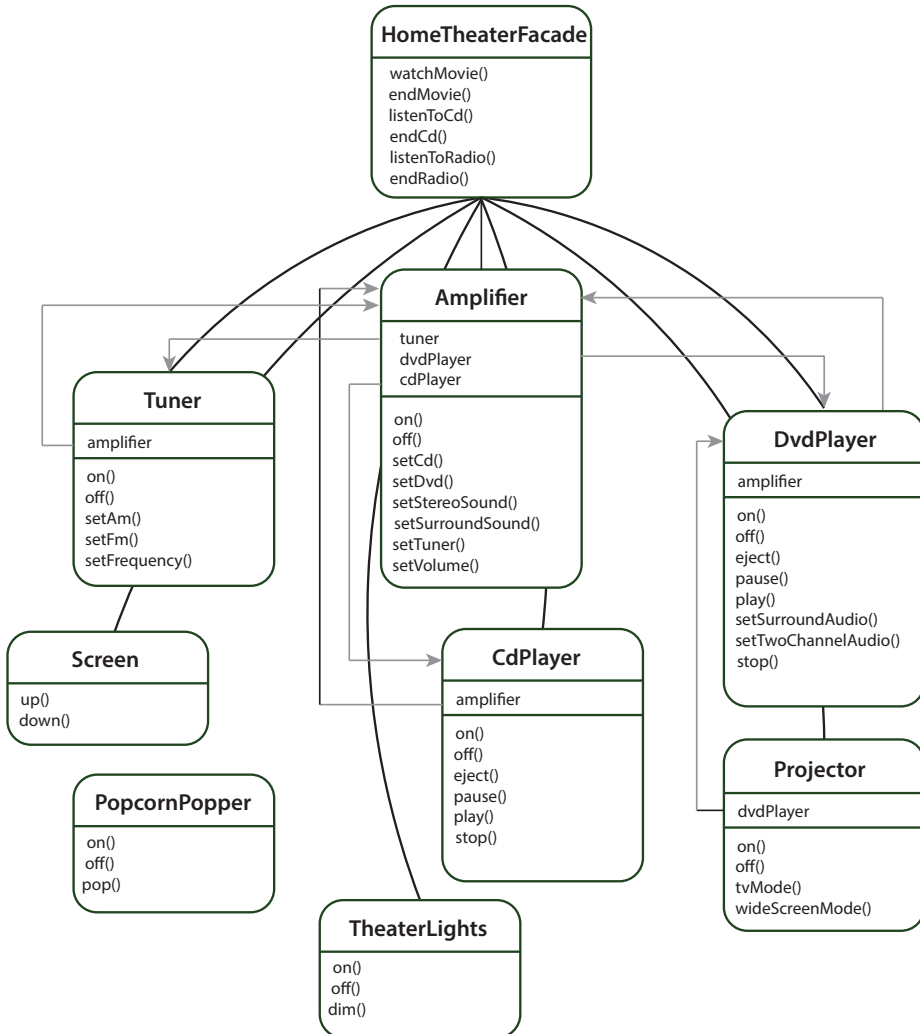
4.3. FACADE

Zamislite da vaš kôd mora da radi sa velikim brojem objekata. Na primer, sistem kućnog bioskopa koji sadrži sledeće komponente:



U tom slučaju, morate da inicijalizujete sve te objekte, pratite njihove zavisnosti, izvršavate metode u određenim redosledom itd. Na taj način vaše klase će biti tesno povezane sa detaljima o implementaciji klasa treće strane. Sve ovo znatno otežava razumevanje i održavanje aplikacije.

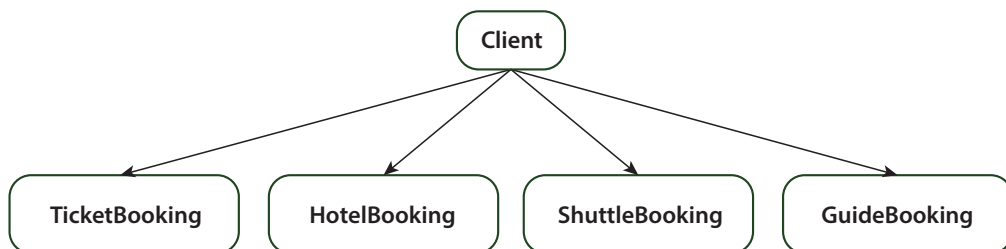
Jedno od rešenja je primena Facade obrasca. Facade je klasa koja omogućava jednostavan interfejs ka složenim podsistemima. Predstavlja način za „maskiranje“ složenosti objekata kreiranjem klase koja će izvršiti sve neophodne proračune za klijenta, s tim što on ne zna šta se zaista dešava u pozadini.



Facade klasa može da pruži ograničeni set funkcionalnosti u poređenju sa direktnim pristupom podsistemu. Međutim, *Facade* uključuje samo one funkcionalnosti koje su zaista neophodne klijentu.

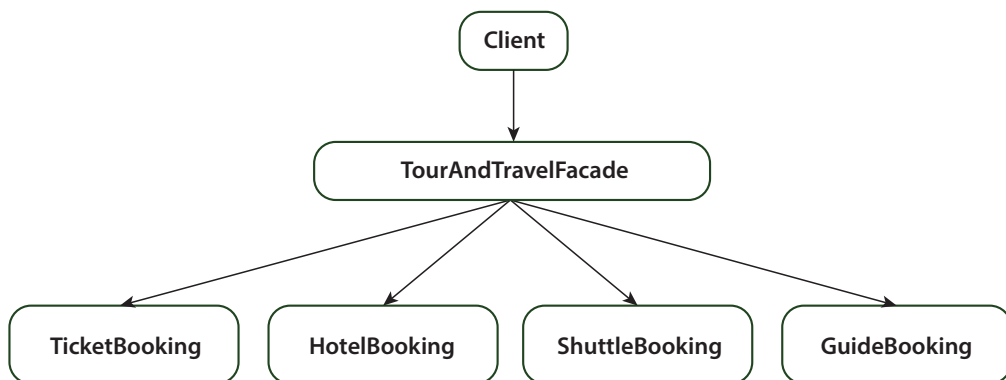
Facade se posebno koristi u slučajevima kada je potrebno da se vaša aplikacija integriše sa nekom složenom bibliotekom koja ima puno funkcionalnosti, a vama je potreban samo mali deo toga.

Na primer, aplikacija za upravljanje putovanjima uključuje sledeće podsisteme: *Ticketing Booking*, *HotelBooking*, *ShuttleBooking* i *GuideBooking*.



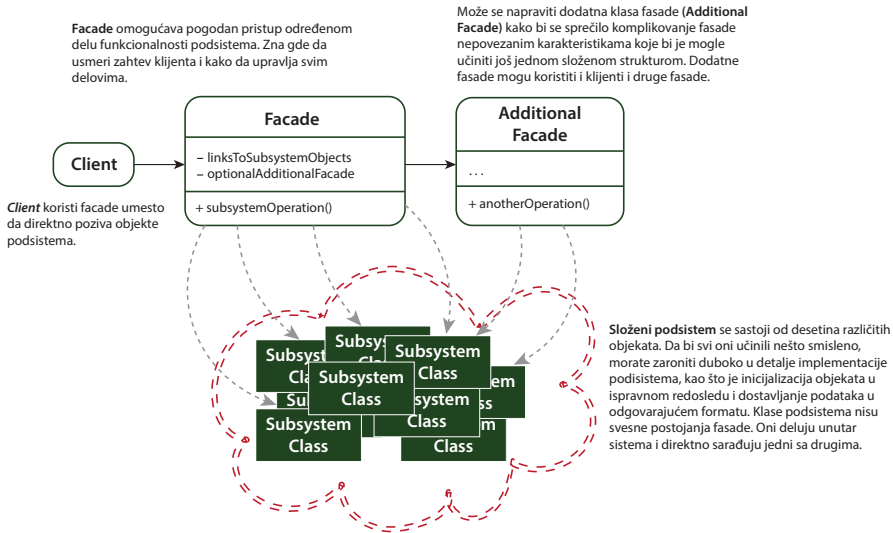
Da bi koristio ove sisteme, klijent mora direktno da komunicira sa svakim od njih. Ukoliko postoji bilo koja promena (npr. dodavanje, brisanje ili ažuriranje) neke nove usluge u ovim podsistemima, onda je potrebno promeniti i klijent aplikaciju - što nije dobar pristup.

U ovom slučaju rešenje je primena *Facade* klase kao interfejsa između klijenta i podsistema. Sva komunikacija klijenta sa podsistemima ide preko *Facade* klase.



Osnovna struktura *Facade* obrasca je:

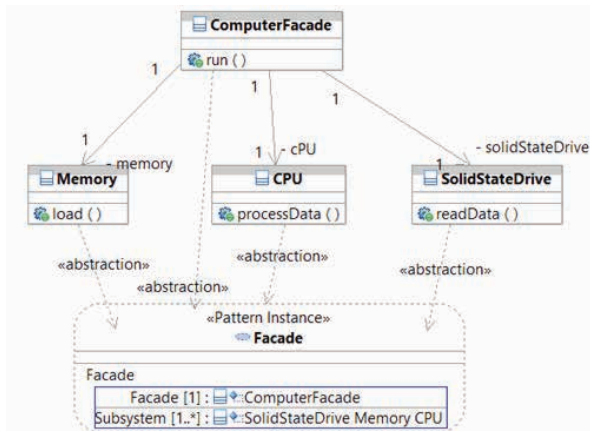
Slika 47. Osnovna struktura *Facade* obrasca.



Vežba 4-3. Primeniti odgovarajući obrazac.

Opis problema: Kada se računar pokrene, to uključuje podizanje i rad CPU, memorije, SSD drajva i drugih podsistema računara. Potrebno je sakriti složenost ovih zadataka od korisnika i prikazati mu samo osnovne informacije o operacijama ovih podsistema.

Rešenje:



```
public class ComputerFacade {
    private Memory memory;
    private CPU cPU;
    private SolidStateDrive sSD;

    public Memory getMemory() {
        return memory;
    }

    public void setMemory(Memory memory) {
        this.memory = memory;
    }

    public CPU getCPU() {
        return cPU;
    }

    public void setCPU(CPU cPU) {
        this.cPU = cPU;
    }

    public SolidStateDrive getSSD() {
        return sSD;
    }

    public void setSSD(SolidStateDrive solidStateDrive) {
        this.sSD = solidStateDrive;
    }

    public void run() {
        cPU.processData();
        memory.load();
        sSD.readData();
        System.out.println("Your computer is ready!");
    }

    public ComputerFacade() {
        this.cPU=new CPU();
        this.memory=new Memory();
        this.sSD=new SolidStateDrive();
    }
}

public class CPU {
    public void processData() {
        System.out.println("CPU process data.");
    }
}

public class Memory {
    public void load() {
        System.out.println("Memory loaded.");
    }
}

public class SolidStateDrive {
    public void readData() {
        System.out.println("SSD drive read data. ");
    }
}
```

```
public class Client {  
    public static void main(String[] args) {  
        ComputerFacade facade= new ComputerFacade();  
        facade.run();  
    }  
}
```

Rezultat:

CPU process data.
Memory loaded.
SSD drive read data.
Your computer is ready!

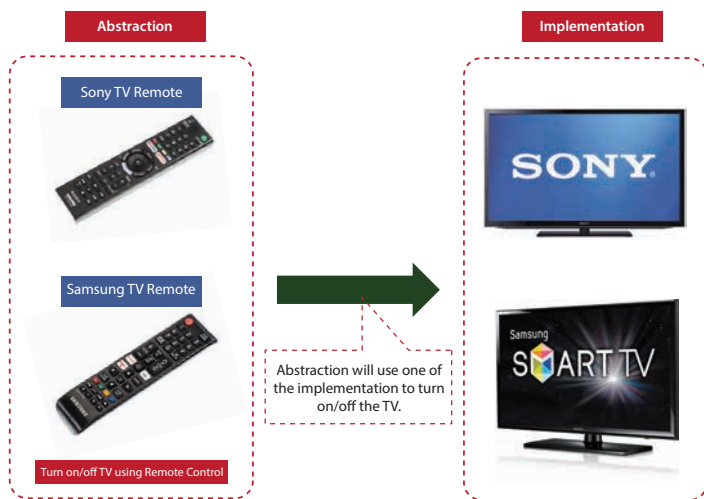
Zadatak 4-3. Primeniti obrazac nad aplikacijom za dizajniranje željenih kola.

Korisnik putem aplikacije može da dizajnira željena kola. Korisnik može da bira željeni model kola, motor i dodatke. Klasa Model sadrži metodu *postaviModel()*, klasa Motor sadrži *postaviMotor()* i klasa Dodaci sadrži *postaviDodatke()*. Ove metode će samo ispisati šta je postavljeno. U okviru FasadeKola, nalazi se metod *dizajnirajKola (carDesign)* koji će prvo ispisati „Kola se dizajniraju...“, potom pozvati metode klase podsistema i na kraju ispisati „Vaša kola su spremna“. Klijent komunicira sa Fasodom gde poziva samo njenu metodu *carDesign()*.

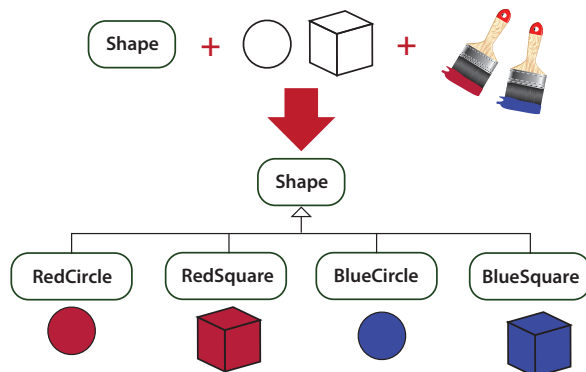
4.4. BRIDGE

Bridge obrazac omogućava da se podeli ogroman broj klasa ili skup tesno povezanih klasa u dve odvojene hijerarhije klasa, gde se jedna bavi apstrakcijom (npr. korisnički interfejs), a druga implementacijom i kao takve da ih razvijamo nezavisno jedna od druge (Slika 48).

Slika 48. Poređenje *Bridge* obrasca sa realnim primerom iz prakse.



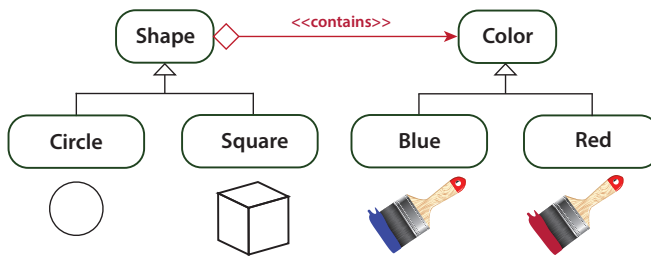
Da bi se bolje razumela primena *Bridge* obrasca, razmotrićemo sledeći problem. Ukoliko imate klasu geometrijski Oblik (*Shape*), sa potklasama Krug (*Circle*) i Kvadrat (*Square*). Zatim planirate da dodate i boje, kako bi kreirali crvene i plave oblike (npr. crveni krug i plavi krug). Umesto dve potklase, sada je potrebno da imate onoliko potklasa koliko imate kombinacija boja oblika.



Dodavanjem novih tipova oblika i boja, hijerarhija klasa će se eksponencijalno uvećavati. Ovaj problem nastaje usled toga što pokušavate da proširite klasu Oblik (*Shape*) sa dve nezavisne dimenzije: po geometrijskom obliku i po boji.

Bridge obrazac premošćava ovaj problem tako što se umesto nasleđivanja fokusira na kompoziciju objekata – ovo znači da razdvaja dve dimenzije u dve odvojene hijerarhije, pre nego da ima sva stanja i ponašanja u jednoj klasi.

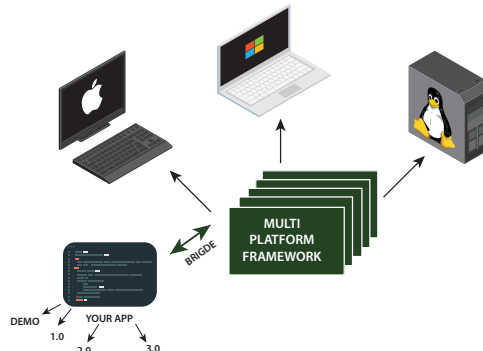
Možete sprečiti eksponencijalni rast hijerarhije klasa, transformišući ih u nekoliko povezanih hijerarhija. U ovom primeru, razdvajamo kôd koji se odnosi na boje u sopstvenu klasu *Color* sa dve potklase *Red* i *Blue*. Klasa *Oblik* referencira na određene objekte boja. Dodavanje nove boje sada neće zahtevati izmenu hijerarhije oblika i obrnuto.

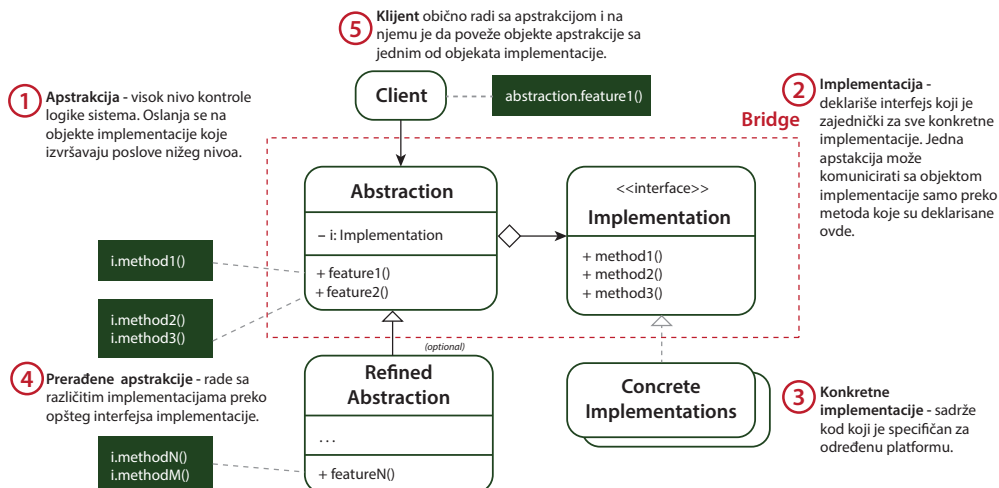


Prema definiciji, *Bridge* obrazac razdvaja apstrakciju od njegove implementacije. Apstrakcija podrazumeva visok nivo interfejsa koji ne sadrži poslovnu logiku ili bilo kakve detalje o implementaciji. Implementacija je odgovorna za poslove nižeg nivoa. Apstrakcija sadrži referencu na objekat implementacije, tako da apstrakcija kontroliše metode objekta implementacije. Na primer, aplikacija se može razvijati u dva nezavisna pravca (Slika 49):

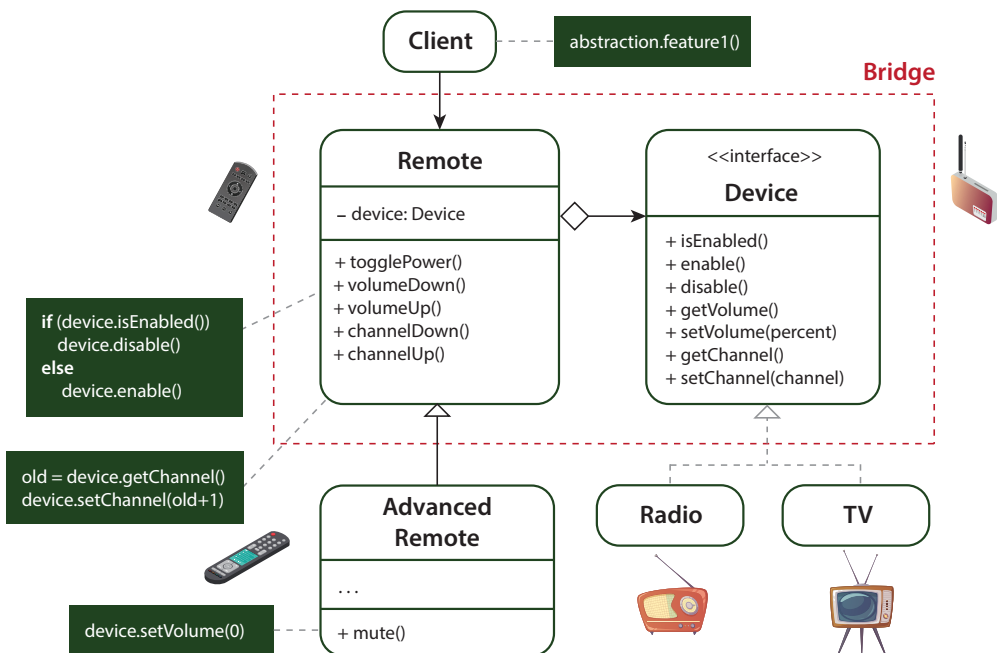
- ♦ Apstrakcija – objekti apstrakcije mogu da kontrolišu izgled aplikacije (npr. GUI sloj aplikacije) koji delegira stvarni posao povezanim objektima implementacije.
- ♦ Implementacija – npr. API operativnih sistema

Slika 49. Primer GUI klasa koje se mogu menjati bez izmene API sistema (Shvets, 2019).



Osnovna struktura *Bridge* obrasca:Slika 50. Osnovna struktura *Bridge* obrasca.

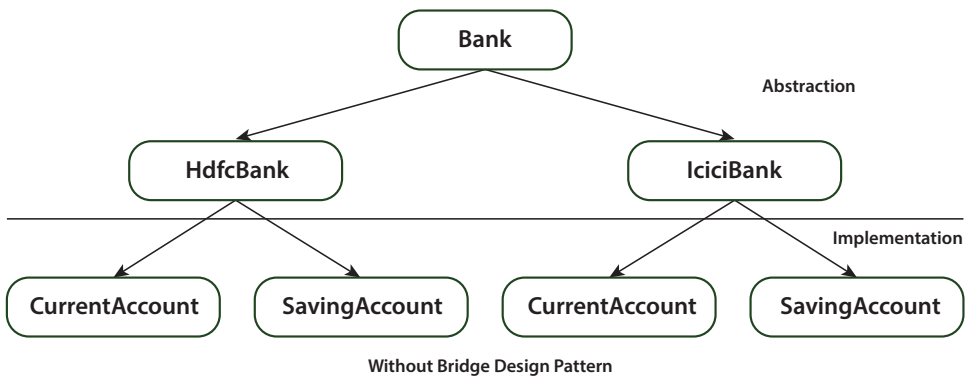
Osnovna struktura primenjena na prvobitnom primeru razdvajanja daljinskih upravljača od implementacije različitih uređaja:



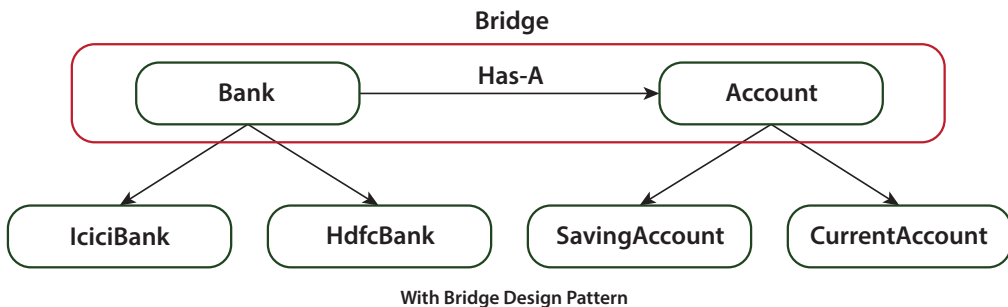
Vežba 4-4. Primeniti odgovarajući obrazac na sledeći model Banke.

Opis problema: Jedan bankarski sistem omogućava klijentima otvaranje računa bilo tekućih ili štednih kod dve banke i to HdfcBank i IciciBank. Svaka od ovih banaka pruža svojim klijentima otvaranje tekućih računa (*Current Account*) i štednih računa (*Saving Account*). Kreirati aplikaciju koja će prikazati u kojoj banci je klijent otvorio koji tip računa.

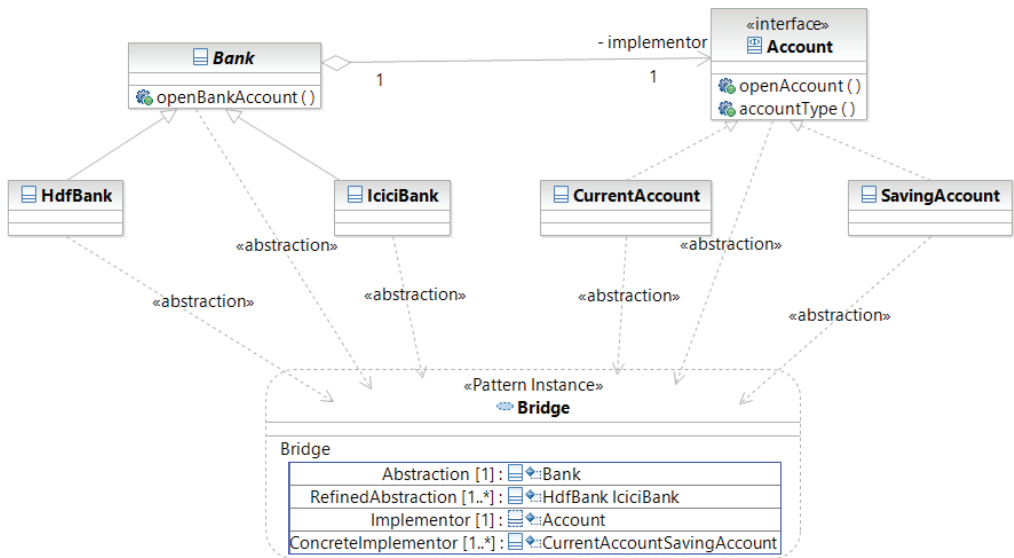
Opis rešenja: Sagledavajući postojeći model, mogu se jasno razgraničiti apstrakcija od implementacije modela.



Primenom Bridge obrasca, prvobitna struktura klasa bi se podelila na apstrakciju Bank sa konkretnim bankama i implementaciju Account sa konkretnim tipovima računa.



Prevlačenjem *Bridge* obrasca na dijagram klasa, u IBM RSA alatu, unećemo nazive klasa u osnovne elemente obrasca i prikazati relacije između klasa. Nakon generisanog dijagrama, dopunićemo klase sa osnovnim metodama, kao što je prikazano na donjem dijagramu.



Generisani i dopunjen kod:

```
public interface Account {
    public void openAccount();
    public void accountType();
}

public class CurrentAccount implements Account {
    public void openAccount() {
        System.out.println("OPENED: CURRENT ACCOUNT");
    }
    public void accountType() {
        System.out.println(" It is a CURRENT Account ");
    }
}

public class SavingAccount implements Account {
    public void openAccount() {
        System.out.println("OPENED: SAVING ACCOUNT ");
    }
    public void accountType() {
        System.out.println(" It is a SAVING Account ");
    }
}

public abstract class Bank {
    //obratiti pažnju da je implementor protected
    protected Account implementor;
    public Account getImplementor() {
        return implementor;
    }
    public void setImplementor(Account implementor) {
        this.implementor = implementor;
    }
}
```

```

    }
    //pretvoreno u apstraktnu klasu
    abstract Account openBankAccount();
    //Dodat je konstruktor
    public Bank(Account implementor) {
        super();
        this.implementor = implementor;
    }
}

public class HdfBank extends Bank {
    public HdfBank(Account implementor) {
        super(implementor);
    }
    @Override
    Account openBankAccount() {
        System.out.println("Open your account with HDFC Bank. ");
        return implementor;
    }
}

public class IciciBank extends Bank {
    public IciciBank(Account implementor) {
        super(implementor);
    }
    @Override
    Account openBankAccount() {
        System.out.println("Open your account with ICICI Bank. ");
        return implementor;
    }
}

public class BridgePatternMain {
    public static void main(String[] args) {
        Bank icici=new IciciBank(new CurrentAccount());
        Account current=icici.openBankAccount();
        current.openAccount();
        current.accountType();
        Bank hdfc=new HdfBank(new SavingAccount());
        Account saving = hdfc.openBankAccount();
        saving.accountType();
    }
}

```

Rezultat:

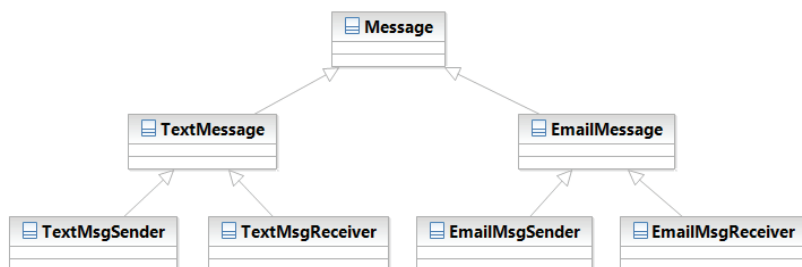
```

Open your account with ICICI Bank.
OPENED: CURRENT ACCOUNT
    It is a CURRENT Account
Open your account with HDFC Bank.
OPENED: SAVING ACCOUNT
    It is a SAVING Account

```

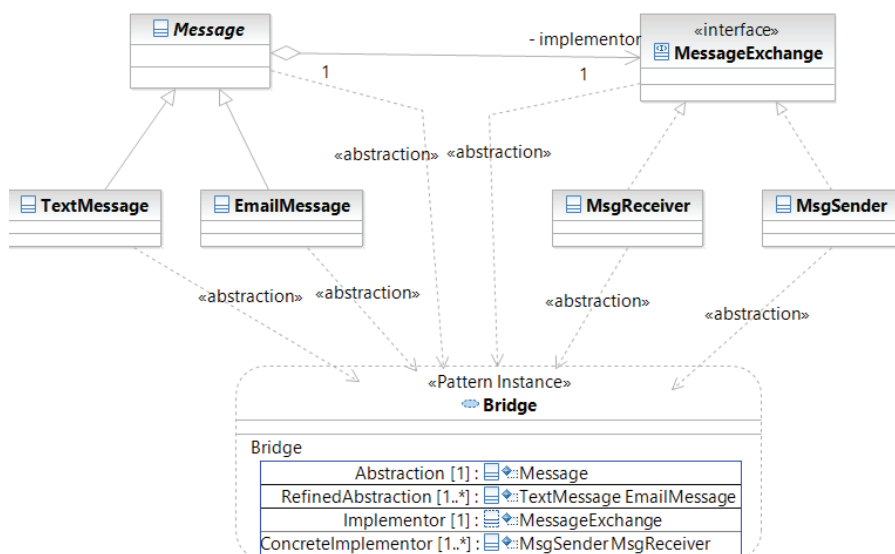
Vežba 4-5. Primeniti odgovarajući obrazac projektovanja na aplikaciju za slanje različitih vrsta poruka.

Modelovati aplikaciju za razmenu poruka koju klijenti mogu da koriste za slanje i primanje različitih vrsta poruka, kao što su tekstualne ili e-mail poruke. Početni model klasa je prikazan na slici:



U narednoj iteraciji razvoja aplikacije bi trebalo omogućiti klijentima opciju enkriptovanja (šifrovanja) poruka pre slanja. Takođe, u planu je razvoj slične aplikacije, gde bi mogli da iskoristite samo deo koda za slanje poruka. Primeniti odgovarajući obrazac koji će vam omogućiti lako proširenje funkcionalnosti i ponovnu upotrebljivost dela aplikacije.

Rešenje:



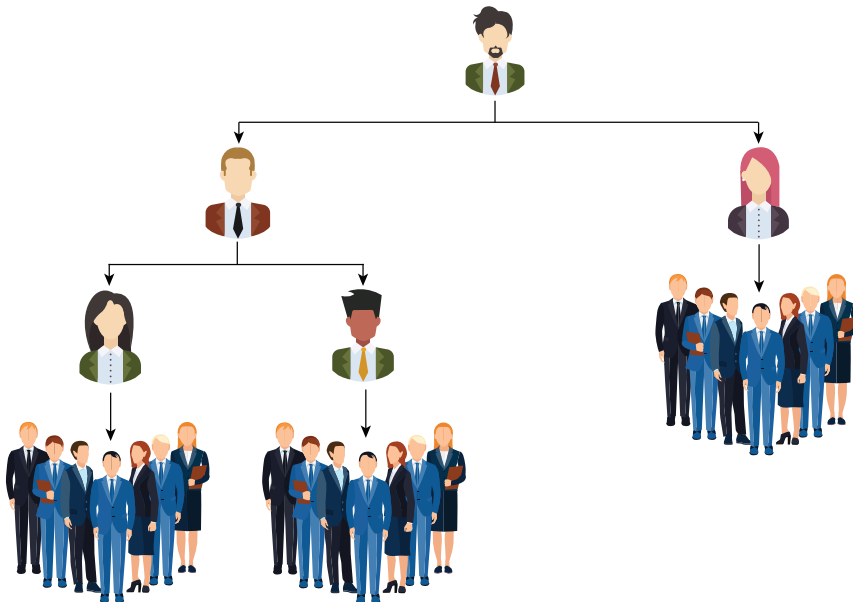
Zadatak 4-4. Primeniti odgovarajući obrazac na aplikaciju koja iscrtava obojene geometrijske oblike.

Aplikacija iscrtava dva geometrijska oblika: trougao i pentagon, a zatim ih boji crvenom i/ili zelenom bojom. Kreirati crveni trougao i pentagon i zeleni pentagon.

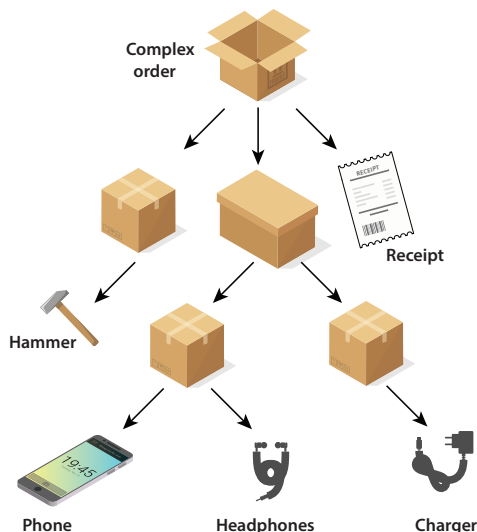
4.5. COMPOSITE

Composite obrazac jedino ima smisla primeniti tamo gde je model aplikacije predstavljen kao stablo (Slika 51).

Slika 51. Poređenje *Composite* obrasca sa realnim primerom u praksi.



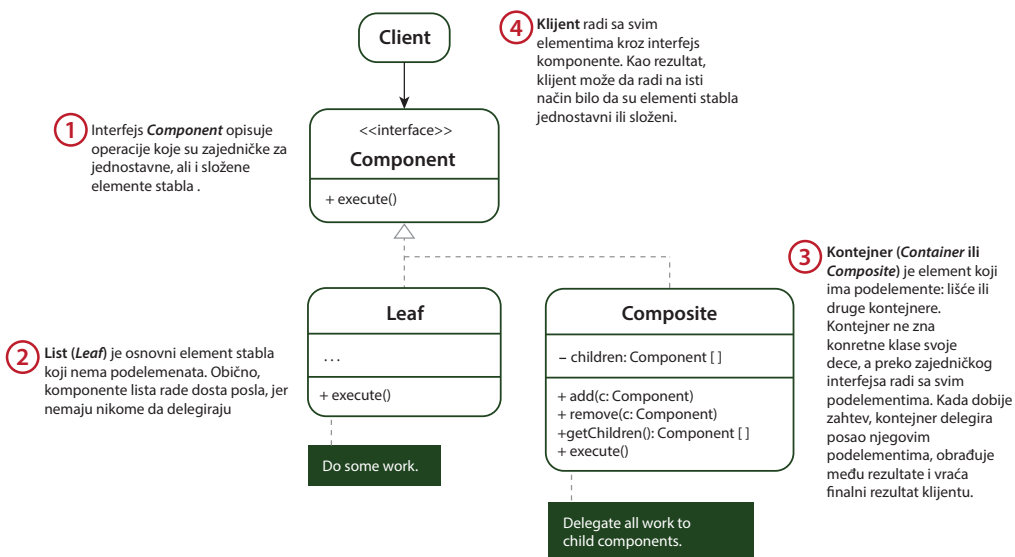
Zamislite da imate dva tipa objekata: Proizvodi i Kutije. Kutija može da sadrži nekoliko proizvoda, ali i manje kutije. Manja kutija takođe može da sadrži neke proizvode i još manje kutije itd. Ukoliko odlučite da napravite sistem za poručivanje koji koristi ove klase, kako bi ste odredili ukupnu cenu ovakvog navedenog redosleda? Možete redom odmotavati kutije i računati cenu, ali u kodu bi trebalo znati koje klase proizvoda i kutije su ugnježdene na kom nivou kutija.



Composite obrazac predlaže da sa proizvodima i kutijama radite preko zajedničkog interfejsa koji deklariše metodu za izračunavanje cene. Kada zovete metodu, objekti prosleđuju zahtev kroz stablo.

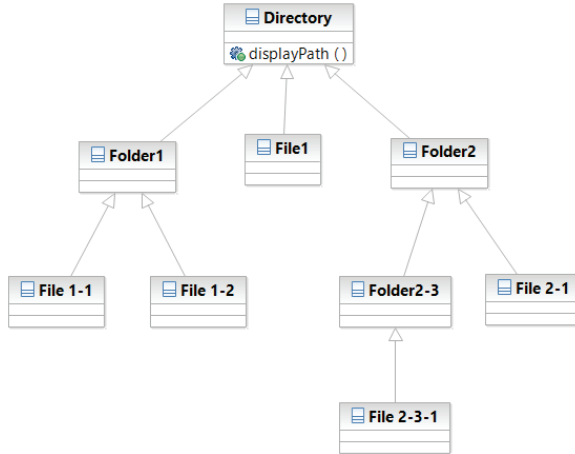
Osnovna struktura *Composite* obrasca je:

Slika 52. Osnovna struktura *Composite* obrasca.



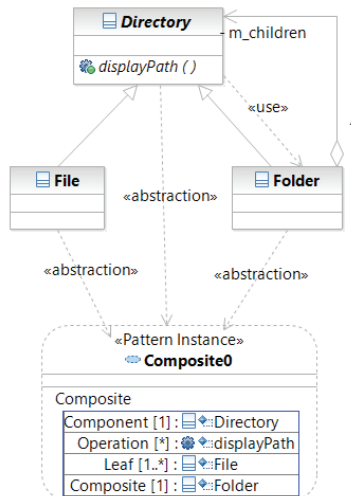
Vežba 4-6. Primeniti *Composite* obrazac na deo aplikacije za prikaz putanje fajla.

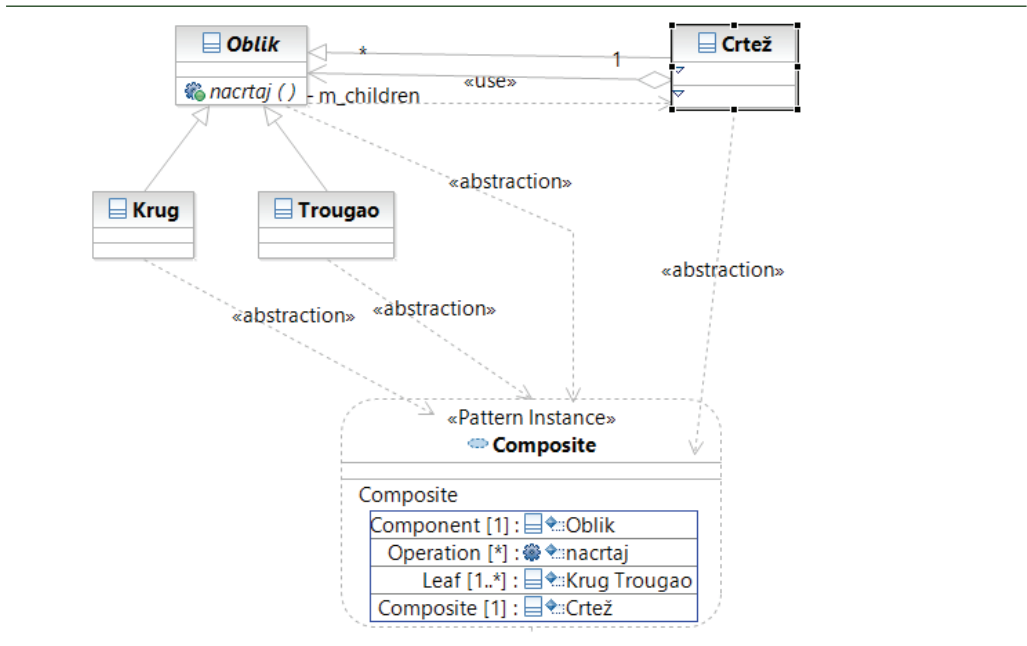
Primeniti *Composite* obrazac za deo aplikacije koja će prikazati detaljnu putanju fajla, na osnovu početnog modela klasa:



Rešenje:

U ovom primeru list (*Leaf*) stabla je *File*, a *Composite* klasa je *Folder*. Klasu tipa *Component*, koju ćemo imenovati kao direktorijum (*Directory*), sadrži jednu metodu *displayPath()*. Sa ovim osnovnim klasama i metodom popunićemo obrazac *Composite* i prikazati relacije:



Vežba 4-7. Primeniti *Composite* obrazac za crtanje geometrijskih oblika na crtežu.

```

public abstract class Oblik {
    public abstract void nacrtaj();
    private Oblik m_parent;
    public Oblik getParent() {
        return m_parent;
    }
    public void setParent(Oblik a_parent) {
        m_parent = a_parent;
    }
    public abstract Crtež getComposite();
}

public class Krug extends Oblik {
    public void nacrtaj() {
        System.out.println("Iscrtava krug");
    }
    public Crtež getComposite() {
        return null;
    }
}

public class Trougao extends Oblik {
    public void nacrtaj() {
        System.out.println("Iscrtava trougao");
    }
}

```

```

        public Crtež getComposite() {
            return null;
        }
    }

import java.util.List;
public class Crtež extends Oblik {
    private List<Oblik> m_children;
    public List<Oblik> getM_children() {
        return m_children;
    }
    public void setM_children(List<Oblik> m_children) {
        this.m_children = m_children;
    }
    public void nacrtaj() {
        for (Oblik mch:m_children){
            mch.nacrtaj();
        }
    }
    public Crtež getComposite() {
        return this;
    }
    public Crtež() {
        m_children = new java.util.ArrayList<Oblik>();
    }
    public synchronized void addChild(Oblik a_component) {
        synchronized (a_component) {
            m_children.add(a_component);
            a_component.setParent(this);
        }
    }
    public Oblik getChild(int a_childIndex) {
        return (Oblik) m_children.get(a_childIndex);
    }
    public synchronized void removeChild(Oblik a_component) {
        synchronized (a_component) {
            if (this == a_component.getParent()) {
                a_component.setParent(null);
            }
            m_children.remove(a_component);
        }
    }
}

public class Main {
    public static void main(String[] args) {
        Oblik tro=new Trougao();
        Oblik kru=new Krug();
        Crtež crtez=new Crtež();
        crtez.addChild(tro);
        crtez.addChild(kru);
        crtez.nacrtaj();
    }
}

```

Rezultat:**Iscrtava trougao****Iscrtava krug****Zadatak 4-5. Primeniti *Composite* obrazac na aplikaciju za opis zaposlenih.**

U hijerarhijskoj organizacionoj strukturi jedne IT kompanije, na vrhu se nalazi generalni menadžer, zatim se pod njim nalaze drugi menadžeri, a pod menadžerima se nalaze ostali zaposleni - developeri. Potrebno je napraviti aplikaciju koja će od svakog čvora u stablu zatražiti da izvrši uobičajenu operaciju, kao što je prikazivanje detaljnih informacija o zaposlenima i to: id, ime, prezime i pozicija.

4.6. FLYWEIGHT

Da bi se lakše razumeo *Flyweight* obrazac (obrazac lake težine), uzećemo u obzir sledeći scenario. Zamislite da ste od vašeg menadžera dobili projekat da reorganizujete front-end deo Amazon aplikacije koja se bavi prikazivanjem, preuzimanjem i skladištenjem knjiga.

Prvo, odlučili ste da kreirate klasu Knjiga (*Book*) koja će čuvati sve relevantne attribute određene knjige:

```
@Data
public class Book {
    private final String name;
    private final double price;
    private final String type;
    private final String distributor;
    private final String otherData;
}
```

Dodali ste i željene metode, npr. *storeBook()* i *displayBooks()*:

```
public class Store {
    private final List<Book> books = new ArrayList();
    public void storeBook(String name, double price, String type,
        String distributor, String otherData){
        books.add(new Book(name, price, type, distributor, otherData));
    }
    public void displayBooks(){
        for (Book book : books){
            //render the books on the store website
        }
    }
}
```

Kreirali glavnu metodu:

```
private static final int BOOK_TYPES=2;
private static final int BOOKS_TO_INSERT = 10000000;

public static void main(String[] args) {
    Store store = new Store();
    for (int i=0; i< BOOKS_TO_INSERT / BOOK_TYPES; i++){
        store.storeBook(getRandomName(),getRandomPrice(),
            "Action", "Follett", "Stuff");
        store.storeBook(getRandomName(),getRandomPrice(),
            "Fantasy", "Ingram", "Extra");
    }
    store.displayBooks();
}
```

Nakon dodavanja i ostalih servisa, aplikaciju ste poslali menadžeru na uvid. Međutim, iako je sve besprekorno radilo kod vas, na vašoj lokalnoj mašini, kod menadžera je aplikacija stalno pucala. Nakon što ste proveli nekoliko sati pokušavajući da rešite greške u kodu, otkrili ste da je pravi razlog pucanja nedovoljna količina RAM memorije.

Pravi razlog problema leži u klasi *Book*. Svaka knjiga je predstavljena posebnim objektom koji sadrži mnogo podataka i ovde se ne pokušava kreirati ili prikazati 10-tak knjiga već milioni njih (`private static final int BOOKS_TO_INSERT = 10000000;`). Dakle, ogromna količina velikih objekata se nisu uklopili u preostali deo RAM memorije, te je program pao.

Rešenje ovog problema leži u primeni i implementaciji obrasca *Flyweight*, koji omogućava da se više objekata postavi u dostupnu količinu RAM-a, deljenjem njihovih zajedničkih stanja.

Da bi primenili ovaj obrazac na našem primeru, potrebno identifikovati zajedničke atribute koji dele objekti knjige. Uočeno je da atribut `BOOK_TYPES` uzima samo dve vrednosti i to „Action” i „Fantasy”, a zatim da tip knjige određuje vrednosti atributa za distributera i druge podatke (*otherData*). Na primer, distributer je isti za sve knjige tipa „Action”.

U ovom slučaju, primena obrasca *Flyweight* bi sugerisala da prestanete da čuvate stanja atributa koja se retko menjaju u originalnom objektu, već ih premestite u poseban objekat. U našem primeru, tri ponavljajuća atributa će se premestiti u novu klasu *BookType*:

```
public class Book {
    private final String name;
    private final double price;
    private final BookType type;
}

public class BookType {
    private final String type;
    private final String distributor;
    private final String otherData;
}
```

Klasa *BookType* se naziva *Flyweight* klasa i ona čuva unutrašnje stanje objekta koje je nepromenljivo, nezavisno od konteksta, deljivo i nikada se ne menja tokom izvršavanja.

S obzirom da će mnoge knjige koristiti isti *Flyweight* objekat, potrebno je osigurati da se njegovo stanje ne menja. Stoga, klasa *BookType* ne bi trebalo da ima javne set metode ili promenljive. Kreiranje vrednosti klase *BookType*, prebacuje se na novu fabričku klasu *FlyweightFactory*. Atributi koji su ostavljeni u klasi *Book* (naziv i cena) su jedinstveni za objekte knjiga i njihove vrednosti se međusobno razlikuju i mogu se menjati u trenutku izvršavanja aplikacije.

Fabričku klasu za kreiranje *flyweight* objekta *BookType*, imenovaćemo kao *BookFactory*:

```
public class BookFactory {
    private static final Map<String, BookType> bookTypes = new HashMap();
    public static BookType getBookType(String type, String distributor,
                                       String otherData) {
        if(bookTypes.get(type)==null){
            bookTypes.put(type, new BookType(type, distributor,
                                             otherData));
        }
        return bookTypes.get(type);
    }
}
```

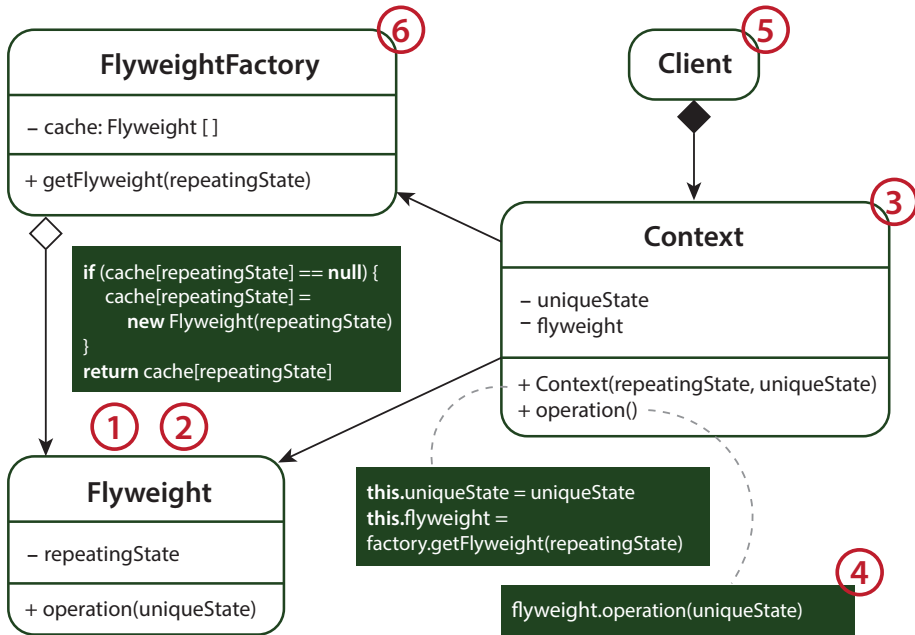
Klasa *BookFactory* koristi *HashMap* strukturu za čuvanje referenci na već kreirane *flyweight* objekte. Klijent (*Client*) klasa ne može direktno da kreira *flyweight* klasu, već poziva *BookFactory* klasu koja pregleda prethodno kreirane *flyweight* klase i ukoliko pronađe klasu koja odgovara navedenim kriterijumima vraća postojeću *flyweight* klasu, u suprotnom kreira novu.

Klasa *Store* će sada umesto da kreira i skladišti sve vrednosti knjige, prvo pozvati odgovarajući *Flyweight* objekat putem metode *getBookType*, a zatim nastaviti sa skladištenjem ostalih vrednosti atributa.

```
public class Store {
    private final List<Book> books = new ArrayList();
    public void storeBook(String name, double price, String type,
                         String distributor, String otherData){
        BookType bookType = BookFactory.getBookType(type, distributor,
                                                    otherData);
        books.add(new Book(name, price, bookType));
    }
}
```

Primena *Flyweight* obrasca dovodi do optimizacije memorijskog prostora, posebno kada je potrebno istovremeno koristiti veliki broj sličnih objekata.

Osnovna struktura *Flyweight* obrasca je prikazana na slici 53.

Slika 53. Osnovna struktura *Flyweight* obrasca (Shvets, 2019).

Ukoliko uporedimo sa prethodnim primerom, *Flyweight* klasa je *BookType* klasa, koja sadrži deo stanja originalnog objekta koji se može deliti između više objekata. Isti *flyweight* objekat se može ponovo upotrebiti u mnogim različitim kontekstima, a stanje koje je uskladišteno unutar *flyweight* klase se naziva „unutrašnje“ (*intrinsic*). Klasa *Context* je klasa *Book* koja sadrži tzv. spoljašnja (*extrinsic*) stanja, jedinstvena za sve originalne objekte. Zajedno klase *Context* i *Flyweight* čine potpuno stanje originalnog objekta.

Da bi se kreirala *Flyweight* klasa neophodna je klasa *FlyweightFactory*, koja upravlja setom postojećih *flyweight* objekata. U našem primeru *FlyweightFactory* je *BookFactory* klasa. Klijent (*Client*) klasa ne može direktno da kreira *flyweight* objekat, već poziva *FlyweightFactory* klasu koja pregleda prethodno kreirane *flyweight* objekte i ukoliko pronađe objekat koja odgovara navedenim kriterijumima vraća postojeći *flyweight* objekat, u suprotnom kreira novi.

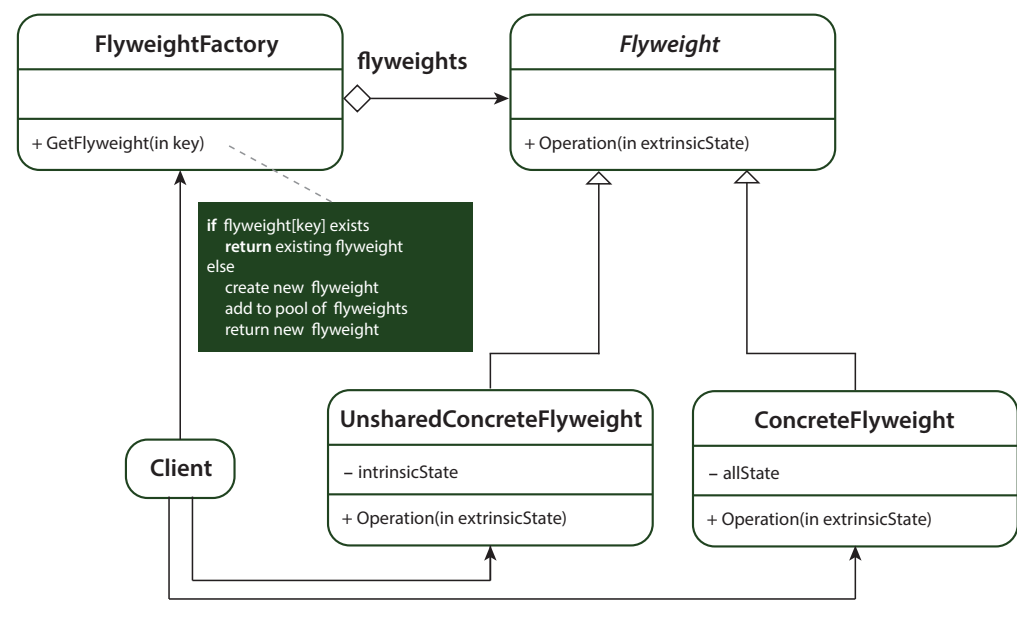
Prošireni dijagram klasa osnovne strukture *Flyweight* obrasca je prikazan na slici 54.

Osnovni elementi *Flyweight* obrasca, prema proširenoj osnovnoj strukturi, su:

- ♦ *Flyweight* interfejs – sadrži operacije za rad za spoljnim (jedinstvenim) stanjima objekata.
- ♦ *ConcreteFlyweight* – implementira *Flyweight* interfejs i dodaje skladište za unutrašnja stanja, ako postoje. *ConcreteFlyweight* objekat mora biti deljiv. Bilo koje stanje koje skladišti mora biti unutrašnje (*intrinsic*), tj. mora biti nezavisno od konteksta *ConcreteFlyweight* objekta.

- ♦ *UnsharedConcreteFlyweight* (ne koristi se često) – ne moraju se deliti sve *Flyweight* potklase. Obično objekti *UnsharedConcreteFlyweight* (kao roditelj klase) imaju podređene (kao decu) *ConcreteFlyweight* objekte, na nekom nivou u strukturi *flyweight* objekta.
- ♦ *FlyweightFactory* – kreira i upravlja *flyweight* objektima i osigurava da se *flyweight* pravilno deli. Kada klijent zahteva *flyweight*, tada *FlyweightFactory* deli postojeću instancu ili kreira novu, ukoliko ne postoji tražena.
- ♦ Klijent – održava referencu na *flyweight* objekte i obrađuje ili skladišti spoljašnje (*extrinsic*) stanje *flyweight* objekata.

Slika 54. Proširena osnovna sturktura *Flyweight* obrasca.



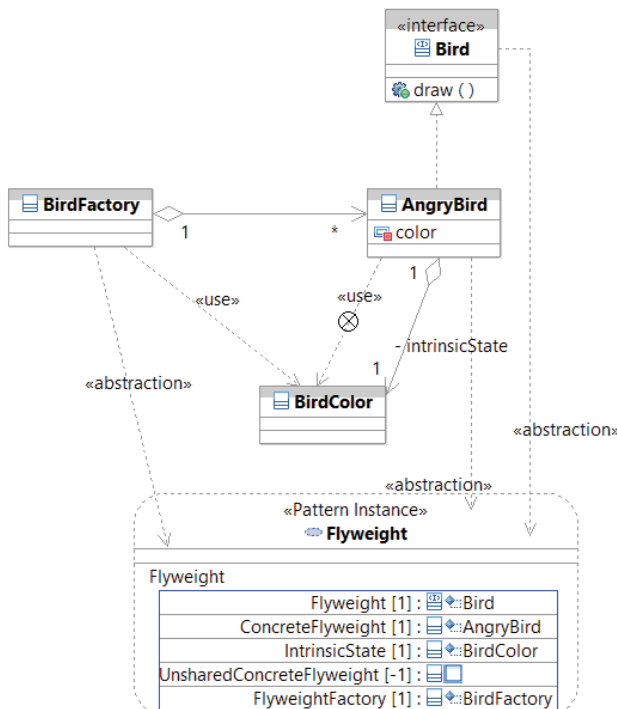
Vežba 4-8. Primeniti *Flyweight* obrazac na igricu *AngryBirds* sa funkcionalnostima iscrtavanja i bojenja *AngryBirds* objekata.

Igrica *Angry Birds* je jedno vreme bila popularna širom sveta. Ukoliko u igrici želimo da kreiramo 20.000 ljutih ptica (*AngryBirds*), crvene boje, osnovna greška bi bila da se kreira svih 20.000 *AngryBirds* objekata, koji će se ispuniti crvenom bojom, jer će ti objekti zauzeti više memorije, a uticaće i na performanse same igrice. Jedno od rešenja je da se primeni *Flyweight* obrazac. Vrednosti koje mogu biti deljene među *AngryBirds* objektima je boja, koja može uzeti sledeće finalne vrednosti: *Red*, *Orange*, *Blue*, *Yellow* i *Pink* (Joshi, 2018).

Koraci rešenja:

Na dijagramu klasa (Slika 55), *Flyweight* obrazac u IBM RSA alatu, zahteva unos *IntrinsicState* klase, gde će se definisati deljena stanja. Ova klasa se može pretvoriti, po želji, u *enum* klasu sa fiksnim vrednostima ili se statične vrednosti mogu definisati u glavnoj klasi, te se ona kao takva može izostaviti u kodu. Ovde je korišćen poslednji pristup, iako je na dijagramu klasa prikazana *IntrinsicState* klasa: *BirdColor*.

Slika 55. Primena *Flyweight* obrasca na *AngryBird* klasu.



Izgenerisani i dopunjen izvorni kod:

Flyweight interfejs:

```
public interface Bird {  
    public void draw ();  
}
```

ConcreteFlyweight klasa:

```
public class AngryBird implements Bird {  
    private String color;  
    public AngryBird (String color) {  
        this.color=color;  
    }  
    @Override  
    public void draw () {  
        System.out.println("Draw Angry Bird in color: " + color);  
    }  
}
```

Flyweight Factory klasa:

```
import java.util.HashMap ;  
  
public class BirdFactory {  
    private static final HashMap<String, AngryBird> angrybirdMap = new  
        HashMap<String, AngryBird>();  
    public static Bird getAngryBird(String color) {  
        AngryBird angrybird = (AngryBird) angrybirdMap.get(color);  
        if(angrybird == null) {  
            angrybird = new AngryBird(color);  
            angrybirdMap.put(color, angrybird);  
            System.out.println("Creating Angry Bird of color: " + color);  
        }  
        return angrybird;  
    }  
}
```

Glava klasa:

```
public class FlyweightDesignPattern {  
    private static final String colors[] = {"Red", "Orange", "Blue", "Yellow",  
        "Pink" };  
    public static void main (String[] args) {  
        for(int i=0; i < 5; ++i) {  
            AngryBird angrybird =  
                (AngryBird) BirdFactory.getAngryBird(getRandomColor());  
            angrybird.draw();  
        }  
    }  
}
```

```
private static String getRandomColor() {  
    return colors[(int)(Math.random()*colors.length)];  
}  
}
```

Rezultat izvršavanja aplikacije:

```
Creating Angry Bird of color : Orange  
Draw Angry Bird in color : Orange  
Creating Angry Bird of color : Blue  
Draw Angry Bird in color : Blue  
Creating Angry Bird of color : Pink  
Draw Angry Bird in color : Pink  
Draw Angry Bird in color : Pink  
Creating Angry Bird of color : Red  
Draw Angry Bird in color : Red
```

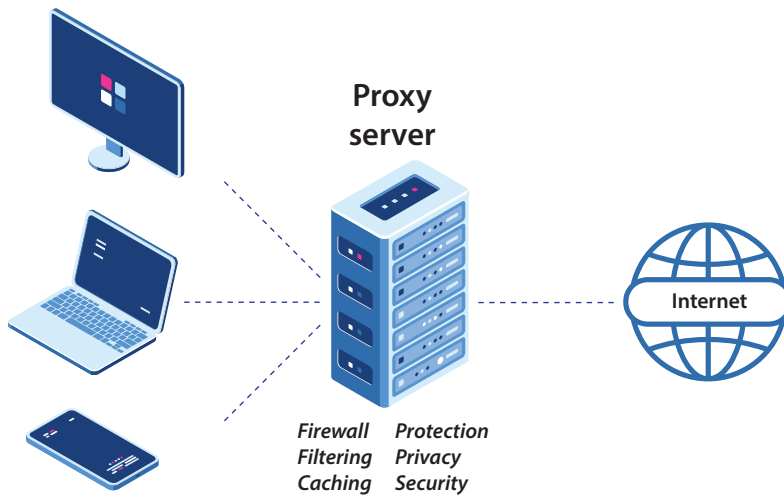
Zadatak 4-6. Primeniti *Flyweight* obrazac na *Counter Strike* igricu.

Kreirati interfejs klasu *Player* koja ima metode *assignWeapon (String weapon)* i *misson()*. Zadaci (*Task*) su deljeni i uzimaju dve vrednosti BOMB i ATTACK. Kreirati ih kroz klasu enumeracije. Klasa *CounterTerrorist* nasleđuje interfejs i izvršava zadatak BOMB. U glavnoj klasi proširiti statične vrednosti za tip igrača (*playerType*) te pored *CounterTerrorist* dodati i *Terrorist*. Takođe dodati statične vrednosti za *weapon*. Kreirati 10 igrača sa random vrednostima oružja.

4.7. PROXY

Kad god se upotrebi reč *proxy*, prvo na šta se pomisli su *proxy* serveri (Slika 56). *Proxy* je posrednički server koji reguliše saobraćaj između korisnika i internet servisa. Njegova uloga nije samo posrednička, već deluju i kao zaštitni zid (*firewall*), web filteri, obezbeđuju deljene mrežne konekcije, mogu da keširaju često zahtevane podatke, štite korisnika od malvera i drugih zloupotreba, obezbeđuju visok nivo privatnosti i bezbednosti surfovanja mrežom i dr.

Slika 56. Položaj *proxy* servera



Ukoliko bi u nekoj aplikaciji deklarirali Internet kao interfejs klasu, a konkretnu uslugu kao *RealInternet* klasu koja implementira dati interfejs:

```
public interface Internet {
    void connectTo(String host);
}

public class RealInternet implements Internet {
    public void connectTo(String host){
    }
}
```

Svaki put kada korisnik pokuša da se poveže na određenu veb lokaciju, kreira se instanca te klase, koja poziva metodu *connectTo()*. Na primer:

```
public static void main (String[] args){
    Internet internet = new RealInternet();
    Internet.connectTo("google.com");
}
```

Ukoliko ste, na primer, dobili zahtev da određenoj grupi korisnika ograničite pristup nekim veb lokacijama u okviru ove Internet klase, onda na ovaj način niko neće moći da im pristupi:

```
public class RealInternet implements Internet {
    public void connectTo(String host){
        if ("banned.com".equals(host)){
            System.out.println("Access Denied!");
        }
        System.out.println("Opened connection to "+host);
    }
}
```

Da bi rešili ovaj problem, uvešćemo posredničku klasu *ProxyInternet*, koja proverava korisnike i njihove zahteve ka listi zabranjenih veb lokacija. Ukoliko određeni korisnik pokuša da pristupi zabranjenim veb lokacijama, pojaviće se poruka o odbijanju pristupa:

```
public class ProxyInternet implements Internet {
    private static final List<String> bannedSites;
    private final Internet internet=new RealInternet();
    static{
        bannedSites=new ArrayList<>();
        bannedSites.add("banned.com");
    }
    public void connectTo(String host){
        if (bannedSites.contains(host)) {
            System.out.println("Access Denied!");
            return;
        }
        Internet.connectTo(host);
    }
}

public static void main (String[] args){
    Internet internet = new ProxyInternet();
    internet.connectTo("google.com");
    internet.connectTo("banned.com");
}
```

Na ovaj način funkcionišu i *Proxy* obrasci projektovanja. *Proxy* je strukturalni obrazac projektovanja koji kontroliše pristup originalnom objektu i omogućava da izvršite neke operacije pre ili nakon što zahtev stigne do originalnog objekta.

Radi boljeg razumevanja rada *Proxy* obrasca, razmotrićemo još jedan primer. Pretpostavimo da imate klasu *VideoDownloader*, koja uzima naziv videa koji želite da preuzmete, zatim se povezuje na *YouTube* radi preuzimanja tog videa, metapodataka i lokacije i zatim vam vraća sve ove informacije zajedno sa video snimkom.

```
public interface VideoDownloader {
    Video getVideo(String videoName);
}

public class RealVideoDownloader implements VideoDownloader{
    public Video getVideo(String videoName){
        System.out.println("Connecting to https://www.youtube.com/");
        System.out.println("Downloading video");
        System.out.println("Retrieving video metadata");
        return new Video(videoName);
    }
}
```

Svaki put kada vam zatreba određeni video, taj video se ponovo preuzima, pa čak iako je prethodno bio preuzet.

```
public static void main(final String[] arguments){
    VideoDownloader videoDownloader = new RealVideoDownloader();
    videoDownloader.getVideo("Geekific");
    videoDownloader.getVideo("Geekific");
    videoDownloader.getVideo("Sean Study");
    videoDownloader.getVideo("Sean Study");
    videoDownloader.getVideo("Geekific");
}
```

Da bi se ovo izbeglo, potreban je neki mehanizam za keširanje koji čuva informacije o preuzetim video snimcima. *Proxy* klasa će omogućiti ovu funkcionalnost:

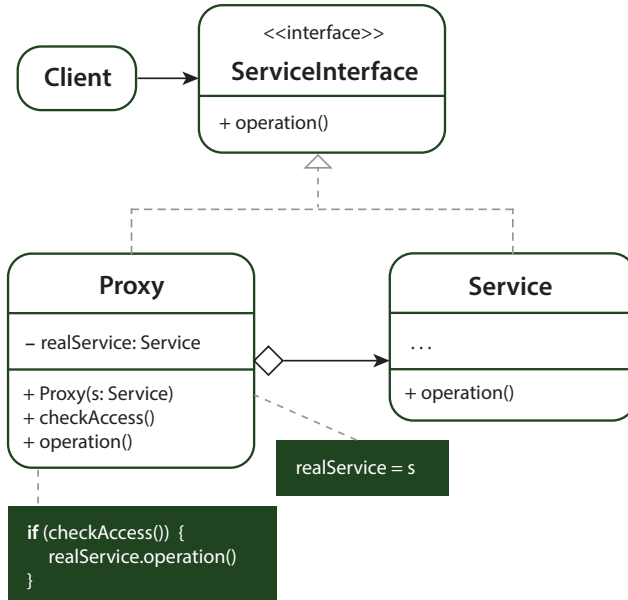
```
public class ProxyVideoDownloader implements VideoDownloader{
    private final VideoDownloader downloader = new RealVideoDownloader();
    public Video getVideo(String videoName){
        ...
    }
}
```

Proxy će implementirati isti interfejs (*VideoDownloader*) i pratiće sve preuzete fajlove koristeći *HashMap*-u. Ukoliko korisnik zahteva isti video više puta, *proxy* će mu vratiti taj video iz keša.

```
public class ProxyVideoDownloader implements VideoDownloader {
    private final Map<String, Video> videoCache = new HashMap<>();
    private final VideoDownloader downloader=new RealVideoDownloader();
    public Video getVideo(String videoName){
        if (!videoCache.containsKey(videoName)){
            videoCache.put(videoName, downloader.getVideo(videoName));
        }
        return videoCache.get(videoName);
    }
}
```

Dakle, osnovna struktura *Proxy* obrasca je:

Slika 57. Osnovna struktura *Proxy* obrasca (Shvets, 2019)



U prethodnim primerima, a prateći ovu osnovnu strukturu, Interfejs servisa (*ServiceInterface*) su bile klase *Internet* ili *VideoDownloader*. Klasa *Proxy* mora da implementira ovaj interfejs kako bi prikrla objekat servisa (*Service*). Klasa *Service* sa druge strane predstavlja stvarnu implementaciju tog interfejsa. Ova klasa obezbeđuje neku korisnu logiku aplikaciji. U prethodnim primerima to su bile klase *RealInternet* i *RealVideoDownloader*.

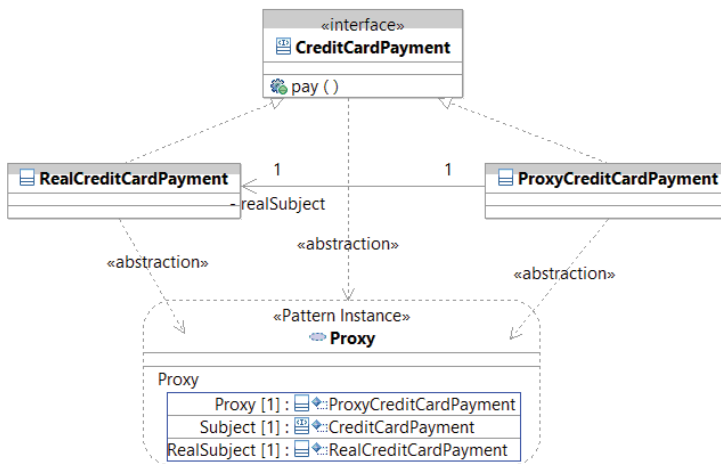
Proxy klasa ima referencirani atribut koji ukazuje na objekat *Service* i nakon što završi svoj posao, dalje prosleđuje zahtev *Service* klasi. U prethodnim primerima to su bile klase *ProxyInternet* i *ProxyVideoDownlaoder*.

Klasa *Client* treba da radi i sa *Proxy* i *Service* klasama preko istog interfejsa. Klijent može da prosledi *Proxy* objekat kodu koji očekuje *Service* objekat.

Dakle, *Proxy* obrazac omogućava kontrolu pristupa određenom objektu tako što izvršava neke operacije pre ili nakon što zahtev stigne do tog objekta. Pored toga, *Proxy* može da upravlja životnim ciklusom *Service* objekta ukoliko klijent ne brine o tome. Takođe, *Proxy* objekat će i dalje raditi iako *Service* objekat nije dostupan. Mogu se dodavati i nove *Proxy* klase bez menjanja klijentske i servisne klase.

Vežba 4-9. Primeniti *Proxy* obrazac na autorizaciju plaćanja kreditnom karticom.

Kreirati interfejs *CreditCardPayment* koji ima jednu metodu *pay()*. Konkretni servis nazvati *RealCreditCardPayment* koji će u okviru *pay()* metode ispisati sledeće "Payment is done by credit card!". U okviru *Proxy* obrasca proveriti da li je korisnik uspešno autentifikovan na sistem, ako jeste, plaćanje će se izvršiti kreditnom karticom, a ako nije proslediti korisniku sledeću poruku: "Please go to Bank and pay using receipt!".



Koraci rešenja:

Korak 1. Kreirati interfejs *CreditCardPayment* sa metodom *pay()*.

```
public interface CreditCardPayment {
    public void pay();
}
```

Korak 2. Kreirati konkretni servis za plaćanje kreditnom karticom, koji će u, našem primeru, ispisati da je plaćanje izvršeno kreditnom karticom.

```
public class RealCreditCardPayment implements CreditCardPayment {
    public void pay() {
        System.out.println("Payment is done by credit card!");
    }
}
```

Korak 3. Kreirati *Proxy* klasu koja će pozvati metodu *pay()* klase *RealCreditCardPayment* u slučaju da je korisnik uspešno autentifikovan, a ako nije u tom slučaju će ispisati da korisnik plati u banci. Kreirana je i klasa *AuthService*, sa *boolean* vrednošću polja *authenticated*.

```
//Proxy klasa
public class ProxyCreditCardPayment implements CreditCardPayment {
    private RealCreditCardPayment realCreditCardNo;
    private AuthService authService;
    public ProxyCreditCardPayment(AuthService authService) {
        this.authService = authService;
        realCreditCardNo = new RealCreditCardPayment();
    }
    public void pay() {
        if (authService.isAuthenticated()){
            realCreditCardNo.pay();
        } else {
            System.out.println("Please go to Bank and pay using receipt!");
        }
    }
}

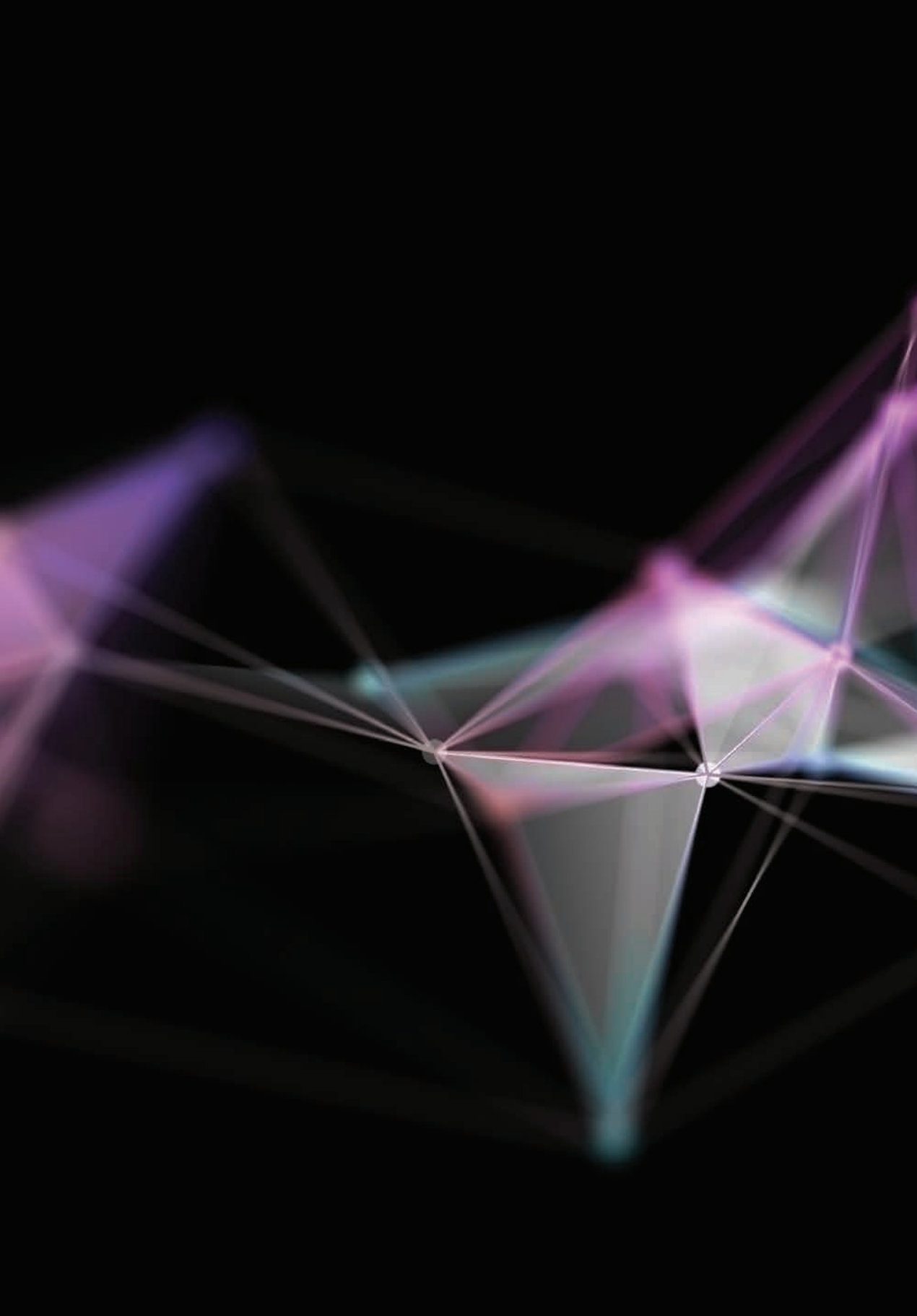
//Klasa AuthService sa boolean vrednošću polja authenticated:
public class AuthService {
    private boolean authenticated;
    public AuthService(final boolean authenticated) {
        this.authenticated = authenticated;
    }
    public boolean isAuthenticated() {
        return authenticated;
    }
    public void setAuthenticated(boolean authenticated) {
        this.authenticated = authenticated;
    }
}
```

Korak 4. Kreirati glavnu klasu *ProxyPatternApp* sa main metodom.

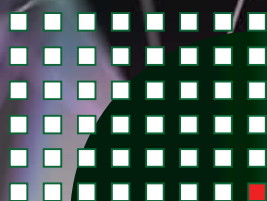
```
public class ProxyPatternApp {
    public static void main(String[] args) {
        CreditCardPayment ccp1 = new ProxyCreditCardPayment(new
            AuthService(true));
        ccp1.pay();
        CreditCardPayment ccp2 = new ProxyCreditCardPayment(new
            AuthService(false));
        ccp2.pay();
    }
}
```

Korak 5. Rezultat izvršavanja.

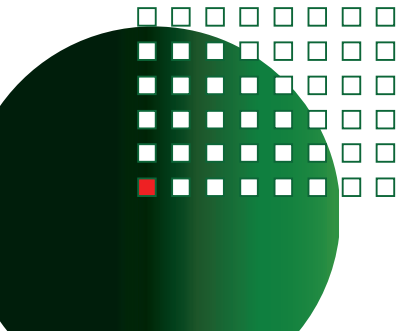
Payment is done by credit card!
Please go to Bank and pay using receipt!



POGLAVLJE 5



OBRASCI
PROJEKTOVANJA
PONAŠANJA



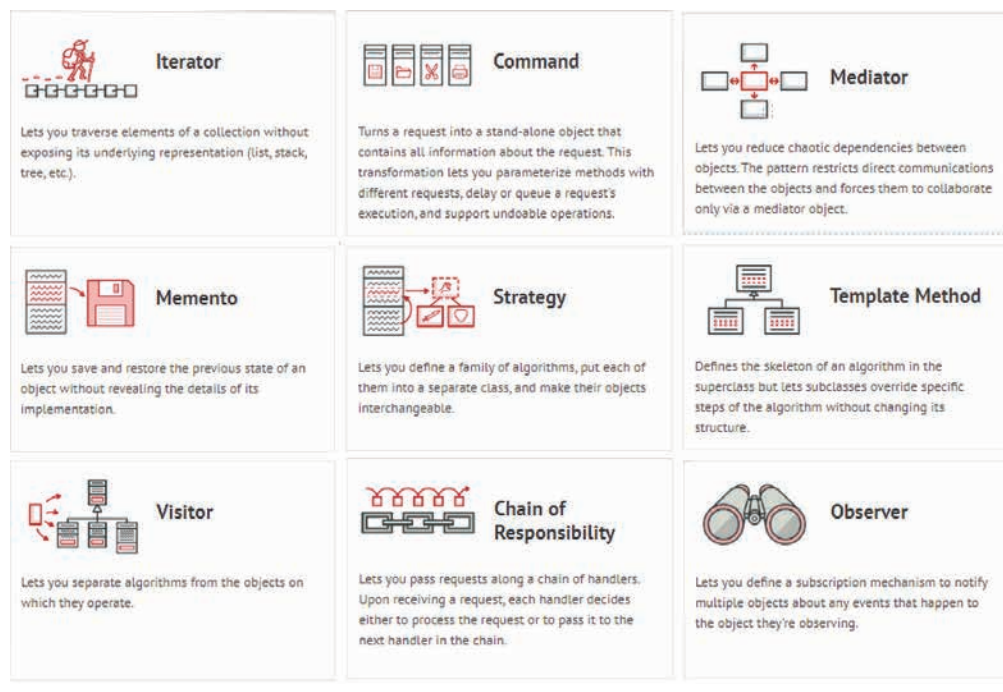
5. OBRASCI PROJEKTOVANJA PONAŠANJA

Obrasci projektovanja ponašanja (*Behavioral design patterns*) identifikuju zajedničke obrasce komunikacije između objekata. Oni se pre svega bave algoritmima, interakcijama i raspoređivanjem odgovornosti između klasa. Interakcija između klasa bi trebala da bude takva da klase mogu lako da komuniciraju, ali da i dalje ostane labava povezanost između njih. Obrasci ponašanja rešavaju različite probleme pomoću uspostavljanja odgovarajućih odnosa i načina razmenjivanja informacija (poruka) između objekata.

Neki od obrazaca koji spadaju u ovu grupu su:

- ♦ **Iterator** - omogućava iterativni (sekvencijalni) pristup elementima kolekcije bez poznavanja načina kako je kolekcija struktuirana (lista, stek, stablo, itd.).
- ♦ **Command** - enkapsulira zahtev komande u objekat.
- ♦ **Mediator** - smanjuje zavisnosti između objekata, tako što sprečava njihovu direktnu komunikaciju, već se ta komunikacija odvija preko objekta posrednika – *Mediator*.
- ♦ **Memento** - omogućava da se, bez narušavanja enkapsulacije, hvata i čuva interno stanje objekta kako bi ono kasnije moglo biti vraćeno u sačuvano stanje. Ovaj obrazac se koristi kada je potrebno da objekat vratimo u neko od njegovih prethodnih stanja, na primer, kada korisnik zahteva „undo“ operaciju nad objektom.
- ♦ **Strategy** - enkapsulira familiju algoritama i selektuje jedan od njih tokom izvršavanja programa.
- ♦ **Template method** - definiše redosled izvršavanja seta metoda nekog konkretnog algoritma tj. dozvoljava potklasama da predefinišu određene korake u algoritmu, ali bez menjanja strukture tog algoritma.
- ♦ **Visitor** - kada je potrebno izvršiti operacije nad grupom sličnih vrsta objekata, primenom ovog obrasca te operacije se izmeštaju iz objekata i stavljaju u posebnu klasu koja se zove *Visitor*.
- ♦ **Chain of responsibility** - omogućava prosleđivanje zahteva duž lanca rukovaoca. Nakon prijema zahteva, svaki rukovaoc odlučuje da li će ga obraditi ili proslediti sledećem rukovaocu u lancu.
- ♦ **Observer** - omogućava da se definiše mehanizam automatskog obaveštavanja više zainteresovanih klasa o svim promenama koje se dešavaju sa klasom koju posmatraju.
- ♦ **State** - omogućava objektu da promeni svoje ponašanje kada se njegovo unutrašnje stanje promeni.

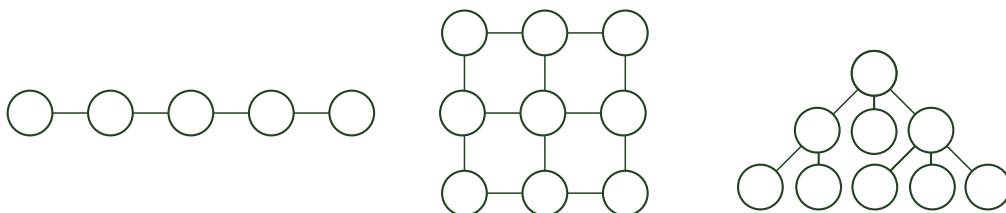
Slika 58. Obrasci ponašanja objašnjeni u ovom poglavlju.



5.1. ITERATOR

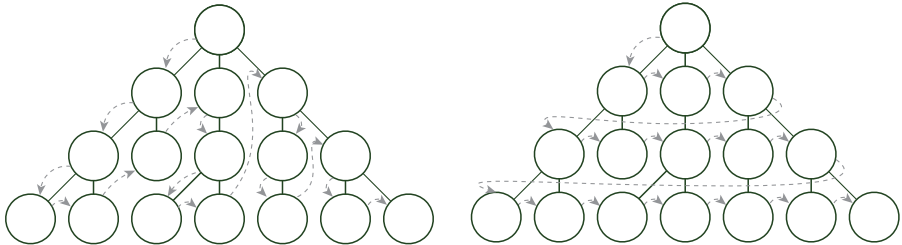
Iterator je veoma jednostavan i često korišćen obrazac. S obzirom da postoji puno različitih struktura podataka ili kolekcija (npr. stablo, lista, *ArrayList*, *HashMap*), Iterator vam omogućava da prođete elemente kolekcije bez potreba da znate kako je kolekcija struktuirana.

Slika 59. Različiti tipovi kolekcija.



Ukoliko su klase organizovane u listu, iterator bi bio lak zadatak (koristila bi se *loop* petlja za prolazak kroz kolekciju objekata). Međutim, da li je moguće sekvencionalno proći kroz kolekciju, kada je ona složene strukture ili kada je potrebno drugačije proći kroz kolekciju?

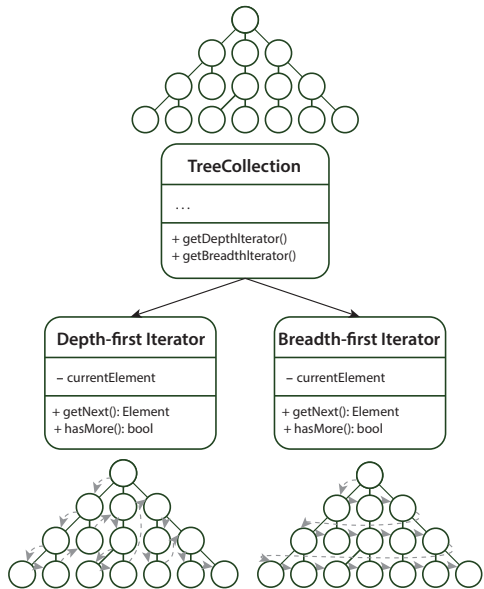
Slika 60. Različiti načini prolaska kroz istu sturkturu kolekcije objekata.



Osnovna ideja iteratora je da se izvuče ponašanje kolekcije u posebnu klasu koja se naziva *Iterator*. Objekat iteratora enkapsulira sve detalje prolaska kroz kolekciju, kao što su trenutna pozicija i koliko elemenata je ostalo do kraja. Usled toga, nekoliko iteratora može proći kroz istu kolekciju, u isto vreme, nezavisno jedna od druge.

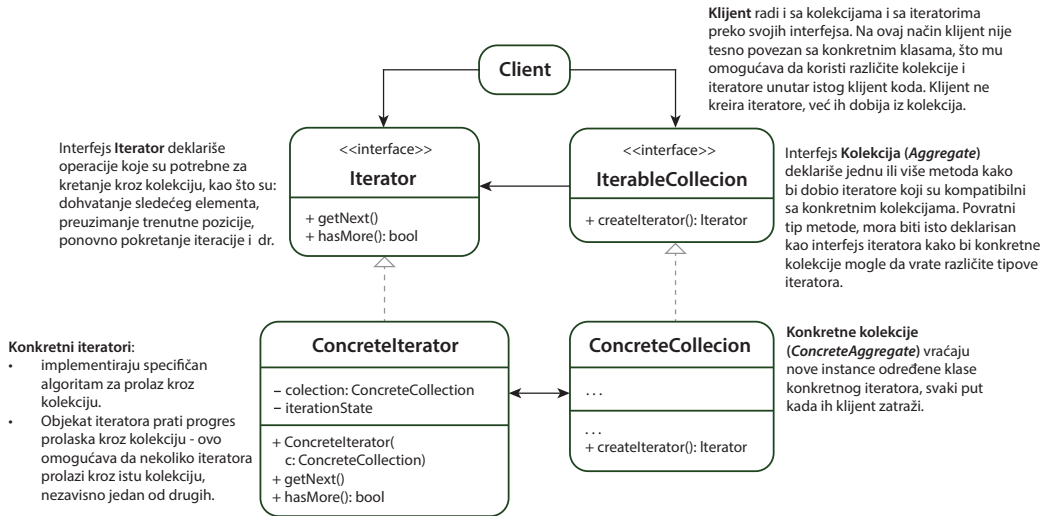
Svi iteratori moraju da implementiraju isti interfejs, s toga je kôd kompatibilan sa bilo kojim tipom kolekcije. Ukoliko je potreban drugačiji način prolaska kroz kolekciju, dovoljno je da se kreira nova iterator klasa, bez potrebe da se menja kolekcija ili klijent.

Slika 61. Iteratori implementiraju različite algoritme prolaska kroz kolekciju (Shvets, 2019).



Osnovna struktura *Iterator* obrasca je prikazana na slici 62.

Slika 62. Osnovna struktura *Iterator* obrasca (modifikovano sa Shvets, 2019).



Objašnjenje osnovne strukture koja je praćena alatom IBM RSA:

- ♦ Klasa *ConcreteAggregate* implementira *createIterator()* koji vraća iterator za svoju kolekciju. Svaki konkretni agregator (Kolekcija) je odgovoran za intanciranje konkretnog iteratora.
- ♦ Interfejs *Iterator* omogućava set metoda za prolazak ili modifikaciju kolekcije, tj. pored *next()* ili *hasNext()* metode, takođe omogućava funkcije za pretragu, uklanjanje elemenata kolekcije i dr.

Vežba 5-1. Razviti aplikaciju koja prikazuje nazive tema prolaskom kroz kolekciju tema.

Razmotrimo sledeći problem:

- ♦ Ukoliko se koristi jednostavan niz (*Array*) za čuvanje tema, onda se kroz kolekciju tema može proći na sledeći način:

```
for (int i = 0; i < TopicList.length; i++)
    Topic currentTopic = TopicList[i];
```

- ♦ Ukoliko se koristi *ArrayList*-a, onda se kroz kolekciju može proći na sledeći način:

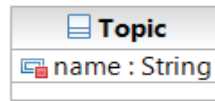
```
for (int i = 0; i < TopicList.size(); i++)
    Topic currentTopic = (Topic)TopicList.get(i);
```

Ukoiko bi imali i druge kolekcije, onda bi se kôd menjao. Stoga je rešenje korišćenje Iterator obrasca.

Koraci primene *Iterator* obrasca su:

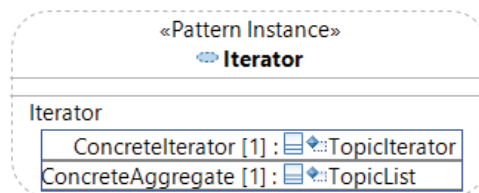
Korak 1. Na dijagramu klasa kreirati klasu *Topic* i prevući i popuniti Iterator obrazac.

Dodati na dijagram klasa novu klasu *Topic* koja ima samo jedan atribut: naziv teme (*name* tipa *String*).



Prevući *Iterator* obrazac i uneti:

- ◆ konkretnu implementaciju iteratora *TopicIterator* u *ConcreteIterator*.
- ◆ konkretnu listu ili kolekciju *TopicList* u *ConcreteAggregate*. Konkretna kolekcija vraća nove instance tema (*Topics*) prema konkretnom iteratoru, svaki put kada ih klijent zatraži.

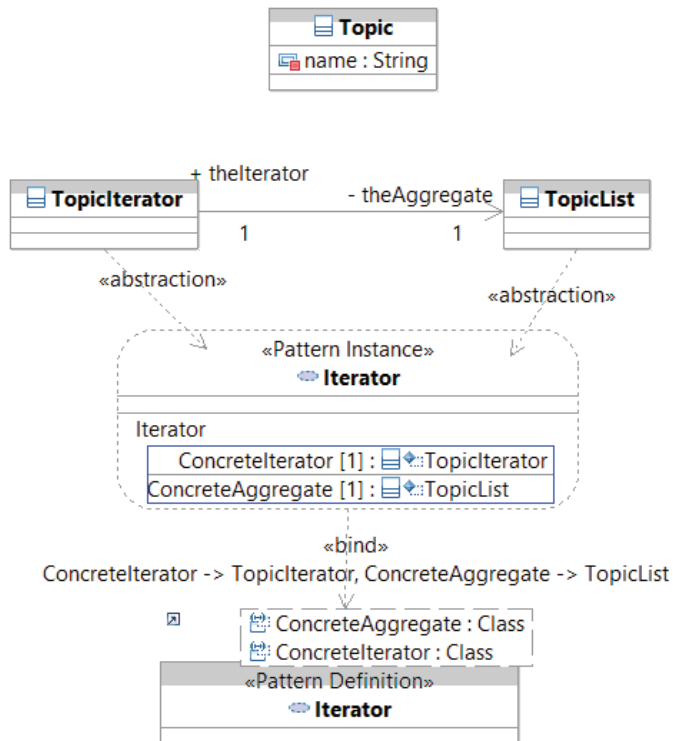


Kako je obrazac *Iterator* direktno podržan u javinim standardnim bibliotekama, *Interfejsi* *Iterator* i *AbstractCollection* se neće prikazati na dijagramu klasa, već će samo biti importovani u kodu:

```
import java.util.AbstractCollection;
import java.util.Iterator;
```

Interfejs *Iterator* definiše metode kretanja i rada sa kolekcijom tema, kao što su: *void remove()*; *Object next()*; *Object currentItem()* i *boolean hasNext()*. *Interfejs* *AbstractCollection* definiše kreiranje iteratora i ima jednu metodu: *iterator():IteratorObject*;

Prilikom generisanja svih relacija, dobiće se sledeći model:



Korak 2. Izgenerisati, dopuniti i izvršiti kod.

Izgenerisan i dopunjen kod je:

```

public class Topic {
    private String name;

    public Topic(String name) {
        super();
        this.name = name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }
}

```

```
import java.util.AbstractCollection;
import java.util.Iterator;
public class TopicList extends AbstractCollection {
    private Topic[] topics;

    public Iterator iterator() {
        return new TopicIterator(topics);
    }

    public int size() {
        throw new java.lang.UnsupportedOperationException();
    }

    public TopicList(Topic[] topics) {
        super();
        this.topics = topics;
    }
}

import java.util.Iterator;
public class TopicIterator implements Iterator {

    private Topic[] topics;
    private int currentIndex;

    public Topic next() {
        return topics[currentIndex++];
    }

    public boolean hasNext() {
        if (currentIndex>=topics.length){
            return false;
        }
        return true;
    }

    public Topic currentItem(){
        return topics[currentIndex];
    }

    public void remove() {
        currentIndex=0;
    }

    public TopicIterator(Topic[] topicList) {
        this.topics=topicList;
        currentIndex=0;
    }
}
```

Dodati klasu sa *main* metodom:

```
import java.util.AbstractCollection;
import java.util.Iterator;

public class TopicMain {
    public static void main(String[] args) {
        Topic[] topics=new Topic[3];
        topics[0]=new Topic("Topic 1");
        topics[1]=new Topic("Topic 2");
        topics[2]=new Topic("Topic 3");

        AbstractCollection list = new TopicList(topics);
        Iterator iterator=list.iterator();
        while(iterator.hasNext()){
            Topic currentTopic = (Topic) iterator.next();
            System.out.println(currentTopic.getName());
        }
    }
}
```

Rezultat izvršavanja ovog jednostavnog primera je:

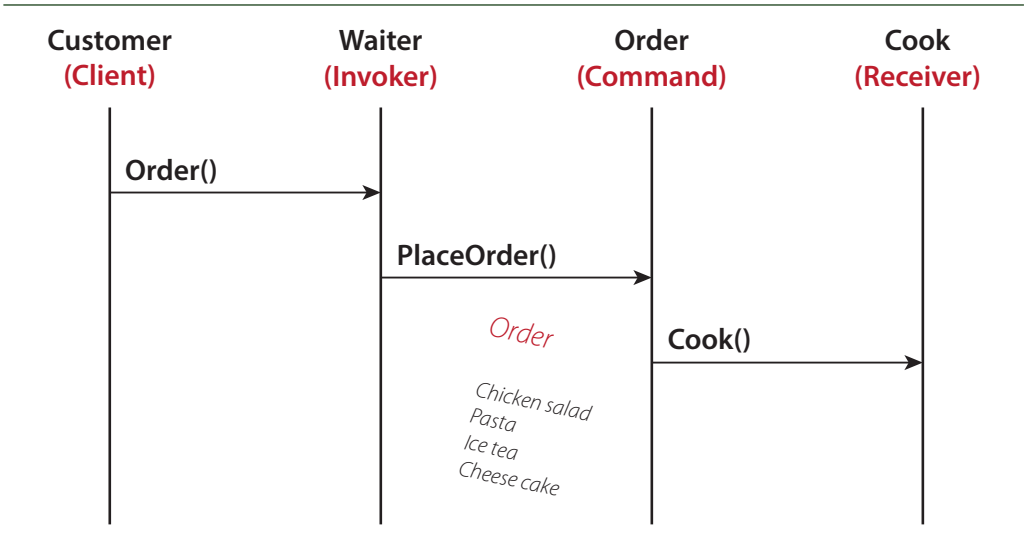
```
Topic 1
Topic 2
Topic 3
```

Zadatak 5-1. Kreirati aplikaciju koja će omogućiti klijentu da lista nazive pesama i njihovih autora, redosledom koji on odabere.

5.2. COMMAND

Command obrazac razdvaja onog ko zahteva akciju od onog ko izvršava akciju (Freeman, Eric, Robson, 2021). Na primeru poručivanja hrane u restoranu mogu se mapirati obavezne klase *Command* obrasca (Slika 63).

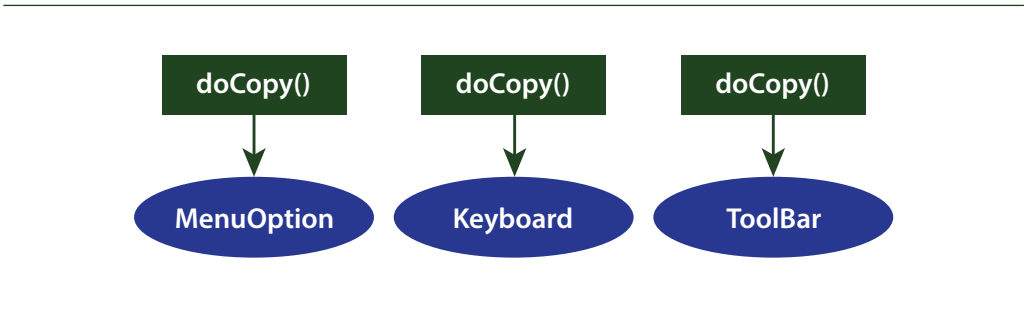
Slika 63. Mapiranje toka procesa poručivanja hrane u restoranu sa klasama *Command* obrasca.



Radi boljeg razumevanja problema koji ovaj obrazac rešava, sagledaćemo primer na koji način se sve može kopirati neki tekst ili slika. Postoji nekoliko opcija (Slika 64), na primer:

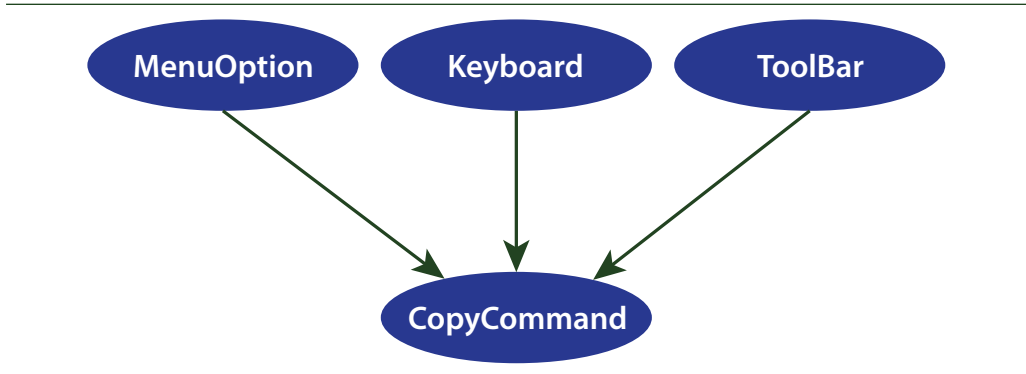
- ♦ može se u meniju odabrati opcija *Copy*
- ♦ mogu se koristiti prečice na tastaturi za kopiranje (npr. Ctrl+C)
- ♦ može se kliknuti na ikonu u okviru *toolbar*-a.

Slika 64. Opcije korišćenja *copy* komande.



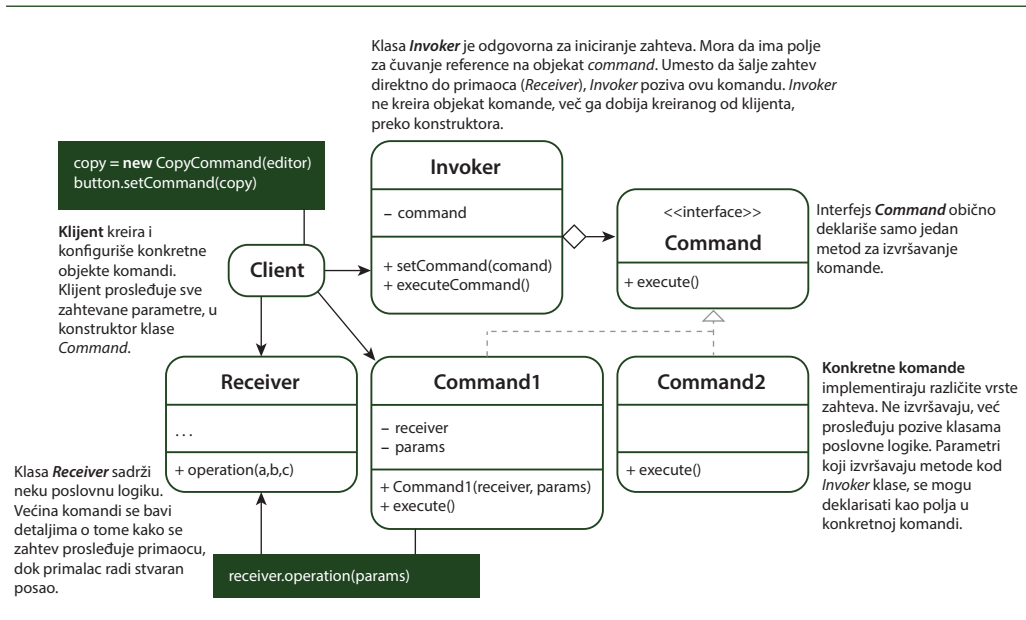
Sve ove spomenute opcije mogu izvršavati komandu *Copy*, međutim činjenica je da ne želite da skladištite kôd ove komande u svaku pojedinačnu opciju. Rešenje je da se komanda *Copy* stavi na jedno odvojeno, jedinstveno mesto, kako bi se izbeglo njeno ponavljanje na više mesta (Slika 65).

Slika 65. Primer izvlačenja *copy* komande iz različitih opcija.

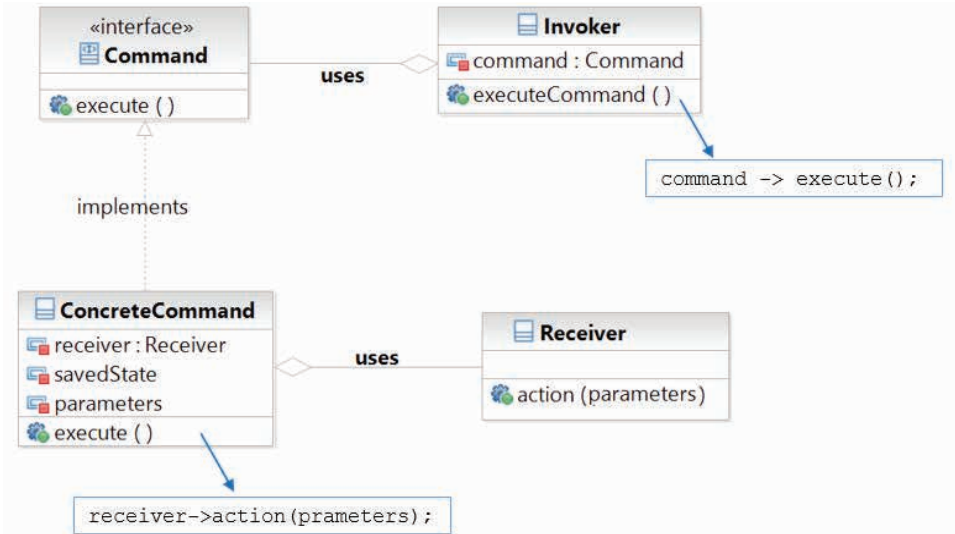


Osnovna struktura Command obrasca je:

Slika 66. Osnovna struktura *Command* obrasca (modifikovano sa Shvets, 2019).



Da bi kreirali komandu, neophodno je prvo da znamo ko je izvršava, tako da nam je neophodna referenca na *Receiver* (Slika 67).

Slika 67. Način rada *Command* obrasca.

Referencu na *receiver* prosleđujemo konstruktoru konkretne komande (*ConcreteCommand* klasa). Komanda (na slici 67 metoda *execute()*) koju implementira klasa *ConcreteCommand* se već nalazi na interfejsu Komande (*Command*). Komanda *executeCommand()* je zatim prosleđena *Invoker* klasi. *Invoker* može biti na primer: UI dugme, UI meni i dr.

Dovoljno je da kreirate kontejner za vaše akcije (klasa *ConcreteCommand*), koji tačno zna ko je *Receiver* tj. ko izvršava vašu akciju i to prosledite *Invoker* klasi. Kada god se poziva neka konkretna komanda, *Invoker* samo poziva dodeljenu komandu (`command -> execute();`) koju kontejner (*ConcreteCommand* klasa) prepoznaje i prosleđuje na izvršavanje *Receiver* klasi.

Dakle, osobine *Command* obrasca projektovanja su:

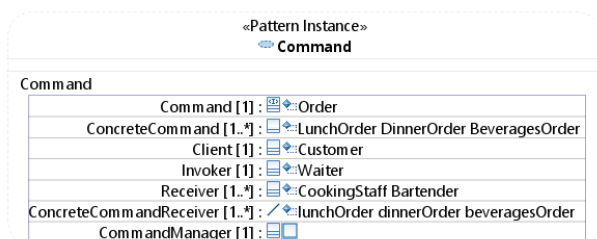
- ♦ implementira labavo povezivanje primenom zahtev-odziv (*request-response*) modela
- ♦ enkapsulira zahteve u formi jednog objekta
- ♦ poznato je ko će da obradi zahtev (*Receiver*), kao i šta je zahtev (*ConcreteCommand*), ali ono što se nije poznato je ko će pozvati komandu.
- ♦ zahtev se šalje *Invoker* klasi, a *Invoker* prosleđuje enkapsuliranom *Command* objektu. *Command* objekat prosleđuje zahtev odgovarajućoj metodi *Receiver*-a da izvrši određenu akciju.
- ♦ Klijent aplikacija kreira *receiver* objekat i pridružuje ga *command* objektu. Zatim kreira *invoker* objekat i pridružuje mu *command* objekat radi izvršavanja akcije.

Vežba 5-2. Primeniti *Command* obrazac na sistem poručivanja hrane u restoranu.

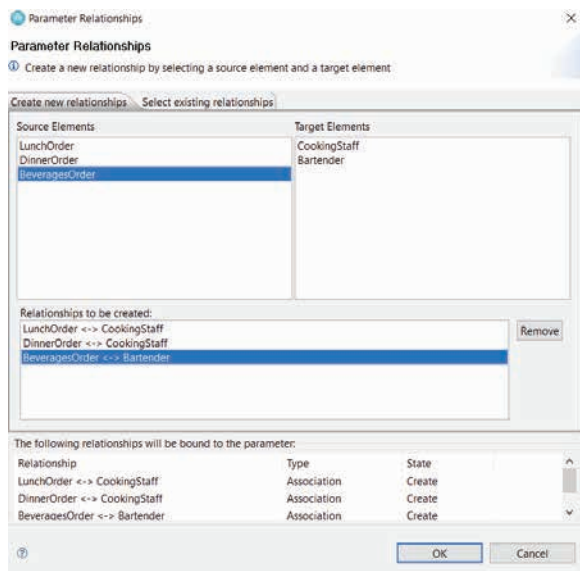
Klijent može poručiti ručak (*LunchOrder*) ili večeru (*DinnerOrder*) od konobara (*Waiter*). Konobar kreira odgovarajuću porudžbinu i prosleđuje je kuvarima (*CookingStaff*).

Koraci rada na vežbi su:

Korak 1. Prevući *Command* obrazac i dopuniti sa odgovarajućim klasama.

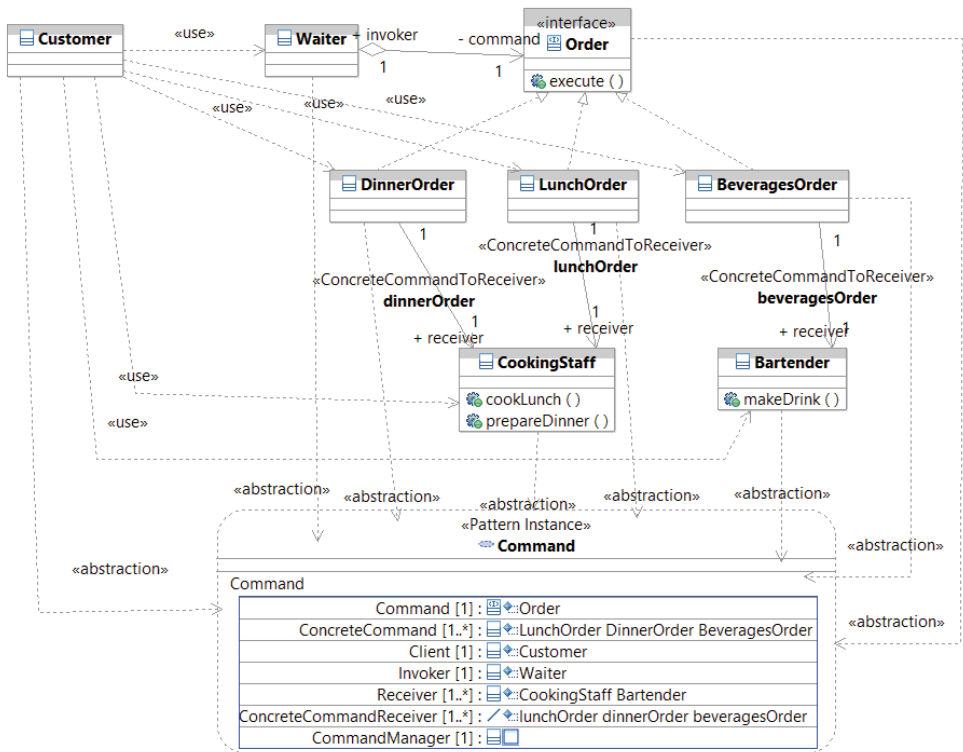


Pod *ConcreteCommandReceiver*, potrebno je mapirati određene komande sa odgovarajućim *Receiverom*, odnosno klasom koja izvršava konkretne komande. Na narednoj slici je prikazano mapiranje, gde *CookingStaff* izvršava komande *LunchOrder* i *DinnerOrder*, a *Bartender* izvršava *BeveragesOrder*.



Klasa *CommandManager* je opcionalna i ona je odgovorna za upravljanje kolekcijom komandnih objekata koje je kreirao *Invoker* objekat. Specifične odgovornosti klase *CommandManager* su upravljanje undo/redo pozivom, sekvenciranje komandi i vremensko raspoređivanje komandi. *Invoker* instancira *Command* objekat sa informacijama koje su mu potrebne. Po potrebi poziva operacije *CommandManager* klase za finalno izvršavanje operacije. Klase *CommandManager* su obično nezavisne od aplikacija u kojima se koriste i mogu da budu visoko ponovno upotrebljive.

Korak 2. Izvršiti povezivanje obrasca sa klasama i dopuniti klase sa metodama.



Korak 3. Generisati, dopuniti i izvršiti kod.

```

public class CookingStaff {
    public void cookLunch() {
        System.out.println("Lunch is in progress!");
    }
    public void prepareDinner() {
        System.out.println("Dinner is in progress!");
    }
}

public class Bartender {
    public void makeDrink() {
        System.out.println("Beverage is in progress!");
    }
}

public interface Order {
    public void execute();
}

public class LunchOrder implements Order {
    public CookingStaff receiver;
    public CookingStaff getReceiver() {
        return receiver;
    }
    public void setReceiver(CookingStaff receiver) {
        this.receiver = receiver;
    }
    //dopuniti metodu sa receiver.cookLunch()
    public void execute() {
        receiver.cookLunch();
    }
    public LunchOrder(CookingStaff receiver) {
        this.receiver = receiver;
    }
}

public class DinnerOrder implements Order {
    public CookingStaff receiver;
    public CookingStaff getReceiver() {
        return receiver;
    }
    public void setReceiver(CookingStaff receiver) {
        this.receiver = receiver;
    }
    //dodati reciver.prepareDinner()
    public void execute() {
        receiver.prepareDinner();
    }
    public DinnerOrder(CookingStaff receiver) {
        this.receiver = receiver;
    }
}

```

```

public class BeverageOrder implements Order {
    public Bartender receiver;
    public Bartender getReceiver() {
        return receiver;
    }
    public void setReceiver(Bartender receiver) {
        this.receiver = receiver;
    }
    //dodati receiver.makeDrink()
    public void execute() {
        receiver.makeDrink();
    }
    public BeverageOrder(Bartender receiver) {
        this.receiver = receiver;
    }
}

public class Waiter {
    private Order command;
    public Order getCommand() {
        return command;
    }
    public void setCommand(Order command) {
        this.command = command;
    }
    //Možemo dodati parametarski konstruktor
    public void invokeCommand(Order command) {
        command.execute();
    }
    public void invokeCommand() {
        this.command.execute();
    }
}

//U okviru klase Customer možemo dodati main() metodu
public class Customer {
    public static void main(String[] args) {
        CookingStaff aCookingStaff = new CookingStaff();
        LunchOrder aLunchOrder = new LunchOrder(aCookingStaff);
        Bartender aBartender=new Bartender();
        BeverageOrder aBeverageOrder = new BeverageOrder(aBartender);
        Waiter waiter = new Waiter();
        waiter.setCommand(aLunchOrder);
        waiter.invokeCommand();
        waiter.invokeCommand(aBeverageOrder);
    }
}

```

Rezultat izvršavanja koda je:

Lunch is in progress!

Beverage is in progress!

Zadatak 5-2. Primeniti *Command* obrazac na komandu butovanje (*Booting*) računara.

Korisnik (*User*) preko *Switch* poziva izvršava konkretnu komandu i to *Restart* ili *ShutDown*. Ove konkretne komande izvršava sam računar (*Computer*). Primeniti obrazac projektovanja i prikazati finalni dijagram klasa.

Zadatak 5-3. Primeniti odgovarajući obrazac na sistem za daljinsko upravljanje svetlom i ventilacijom u kući.

Kreirati aplikaciju za daljinsko automatsko upravljanje svetlom (*Light*) i ventilacijom (*Fan*) u kući. *Light* i *Fan* su klase koje će izvršavati sledeće komande:

- Za klasu *Light*: *turnOn()* i *turnOff()*. Metoda *turnOn* će ispisati „Light is on.“, a metoda *turnOff* „Light is off.“
- Za klasu *Fan*: *start()* i *stop()*. Metoda *start()* će prikazati „Fan started...“, a *stop()* „Fan stopped.“

Klijent koristi daljinski za automatsko upravljanje kućnim resursima (*HomeAutomation Remote*), na kome se izvršava metoda *buttonPressed()*, koja poziva konkretne komande, preko interfejsa *HomeCommand*, da preuzmu izvršavanje (*execute*) zahteva klijenta. Konkretne implementacije komandi su:

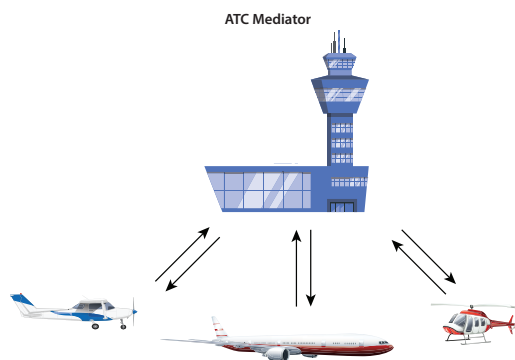
- *StartFan* prikazuje Starting Fan.
- *StopFan* prikazuje Stopping Fan.
- *TurnOnLight* prikazuje Turning on light.
- *TurnOffLight* prikazuje Turning off light.

Omogućiti klijentu da upali svetlo u dnevnoj sobi (*livingRoomLight*), a da ugasi u spavaćoj (*bedRoomLight*). Zatim da uključi ventilaciju u dnevnoj i spavaćoj, a potom da je zaustavi u obe prostorije.

5.3. MEDIATOR

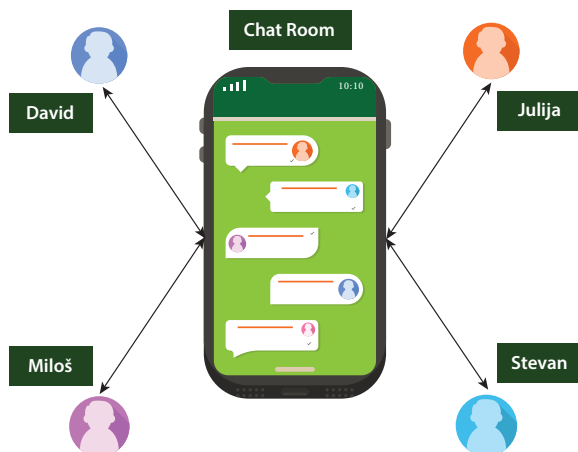
Mediator je obrazac koji smanjuje zavisnosti između objekata, tako što sprečava njihovu direktnu komunikaciju, već se ta komunikacija odvija preko objekta posrednika – *Mediator*. Upoređujući sa realnim primerom, kao na primer, sa tornjem za kontrolu saobraćaja na aerodromima, ukoliko bi svi avioni međusobno komunicirali radi dogovora oko redosleda sletanja, to bi stvorilo popriličan haos. Umesto toga avioni šalju svoj status kontroli leta, a oni im šalju potvrdu o sletanju (Slika 68).

Slika 68. *Mediator* obrazac na primeru tornja za kontrolu leta.



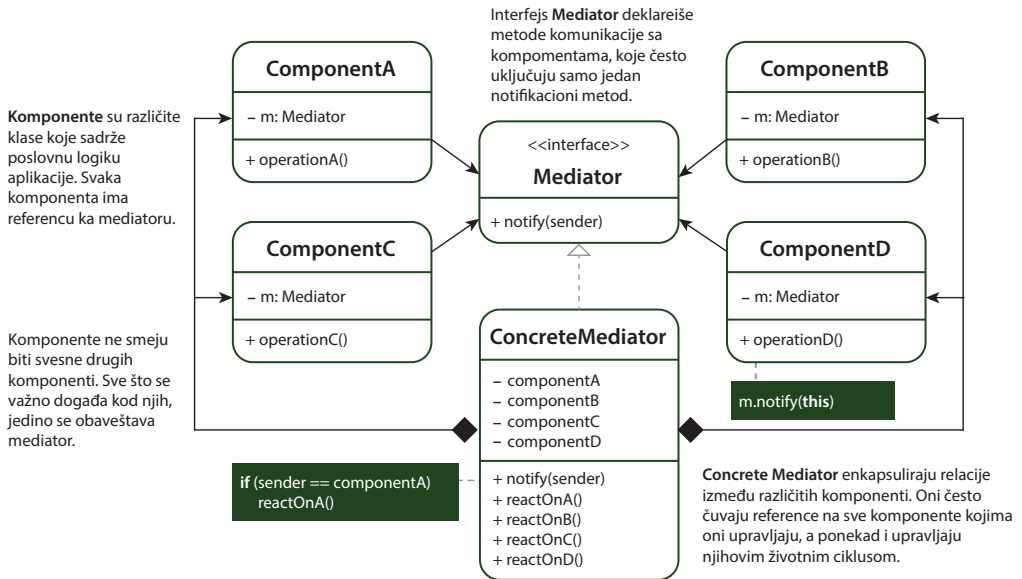
U aplikaciji za četovanje, gde postoji veliki broj učesnika, ne bi bila dobra ideja da se svi međusobno povežu, jer bi broj konekcija samo rastao. Najbolje rešenje je da postoji hab (*ChatRoom*) gde će se svi učesnici povezati (Slika 69). Ovaj primer prikazuje kako obrazac *Mediator* obezbeđuje slabo povezanu komunikaciju svih instanci klase *Users* koje su registrovane u okviru jedne instance klase *Chatroom*.

Slika 69. *ChatRoom* kao *Mediator* klasa.



Prema osnovnoj strukturi *Mediator* obrasca (Slika 70) može se videti da *Mediator* objekat enkapsulira znanje o interakciji grupe objekata i s obzirom da objekti ne referenciraju direktno jedni druge, omogućeno je slabo povezivanje (*loose coupling*) objekata.

Slika 70. Osnovna struktura *Mediator* obrasca (modifikovano sa Shvets, 2019).



Neke osnovne karakteristike Mediator obrasca su:

- ♦ objekti ne pamte reference jedni na druge
- ♦ definiše objekat koji enkapsulira način interakcije između objekata, tj. složena interakcija objekata je centralizovana
- ♦ objekti i mediator se mogu nezavisno menjati
- ♦ veze više-prema-više su zamenjene vezama jedan-prema-više što olakšava razumevanje i izmenu.
- ♦ logika interakcije objekata je odvojena od njihovog individualnog ponašanja.

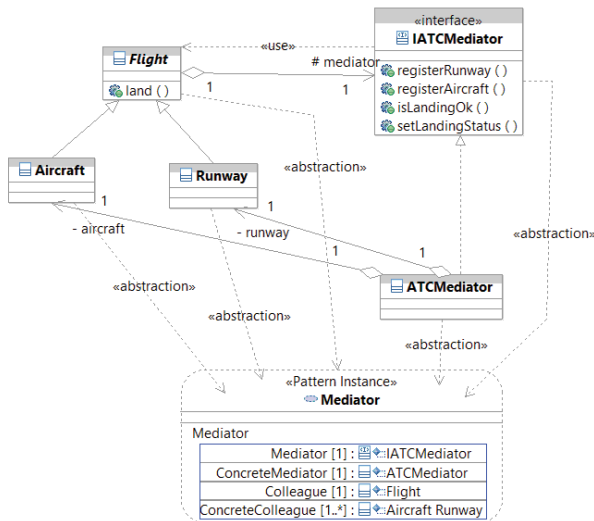
Klase koje učestvuju u ovom obrascu su:

- ♦ *Mediator*: definiše interfejs za komunikaciju među objektima klase *Colleague*
- ♦ *ConcreteMediator*: implementira mehanizme kooperativnog ponašanja
- ♦ *Colleague*: svaka *Colleague* klasa zna svoj *Mediator* objekat
- ♦ *ConcreteColleague*: Svaka instanca klase *Colleague* komunicira samo sa svojim posrednikom bez obzira da li komunicira sa drugim instancama.

Vežba 5-3. Primeniti *Mediator* obrazac na aplikaciju za upravljanje sletanjima aviona.

Kreirati aplikaciju za *AirTrafficControl* (ATC) koja će upravljati sletanjima aviona. Metode koje će implementirati su: *registerRunway* (registrovanje piste), *registerAircraft* (registrovanje aviona), *isLandingOk* (*Boolean* vrednost) i *setLandingStatus* (prihvata *boolean* vrednosti statusa sletanja). Klase *Aircraft* i *Runway* komuniciraju preko ATC klase, a nasleđuju apstraktnu klasu *Flight* koja sadrži metodu *land()*. *Aircraft* klasa sadrži još jednu metodu *getReady*, u okviru koje se prikazuje spremnost aviona za sletanje. U okviru klijent aplikacije, prikazati statuse *getReady* i *land*, aviona *sparrow101*, koji treba da sleti na glavnu pistu (*mainRunway*).

Rešenje vežbe:



```
//Interfejs IATCMediator
```

```
public interface IATCMediator {
    public void registerRunway(Runway runway);
    public void registerAircraft(Aircraft aircraft);
    public boolean isLandingOk();
    public void setLandingStatus(boolean status);
    public void notify(Flight colleague);
}
```

```
//apstraktna klasa Flight komande
```

```
public abstract class Flight {
    protected IATCMediator mediator;
    public IATCMediator getMediator() {
        return mediator;
    }
}
```

```
        public void setMediator(IATCMediator mediator) {
            this.mediator = mediator;
        }
    public Flight(IATCMediator mediator) {
        super();
        this.mediator = mediator;
    }
    public abstract void land();
}

public class Aircraft extends Flight {
    private IATCMediator atcMediator;
    public Aircraft(IATCMediator mediator, IATCMediator atcMediator)
{
        super(mediator);
        this.atcMediator = atcMediator;
    }
    @Override
    public void land() {
        if (atcMediator.isLandingOk()){
            System.out.println("Successfully landed.");
            atcMediator.setLandingStatus(true);
        }
        else
            System.out.println("Waiting for landing");
    }
    public void getReady() {
        System.out.println("Ready for landing.");
    }
}

public class Runway extends Flight {
    private IATCMediator atcMediator;
    public Runway(IATCMediator atcMediator) {
        super(atcMediator);
        this.atcMediator = atcMediator;
        atcMediator.setLandingStatus(true);
    }
    @Override
    public void land() {
        System.out.println("Landing permission granted.");
        atcMediator.setLandingStatus(true);
    }
}

public class ATCMediator implements IATCMediator {
    private Aircraft aircraft;
    private Runway runway;
    private boolean land;
    //getters and setters
    public Aircraft getAircraft() {
        return aircraft;
    }
}
```

```

    public void setAircraft(Aircraft aircraft) {
        this.aircraft = aircraft;
    }
    public Runway getRunway() {
        return runway;
    }
    public void setRunway(Runway runway) {
        this.runway = runway;
    }
    public boolean isLand() {
        return land;
    }
    public void setLand(boolean land) {
        this.land = land;
    }
    public void registerRunway(Runway runway) {
        this.runway= runway;
    }
    public void registerAircraft(Aircraft aircraft) {
        this.aircraft=aircraft;
    }
    public boolean isLandingOk() {
        return land;
    }
    public void setLandingStatus(boolean status) {
        land=status;
    }
    public void notify(Flight colleague) {
        if (colleague instanceof Aircraft) {
        } else if (colleague instanceof Runway) {
        }
    }
}

public class Client {
    public static void main(String[] args) {
        ATCMediator atcMediator = new ATCMediator();
        Aircraft sparrow101 = new Aircraft(atcMediator, atcMediator);
        Runway mainRunway = new Runway(atcMediator);

        atcMediator.registerAircraft(sparrow101);
        atcMediator.registerRunway(mainRunway);
        sparrow101.getReady();
        mainRunway.land();
        sparrow101.land();
    }
}

```

Izlaz:

Ready for landing.
 Landing permission granted.
 Successfully landed.

Zadatak 5-4. Primeniti *Mediator* obrazac na aplikaciju za četovanje.

Chatroom je centralno mesto kroz koje teče svaka vrsta komunikacije između korisnika (*Users*).

5.4. MEMENTO

Memento obrazac omogućava da se, bez narušavanja enkapsulacije, hvata i čuva interno stanje objekta kako bi ono kasnije moglo biti vraćeno u sačuvano stanje. Ovaj obrazac se koristi kada je potrebno da objekat vratimo u neko od njegovih prethodnih stanja, na primer, kada korisnik zahteva „undo“ operaciju nad objektom.

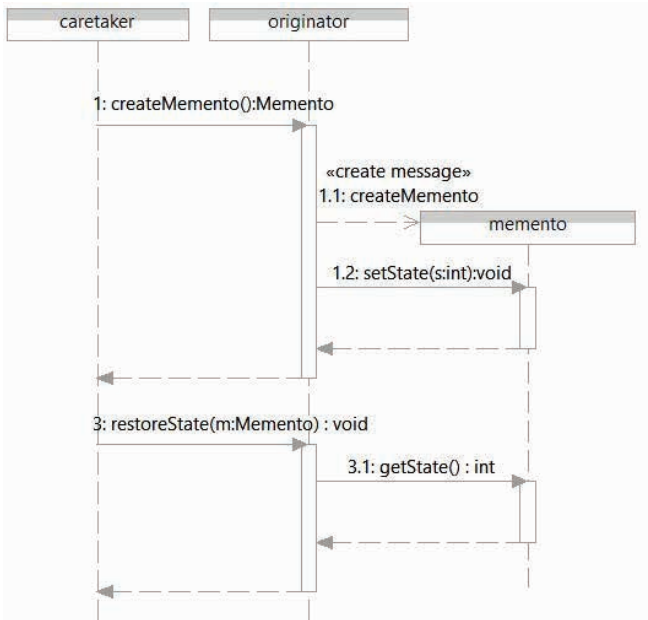
Memento ima sledeće ciljeve:

- ♦ da sačuva bitna stanja sistemski ključnog objekta
- ♦ da održi enkapsulaciju ključnog objekta.

Osnovni elementi, odnosno klase i/ili objekti koje učestvuju u ovom obrascu su (Slika 71):

- ♦ **Originator** - ova klasa predstavlja objekat čije stanje želimo da sačuvamo. *Originator* ima informaciju o svom trenutnom stanju i može kreirati objekat *Memento* koji sadrži snimak tog stanja. Koristi *Memento* da povрати svoje unutrašnje stanje.
- ♦ **Memento** - *Memento* je objekat koji čuva interno stanje objekta klase *Originator* od neželjenog menjanja. *Memento* treba da sačuva dovoljno podataka o stanju objekta *Originator* kako bi kasnije mogao da povрати njegovo prvobitno stanje.
- ♦ **Caretaker** - ova klasa je odgovorna za čuvanje *Memento* stanja objekata. *Caretaker* može čuvati više *Memento* objekata kako bi mogao vratiti *Originator* klasi prethodna stanja. On takođe pruža interfejs preko kojeg *Originator* može zatražiti povrat stanja iz *Memento* objekata. Nikada ne ispituje niti otvara sadržaj *Memento* objekta.

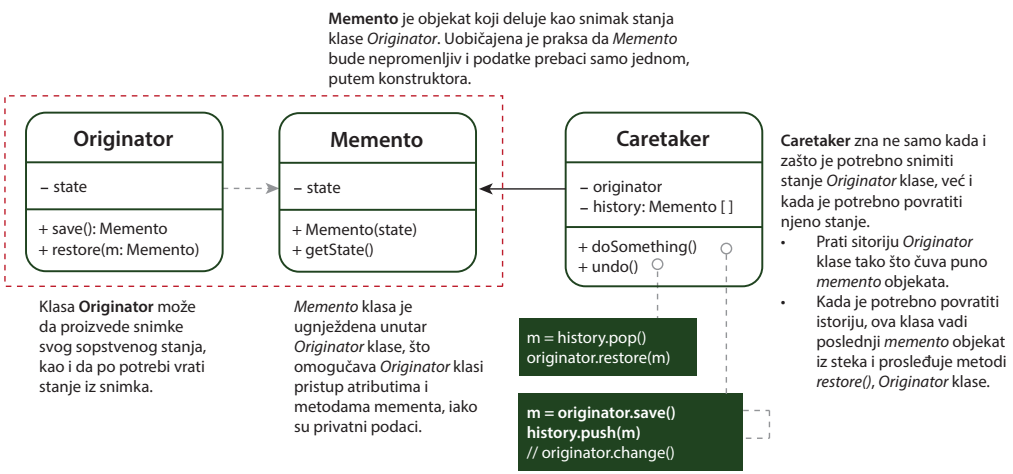
Slika 71. Odnos između osnovnih objekata *Memento* obrasca.



Postoje tri moguće vrste implementacija *Memento* obrasca i to:

- ♦ **Implementacija zasnovana na ugnježđenim klasama** (*Implementation based on nested classes*) – predstavlja klasičnu implementaciju podržanu od strane mnogi popularnih programskih jezika (npr. C++, C#, Java).

Slika 72. Osnovna struktura *Memento* obrasca zasnovana na ugnježđenim klasama (modifikovano sa Shvets, 2019).



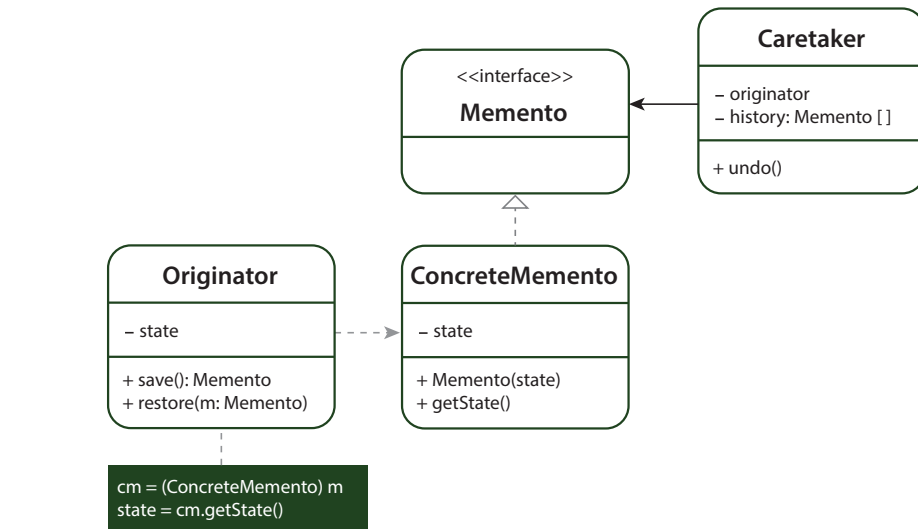
```

m = history.pop()
originator.restore(m)

m = originator.save()
history.push(m)
// originator.change()
    
```

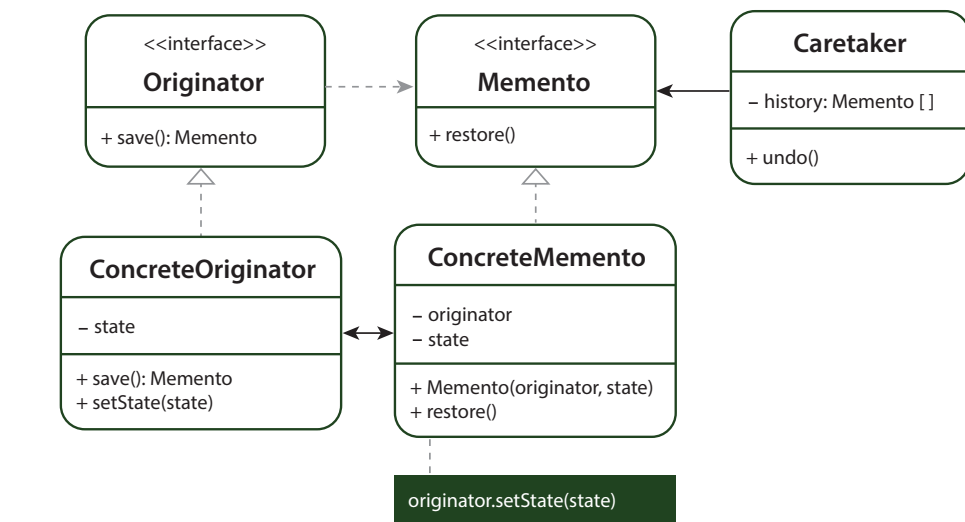
- ♦ **Implementacija zasnovana na posredničkom interfejsu** (*Implementation based on an intermediate interface*) - ova alternativna implementacija je zgodna za programske jezike koji ne podržavaju ugnježdene klase (npr. PHP).

Slika 73. Osnovna struktura *Memento* obrasca zasnovanog na posredničkom interfejsu (Shvets, 2019).



- ♦ **Implementacija sa još strožijom enkapsulacijom** (*Implementation with even stricter encapsulation*) - ova implementacija je korisna kada striktno ne želite da dozvolite drugim klasama da pristupe stanju Originator klase kroz *Memento*.

Slika 74. Osnovna struktura *Memento* obrasca zasnovanog na strožijoj enkapsulaciji (Shvets, 2019).



Osnovne karakteristike ovog obrasca su:

- ♦ Očuvanje enkapsulacije – *Memento* objekat ne otkriva detalje interne implementacije *Originator* objekta iako se njegovo stanje čuva van njega. Pristup *Memento* objektu je moguć samo od strane *Originator* objekta.
- ♦ Pojednostavljen dizajn *Originator* objekta – klijenti čuvaju stanje objekta kroz *Memento* objekat.
- ♦ Bolja skalabilnost - čuvanje stanja se prepušta klijentima omogućavajući fleksibilnost i bolju skalabilnost sistema.
- ♦ Problemi sa velikim količinama podataka - Korišćenje *Memento* objekata može biti problematično ukoliko je stanje *Originator*-a definisano velikom količinom podataka.
- ♦ Poteškoće u ograničavanju pristupa - U nekim programskim jezicima može biti teško sprečiti pristup internim strukturama *Memento* objekta od strane drugih objekata u sistemu.

Vežba 5-4. Primeniti *Memento* obrazac na scenario čuvanja sadržaja Artikla.

Klasa *ArticleOriginator* sadrži dva atributa: *id* i *content*. Potrebno je sačuvati prethodne vrednosti artikla u slučaju undo opcije.

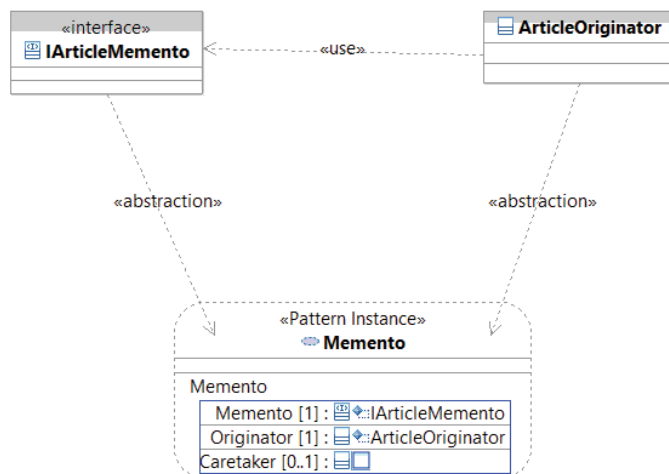
Koraci rešenja ovog zadatka su:

Korak 1. Prevući *Memento* obrazac na dijagram klasa, dopuniti sa odgovarajućim klasama i prikazati veze.

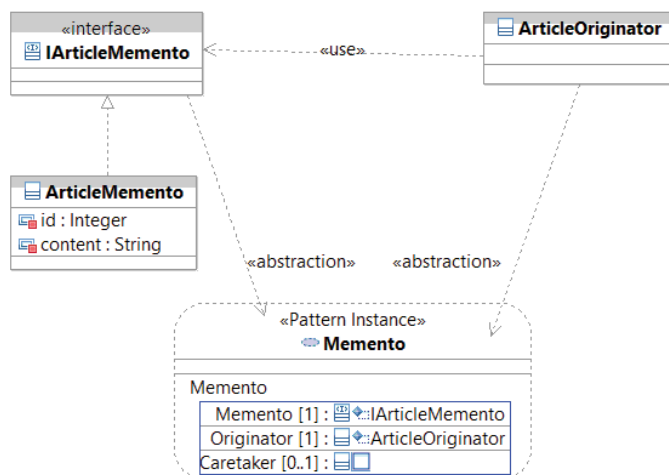
U ovom primeru *Main* klasa će imati ulogu *Caretaker* klase koja će kreirati i vraćati *memento* objekte. Shodno tome popunjene su samo dve vrednosti pod *Memento* obrascem i to:

- ♦ Pod *Memento* tipa interfejs, unet je interfejs *IArticleMemento*
- ♦ Pod *Originator*, uneto je *ArticleOriginator*.

Na osnovu unetih vrednosti izvršeno je povezivanje i prikazivanje svih relacija, kao što je prikazano na sledećoj slici.



Na dijagramu je dodata klasa *ArticleMemento* koja implementira interfejs *IArticleMemento*. U okviru te klase dodata su dva polja: *id* i *content*, kao što je prikazano na sledećoj slici.



Korak 2. Izvršiti, dopuniti kod i prikazati rezultate.

Prilikom generisanja koda, metode koje su kreirane po defaultu u *ArticleOriginator* klasi su: *createMemento()* (kreira i čuva stanje objekta) i *Restore(memento)*. Klasa *ArticleMemento* se ponaša kao zaključana kutija u koju može samo da upisuje i čita *Originator* klasa. *Memento* mora biti nepromenljiv/postojan objekat tako da niko ne može da promeni njegovo stanje, jednom kreirano. *Caretaker* čuva i vraća prethodno stanje originator objekta. U ovom primeru *Main* klasa će imati ulogu *Caretaker* klase koja će kreirati i vraćati prethodno stanje *memento* objekta.

Generisani, izmenjen i dopunjen kod:

```
public interface IArticleMemento {
}

public class ArticleMemento implements IArticleMemento {
    private long id;
    private String content;
    //getters and setters
    public long getId() {
        return id;
    }
    public void setId(long id) {
        this.id = id;
    }
    public String getContent() {
        return content;
    }
    public void setContent(String content) {
        this.content = content;
    }
    //constructor
    public ArticleMemento(long id, String content) {
        super();
        this.id = id;
        this.content = content;
    }
}

public class ArticleOriginator {
    private long id;
    private String content;
    //getters and setters
    public long getId() {
        return id;
    }
    public void setId(long id) {
        this.id = id;
    }
}
```

```

    public String getContent() {
        return content;
    }
    public void setContent(String content) {
        this.content = content;
    }
    //constructor
    public ArticleOriginator(long id, String content) {
        super();
        this.id = id;
        this.content = content;
    }

    public ArticleMemento createMemento() {
        ArticleMemento m= new ArticleMemento(id, content);
        return m;
    }
    public void restore(ArticleMemento m) {
        this.id=m.getId();
        this.content=m.getContent();
    }
    public String toString() {
        return "Article [id = "+id+", content = "+content+ "];"
    }
}

public class ClientMain {
    public static void main(String[] args) {
        ArticleOriginator a=new ArticleOriginator(1,"My Article");
        a.setContent("ABC"); //original content
        System.out.println(a);

        ArticleMemento m = a.createMemento(); //created immutable memento
        a.setContent("123"); //changed content;
        System.out.println(a);
        a.restore(m);
        System.out.println(a);//original content } }
    }
}

```

Izlaz:

```

Article [id = 1, content = ABC]
Article [id = 1, content =123]
Article [id = 1, content = ABC]

```

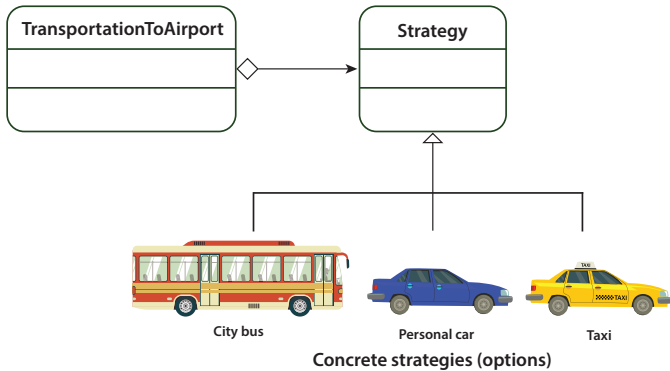
Zadatak 5-5. Primeniti *Memento* obrazac na sledeći scenario:

Kreirati aplikaciju sa undo funkcionalnošću, tako da čuva sva stanja objekta Zaposleni, kako bi moglo da se povрати poslednje stanje tog objekta. Podaci koji se čuvaju o zaposlenom su *id:int* i *salary:float*. Primeniti implementaciju sa strožijom enkapsulacijom.

5.5. STRATEGY

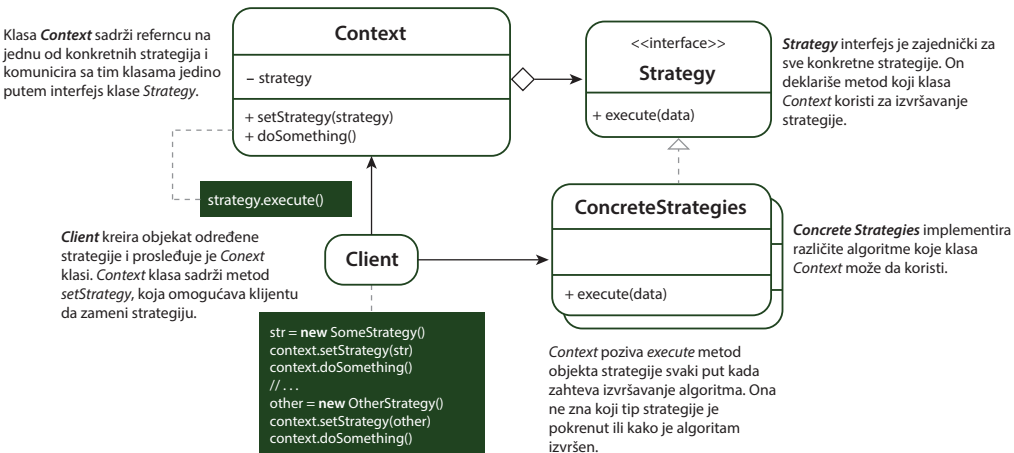
Strategy je tip obrasca ponašanja koji enkapsulira familiju algoritama i selektuje jedan od njih tokom izvršavanja programa.

Slika 75. Poređenje *Strategy* obrasca sa primerom iz realnog života



Strategy obrazac se koristi kada imamo više algoritama za određeni zadatak i klijent bira konkretni algoritam koji će se koristiti u trenutku izvršavanja programa. Primer *Strategy* obrasca u realnom životu bi bio, kada zahtevamo od developera da kreiraju aplikaciju, ali nam nije važno koji jezik će koristiti. Jedan će koristiti Javu, a drugi C#. Prepuštamo detalje oko prisajanja koda developerima i svako od njih je primenio svoju strategiju. U bilo kojoj fazi, developer može da promeni svoju strategiju, odnosno da piše drugim jezikom.

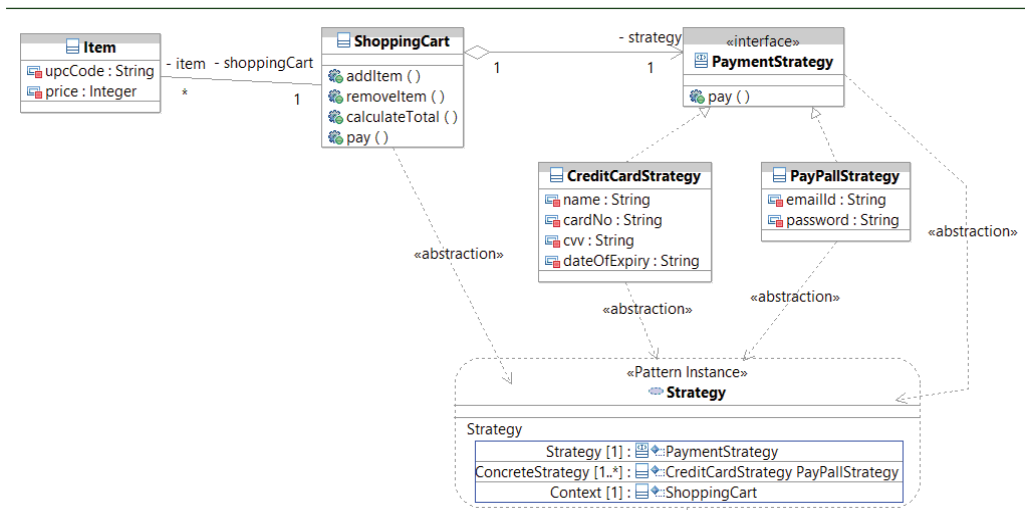
Slika 76. Osnovna struktura *Strategy* obrasca (modifikovano sa Shvets, 2019).



Klase i/ili objekti koje učestvuju u ovom obrascu su:

- ♦ *Strategy*: deklariše zajednički interfejs za sve podržane algoritme. Objekti klase kontekst koriste ovaj interfejs da pozovu algoritme definisane *ConcreteStrategy* familijom klasa.
- ♦ *ConcreteStrategy*: implementira algoritam koristeći interfejs klase *Strategy*.
- ♦ *Context*: konfiguriše se pomoću *ConcreteStrategy* objekata, čuva referencu na objekat klase *Strategy* i može da definiše interfejs koji objektu klase *Strategy* dozvoljava pristup podacima.

Vežba 5-5. Implementirati transakciju Plaćanja koja koristi dve strategije: plaćanje putem kreditne kartice ili putem *PayPal* sistema.



```
public interface PaymentStrategy {
    public void pay(Integer amount);
}
```

```
public class CreditCardStrategy implements PaymentStrategy {
    private String name;
    private String cardNo;
    private String cvv;
    private String dateOfExpiry;
    public CreditCardStrategy(String cardNo, String cvv, String
    dateOfExpiry,String name) {
        super();
        this.cardNo = cardNo;
        this.cvv = cvv;
        this.dateOfExpiry = dateOfExpiry;
        this.name = name;
    }
}
```

```

        public void pay(Integer amount) {
            System.out.println(amount+" paid with credit/debit card");
        }
    }

    public class paypalstrategy implements paymentstrategy {
        private String emailid;
        private String password;
        // Generate getters and setters
        public paypalstrategy(String emailid, String password) {
            super();
            this.emailid = emailid;
            this.password = password;
        }
        public void pay(Integer amount) {
            System.out.println(amount+" paid using paypal.");
        }
    }

    import java.util.ArrayList;
    import java.util.List;
    public class ShoppingCart {
        private PaymentStrategy strategy;
        private List<Item> items;
        // Generate here getters and setters
        //Generate constructors:
        public ShoppingCart() {
            this.items=new ArrayList<Item>();
        }
        public ShoppingCart(List<Item> items, PaymentStrategy strategy) {
            super();
            this.items = items;
            this.strategy = strategy;
        }
        public void addItem(Item item) {
            this.items.add(item);
        }
        public void removeItem(Item item) {
            this.items.remove(item);
        }

        public int calculateTotal() {
            int sum=0;
            for (Item item:items){
                sum+=item.getPrice();
            }
            return sum;
        }
        public void pay(PaymentStrategy paymentMethod){
            int amount=calculateTotal();
            paymentMethod.pay(amount);
        }
    }

```

```
public class Item {
    private String upcCode;
    private Integer price;
    private ShoppingCart shoppingCart;
    // Generate here getters and setters
    public Item(String upcCode, int price) {
        super();
        this.price = price;
        this.upcCode = upcCode;
    }
}

public class Client {

    public static void main(String[] args) {
        ShoppingCart cart=new ShoppingCart();
        Item item1=new Item("1234",10);
        Item item2=new Item("5678",40);
        cart.addItem(item1);
        cart.addItem(item2);
        cart.pay(new PayPallStrategy("myemail@example.com","mypwd"));
        cart.pay(new CreditCardStrategy("Simon Simeunovic", "12345
67890123456", "786", "12/15"));
    }
}
```

Izlaz:

50 paid using PayPal.
50 paid with credit/debit card.

Vežba 5-6. Primeniti odgovarajući obrazac na aplikaciju za upravljanje robotima.

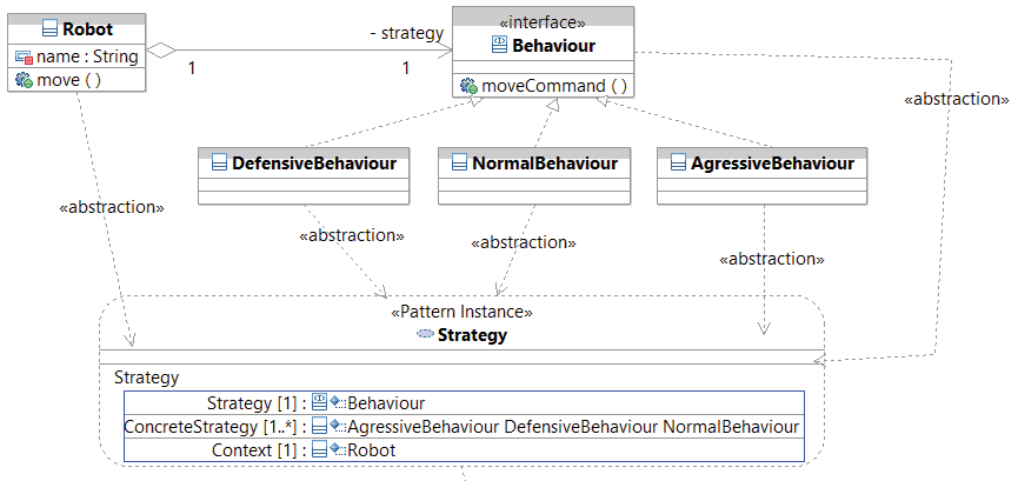
Kreirati aplikaciju koja će zadavati komande robotima, a koje upravljaju njihovim ponašanjima. Ponašanje robota može biti: *Agressive*, *Defensive* i *Normal*. Svaka od ovih konkretnih ponašanja će realizovati metodu `moveCommand()`, koja će prikazati opis ponašanja i vratiti odgovarajuću komandu:

- ♦ Za agresivno ponašanje:
 - Opis je: **Agressive Behaviour**: `if find another robot attack it!`
 - Komanda koja se izdaje je: **1**
- ♦ Za defenzivno ponašanje:
 - Opis je: **Defensive Behaviour**: `if find another robot run from it!`
 - Komanda koja se izdaje je: **-1**

- ♦ Za normalno ponašanje:
 - Opis je: Normal Behaviour: `if find another robot ignore it!`
 - Komanda koja se izdaje je: `0`

Klasa *Robot* sadrži naziv robota, a metoda *move* prikazuje sledeće: „**Robot's name - Based on the current position, the behaviour object decides the next move: //... send the command to mechanisms. The result +command+ returned by behaviour object is sent to the movement mechanisms for the robot 'robot name'.**

Rešenje:



```

public interface Behaviour {
    public int moveCommand();
}

public class Robot {
    String name;
    private Behaviour strategy;
    //Generate getters and setters
    public Behaviour getStrategy() {
        return strategy;
    }
    public void setStrategy(Behaviour strategy) {
        this.strategy = strategy;
    }
}
//Constructor
public Robot(String name) {
    super();
    this.name = name;
}
  
```

```
        public void move() {
            System.out.println(this.name + " - Based on the current
            position, the behaviour object decides the next move: ");
            int command= strategy.moveCommand();
            //... send the command to mechanisms
            System.out.println("\tThe result '"+command+"' returned by
            behaviour object is sent to the movement mechanisms for the
            robot '"+this.name+"'.");
        }
    }

    public class AgressiveBehaviour implements Behaviour {
        public int moveCommand() {
            System.out.println("\tAggressive behaviour: if find another
            robot attack it!");
            return 1;
        }
    }

    public class DefensiveBehaviour implements Behaviour {
        public int moveCommand() {
            System.out.println("\tDefensive Behaviour: if find another
            robot run from it!");
            return -1;
        }
    }

    public class NormalBehaviour implements Behaviour {
        public int moveCommand() {
            System.out.println("\tNormal Behaviour: if find another robot
            ignore it!");
            return 0;
        }
    }

    public class Client {
        public static void main(String[] args) {
            Robot r1=new Robot("Big Robot");
            Robot r2=new Robot ("George v.2.1.");
            Robot r3=new Robot("R2");
            r1.setStrategy(new AgressiveBehaviour());
            r2.setStrategy(new DefensiveBehaviour());
            r3.setStrategy(new NormalBehaviour());
            r1.move();
            r2.move();
            r3.move();
            System.out.println("\nNew behaviour: "+ "\n\n'George v.2.1.'
            becomes really mad because it's always attacked by other
            robots and R2 keeps it calm.\n");
            r1.setStrategy(new DefensiveBehaviour());
            r2.setStrategy(new AgressiveBehaviour());
            r1.move();
            r2.move();
            r3.move();
        }
    }
```

```
Properties Tasks Console Bookmarks Problems
<terminated> Client (23) [Java Application] C:\Program Files (x86)\IBM\SDP\jdk\bin\javaw.exe (22 Nov 2020 18:29:14)
Big Robot - Based on the current position, the behaviour object decides the next move:
    Aggressive behaviour: if find another robot attack it!
    The result '1' returned by behaviour object is sent to the movement mechanisms for the robot 'Big Robot'.
George v.2.1. - Based on the current position, the behaviour object decides the next move:
    Defensive Behaviour: if find another robot run from it!
    The result '-1' returned by behaviour object is sent to the movement mechanisms for the robot 'George v.2.1.'.
R2 - Based on the current position, the behaviour object decides the next move:
    Normal Behaviour: if find another robot ignore it!
    The result '0' returned by behaviour object is sent to the movement mechanisms for the robot 'R2'.

New behaviour:

'George v.2.1.' becomes really mad because it's always attacked by other robots and R2 keeps it calm.

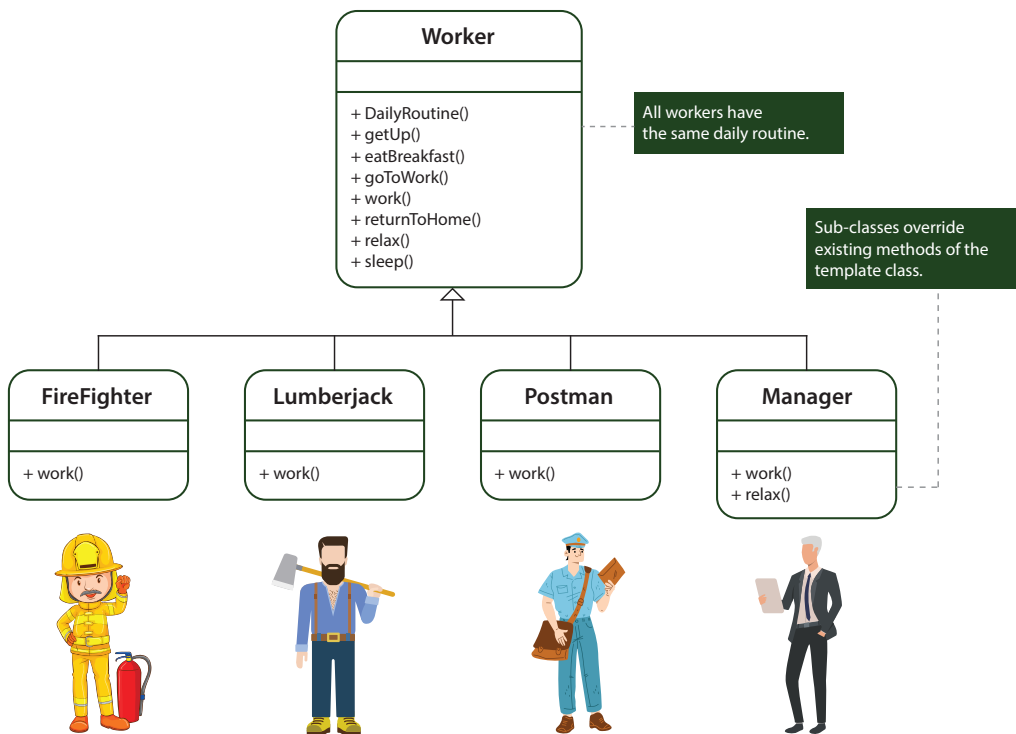
Big Robot - Based on the current position, the behaviour object decides the next move:
    Defensive Behaviour: if find another robot run from it!
    The result '-1' returned by behaviour object is sent to the movement mechanisms for the robot 'Big Robot'.
George v.2.1. - Based on the current position, the behaviour object decides the next move:
    Aggressive behaviour: if find another robot attack it!
    The result '1' returned by behaviour object is sent to the movement mechanisms for the robot 'George v.2.1.'.
R2 - Based on the current position, the behaviour object decides the next move:
    Normal Behaviour: if find another robot ignore it!
    The result '0' returned by behaviour object is sent to the movement mechanisms for the robot 'R2'.
```

Zadatak 5-6. Kreirati aplikaciju koja omogućava korisniku da se poveže sa svojim prijateljima putem SM (*Social Media*) platformi i to preko Instagram-a, Facebook-a i LinkedIn-a. Korisnik će proslediti ime prijatelja i željenu platformu, a aplikacija će ga transparentno povezati. U slučaju da korisnik bude želeo da proširi broj SM platformi, dodati ih bez degradiranja početnog dizajna aplikacije.

5.6. TEMPLATE METHOD

Template Method definiše redosled izvršavanja seta metoda nekog konkretnog algoritma. Ovaj obrazac dozvoljava potklasama da predefinišu određene korake u algoritmu, ali bez menjanja strukture tog algoritma. Na primer, na slici 77, generalizovana klasa je radnik (*Worker*) u okviru koje se nalaze zajedničke metode za potklase vartogasci (*FireFighter*), poštari (*Postman*) i dr. U generalizovanoj klasi se nalazi template metoda, koja je nazvana *DailyRoutine()*, u okviru koje je definisan redosled izvršavanja metoda: *getUp()*, *eatBreakfast()*, *goToWork()*, *work()*, *returnToHome()*, *relax()* i *sleep()*. Određene metode u potklasama mogu biti predefinisane, kao na primer, metoda *work()*, koja će pokazati kako radnici obavljaju svoj svakodnevni posao.

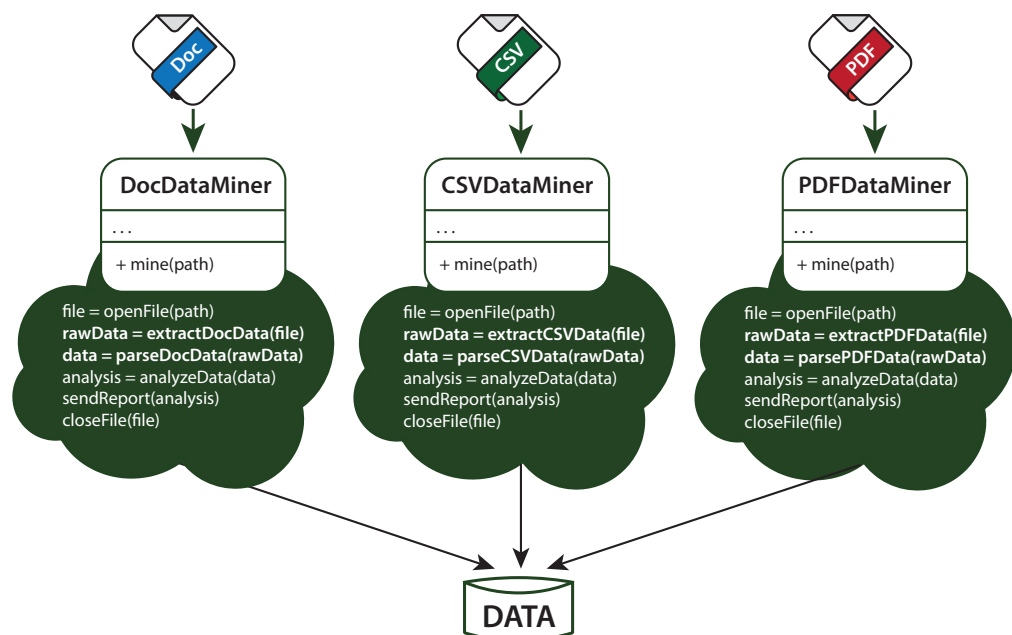
Slika 77. Primer *Template Method* obrasca.



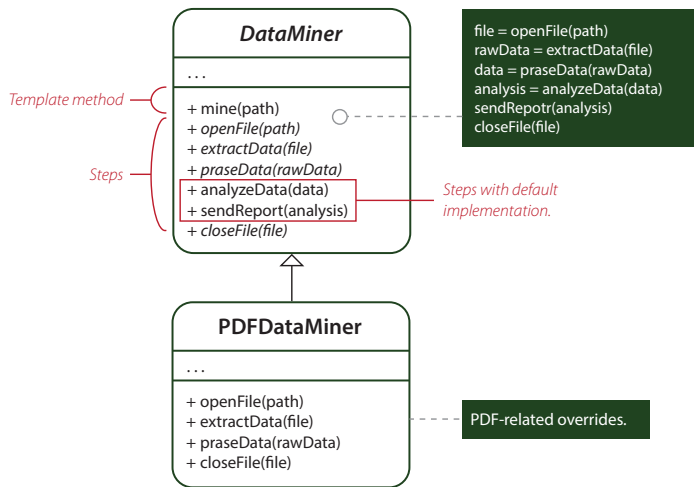
Zamislite da razvijate *data mining* aplikaciju koja analizira dokumenta kompanije. Korisnici učitavaju razne formate dokumenata (pdf, doc, csv) i pokušavaju da izvuku značajne podatke u jedinstven format. Bilo koji dokument da je u pitanju, aplikacija će izvršiti metode sledećim redosledom: *openFile(path)*, *extractData(file)*, *parseData(rawData)*, *analyzeData(data)*, *sendReport(analysis)* i *closeFile(file)*.

Prva verzija aplikacije je radila jedino sa .doc fajlovima, a kasnije je trebalo uvesti i ostale formate. U nekom trenutku ste uočili da sve tri klase imaju dosta sličnog koda, posebno za obradu i analizu podataka (Slika 78). U situacijama gde imamo veoma veliku sličnost pokušaćemo da pronađemo rešenje kako bi se smanjilo ponavljanje i eventualna redundansa.

Slika 78. Problemi data mining aplikacije (Shvets, 2019).

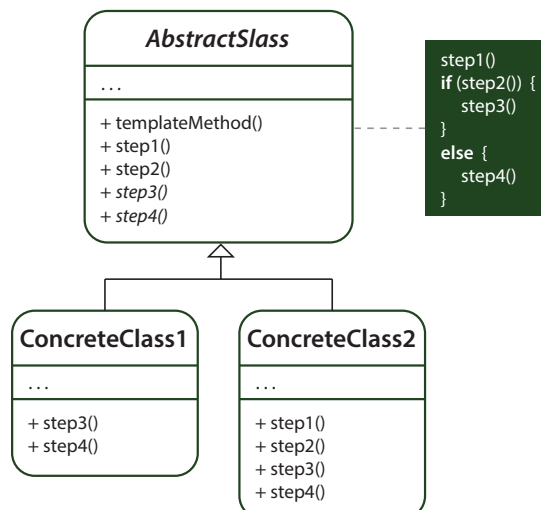


Sve zajedničke metode ćete staviti u apstraktnu klasu, dok će potklase obezbediti sopstvenu implementaciju ovih metoda. Apstraktna klasa deklarise apstraktne metode kao korake algoritma, kao i *template* metod koji poziva ove metode prema određenom redosledu.

Slika 79. Rešenje primenom *template method* obrasca (Shvets, 2019).

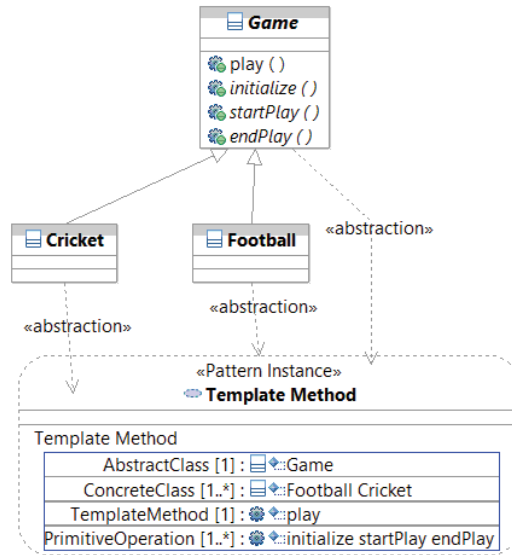
Osnovna struktura *Template method* obrasca sadrži (Slika 80):

- ♦ **Apstraktnu klasu** (*AbstractClass*) deklarise metode kao korake algoritma i aktuelni template metod koji poziva ove metode prema određenom redosledu. Metode mogu biti deklarisanе kao apstraktne ili mogu imati i neku difoltnu implementaciju.
- ♦ **Konkretnе klase** (*ConcreteClasses*) mogu da predefinišu sve korake, ali **ne može i sam template method**.

Slika 80. Osnovna struktura *Template method* obrasca (Shvets, 2019).

Vežba 5-7. Primeniti *Template method* obrazac na aplikaciju za igranje igrica.

Kreirati aplikaciju za izgranje igrica. Korisnik može da odabere igranje fudbala i kriketa. Zajedničke metode kod ovih igrica, a koje moraju da se odigravaju prema tačno utvrđenom redosledu, su: *Initialize()*, *startPlay()* i *stopPlay()*.



```

public abstract class Game {
    public final void play() {
        initialize();
        startPlay();
        endPlay();
    }
    public abstract void initialize();
    public abstract void startPlay();
    public abstract void endPlay();
}

public class Football extends Game {
    public void initialize() {
        System.out.println("Football Game Initialized! Start playing.");
    }
    public void startPlay() {
        System.out.println("Football Game Started. Enjoy the game!");
    }
    public void endPlay() {
        System.out.println("Football Game Finished!");
    }
}

```

```

public class Cricket extends Game {
    public void initialize() {
        System.out.println("Cricket Game Initialized! Start playing.");
    }
    public void startPlay() {
        System.out.println("Cricket Game Started. Enjoy the game!");
    }
    public void endPlay() {
        System.out.println("Cricket Game Finished!");
    }
}

public class Client {
    public static void main(String[] args) {
        Game game = new Cricket();
        game.play();
        System.out.println();
        game=new Football();
        game.play();
    }
}

```

Rezultat:

```

Cricket Game Initialized! Start playing.
Cricket Game Started. Enjoy the game!
Cricket Game Finished!

Football Game Initialized! Start playing.
Football Game Started. Enjoy the game!
Football Game Finished!

```

Zadatak 5-7. Primeniti *Template method* obrazac na različite načine poručivanja robe, a koji prate istu proceduru poručivanja.

Klijent može da poruči robu bilo u prodavnici (*StoreOrder*) ili onlajn (*NetOrder*). Proces poručivanja (*OrderProcess*), bez obzira da li je onlajn ili u prodavnici, prolazi istim redosledom kroz sledeće korake:

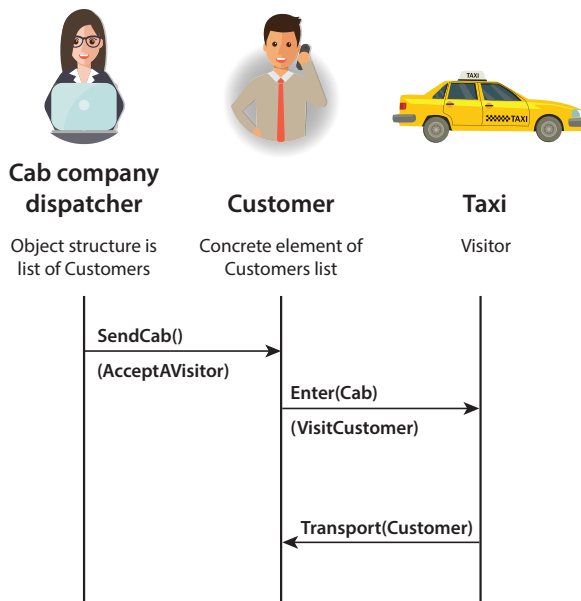
1. Korisnik selektuje proizvode koje želi da doda u svoju šoping korpu (metoda: *doSelect*)
2. Korisnik realizuje plaćanje (metoda: *doPayment*)
3. Ukoliko je proizvod poklon, on će biti posebno upakovan (metoda: *giftWrap*)
4. Nakon uspešno realizovane transakcije plaćanja, proizvodi su spremni za isporuku (metoda: *doDelivery*)

Prikazati odgovarajuće izlaze za oba načina poručivanja.

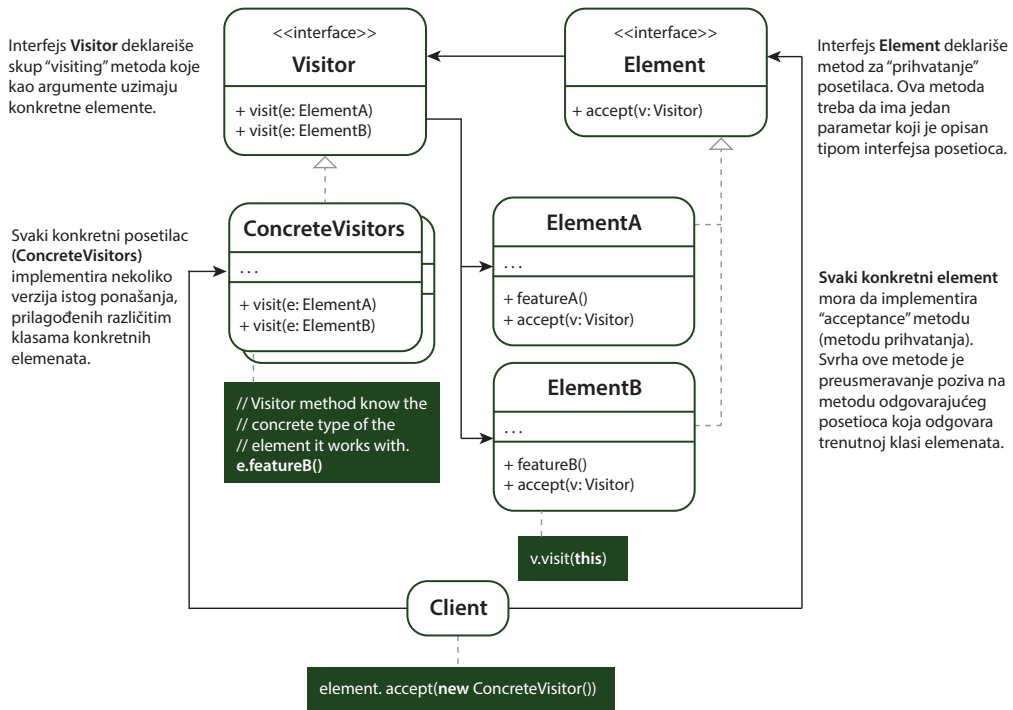
5.7. VISITOR

Visitor obrazac se koristi kada je potrebno izvršiti operacije nad grupom sličnih vrsta objekata. Primenom ovog obrasca te operacije se izmeštaju iz objekata i stavljaju u posebnu klasu koja se zove *Visitor*. Na primer, klijenti taksi usluga su slični objekti jer zahtevaju od taksi vozila da ih preveze sa jedne na drugu destinaciju (Slika 81). Te zajedničke metode objekta klijenti, se izvlače iz klijenta i stavljaju u posebnu tzv. *Visitor* klasu i u ovom primeru to je taksi vozilo. Taksi preuzima ulogu posetioca (*Visitor*) i prevozi klijenta do željene destinacije. Međutim da bi se metode *Visitor* klase realizovale, klijent prethodno mora da prihvati uslugu (metoda *acceptVisitor*).

Slika 81. Primer *Visitor* obrasca na realnom primeru.



Visitor obrazac se koristi kada je neophodno da izvršimo operaciju nad grupom sličnih objekata. Uz pomoć *Visitor* obrasca operativnu logika se premešta sa objekata na tzv. *Visitor* klasu. Osnovna struktura je prikazana na slici 82.

Slika 82. Osnovna struktura *Visitor* obrasca (modifikovano sa Shvets, 2019).

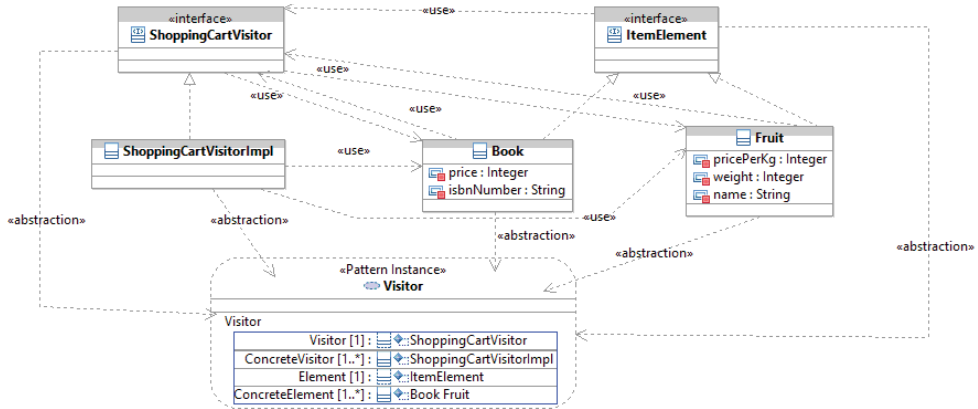
Vežba 5-8. Kreirati aplikaciju jedne online prodavnice koja svojim potrošačima želi da omogući pregled svojih poručenih proizvoda i ukupnu cenu poručene robe.

Kreirati dva konkretna elementa:

- ♦ Knjiga - koja ima attribute ISBN i cena;
- ♦ Voće - koje ima attribute naziv, težina i cena po kg.

Kreirati jednu implementaciju *Visitor* interfejsa pod nazivom *ShoppingCartVisitorImpl* u okviru koje će se:

- ♦ proveravati cena knjige i ukoliko je cena veća od 50e, primeniti popust od 5e.
- ♦ za kupljeno voće će se izračunati i prikazati vrednost (*težina * cena po kg*) i naziv voća.



```
public interface ItemElement {
    public int accept(ShoppingCartVisitor shoppingCartVisitor);
}
```

```
public class Book implements ItemElement {
    private int price;
    private String isbnNumber;
    //create constructors
    // create getters and setters
    //@@Override
    public int accept(ShoppingCartVisitor visitor) {
        return visitor.visit(this);
    }
}
```

```
public class Fruit implements ItemElement {
    private int pricePerKg;
    private int weight;
    private String name;
    // create constructors
    // create getters and setters
    //@@Override
    public int accept(ShoppingCartVisitor visitor) {
        return visitor.visit(this);
    }
}
```

```
public interface ShoppingCartVisitor {
    public int visit(Book book);
    public int visit(Fruit fruit);
}
```

```
public class ShoppingCartVisitorImpl implements ShoppingCartVisitor {
    public int visit(Book book) {
        int cost=0;
        //apply 5e discount if book price is greater than 50
        if (book.getPrice()>50){
            cost=book.getPrice()-5;
        }
        else
            cost=book.getPrice();
        System.out.println("Book ISBN:: "+book.getIsbnNumber()+"
```

```

        cost = "+cost);
        return cost;
    }
    public int visit(Fruit fruit) {
        int cost=fruit.getPricePerKg()*fruit.getWeight();
        System.out.println(fruit.getName()+" cost = "+cost);
        return cost;
    }
}

public class Client {
    public static void main(String[] args) {
        ItemElement[] items = new ItemElement[] {
            new Book("1234",20),
            new Book("5678",100),
            new Fruit("Banana",2,10),
            new Fruit("Apple",5,5)
        };
        int total=calculatePrice(items);
        System.out.println("Total Cost = "+total);
    }
    private static int calculatePrice(ItemElement[] items) {
        ShoppingCartVisitor visitor = new ShoppingCartVisitorImpl();
        int sum=0;
        for(ItemElement item:items){
            sum=sum+item.accept(visitor);
        }
        return sum;
    }
}

```

Rezultat:

```

Book ISBN :: 1234 cost = 20
Book ISBN :: 5678 cost = 95
Banana cost = 20
Apple cost = 25
Total Cost = 160

```

Zadatak 5-8. Primeniti obrazac na aplikaciju za servisiranje vozila. Svako vozilo (*Car*, *Van*, *Motorbike*) je posećeno od strane inspektora (*VehicleInspector*) koji vrši inspekciju (*VehicleInspection*) i izračunava servisnu naknadu za svako vozilo i ukupne troškove servisiranja.

Klasa *Car* ima attribute *color* i *manufacturingDate*. Klasa *Van* ima attribute *storageCapacity* i *numberOfDoors*, a *Motorbike* ima attribute *engineCapacity* i *brand*.

Kada inspektor poseti automobil (*Car*) proverava da li je boja crna i ukoliko jeste naknada za usluge je 100, u suprotnom je 50. Ispisati naknadu za svaki posećeni automobil.

Kada inspektor poseti kombi (*Van*) i ukoliko je broj vrata veći od 4, naknada za usluge je 500, a u suprotnom je 100. Prikazati naknadu za kombi.

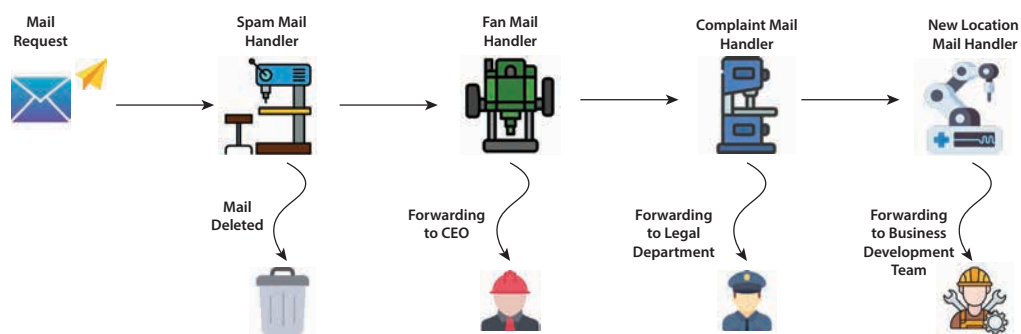
Kada inspektor poseti motor (*Motorbike*) i ukoliko je kapacitet motora veći ili jednako 200, naknada je 200, a u suprotnom je 50. Prikazati naknadu za motor.

Instancirati vozila sa željenim atributima i prikazati ukupan iznos naknada za posećena vozila.

5.8. CHAIN OF RESPONSIBILITY

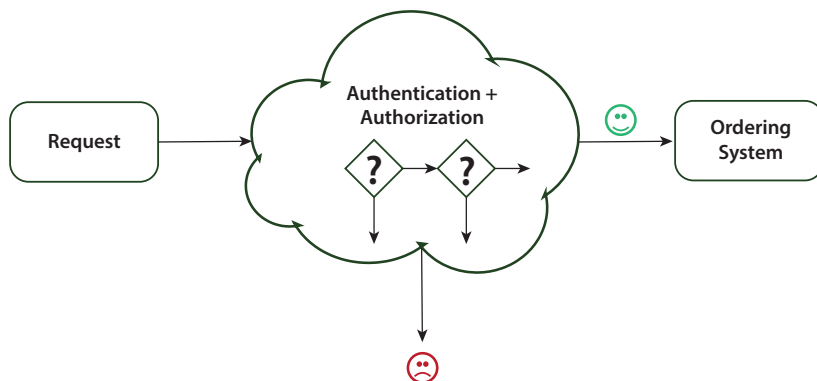
Lanac odgovornosti (*Chain of responsibility*) je obrazac koji omogućava prosleđivanje zahteva duž lanca rukovaoca. Nakon prijema zahteva, svaki rukovaoc odlučuje da li će ga obraditi ili proslediti sledećem rukovaocu u lancu. Na primer, u procesu obrade i filtriranja emailova (Slika 83), potrebno je sve pristigle emailove klijenata proslediti direktno odgovornoj osobi, npr. emailove vezano za žalbe je potrebno proslediti pravnim službama, spam emailove je potrebno brisati itd.

Slika 83. Rukovanje pristiglim e-mailom kroz lanac rukovaoca.



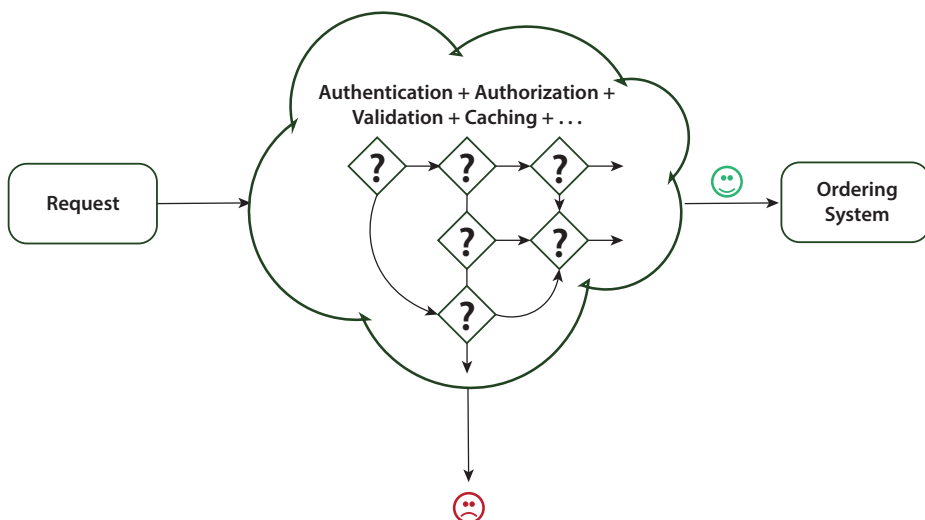
Zamislite da razvijate onlajn sistem za poručivanje. Prvi korak je da ograničite pristup tako da samo autentifikovani korisnici mogu kreirati porudžbine, a administratori da imaju potpuno pravo pristupa. Ukoliko kredencijali nisu korektni, nema razloga da se nastavi dalje (*Authentication i Autorization*) (Slika 84).

Slika 84. Primer rukovaoca autentikacije i autorizacije u sistemu za poručivanje.



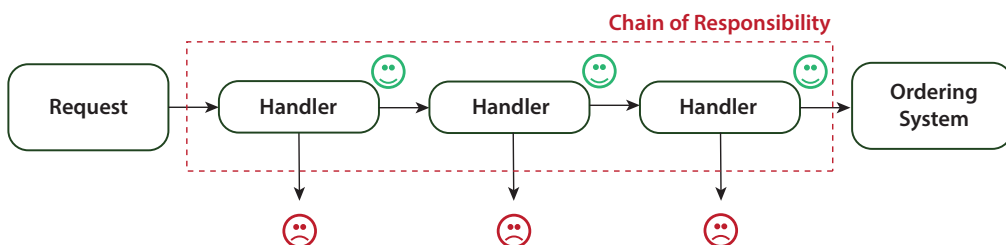
Nakon nekog vremena, dobili ste zahtev da ne prosleđujete sirove podatke direktno u sistem za poručivanje, stoga ste dodali još jedan korak validacije (*validation*) odnosno čišćenja podataka. Nakon toga, primetili ste da je sistem ranjiv na brute force napade (napadi sa uzastopnim pokušavanjem), te ste dodali još jedan korak koji filtrira ponavljajuće neuspešne zahteve za logovanjem koji dolaze sa iste IP adrese. Potom je predloženo da se sistem može ubrzati vraćanjem keširanih rezultata ponavljajućih zahteva koji sadrže iste podatke (*caching*) (Slika 85).

Slika 85. Primer dodavanja lanaca rukovaoca zahteva u sistemu za poručivanje.



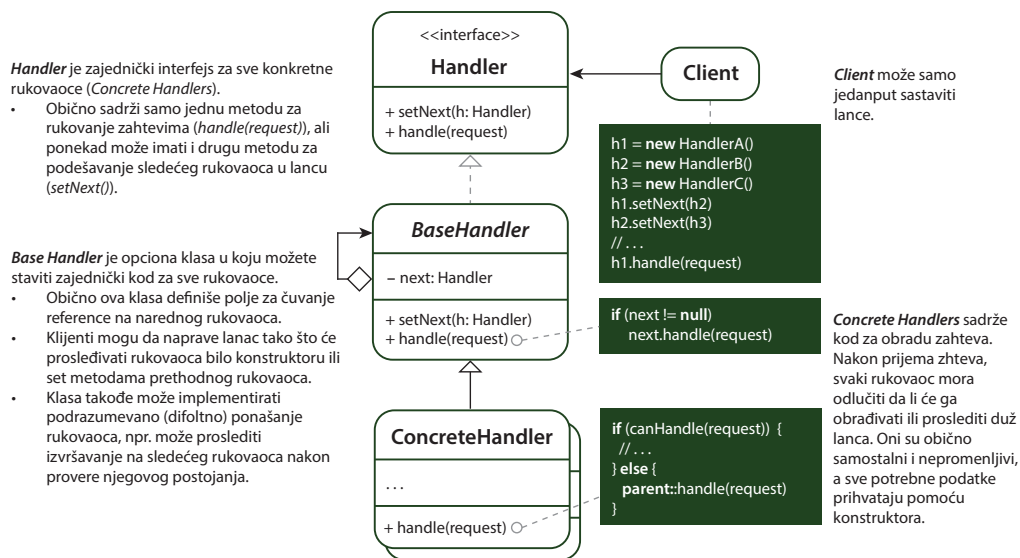
Rešenje ovog problema leži u primeni obrasca *Chain of responsibility* koji pretvara određena ponašanja sistema u samostalne rukovaoce (*Handlers*) koji su povezani u lanac. U ovom primeru svaka provera treba da se izdvoji u svoju klasu sa jednom metodom koja vrši konkretnu proveru. Zahtev, zajedno sa podacima, prosleđuje se ovoj metodi kao argument. Svaki rukovaoc u lancu sadrži polje za čuvanje reference na sledećeg rukovaoca u lancu. Zahtev putuje kroz lanac sve dok ga neki od rukovaoca ne obradi (Slika 86).

Slika 86. Primer kreiranja lanaca rukovaoca zahtevima aplikacije za poručivanje.



Osnovna struktura obrasca *Chain of Responsibility* je prikazana na slici 87.

Slika 87. Osnovna struktura *Chain of Responsibility* obrasca (modifikovano sa Shvets, 2019).

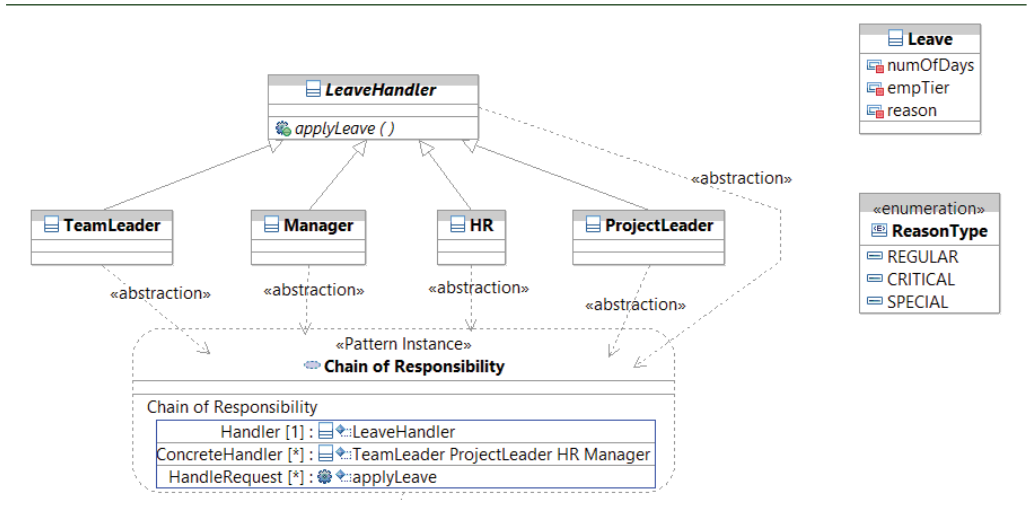


Vežba 5-9. Primeniti obrazac *Chain of Responsibility* na sistem za odobravanje odsustva zaposlenih.

Kreirati aplikaciju za odobravanje zahteva za odsustvo zaposlenih, tako što će se zahtev na osnovu broja dana odsustva, nivoa zaposlenog i razloga za odsustvo, proslediti odgovarajućoj osobi i to rukovodiocu tima (*TeamLeader*), vođi projekta (*ProjectLeader*), HR-u ili menadžer. Ukoliko zaposleni zatraži odsustvo u trajanju od 7 ili manje dana, vođa tima će rešiti taj zahtev. Međutim, pre nego što odobri zahtev vođa tima proverava nivo zaposlenog. Ako je nivo zaposlenog u prihvatljivom opsegu, tada će vođa tima odobriti zahtev za odsustvo.

Ako zaposleni zatraži odsustvo u trajanju od 14 ili manje dana, vođa tima ne može da odgovori na taj zahtev, jer ima ovlašćenje da odobri samo odsustvo do 7 dana. U tom slučaju zahtev će biti prosleđen vođi projekta, koji ima ovlašćenje da obradi taj zahtev. Zatim će započeti proces odobravanja koji je isti kao prethodni.

Ako zaposleni zahteva do 21 dan, ljudski resursi će se pobrinuti za to, a ako je više od 21 dana, menadžer će to rešiti. Kada obrađuju zahtev, proveriće da li je razlog redovan, kritičan ili poseban. Onda će na osnovu tih činjenica odobriti ili odbiti zahtev.



```

public class Leave {
    private int numberOfDays;
    private int empTier;
    private ReasonType reason;

    public Leave(int numberOfDays, int empTier, ReasonType reason) {
        this.numberOfDays = numberOfDays;
        this.empTier = empTier;
        this.reason = reason;
    }

    public int getNumberOfDays() {
        return numberOfDays;
    }

    public int getEmpTier() {
        return empTier;
    }

    public ReasonType getReason() {
        return reason;
    }
}

public enum ReasonType {
    REGULAR, CRITICAL, SPECIAL
}

public abstract class LeaveHandler {
    protected LeaveHandler supervisor;
    public void setSupervisor(LeaveHandler supervisor) {
        this.supervisor = supervisor;
    }
    public abstract String applyLeave(Leave leave);
}
  
```

```

public class TeamLeader extends LeaveHandler {
    @Override
    public String applyLeave(Leave leave) {
        //"reasonType" is not going to be consider under TeamLeader &
        ProjectLeader
        //Team Leader can approve up to 7 days, otherwise it will pass
        to the Project Leader
        if(leave.getNumberOfDays() <= 7){
            //Employee tier should be 4 or above to get approved
            if(leave.getEmpTier() <= 4){
                return "Your leave days has been APPROVED by TeamLeader";
            }else{
                return "You employee Tier level is too low for request "
                + leave.getNumberOfDays() + " days";
            }
        }else{
            return superVisor.applyLeave(leave);
        }
    }
}

```

```

public class ProjectLeader extends LeaveHandler {
    @Override
    public String applyLeave(Leave leave) {
        //"reasonType" is not going to be consider under TeamLeader &
        ProjectLeader
        //Project Leader can approve up to 14 days, otherwise it will
        pass to the HR
        if(leave.getNumberOfDays() <= 14){
            //Employee tier should be 3 or above to get approved
            if(leave.getEmpTier() <= 3){
                return "Your leave days has been APPROVED by Project Leader";
            }else{
                return "You employee Tier level is too low for request "
                + leave.getNumberOfDays() + " days";
            }
        }else{
            return superVisor.applyLeave(leave);
        }
    }
}

```

```

public class HR extends LeaveHandler {
    @Override
    public String applyLeave(Leave leave) {
        //HR can approve up to 21 days, otherwise it will pass to the Manager
        if(leave.getNumberOfDays() <= 21){
            //Employee tier should be 3 or above & reason type should
            not be "Regular" to get approved
            if(leave.getEmpTier() <= 3 && !leave.getReason().
            equals(ReasonType.REGULAR)){
                return "Your leave days has been APPROVED by HR";
            }
        }
    }
}

```

```
        }else{
            return "Your leave request has been DENIED by HR";
        }
    }else{
        return superVisor.applyLeave(leave);
    }
}
}

public class Manager extends LeaveHandler {
    @Override
    public String applyLeave(Leave leave) {
        //Only Manager has the authority to approve more than 21 days
        if (leave.getNumberOfDays() > 21) {
            //Employee tier should be 2 or above & reason type should be
            "Special" to get approved
            if (leave.getEmpTier() <= 2 && leave.getReason().
                equals(ReasonType.SPECIAL)) {
                return "Your leave days has been APPROVED by Manager";
            }
        }
        return "Your leave request has been DENIED by Manager";
    }
}
```

```
public class LeaveRequestApp {
    public static void main(String[] args) {
        TeamLeader teamLeader = new TeamLeader();
        ProjectLeader projectLeader = new ProjectLeader();
        HR hr = new HR();
        Manager manager = new Manager();

        teamLeader.setSuperVisor(projectLeader);
        projectLeader.setSuperVisor(hr);
        hr.setSuperVisor(manager);

        Leave leave1 = new Leave(5,4, ReasonType.REGULAR);
        System.out.println(teamLeader.applyLeave(leave1));

        Leave leave2 = new Leave(5,5, ReasonType.REGULAR);
        System.out.println(teamLeader.applyLeave(leave2));

        Leave leave3 = new Leave(12,3, ReasonType.REGULAR);
        System.out.println(teamLeader.applyLeave(leave3));

        Leave leave4 = new Leave(18,2, ReasonType.CRITICAL);
        System.out.println(teamLeader.applyLeave(leave4));

        Leave leave5 = new Leave(18,2, ReasonType.REGULAR);
        System.out.println(teamLeader.applyLeave(leave5));

        Leave leave6 = new Leave(30,2, ReasonType.SPECIAL);
        System.out.println(teamLeader.applyLeave(leave6));
    }
}
```

Izlaz:

Your leave days has been APPROVED by TeamLeader
Your employee Tier level is too low for request 5 days
Your leave days has been APPROVED by Project Leader
Your leave days has been APPROVED by HR
Your leave request has been DENIED by HR
Your leave days has been APPROVED by Manager

Zadatak 5-9. Primeniti obrazac *Chain of Responsibility* na aplikaciju za obradu i filtriranje pristiglih emailova klijenata.

Kreirati aplikaciju koja će sve pristigle email-ove klijenata proslediti direktno odgovornoj osobi. Npr. email-ove vezano za žalbe je potrebno proslediti pravnoj službi, spam email-ove je potrebno brisati itd. Kreirati četiri lančana rukovaoca i to *SpamHandler*, *FanHandler* i *ComplaintHandler*. Svaki lanac rukovaoca proverava predmet e-maila i u skladu s tim će ili obraditi zahtev ili ga proslediti sledećem rukovaocu (metoda: *forwardMail()*).

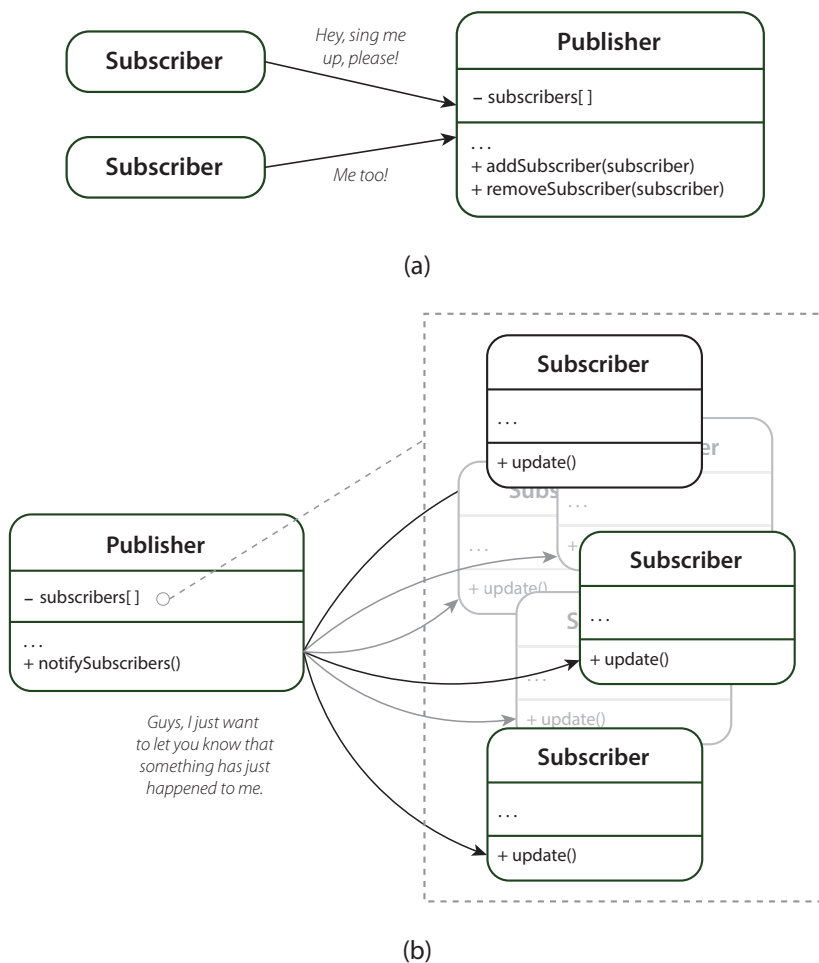
5.9. OBSERVER

Observer (posmatrač) je obrazac ponašanja koji omogućava da se definiše mehanizam automatskog obaveštavanja više zainteresovanih klasa o svim promenama koje se dešavaju sa klasom koju posmatraju.

Zamislite da imate dva tipa objekata: Klijent i Prodavnicu. Klijent je veoma zainteresovan za određeni brend proizvoda (npr. novi model *iPhone*), koji bi uskoro trebao da se nađe u prodavnicama. Jedno rešenje je da klijent posećuje često prodavnicu. Drugo rešenje je da prodavnica šalje tonu e-mailova (što se može smatrati *spamom*) kako bi obavestio klijente svaki put kada u prodavnicu stigne neki novi proizvod.

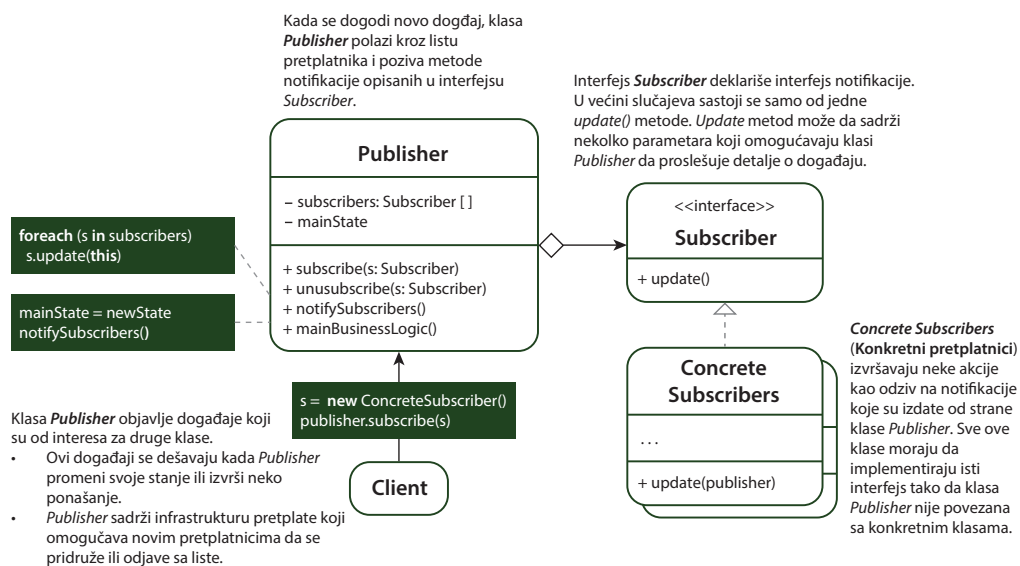
Observer obrazac predlaže da se uvede neki sistem pretplate na obaveštenja za posmatrače, gde oni u svakom trenutku mogu da se odjave (Slika 88).

Slika 88. Primer *Observer* obrasca na primeru *Publisher-Subscriber* (Shvets, 2019).



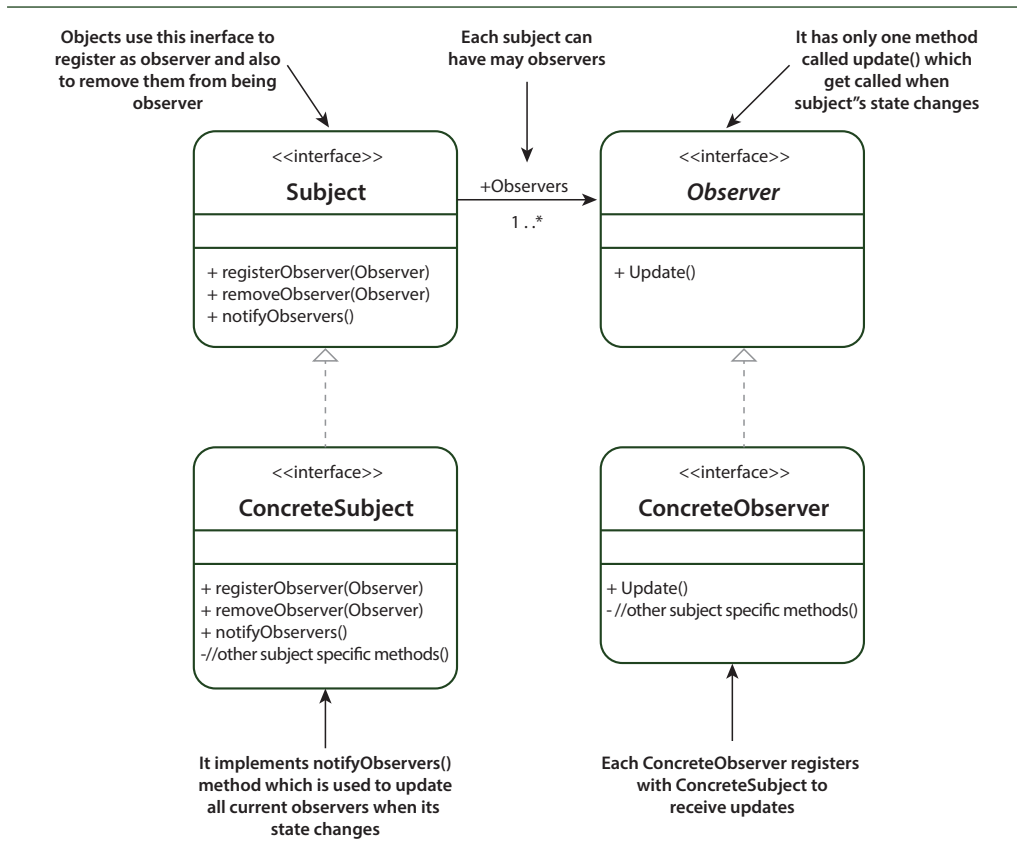
Osnovna struktura *Observer* obrasca, sa objašnjenima njegovih osnovnih elemenata je prikazana na slici 89.

Slika 89. Osnovna struktura *Observer* obrasca



Dakle, osnovni elementi *Observer* obrasca su:

- ♦ **Subject/Observable/Publisher** –registruje posmatrače (`registerObserver`) uklanja ih (`removeObserver`) i obaveštava (`notifyObservers`) (Slika 90).
- ♦ **Observer/Subscriber** – sadrži `update` metodu, koja inicijalizuje slanje obaveštenja posmatračima, odnosno kada se desi promena na **Subject** klasi, tada se posmatrači obaveštavaju o toj promeni (Slika 90).

Slika 90. Osnovni elementi *Observer* obrasca.

Interfejsi *Observable* (posmatrana klasa) i *Observer* (posmatrač) neće biti prikazivani na dijagramu klasa jer se importuju iz paketa `java.util`. Metode ovih interfejsa su:

Observer:

- ♦ **public void update(*Observable* obs, *Object* obj)** - Ova metoda se poziva kada se desi promena u stanju posmatranog objekta.

Observable:

- ♦ **public void addObserver(*Observer* obs)** - dodaje observer-a na internu listu posmatrača.
- ♦ **public void deleteObserver(*Observer* obs)** - briše observer objekat sa interne liste posmatrača.
- ♦ **public void deleteObservers()** - briše sve posmatrače sa interne liste posmatrača.
- ♦ **public int countObservers()** - vraća broj posmatrača na internoj listi posmatrača.

- ♦ **protected void setChanged()** - postavlja interni pokazatelj (*flag*) koji ukazuje da je posmatrani objekat promenio stanje.
- ♦ **protected void clearChanged()** - briše interni pokazatelj koji ukazuje da je posmatrani objekat promenio stanje.
- ♦ **public boolean hasChanged()** - vraća logičku vrednost *true* ako je posmatrani objekat promenio stanje.
- ♦ **public void notifyObservers()** - proverava interni pokazatelj (*flag*) da vidi da li je posmatrač promenio stanje i obaveštava sve posmatrače.
- ♦ **public void notifyObservers(Object obj)** - Proverava interni pokazatelj (*flag*) da vidi da li je posmatrani objekat promenio stanje i obaveštava sve posmatrače. Prosleđuje objekat **obj** (naveden u listi parametara) metodi **notify()** posmatrača.

Klasa *ObservableValue*:

```
import java.util.Observable;
public class ObservableValue extends Observable {
    private int n = 0;
    public ObservableValue(int n) {
        this.n = n;
    }
    public void setValue(int n) {
        this.n = n;
        setChanged();
        notifyObservers();
    }
    public int getValue() {
        return n;
    }
}
```

Interno stanje klase **ObservableValue** (koji nasleđuje *Observable*) je obuhvaćeno parametrom **n**. Ukoliko se vrednost ovog parametra promeni, posmatrani objekat poziva sopstvenu metodu **setChanged()** kako bi ukazao da se stanje objekta promenilo. Zatim se poziva sopstvenu metodu **notifyObservers()** da bi obavestila sve registrovane posmatrače.

Klasa *TextObserver*:

```
import java.util.Observer;
import java.util.Observable;
public class TextObserver implements Observer {
    private ObservableValue ov = null;
    public TextObserver (ObservableValue ov) {
        this.ov = ov;
    }
    public void update(Observable obs, Object obj) {
        if (obs == ov) {
            System.out.println(ov.getValue());
        }
    }
}
```

Kreirana je nova klasa objekata *TextObserver* kao posmatrač, koja posmatra promene stanja *ObservableValue* klase i želi da o tome bude obavešten. *TextObserver* klasa implementira *Observer* interfejs koji zahteva da *update()* metoda bude predefinisana u novoj klasi. Metoda *update()* se poziva svaki put kada posmatrani objekat promeni stanje i obavesti sve posmatrače pozivom metode *notifyObservers()*.

U glavnoj klasi se prikazuje kako se koristi metoda *addObserver()*:

Klasa *Main*:

```
public class Main {  
    public Main () {  
        ObservableValue ov = new ObservableValue(0);  
        TextObserver to = new TextObserver(ov);  
        ov.addObserver(to);  
    }  
    public static void main(String [] args) {  
        Main m = new Main();  
    }  
}
```

Vežba 5-10. Primeniti *Observer* obrazac na aplikaciju za slanje obaveštenja | klijentima o rasprodajama koje su u toku.

Prodavnica (*Store*) želi da svoje pasivne kupce (*PassiveCustomer*) i šopingholičare (*ShopaholicCustomer*) obaveštava o rasprodajama koje su u toku. Aplikacija treba da šalje poruku svim pretplaćenim kupcima kad god bi rasprodaja bila aktivirana. Primeniti odgovarajući obrazac i izgenerisati Java kôd.

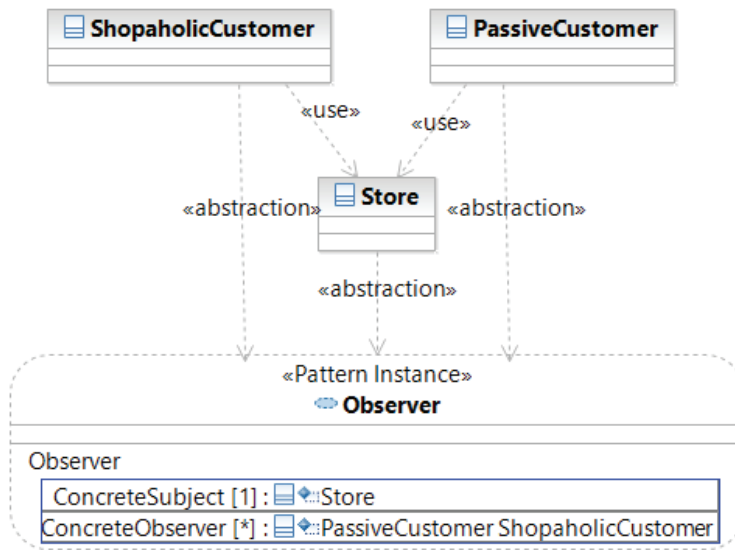
Prikažaćemo dva načina rešavanja ovog zadatka:

1. Prvi način je kada se koriste predefinisani interfejsi *Observable* (posmatrana klasa) i *Observer* (posmatrači) koji su sastavni deo paketa *java.util*. Ti interfejsi se neće prikazati na dijagramu klasa, već će biti dovoljno da ih klase implementiraju.
2. Drugi način je kada se ne koristi paket *java.util*, već se ručno kreiraju interfejsi, odnosno logika rada *Observer* obrasca.

Koraci rada na ovoj vežbi koristeći prvi način je:

Korak 1. Prevući obrazac, popuniti sa traženim klasama i prikazati veze.

Prilikom prikazivanja svih veza, interfejsi *Observable* (posmatrana klasa) i *Observer* (posmatrač) neće biti prikazani na dijagramu klasa, jer se importuju iz paketa java.util, kao što je i prikazano na sledećem dijagramu klasa.



Korak 2. Izgenerisati Java kôd.

Izgenerisana klasa *Store* nasleđuje *Observable* interfejs:

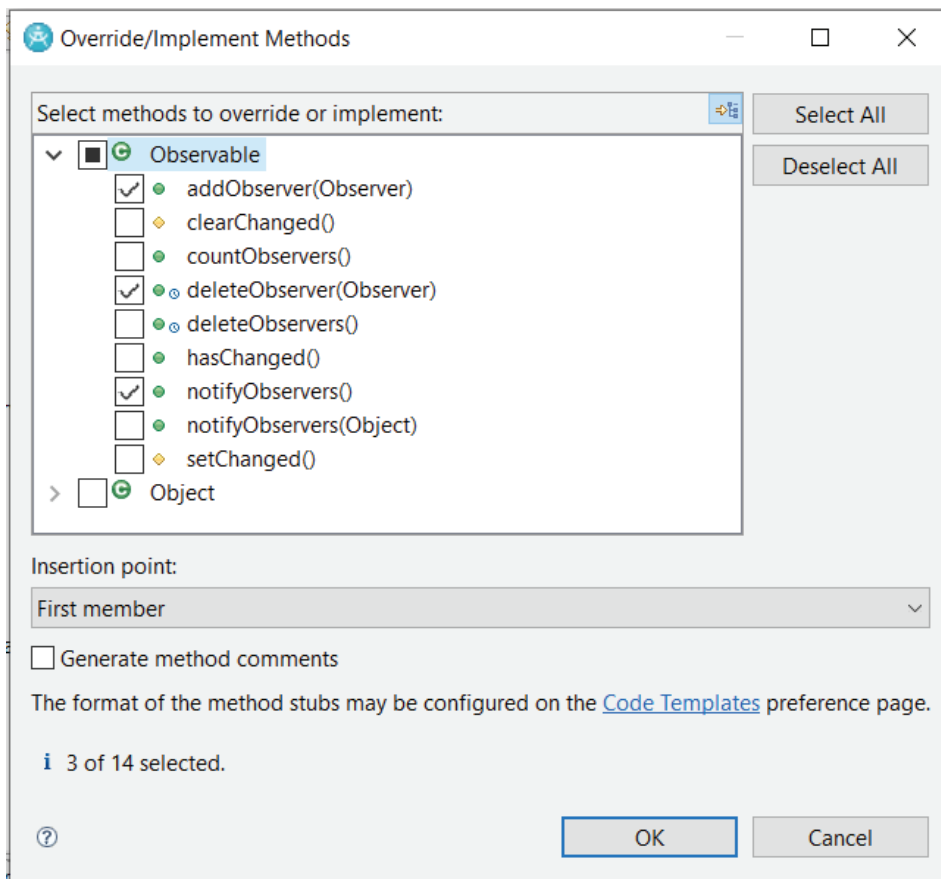
```

import java.util.Observable;

public class Store extends Observable {
}

```

Ukoliko želite da predefinišete neku od postojećih metoda *Observable* interfejsa, desnim klikom odaberite *Source*, a zatim *Override/Implement Methods*, prilikom čega će vam se otvoriti prozor, kao na sledećoj slici, odakle možete selektovati metode koje želite da predefinišete.



Izgenerisana klasa *PassiveCustomer* implementira *Observer* interfejs, prilikom čega prikazuje *update* metodu:

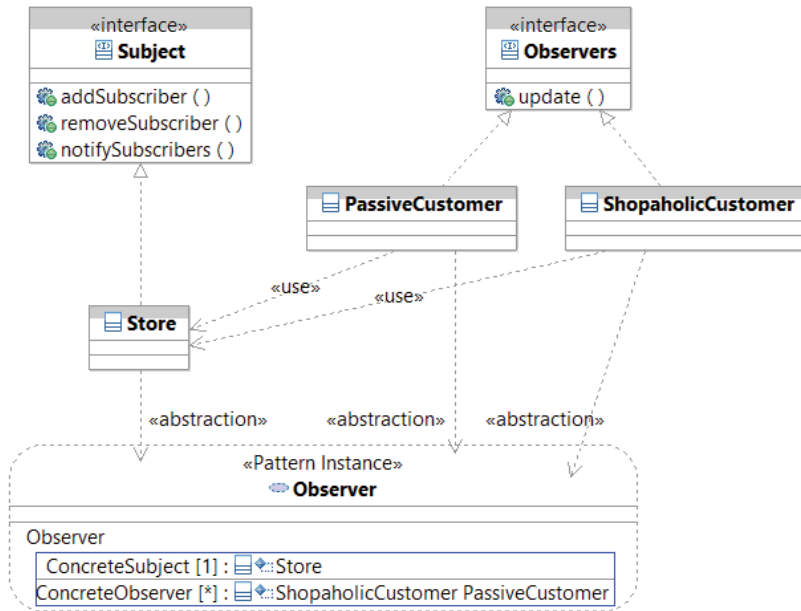
```
import java.util.Observable;
import java.util.Observer;
public class PassiveCustomer implements Observer {
    public void update(Observable o, Object arg) {
    }
}
```

Na identičan način će biti izgenerisana i klasa *ShopaholicCustomer*.

Koraci rada na ovoj vežbi koristeći drugi način je:

Korak 1. Dopuniti dijagram klasa sa interfejsima *Subject* i *Observers*.

Da bi se izgenerisali interfejsi u programskom kodu, na dijagramu klasa će se dodati interfejsi i njihove metode:



```

public interface Subject {
    public void addSubscriber(Observers observer);
    public void removeSubscriber(Observers observer);
    public void notifySubscribers();
}

public interface Observers {
    public void update(String message);
}

public class ShopaholicCustomer implements Observers {
    public void update(String message) {
        processMessage(message);
        //since of a complex psychologic response to a sale by a
        //shopaholic, function processMessage() is added.
    }
}
  
```

```
        private void processMessage(String message) {
            System.out.println("Shopaholic customer is interested in
                                buying the product on sale!");
        }
    }

    public class PassiveCustomer implements Observers {
        public void update(String message) {
            //Passive customer does not react to the message too much
            System.out.println("Passive customer made note of the sale.");
        }
    }

    import java.util.ArrayList;
    import java.util.List;

    public class Store implements Subject {
        private List<Observers> customers = new ArrayList<Observers>();

        public void addSubscriber(Observers customer) {
            customers.add(customer);
        }

        public void removeSubscriber(Observers customer) {
            customers.remove(customer);
        }

        @Override
        public void notifySubscribers() {
            System.out.println("A new item is on sale! Act fast before
                                it sells out!");
            for (Observers customer: customers){
                customer.update("Sale!");
            }
        }
    }

    public class Main {
        public static void main(String[] args) {
            Store fashionStore = new Store();
            PassiveCustomer customer1 = new PassiveCustomer();
            ShopaholicCustomer customer2 = new ShopaholicCustomer();
            ShopaholicCustomer customer3 = new ShopaholicCustomer();

            // Adding two customers to the newsletter
            fashionStore.addSubscriber(customer1);
            fashionStore.addSubscriber(customer2);

            // Notifying customers (observers)
            fashionStore.notifySubscribers();

            // A customer1 has decided not to continue following the newsletter
            fashionStore.removeSubscriber(customer1);

            // customer2 told customer3 that a sale is going on
            fashionStore.addSubscriber(customer3);

            // Notifying the updated list of customers
            fashionStore.notifySubscribers();
        }
    }
```

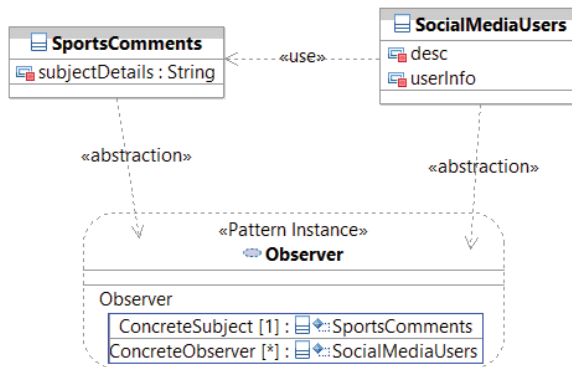
Izlaz:

A new item is on sale! Act fast before it sells out!
 Passive customer made note of the sale.
 Shopaholic customer is interested in buying the product on sale!
 A new item is on sale! Act fast before it sells out!
 Shopaholic customer is interested in buying the product on sale!
 Shopaholic customer is interested in buying the product on sale!

Vežba 5-11. Primeniti *Observer* obrazac za registrovanje korisnika na sportske vesti.

Kreirati aplikaciju za automatsko slanje sportskih komentara (*SportsComments*) pretplaćenim korisnicima sa društvenih medija (*SocialMediaUsers*). Klasa *SportsComments* će imati samo jedan atribut *subjectDetails* tipa *String*, dok će klasa *SocialMediaUsers* imati dva atributa i to informacije o korisnicima (*userInfo*) i opis (*desc*). Metode *SocialMediaUsers* klase su *subscribe()*, *unsubscribe()* i *update()*. Koristiti prvi način rada na ovoj vežbi.

Rešenje ove vežbe:



```

import java.util.Observable;

public class SportsComments extends Observable {
    private String subjectDetails;
    //getters and setters
    public void setSubjectDetails(String subjectDetails) {
        this.subjectDetails = subjectDetails;
        setChanged();
        notifyObservers(subjectDetails);
    }
    public String getSubjectDetails() {
        return subjectDetails;
    }
    //constructors
  
```

```
    public SportsComments() {
        super();
    }
    public SportsComments(String subjectDetails) {
        super();
        this.subjectDetails = subjectDetails;
    }
}
import java.util.*;
public class SocialMediaUsers implements Observer {
    private String desc;
    private String userInfo;
    private final Observable observable;
    public void subscribe() {
        System.out.println("Subscribing "+userInfo+" to "+
            ((SportsComments)(observable)).getSubjectDetails()+" ...");
        this.observable.addObserver(this);
        System.out.println("Subscribed successfully.");
    }
    public void unsubscribe() {
        System.out.println("Unsubscribing "+userInfo+" to "+
            ((SportsComments)(observable)).getSubjectDetails()+" ...");
        this.observable.deleteObserver(this);
        System.out.println("Unsubscribed successfully.");
    }
    public void update(Observable o, Object arg) {
        desc=(String)arg;
        System.out.println("[ "+userInfo+"]: "+desc);
    }
}
//constructor
public SocialMediaUsers(Observable observable,String userInfo){
    this.observable = observable;
    this.userInfo = userInfo;
}
}
public class Main {
    public static void main(String[] args) {
        SportsComments obj = new SportsComments("Derbi [31.08.2022]");
        SocialMediaUsers observer1 = new SocialMediaUsers(obj, "Petar Petrovic [Niksic]");
        SocialMediaUsers observer2 = new SocialMediaUsers(obj, "Nikola Nikolic [Beograd]");
        observer1.subscribe();
        observer2.subscribe();
        obj.setSubjectDetails("Welcome to live Soccer match");
        obj.setSubjectDetails("Current score 0-0");
        observer1.unsubscribe();
        obj.setSubjectDetails("It's a goal!!");
        obj.setSubjectDetails("Current score 1-0");
    }
}
```

Rezultat:

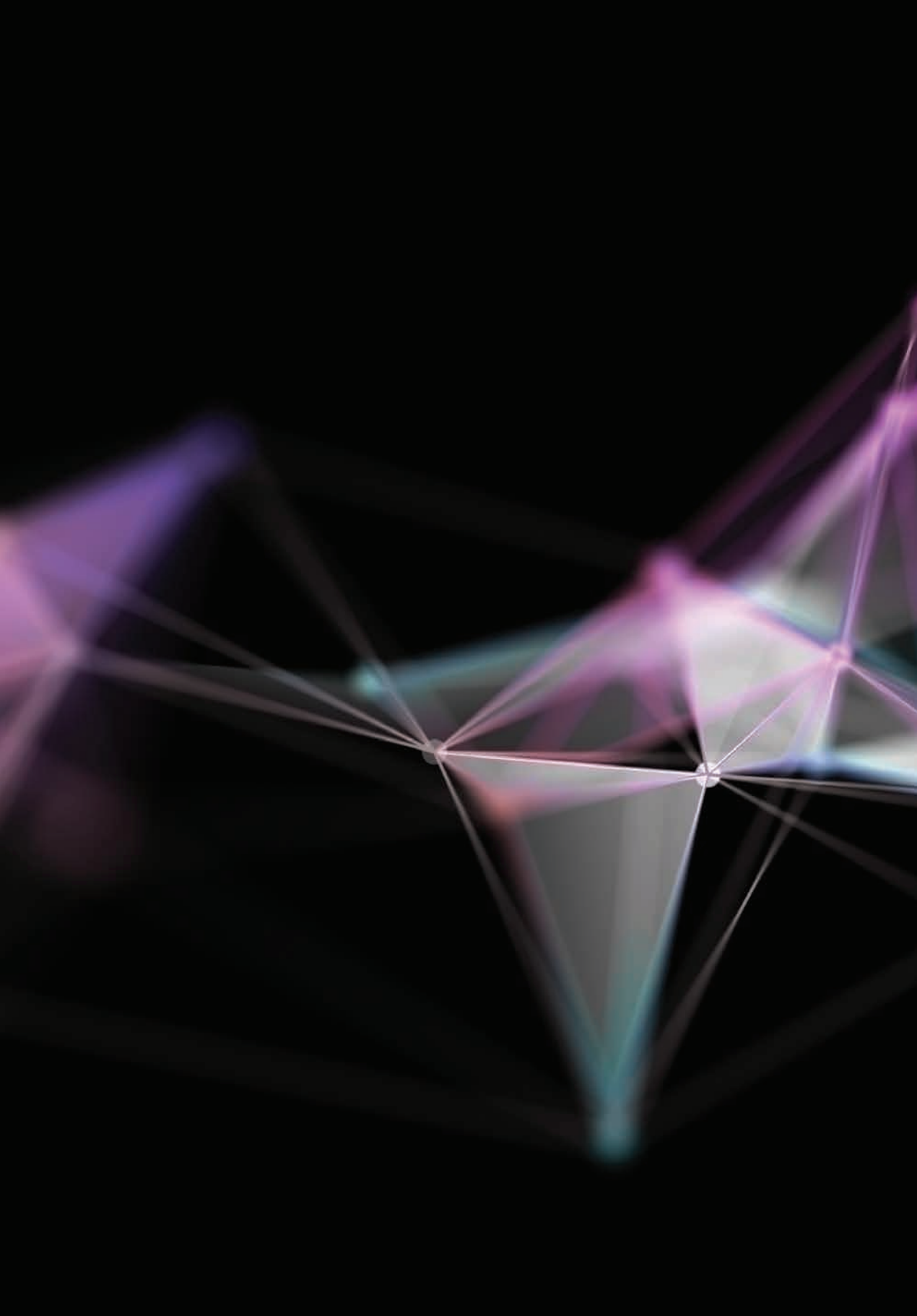
```
Subscribing Petar Petrovic [Niksic] to Derbi [31.08.2022] ...
Subscribed successfully.
Subscribing Nikola Nikolic [Beograd] to Derbi [31.08.2022] ...
Subscribed successfully.
[Petar Petrovic [Niksic]]: Welcome to live Soccer match
[Nikola Nikolic [Beograd]]: Welcome to live Soccer match
[Petar Petrovic [Niksic]]: Current score 0-0
[Nikola Nikolic [Beograd]]: Current score 0-0
Unsubscribing Petar Petrovic [Niksic] to Current score 0-0 ...
Unsubscribed successfully.
[Nikola Nikolic [Beograd]]: It's a goal!!
[Nikola Nikolic [Beograd]]: Current score 1-0
```

Zadatak 5-10. Kreirati aplikaciju za banku koja će redovno obavještavati svoje klijente (*BankClients*) i društvene medije (*SocialMedia*) o promeni kamate (*interest*) na pozajmljena sredstva.

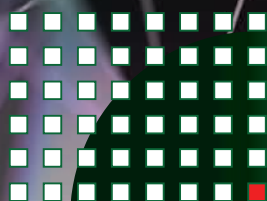
Literatura

- Baron, D., & Safari, an O. M. C. (2021). *Game Development Patterns with Unity 2021* (second). O'Reilly for Higher Education.
- Bass, L., Clements, P., Kazman, R., & Safari, an O. M. C. (2021). *Software Architecture in Practice, 4th Edition*. <https://www.oreilly.com/library/view/software-architecture-in/9780136885979/>
- Bin Uzayr, S. (2023). *Software design patterns: the ultimate guide*. CRC Press. <https://www.routledge.com/Software-Design-Patterns-The-Ultimate-Guide/Uzayr/p/book/9781032311760>
- Design Patterns - Adapter pattern*. (n.d.). Retrieved May 14, 2023, from https://scweb.sce.uhcl.edu/helm/WEBPAGES-SoftwareDesignPatterns/myfiles/TableContents/Module-8/adapter_pattern_works_as_a_bridg.html
- Freeman, Eric, Robson, E. (2021). *Head First Design Patterns: A Brain-Friendly Guide: Building Extensible and Maintainable Object-Oriented Software* (second). O'Reilly. https://www.amazon.co.uk/Head-First-Design-Patterns-Object-Oriented/dp/149207800X/ref=sr_1_6?crid=3IRO57VE81Y5G&keywords=software+design+patterns&qid=1684089359&s=books&prefix=software+design+patterns+%2Cstripbooks%2C108&sr=1-6
- Hadzi-Puric, J. (n.d.). *Razvoj softvera 2*. Retrieved May 14, 2023, from <https://poincare.matf.bg.ac.rs/~jelenagr/rs2/>
- Ivanković, Z., & Lacmanović, D. (n.d.). *Softversko inženjerstvo 2*. In Univerzitet u Novom Sadu. <http://www.tfzr.uns.ac.rs/Content/files/1/SE2 Ivankovic Lacmanovic.pdf>
- Java Design Pattern - Object Pool*. (2018). GP Coder. <https://gpcoder.com/4456-huong-dan-java-design-pattern-object-pool/>
- Joshi, R. (2018). *Java design patterns reusable solutions to common problems*. Java Code Geeks. <https://enos.itcollege.ee/~jpoial/java/naited/Java-Design-Patterns.pdf>
- Mehra, L. . (2021). *Software Design Patterns for Java Developers*. BPD Publications.
- Meier, J. D., Hill, D., Homer, A., Taylor, J., Wall, L., Boucher, R. J., & Bogawat, A. (2009). Chapter 3: Architectural Patterns and Styles. *Microsoft Press*, January, 45. <http://msdn.microsoft.com/en-us/library/ee658117.aspx>
- Milanov, G., & Njeguš, A. (2012). Analysis of Return on Investment in Different Types of Agile Software Development Project Teams. *Informatica Economica*, 16(4).

- Njegus, A. (2021). *Business Information Systems*. Singidunum University. <https://singipedia.singidunum.ac.rs/izdanje/40758-poslovni-informacioni-sistemi>
- Njeguš, A., & Milanov, G. (2011). Qualitative comparison of agile and iterative software development methodologies. *2011 19th Telecommunications Forum, TELFOR 2011 - Proceedings of Papers*. <https://doi.org/10.1109/TELFOR.2011.6143783>
- Sataloff, R. T., Johns, M. M., & Kost, K. M. (2005). *Design Patterns Explained: A New Perspective on Object Oriented Design* (2nd ed.). Addison-Wesley.
- Shvets, A. (2019). Dive Into Design Patterns. In *Refactoring.guru*. <https://refactoring.guru/design-patterns/book>
- Swithinbank, P., Chessell, M., Gardner, T., Griffin, C., Man, J., Wylie, H., & Yusuf, L. (2005). *Patterns: Model-Driven Development Using IBM Rational Software Architect*.
- Wengner, M. (2023). *Practical design patterns for Java developers : hone your software design skills by implementing popular design patterns in Java*. Packt Publishing.



PRILOZI



**PRILOG 1: PITANJA ZA
TESTIRANJE ZNANJA**

**PRILOG 2: REŠENJA
ZADATAKA**

PRILOG1: PITANJA ZA TESTIRANJE ZNANJA

1. Šta su obrasci projektovanja (*Design patterns*)?

- a) opšte, ponovo upotrebljivo rešenje za probleme koji se često mogu pojaviti u arhitekturi softvera.
- b) ponovo upotrebljiv softverski kod koji nudi opšte rešenje za učestale probleme u razvoju softvera.
- c) opisno rešenje problema koje zavisi od programskog jezika i mogu se primeniti samo na određene tehnologije.

2. Koji su osnovni tipovi obrazaca projektovanja softvera?

- a) Strukturni obrasci
- b) Obrasci arhitekture
- c) Kreacioni obrasci
- d) Obrasci ponašanja
- e) Obrasci uvođenja
- f) Obrasci sistema

3. Koji su osnovni elementi obrazaca projektovanja?

- a) Problem koji obrazac rešava
- b) Arhitektura softvera
- c) Resursi koji su potrebni za rešavanje problema
- d) Programski kod koji se odmah može izvršiti
- e) Pravila kako treba koristiti resurse za rešavanje problema
- f) Smernice gde treba koristiti obrazac

4. Šta su kreacioni obrasci?

- a) Tip obrazaca koji se bave kompozicijom klasa u fleksibilne i efikasne strukture podataka.
- b) Tip obrasca koji se bave komunikacijom između objekata.
- c) Tip obrazaca koji obezbeđuju način kreiranja objekata sakrivajući pri tom logiku kreiranja, pre nego da se bave direktnim instanciranjem objekata koristeći operator *new()*.
- d) Tip obrasca koji se isključivo bavi prezentacionim slojem arhitekture softvera.

5. Koji obrasci spadaju u grupu kreacionih obrazaca?

- a) Factory Method
- b) Facade
- c) Builder
- d) Bridge
- e) Memento
- f) Decorator
- g) Chain of responsibility

6. Koji su strukturni obrasci?

- a) Factory Method
- b) Facade
- c) Builder
- d) Bridge
- e) Memento
- f) Decorator
- g) Chain of responsibility

7. Koji su obrasci ponašanja?

- a) Adapter
- b) Iterator
- c) Composite
- d) Strategy
- e) Facade
- f) Mediator

8. Ukoliko želite da klijenti koriste istu instancu klase A, šta bi bilo rešenje:

- a) Označili bi klasu A kao final
- b) Označili bi klasu A kao apstraktnu
- c) Primenili bi Singleton obrazac na klasu A
- d) Primenili bi Memento obrazac na klasu A

9. Ukoliko implementacija metoda zavisi od promene stanja u vrednostima atributa i ukoliko imate puno uslova koje morate da proveravate, koji obrazac bi primenili:

- a) State
- b) Singleton
- c) Observer
- d) Iterator
- e) Adapter

10. Koji obrazac nam omogućava da kreiramo i vratimo instancu kreirane klase, sakrivajući od klijenta postupak kreiranja te klase? Klijent koristi samo jedan zajednički interfejs za kreiranje novog tipa objekata.
- a) Singleton
 - b) Builder
 - c) Factory Method
 - d) Abstract Factory
11. Koji bi obrazac primenili, ukoliko konkretno odeljenje u nekoj kompaniji treba da kreira njemu odgovarajući dokument, npr. Računovodstvo treba da kreira račun, Magacin treba da kreira prijemnicu/otpremnicu robe, Marketing kreira porudžbinu itd. Pri tome svako odeljenje jedino komunicira preko zajedničkog interfejsa i ne zanimaju ga detalji oko kreiranja ovih dokumenata.
- a) Singleton
 - b) Builder
 - c) Factory Method
 - d) Abstract Factory
12. Ukoliko pravite aplikaciju za konkretnog proizvođača automobila, koji nabavlja delove od fabrike za proizvodnju točkova, fabrike za proizvodnju motora i fabrike koja proizvodi haube, koji obrazac će vam omogućiti interfejs za kreiranje ovih delova automobila?
- a) Singleton
 - b) Builder
 - c) Factory Method
 - d) Abstract Factory
13. Ukoliko pravite aplikaciju za McDonald's, konkretno za Happy Meal obrok, koji se sastoji od poznatih sastojaka, s tim što klijent može birati piće, vrstu hamburgera i igračku, koji bi obrazac primenili za kreiranje željenog Happy Meal obroka?
- a) Singleton
 - b) Builder
 - c) Factory Method
 - d) Abstract Factory
14. Koji su osnovni elementi strukture Factory Method obrasca?
- a) ConcreteFactory
 - b) Director
 - c) Creator
 - d) Product
 - e) FactoryMapping
 - f) ProductMapping
 - g) ConcreteBuilder

15. Koji su osnovni elementi strukture Abstract Factory obrasca?

- a) ConcreteFactory
- b) Director
- c) Creator
- d) Product
- e) FactoryMapping
- f) ProductMapping
- g) ConcreteBuilder

16. Koji obrazac nam omogućava da klase sa nekompatibilnim interfejsima međusobno sarađuju bez potrebe za modifikacijom njihovog izvornog koda?

- a) Iterator
- b) Command
- c) Strategy
- d) Adapter
- e) Bridge
- f) Composite

17. Kompanija Wizzair na osnovnu cenu karte pruža i dodatne usluge, kao što su rukovanje prtljagom, catering u avionu i druge brojne naknade. Želimo da kreiramo aplikaciju koja će izračunati ukupnu cenu svih željenih usluga klijenata. Koji obrazac bi primenili?

- a) Command
- b) Strategy
- c) Builder
- d) Decorator
- e) Bridge
- f) Composite

18. Koji obrazac omogućava da korisnik dodaje nove funkcionalnosti postojećem objektu bez menjanja njegove strukture, odnosno da deluje kao omotač na originalnu klasu i omogući dodavanje dodatnih funkcionalnosti?

- a) Command
- b) Strategy
- c) Builder
- d) Decorator
- e) Bridge
- f) Composite

- 19. Ukoliko razvijate aplikaciju za rezervaciju putovanja, gde će klijent, direktnim pristupom različitim podsistemima, moći da rezerviše avio let, smeštaj, rentakar i druge usluge koje se nude na samoj destinaciji, koji obrazac bi rešio problem ovako tesno povezanih klasa?**
- a) Command
 - b) Strategy
 - c) Builder
 - d) Decorator
 - e) Bridge
 - f) Facade
- 20. Koji strukturni obrazac omogućava jednostavan interfejs ka većem broju složenih podsistemima, uključujući samo one funkcionalnosti koje su zaista neophodne klijentu?**
- a) Command
 - b) Facade
 - c) Strategy
 - d) Builder
 - e) Decorator
 - f) Bridge
- 21. Koji obrazac razdvaja ogroman broj klasa, često i tesno povezanih, u dve odvojene hijerarhije klasa: Apstrakcija i Implementacija.**
- a) Command
 - b) Facade
 - c) Strategy
 - d) Builder
 - e) Decorator
 - f) Bridge
- 22. Kada je model aplikacije predstavljen kao stablo, koji obrazac bi ste primenili?**
- a) Command
 - b) Composite
 - c) Strategy
 - d) Builder
 - e) Decorator
 - f) Bridge

23. Koji obrazac rešava problem prolaska kroz različite strukture kolekcije?

- a) Command
- b) Facade
- c) Strategy
- d) Builder
- e) Iterator
- f) Bridge

24. Potrebno je proći kroz sve elemente kolekcije Zaposleni i svakom ko ima platu manje od 500e potrebno je povećati za 100e. Koji obrazac bi koristitli za prolazak kroz datu kolekciju?

- a) Command
- b) Facade
- c) Strategy
- d) Builder
- e) Iterator
- f) Bridge

25. Želite da razvijate aplikaciju za upravljanje kućom i da omogućite klijentu da kada klikne na opciju npr. **Turn-off**, da se svi upaljeni elementi ugase, bez potrebe da klijent mora da ih gasi jedan po jedan ili da ima znanja o detaljima određenih uređaja, kako bi uspeo sve da ih pogasi. Objekat konkretne naredbe će proslediti zahteve primaocu (*receiver*) koji će realizovati celokupan posao tj. voditi računa da svi elementi budu ugašeni. Koji obrazac bi vam pomogao u realizaciji ove aplikacije?

- a) Command
- b) Composite
- c) Strategy
- d) Decorator
- e) Bridge
- f) Builder

26. Želite da napravite aplikaciju koja će posredovati u komunikaciji između raznih uređaja (npr. dron, kola) sa ciljem da umanjite složenost i jaku zavisnost između objekata koji međusobno komuniciraju. Koji obrazac bi primenili?

- a) Template method
- b) Facade
- c) Strategy
- d) Builder
- e) Mediator
- f) Bridge

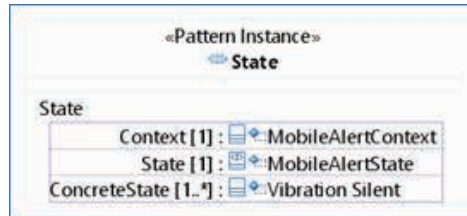
- 27. Koji obrazac ponašanja vam omogućava da implementirate „undo“ akcije, odnosno da povratite prethodno stanje objekta?**
- a) Comand
 - b) Mediator
 - c) Memento
 - d) Strategy
 - e) Template Method
 - f) Visitor
 - g) Chain of Responsibility
- 28. Koji obrazac se koristi kada za konkretni zadatak postoji više načina njegove realizacije, a klijent odlučuje o aktuelnoj implementaciji koja će se koristiti u trenutku izvršavanja?**
- a) Comand
 - b) Mediator
 - c) Memento
 - d) Strategy
 - e) Template Method
 - f) Visitor
 - g) Chain of Responsibility
- 29. Ukoliko razvijate aplikaciju, gde će korisnik, u zavisnosti od analize troškova, vremena i pogodnosti, selektovati jedan način prevoza do aerodroma (taxi, autobus, limuzina, helikopter ...) i platiti za uslugu, koji obrazac bi primenili?**
- a) Comand
 - b) Mediator
 - c) Memento
 - d) Strategy
 - e) Template Method
 - f) Visitor
 - g) Chain of Responsibility
- 30. Želite da razvijete aplikaciju koja će omogućiti povezivanje sa prijateljima na bilo kojoj platformi za društvene medije (npr. Facebook, Twitter, Instagram...). Klijent će samo odabrati prijatelja i željenu platformu, a aplikacija će ga transparentno povezati. Koji obrazac bi koristili?**
- a) Comand
 - b) Mediator
 - c) Memento
 - d) Strategy
 - e) Template Method
 - f) Visitor

31. Koji obrazac vam omogućava da u apstraktnoj klasi definišete final metodu sa jasnim koracima (algoritmom) realizacije metoda, koji je zajednički za sve potklase?
- a) Comand
 - b) Mediator
 - c) Memento
 - d) Strategy
 - e) Template Method
 - f) Visitor
32. Potrebno je da kreirate jedinstvenu aplikaciju za praćenje bilo kojih utakmica (npr. Fudbal, Kriket...). Sve utakmice imaju zajedničke metode kao što su *initialize()*, *startPlay()*, *endPlay()* i *play()* čiji ste redosled jasno definisali u apstraktnoj klasi *Game*. Koji obrazac bi primenili kako bi osigurali redosled realizacije metoda?
- a) Comand
 - b) Mediator
 - c) Memento
 - d) Strategy
 - e) Template Method
 - f) Visitor
33. Želite da kreirate aplikaciju za taksi usluge, a da pri tom, kada hoćete da uvedete nove funkcionalnosti, ne želite da menjate osnovnu strukturu klase Klijent. Koji obrazac vam omogućava da deifnišete nove operacije bez menjaja postojeće stukture objekta?
- a) Comand
 - b) Mediator
 - c) Memento
 - d) Strategy
 - e) Template Method
 - f) Visitor
34. Koji obrazac vam omogućava da pošaljete zahtev klasama, koje će odlučiti da li će obraditi pristigli zahtev ili ga proslediti narednoj klasi u lancu?
- a) Comand
 - b) Mediator
 - c) Memento
 - d) Strategy
 - e) Template Method
 - f) Visitor
 - g) Chain of Responsibility

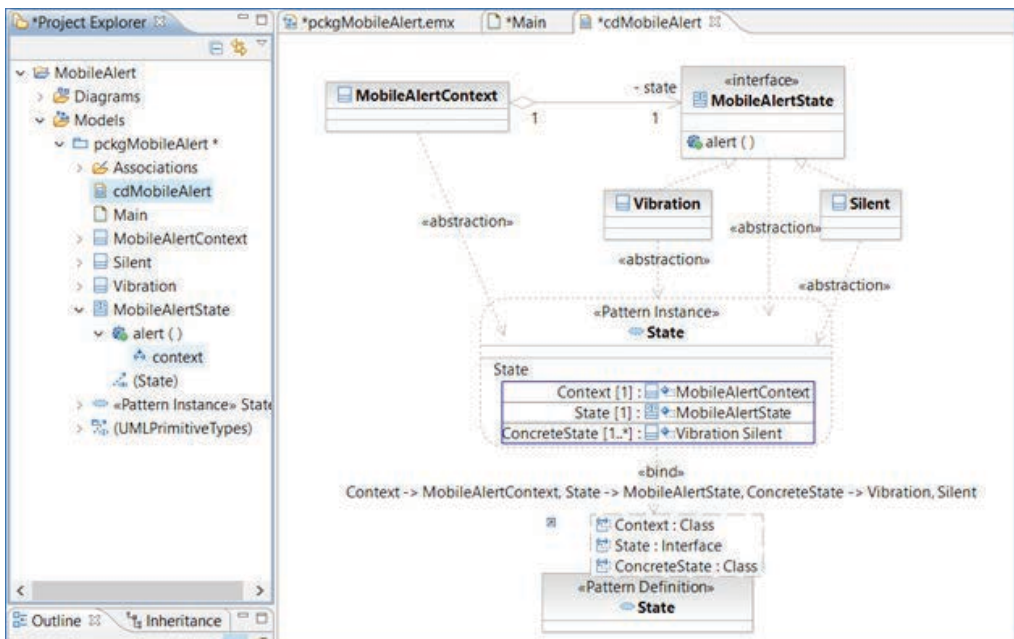
PRILOG 2: REŠENJA ZADATAKA

Zadatak 2-1. Primeniti *State* obrazac nad klasom *MobileAlert*.

Korak 1: Prevući *State* patern i uneti nazive klasa



Korak 2: Prikazati klase i njihove međusobne veze



Korak 3: Izgenerisati kod i prilagoditi ga po želji i prikazati rezultat izvršenja

Interfejs:

```
public interface MobileAlertState {
    public void alert(MobileAlertContext context);
}
```

Potklasa *Silent.java*:

```
public class Silent implements MobileAlertState {
    public void alert(MobileAlertContext context) {
        System.out.println("Silent mode");
    }
    public Silent() {
    }
}
```

Potklasa *Vibration.java*:

```
public class Vibration implements MobileAlertState {
    public void alert(MobileAlertContext context) {
        System.out.println("Vibration mode");
    }
    public Vibration() {
    }
}
```

Kontekstna klasa:

```
public class MobileAlertContext {
    private MobileAlertState state;
    public MobileAlertState getState() {
        return state;
    }
    public void setState(MobileAlertState state) {
        this.state = state;
    }
    public MobileAlertContext(MobileAlertState aMobileAlertState) {
        this.state = aMobileAlertState;
    }
    public MobileAlertContext() {
        this.state=new Vibration();
    }
    public void alert() {
        state.alert(this);
    }
}
```

Glavna klasa:

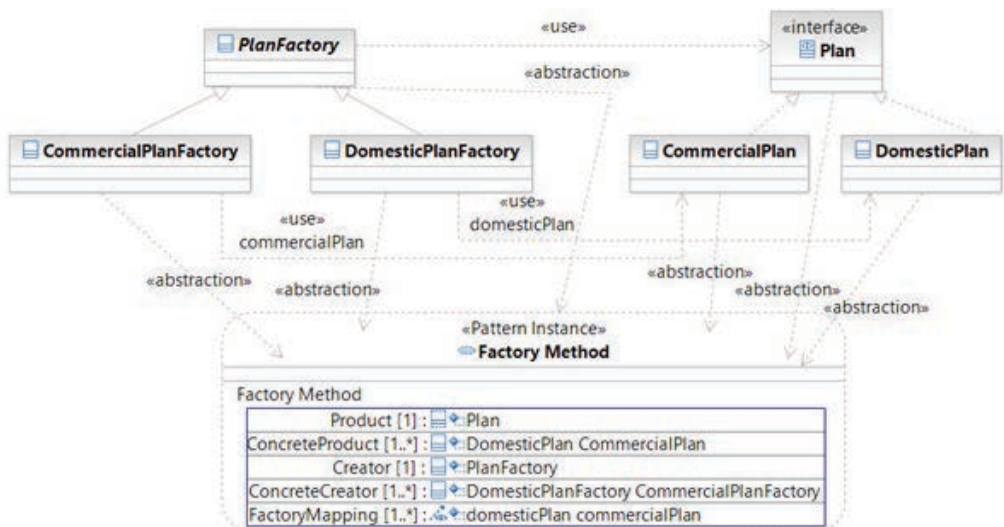
```
public class MobileAlertClient {
    public static void main(String[] args) {
        MobileAlertContext stateContext = new MobileAlertContext();
        stateContext.alert();
        stateContext.setState(new Silent());
        stateContext.alert();
        stateContext.alert();
        stateContext.setState(new Vibration());
        stateContext.alert();
    }
}
```

Rezultat izvršenja aplikacije:

Vibration mode
Silent mode
Silent mode
Vibration mode

Zadatak 3-1. Kreirati aplikaciju za generisanje računa za električnu energiju u zavisnosti od plana potrošnje primenom *Factory Method* obrasca.

Dijagram klasa primenom *Factory Method* obrasca:



Prilikom generisanja Java koda, klasa **Plan** će po defaultu biti kreirana kao interfejs. U ovom zadatku je prikazan primer klase **Plan** kao apstraktne klase.

Product klasa *Factory Method* obrasca:

```
public abstract class Plan {
    protected double rate;
    abstract void getRate();
    public void calculateBill(int units){
        System.out.println(units*rate);
    }
}
```

ConcreteProduct klase:

```
public class DomesticPlan extends Plan {
    public void getRate() {
        rate = 3.50;
    }
    public DomesticPlan() {
    }
}
```

```
public class CommercialPlan extends Plan {
    public void getRate() {
        rate = 7.50;
    }
    public CommercialPlan() {
    }
}
```

Creator klasa:

```
public abstract class PlanFactory {
    public abstract Plan createPlan(String planType);
}
```

Concrete Creator klasa:

```
public class GenerateBillFactory extends PlanFactory {
    public Plan createPlan(String planType) {
        if(planType==null) {
            return null;
        }
        if (planType.equalsIgnoreCase("Domestic")){
            return new DomesticPlan();
        }
        if (planType.equalsIgnoreCase("Commercial")){
            return new CommercialPlan();
        }
        return null;
    }
}
```

Glavna klasa:

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

public class PlanMain {
    public static void main(String[] args) throws IOException {
        PlanFactory generateBillFactory = new GenerateBillFactory();
        System.out.println("Enter the name of plan for which the
        bill will be generated: ");
        BufferedReader br=new BufferedReader(new InputStreamReader
        (System.in));

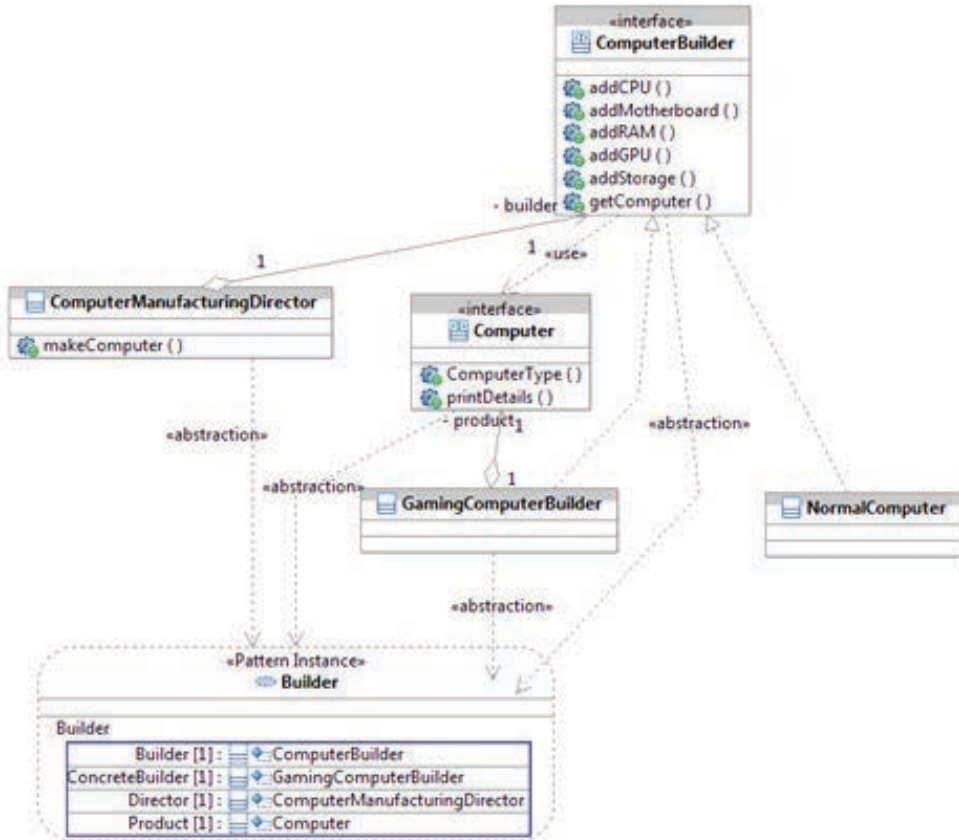
        String planName=br.readLine();
        System.out.println("Enter the number of units for bill will
        be calculated: ");
        int units=Integer.parseInt(br.readLine());

        Plan aBill=generateBillFactory.createPlan(planName);
        System.out.print("Bill amount for "+planName+" of "+units+
        " units is: ");
        aBill.getRate();
        aBill.calculateBill(units);
    }
}
```

Rezultat izvršenja aplikacije:

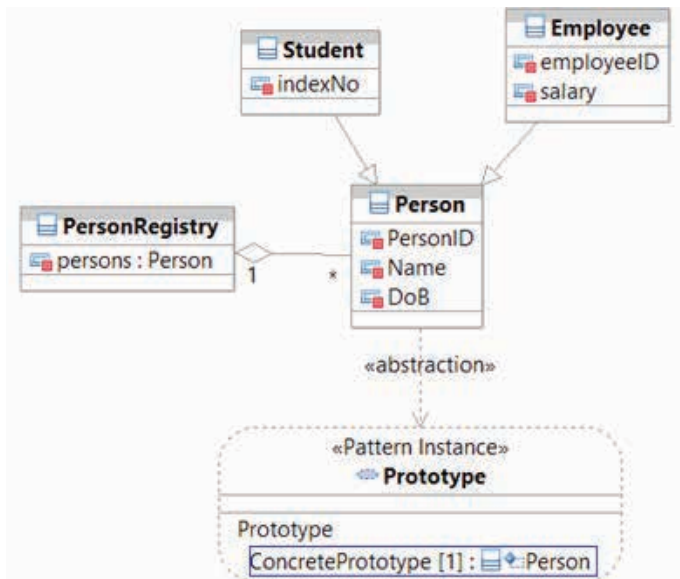
```
Enter the name of plan for wich the bill will be generated:
domestic
Enter the number of units for bill will be calculated:
1000
Bill amount for domestic of 1000 units is: 3500.0
```

Zadatak 3-2. Primeniti odgovarajući obrazac projektovanja na aplikaciju koja se bavi sklapanjem normalnih i gejming računara.



Zadatak 3-3. Primeniti odgovarajući obrazac projektovanja na aplikaciju koja klonira objekte tipa Osobe (Person) na Univerzitetu.

Objekti tipa *Person* na Univerzitetu mogu biti studenti ili zaposleni. Atributi apstraktne klase *Person* su *PersonID*, *Name*, *DoB* (Date of Birth). Klasa *Student* sadrži *IndexNo*, a klasa *Zaposleni* (*Employee*) sadrži *EmpID* i *Salary*. Klonirati objekat *Zaposleni*, promeniti iznos plate i prikazati rezultate.



```

public class Person implements Cloneable {
    private String PersonID;
    private String Name;
    private String DoB;

    public void setPersonID(String personID) {
        PersonID = personID;
    }
    public String getPersonID() {
        return PersonID;
    }
    public void setName(String name) {
        Name = name;
    }
    public String getName() {
        return Name;
    }
}

```

```

    public void setDoB(String doB) {
        DoB = doB;
    }
    public String getDoB() {
        return DoB;
    }
    public Object clone() {
        Object clone = null;
        try{
            clone = super.clone();
        }
        catch (CloneNotSupportedException e) {
            e.printStackTrace();
        }
        return clone;
    }
    public String toString() {
        return "Person cloned and creates "+this.PersonID;
    }
}

public class Employee extends Person {
    private String EmployeeID;
    private double Salary;
    public void setEmployeeID(String employeeID) {
        EmployeeID = employeeID;
    }
    public String getEmployeeID() {
        return EmployeeID;
    }
    public void setSalary(double salary) {
        Salary = salary;
    }
    public double getSalary() {
        return Salary;
    }
    @Override
    public String toString() {
        return "Employee["+EmployeeID+", "+Salary+"]";
    }
}

public class Student extends Person {
    private String indexNo;
    public void setIndexNo(String indexNo) {
        this.indexNo = indexNo;
    }
    public String getIndexNo() {
        return indexNo;
    }
    @Override
    public String toString() {

```

```
        return "Student["+ "indexNo="+indexNo+"]";
    }
}
import java.util.*;

public class PersonRegistry {
    private Map<PersonType, Person> persons=new HashMap<PersonType,
        person>();
    public PersonRegistry(){
        this.initialize();
    }

    public Person getPerson(PersonType personType) throws CloneNot
        SupportedException {
        Person person = null;
        person=(Person)persons.get(personType).clone();
        return person;
    }

    private void initialize(){
        Employee employee = new Employee();
        employee.setPersonID("100");
        employee.setName("Ana Jovanovic");
        employee.setDoB("5/10/1997");
        employee.setEmployeeID("AJ100");
        employee.setSalary(60000);

        Student student=new Student();
        student.setPersonID("200");
        student.setName("Petar Petrovic");
        student.setDoB("25/02/2004");
        student.setIndexNo("2023/123466");

        persons.put(PersonType.EMPLOYEE, employee);
        persons.put(PersonType.STUDENT, student);
    }
}

public enum PersonType {
    STUDENT,
    EMPLOYEE;
}

public class Application {
    public static void main(String[] args) throws CloneNotSupporte
        dException {
        PersonRegistry personRegistry=new PersonRegistry();
        Employee employee=(Employee)personRegistry.getPerson(PersonType.
            EMPLOYEE);
        System.out.println(employee);
        employee.setSalary(70000.00);
        System.out.println(employee);
        Employee employee1=(Employee) personRegistry.getPerson(PersonType.
            EMPLOYEE);
        System.out.println(employee1);
    }
}
```

Rezultat izvršavanja aplikacije:

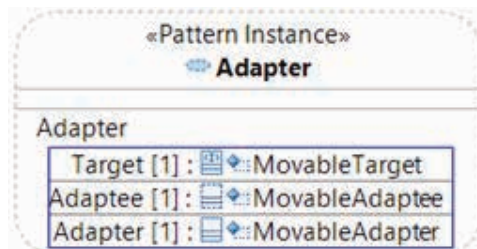
```
Employee[EmployeeID=AJ100, Salary=60000.0]
Employee[EmployeeID=AJ100, Salary=70000.0]
Employee[EmployeeID=AJ100, Salary=60000.0]
```

Zadatak 4-1. Primeniti Adapter obrazac za prikaz brzine kola u MPH i km/h.

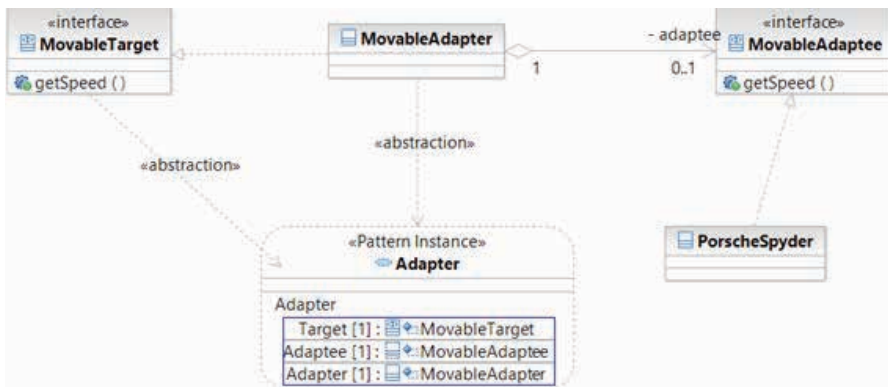
Postojeća aplikacija, koja je razvijena za SAD tržište, vraća najveću brzinu luksuznih kola u miljama na sat (Miles Per Hour, MPH). Kako se tržište proširilo, neophodno je istu aplikaciju koristiti i za klijenta u Velikoj Britaniji, koji želi iste rezultate, ali u kilometrima na sat (km/h). Upotrebiti odgovarajući obrazac koji će prikazati maksimalnu brzinu kola Porsche 918 Spyder u km/h, ukoliko je poznato da PorscheSpyder ide 214 mph. Funkcija `convertMPHtoKMPH` vraća $\text{mph} * 1.60934$.

Koraci rešavanja zadatka su:

1. Prevući Adapter obrazac i popuniti kao na slici:



2. Prikazati sve relacije i dodati ostale klase. Interfejsima dodati metodu `getSpeed()`:



3. Generisati Java programski kod i prilagoditi kod potrebi aplikacije.

- ♦ Ukoliko prethodno nije naznačen tip podatka koju metoda *getSpeed()* vraća, onda izmeniti difoltni tip u *double*.
- ♦ Izgled interfejsa je:

```
public interface MovableTarget {  
    public double getSpeed();  
}  
public interface MovableAdaptee {  
    public double getSpeed();  
}
```

- ♦ Metodu *getSpeed()* klase *PorscheSpyder* dopuniti sa povratnom vrednošću, gde se prikazuje koju maksimalnu brzinu ovaj model razvija u mph:

```
public class PorscheSpyder implements MovableAdaptee {  
    public double getSpeed() {  
        return 214;  
    }  
}
```

- ♦ U klasi *MovableAdapter* izgenerisano je polje *adaptee*, kao i njegove *get* i *set* metode. Takođe, izgenerisani su i konstruktori (parametarski i neparametarski). Potrebno je dopuniti metodu *getSpeed()* sa funkcijom koja konvertuje MPH u KMPH. Voditi računa o tipu povratne vrednosti vezano za brzinu, odnosno voditi računa da bude *double*.

```
public class MovableAdapter implements MovableTarget {  
    private MovableAdaptee adaptee;  
    public MovableAdaptee getAdaptee() {  
        return adaptee;  
    }  
    public void setAdaptee(MovableAdaptee adaptee) {  
        this.adaptee = adaptee;  
    }  
    public double getSpeed() {  
        return convertMPHtoKMPH(adaptee.getSpeed());  
    }  
    private double convertMPHtoKMPH(double mph){  
        return mph*1.60934;  
    }  
    public MovableAdapter() {  
    }  
    public MovableAdapter(MovableAdaptee aMovableAdaptee) {  
        this.adaptee = aMovableAdaptee;  
    }  
}
```

4. Kreirati *Client* klasu koja će sadržati *main* metodu i izvršiti program.

♦ Izgled *Client* klase:

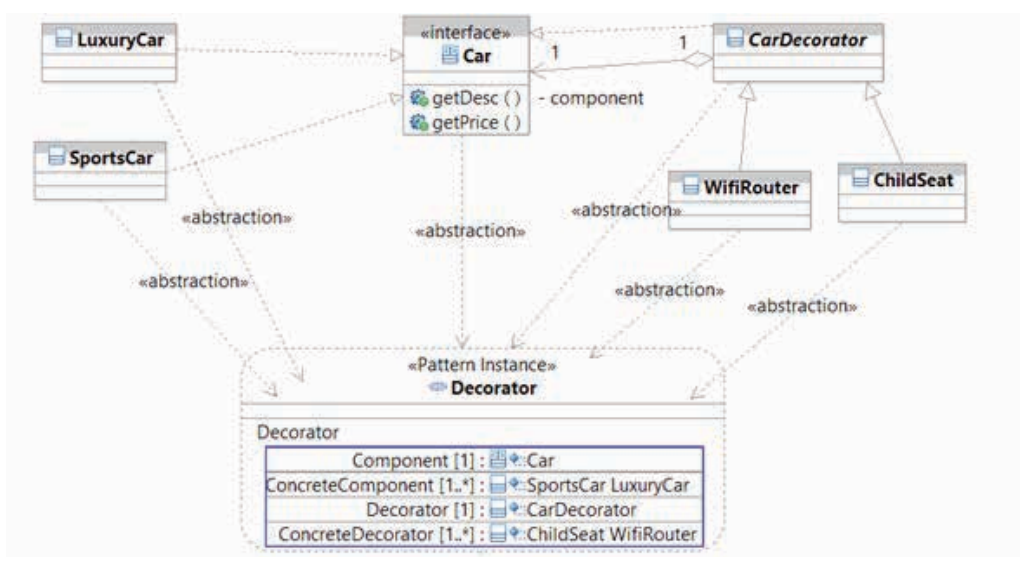
```
public class Client {
    public static void main(String[] args) {
        MovableAdaptee porscheSpyder = new PorscheSpyder();
        MovableTarget porscheSpyderTarget = new MovableAdapter
            (porscheSpyder);
        System.out.println("Maksimalna brzina Porsche 918 Spyder je "+
            porscheSpyderTarget.getSpeed() + " KM/H");
    }
}
```

♦ Rezultat izvršavanja aplikacije:

Maksimalna brzina Porche 918 Spyder je 344.39876 KM/H

Zadatak 4-2. Primeniti odgovarajući obrazac na aplikaciju za rentiranje kola.

Primenjuje se *Decorator* obrazac.



```
public interface Car {
    public String getDesc();
    public double getPrice();
}

public class SportsCar implements Car {
    public String getDesc() {
        return "Sports car: 30e";
    }
    public double getPrice() {
        return 30;
    }
}

public class LuxuryCar implements Car {
    public String getDesc() {
        return "Luxury car: 50e";
    }
    public double getPrice() {
        return 50;
    }
}

public abstract class CarDecorator implements Car {
    protected Car component;
    public Car getComponent() {
        return component;
    }
    public void setComponent(Car component) {
        this.component = component;
    }
    public CarDecorator(Car rentedCar) {
        this.component = rentedCar;
    }
    public String getDesc(){
        return component.getDesc();
    }
}

public class ChildSeat extends CarDecorator {
    public ChildSeat(Car rentedCar) {
        super(rentedCar);
    }
    @Override
    public String getDesc() {
        return component.getDesc()+" , Child seat: 10";
    }
    @Override
    public double getPrice() {
        return component.getPrice()+10;
    }
}
```

```

public class WifiRouter extends CarDecorator {
    public WifiRouter(Car rentedCar) {
        super(rentedCar);
    }
    @Override
    public String getDesc() {
        return component.getDesc()+" Wifi route: 15";
    }
    @Override
    public double getPrice() {
        return component.getPrice()+15;
    }
}

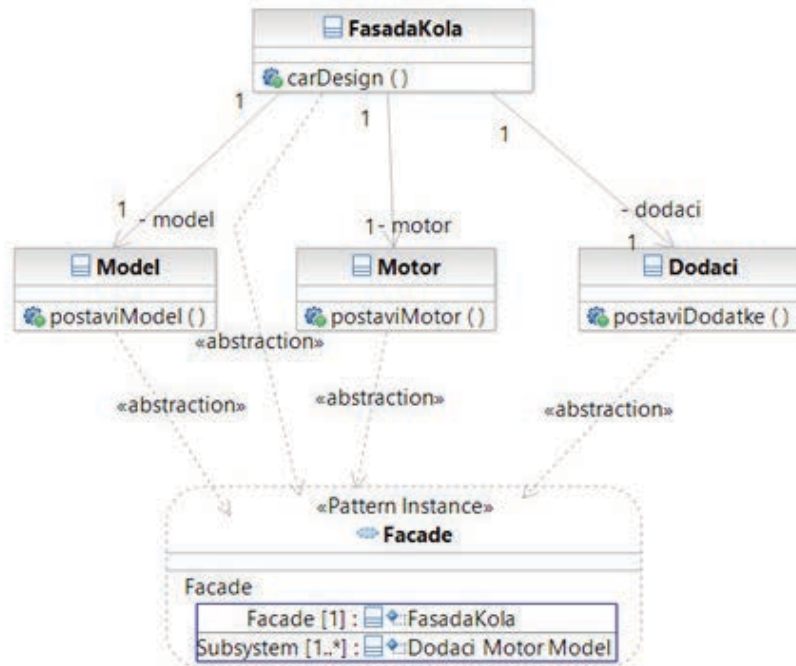
public class Client {
    public static void main(String[] args) {
        Car sportscar=new SportsCar();
        sportscar=new WifiRouter(sportscar);
        sportscar=new ChildSeat(sportscar);
        System.out.println("Description of the rented car "+"\\n"
        +sportscar.getDesc());
        System.out.println("Price: "+sportscar.getPrice());
    }
}

```

Rezultat:**Description of the rented car**

Sports car: 30e, Wifi route: 15; Child seat: 10
 Price: 55.0

Zadatak 4-3. Primeniti obrazac nad aplikacijom za dizajniranje željenih kola.



```
public class Model {
    public void postaviModel() {
        System.out.println("Model kola je postavljen.");
    }
}

public class Motor {
    public void postaviMotor() {
        System.out.println("Motor je postavljen.");
    }
}

public class Dodaci {
    public void postaviDodatke() {
        System.out.println("Dodaci su postavljeni.");
    }
}

public class FasadaKola {
    private Dodaci dodaci;
    public Dodaci getDodaci() {
        return dodaci;
    }
    public void setDodaci(Dodaci dodaci) {
```

```

        this.dodaci = dodaci;
    }
    private Motor motor;
    public Motor getMotor() {
        return motor;
    }
    public void setMotor(Motor motor) {
        this.motor = motor;
    }
    private Model model;
    public Model getModel() {
        return model;
    }
    public void setModel(Model model) {
        this.model = model;
    }
    public void carDesign() {
        System.out.println("Kola se dizajniraju...");
        motor.postaviMotor();
        model.postaviModel();
        dodaci.postaviDodatke();
        System.out.println("Vaša kola su spremna.");
    }

    public FasadaKola() {
        this.dodaci = new Dodaci();
        this.model = new Model();
        this.motor = new Motor();
    }
}

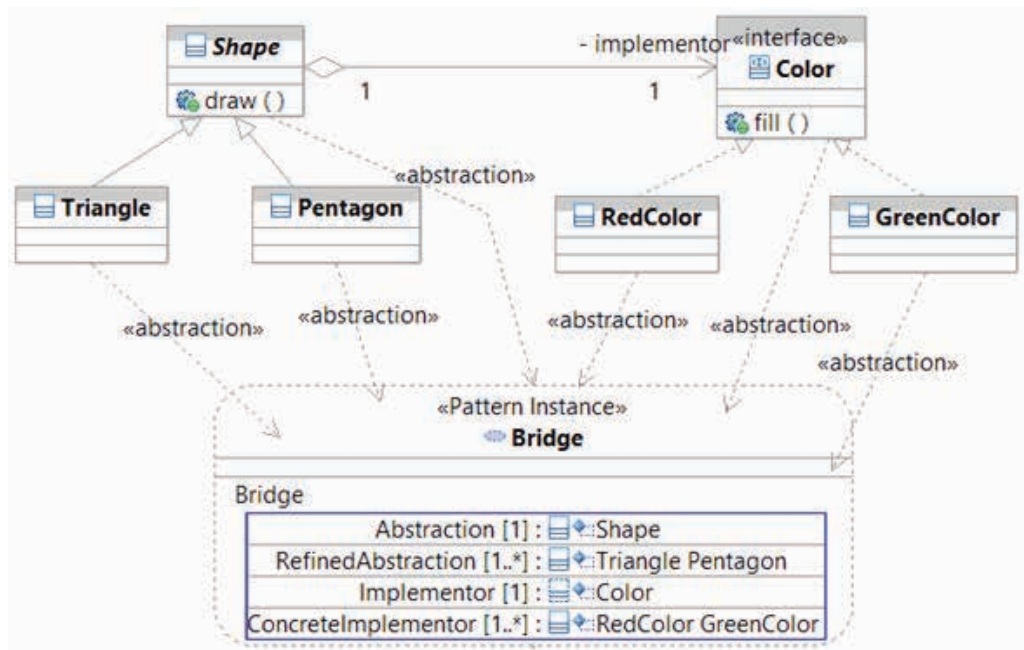
public class Client {
    public static void main(String[] args) {
        FasadaKola fasada=new FasadaKola();
        fasada.carDesign();
    }
}

```

Rezultat:

Kola se dizajniraju...
 Motor je postavljen.
 Model kola je postavljen.
 Dodaci su postavljeni.
 Vaša kola su spremna.

Zadatak 4-4. Primeniti odgovarajući obrazac na aplikaciju koja iscrtava obojene geometrijske oblike.



```
public interface Color {
    String fill();
}

public class Green implements Color {
    public String fill() {
        return "Color is Green.";
    }
}

public class Red implements Color {
    @Override
    public String fill() {
        return "Color is Red.";
    }
}

public abstract class Shape {
    protected Color color;
    public Color getColor() {
        return color;
    }
    public void setColor(Color c) {
        this.color = c;
    }
}
```

```

    public Shape(Color c) {
        this.color=c;
    }
    abstract public String draw();
}

public class Triangle extends Shape {
    public Triangle(Color color) {
        super(color);
    }
    @Override
    public String draw() {
        return "Triangle drawn. "+color.fill();
    }
}

public class Pentagon extends Shape {
    public Pentagon(Color color) {
        super(color);
    }
    public String draw() {
        return "Pentagon drawn. "+ color.fill() ;
    }
}

public class Client {
    public static void main(String[] args) {
        Shape triangle = new Triangle(new Red());
        Shape pentagon=new Pentagon(new Red());
        Shape triangle1 = new Triangle(new Green());

        System.out.println(triangle.draw());
        System.out.println(pentagon.draw());
        System.out.println(triangle1.draw());
    }
}

```

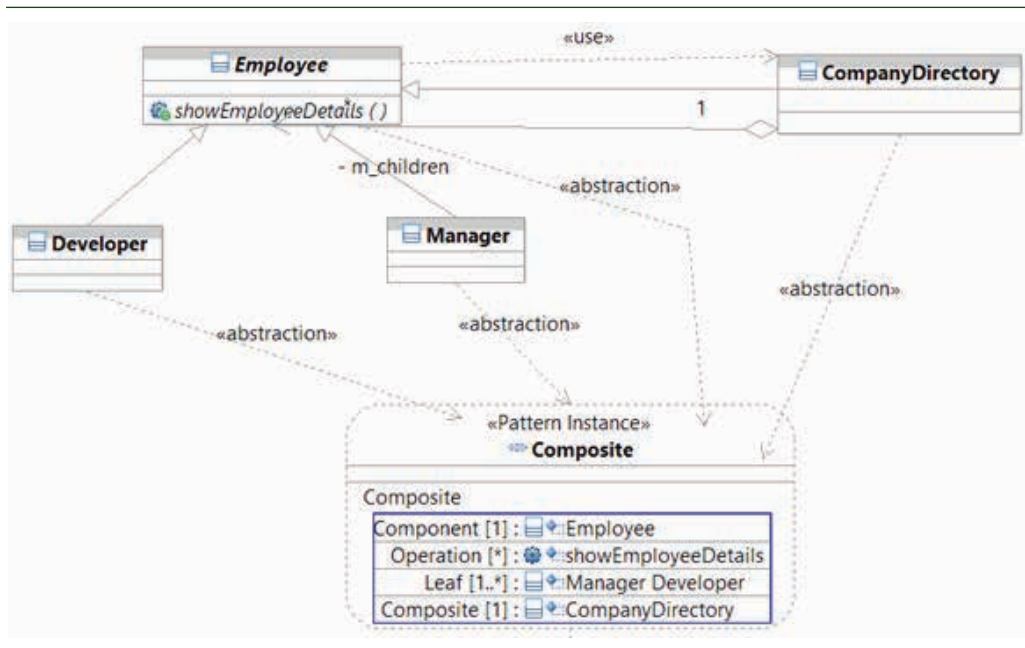
Rezultat:

Triangle drawn. Color is Red.
Pentagon drawn. Color is Red.
Triangle drawn. Color is Green.

Zadatak 4-5. Primeniti *Composite* obrazac na aplikaciju za opis zaposlenih.

Rešenje:

Generalni menadžer i menadžeri su zaposleni u kompaniji, tako da ćemo imati jednu apstraktnu klasu *Employee*, koju će nasleđivati klase: *Developer* i *Manager*. Operacija koju svaki čvor treba da implementira je *showEmployeeDetails()*. Kompozitna klasa je imenik kompanije (*CompanyDirectory*).



Izgenerisani i dopunjen kod:

```

public abstract class Employee {
    public abstract void showEmployeeDetails();
    private Employee m_parent;
    public Employee getParent() {
        return m_parent;
    }
    public void setParent(Employee a_parent) {
        m_parent = a_parent;
    }
    public abstract CompanyDirectory getComposite();
}

```

```

public class Manager extends Employee {
    private long empId;
}

```

```

    private String fname;
    private String lname;
    private String position;
public Manager(long empId, String fname, String lname, String position)
{
    super();
    this.empId = empId;
    this.fname = fname;
    this.lname = lname;
    this.position = position;
}
public void showEmployeeDetails() {
    System.out.println("Employee ID: "+empId+"\n First name:
    "+fname+"\n Last name: "+lname+"\n Position: "+position);
}
public CompanyDirectory getComposite() {
    return null;
}
}

public class Developer extends Employee {
    private long empId;
    private String fname;
    private String lname;
    private String position;
    public Developer(long empId, String fname, String lname, String
    position)
    {
        super();
        this.empId = empId;
        this.fname = fname;
        this.lname = lname;
        this.position = position;
    }
    public void showEmployeeDetails() {
        System.out.println("Employee ID: "+empId+"\n First name:
        "+fname+"\n Last name: "+lname+"\n Position: "+position);
    }
    public CompanyDirectory getComposite() {
        return null;
    }
}

//Kompozitna klasa
import java.util.List;
public class CompanyDirectory extends Employee {
    private List<Employee> m_children;
    public List<Employee> getM_children() {
        return m_children;
    }
    public void setM_children(List<Employee> m_children) {
        this.m_children = m_children;
    }
}

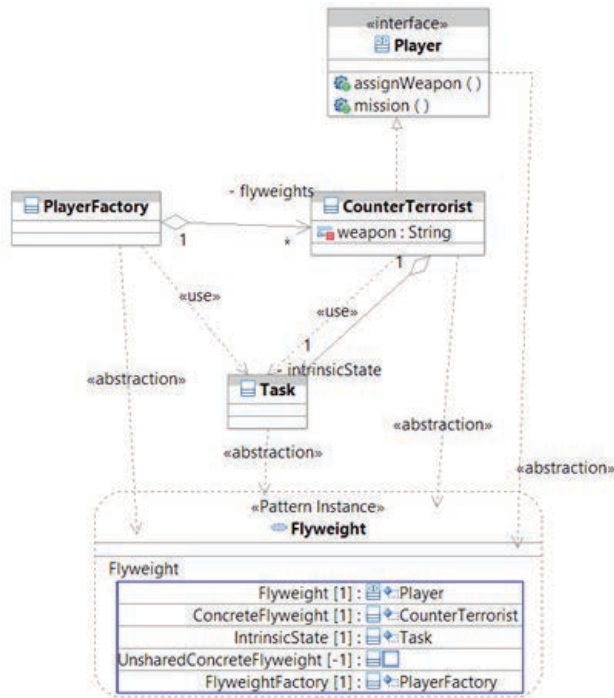
```

```
        public void showEmployeeDetails() {
            for (Employee emp:m_children){
                emp.showEmployeeDetails();
            }
        }
        public CompanyDirectory getComposite() {
            return this;
        }
        public CompanyDirectory() {
            m_children = new java.util.ArrayList<Employee>();
        }
        public synchronized void addChild(Employee a_component) {
            synchronized (a_component) {
                m_children.add(a_component);
                a_component.setParent(this);
            }
        }
        public Employee getChild(int a_childIndex) {
            return (Employee) m_children.get(a_childIndex);
        }
        public synchronized void removeChild(Employee a_component) {
            synchronized (a_component) {
                if (this == a_component.getParent()) {
                    a_component.setParent(null);
                }
                m_children.remove(a_component);
            }
        }
    }
}

public class Client {
    public static void main(String[] args) {
        Developer dev1=new Developer(100, "Marko", "Markovic", "Senior
        Java developer");
        Developer dev2=new Developer(110, "Stefan", "Stefanovic",
        "Junior Java developer");
        CompanyDirectory coEmpl=new CompanyDirectory();
        coEmpl.addChild(dev1);
        coEmpl.addChild(dev2);
        Manager mng1=new Manager(200, "Sava", "Savanovic","SEO
        Manager");
        Manager mng2=new Manager(210, "Petar", "Petrovic","Software
        Manager");
        CompanyDirectory coMng=new CompanyDirectory();
        coMng.addChild(mng1);
        coMng.addChild(mng2);
        CompanyDirectory coDir=new CompanyDirectory();
        coDir.addChild(coEmpl);
        coDir.addChild(coMng);
        coDir.showEmployeeDetails();
    }
}
```

Rezultat:

Employee ID: 100
 First name: Marko
 Last name: Markovic
 Position: Senior Java developer
 Employee ID: 110
 First name: Stefan
 Last name: Stefanovic
 Position: Junior Java developer
 Employee ID: 200
 First name: Sava
 Last name: Savanovic
 Position: SEO Manager
 Employee ID: 210
 First name: Petar
 Last name: Petrovic
 Position: Software Manager

Zadatak 4-6. Primeniti *Flyweight* obrazac na *Counter Strike* igricu (Baron & Safari, 2021).


```
public interface Player {
    public void assignWeapon(String weapon);
    public void mission();
}

public enum Task {
    BOMB, ATTACK
}

public class CounterTerrorist implements Player {
    final Task task = Task.BOMB; //intrinsic state - shareable
    private String weapon; //extrinsic state - supplied by client
    public String getWeapon() {
        return weapon;
    }
    public void setWeapon(String weapon) {
        this.weapon = weapon;
    }
    public void assignWeapon(String weapon) {
        this.weapon = weapon;
    }
    public void mission() {
        System.out.println("Counter Terrorist with weapon "+weapon+
            ". Task is "+task);
    }
    public CounterTerrorist() {
        super();
    }
}

import java.util.*;
public class PlayerFactory {
    //HashMap stores the reference to the object of CounterTerrorist(CT).
    private static HashMap <String, Player> hm = new HashMap<String, Player>();
    // Method to get a player
    public static Player getPlayer(String type) {
        Player p = null;
        /* If an object for CT has already been
           created simply return its reference */
        if (hm.containsKey(type))
            p = hm.get(type);
        else {
            System.out.println("Counter Terrorist Created");
            p = new CounterTerrorist();
            // Once created insert it into the HashMap
            hm.put(type, p);
        }
        return p;
    }
}
```

```

import java.util.Random;
public class CounterStrike {
    // All player types and weapon used by getRandPlayerType() and
    // getRandWeapon()
    private static String[] playerType = {"Terrorist", "CounterTerrorist"};
    private static String[] weapons = {"AK-47", "Maverick", "Gut Knife",
    "Desert Eagle"};

    // Driver code
    public static void main(String args[])
    {
        // Assume that we have a total of 10 players in the game
        for (int i = 0; i < 10; i++)
        {
            // getPlayer() is called simply using the class name since the
            // method is a static one
            Player p = PlayerFactory.getPlayer(getRandPlayerType());
            // Assign a weapon chosen randomly uniformly from the weapon array
            p.assignWeapon(getRandWeapon());
            // Send this player on a mission
            p.mission();
        }
    }

    // Utility methods to get a random player type and weapon
    public static String getRandPlayerType()
    {
        Random r = new Random();
        // Will return an integer between [0,2)
        int randInt = r.nextInt(playerType.length);
        // return the player stored at index 'randInt'
        return playerType[randInt];
    }

    public static String getRandWeapon()
    {
        Random r = new Random();
        // Will return an integer between [0,5)
        int randInt = r.nextInt(weapons.length);
        // Return the weapon stored at index 'randInt'
        return weapons[randInt];
    }
}

```

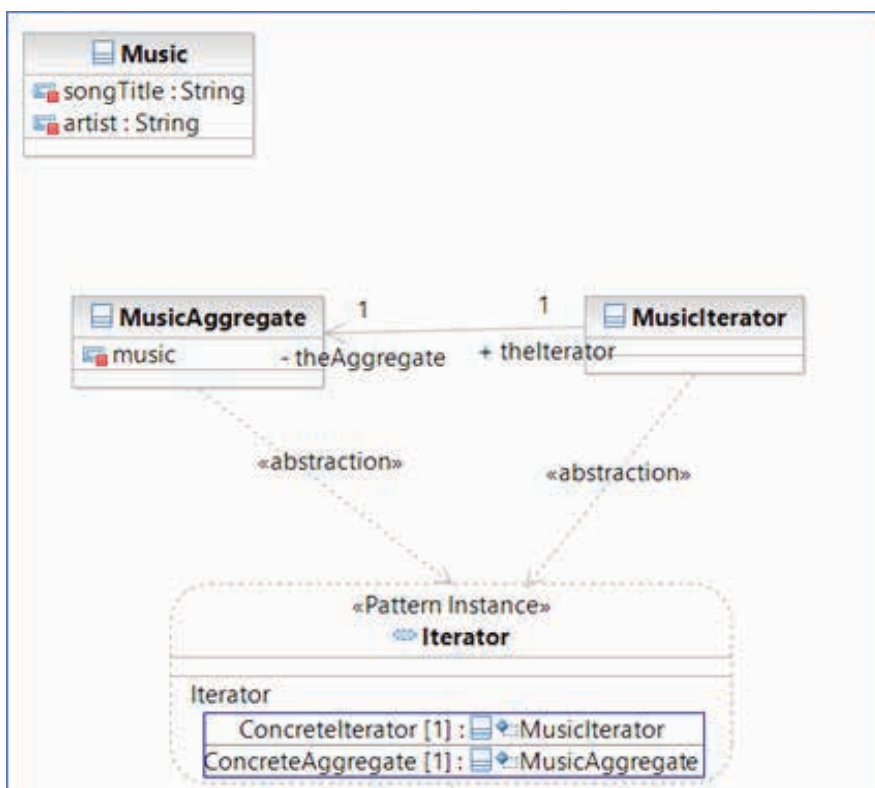
Rezultat izvršavanja:

```

Counter Terrorist Created
Counter Terrorist with weapon Maverick. Task is BOMB
Counter Terrorist with weapon Maverick. Task is BOMB
Counter Terrorist with weapon AK-47. Task is BOMB
Counter Terrorist with weapon AK-47. Task is BOMB
Counter Terrorist Created
Counter Terrorist with weapon Gut Knife. Task is BOMB
Counter Terrorist with weapon Maverick. Task is BOMB
Counter Terrorist with weapon Maverick. Task is BOMB
Counter Terrorist with weapon AK-47. Task is BOMB
Counter Terrorist with weapon AK-47. Task is BOMB
Counter Terrorist with weapon AK-47. Task is BOMB

```

Zadatak 5-1. Kreirati aplikaciju koja će omogućiti klijentu da lista nazive pesama i njihovih autora, redosledom koji on odabere.



Klasa *Music*:

```
public class Music {
    private String songTitle;
    private String artist;

    public String getSongTitle() {
        return songTitle;
    }
    public void setSongTitle(String songTitle) {
        this.songTitle = songTitle;
    }
    public String getArtist() {
        return artist;
    }
    public void setArtist(String artist) {
        this.artist = artist;
    }

    // dodati konstruktore
    public Music(String artist, String songTitle) {
        super();
        this.artist = artist;
        this.songTitle = songTitle;
    }
    public Music() {
        super();
    }
}
```

Klasa *MusicAggregate*:

```
import java.util.AbstractCollection;
import java.util.Iterator;

public class MusicAggregate extends AbstractCollection {
    //promeniti difoltni tip Object u listu: Music[] i kod getters and setters
    private Music[] music;

    public Music[] getMusic() {
        return music;
    }
    public void setMusic(Music[] music) {
        this.music = music;
    }
    transient int count;
    //prijavljuje gresku zbog toga sto nije podeseno u MusicIterator
    public Iterator iterator() {
        return new MusicIterator(music);
    }
    public int size() {
        throw new java.lang.UnsupportedOperationException();
    }
    //dodati parametarski konstruktor
    public MusicAggregate(Music[] music) {
        super();
        this.music = music;
    }
}
```

Klasa *MusicIterator*:

```
import java.util.Iterator;
//dodati <Music>
public class MusicIterator implements Iterator<Music> {

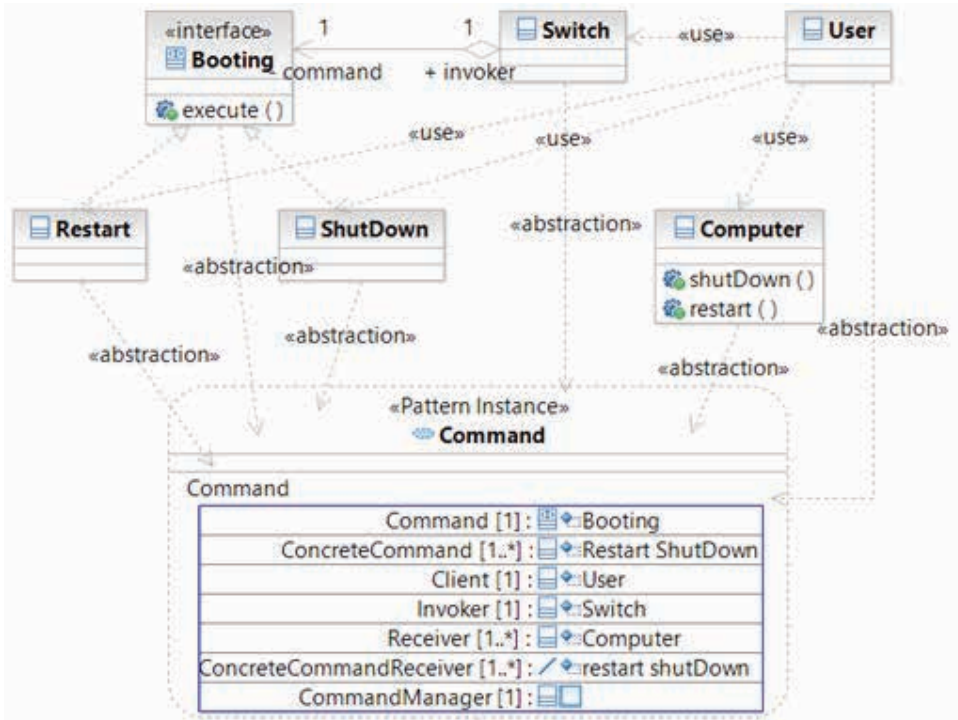
    protected transient int modCount;
    private int currentIndex;
    private Music[] music;

    public Music[] getMusic() {
        return music;
    }
    public void setMusic(Music[] music) {
        this.music = music;
    }
    public Music next() {
        return music[currentIndex++];
    }
    public boolean hasNext() {
        if(currentIndex < music.length && music[currentIndex]!=null){
            return true;
        }
        else return false;
    }
    public void remove() {
        currentIndex=0;
    }
    public MusicIterator(Music[] music) {
        super();
        this.music = music;
    }
}
```

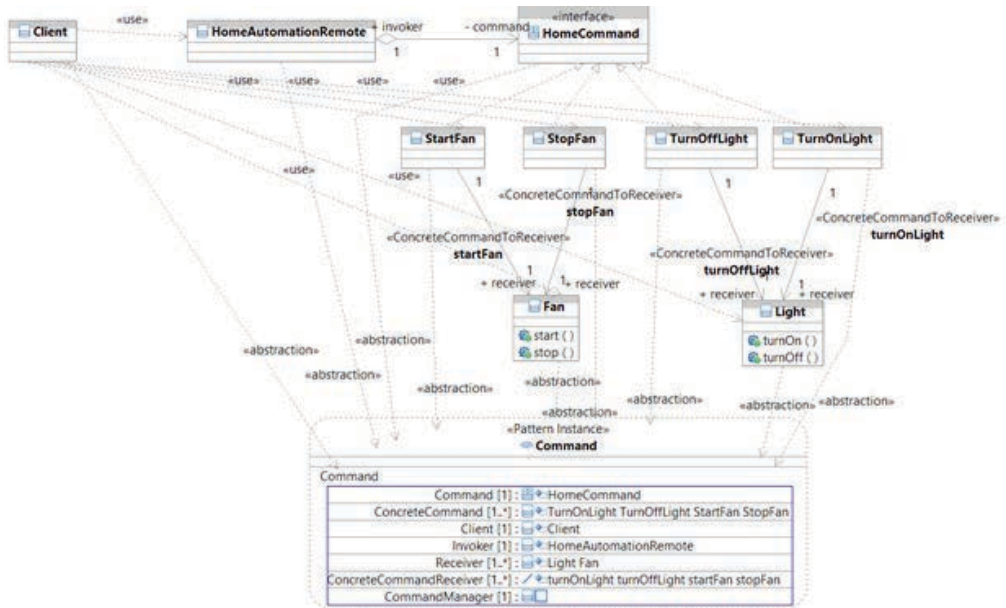
Klasa *Main*:

```
import java.util.Iterator;
public class Main {
    public static void main(String[] args) {
        Music[] music=new Music[2];
        music[0]=new Music("Swedish House Mafia ft The Weeknd", "Moth
        To A Flame");
        music[1]=new Music("James Vincent McMorrow, Rudimental",
        "Be Somebody");
        MusicAggregate list=new MusicAggregate(music);
        Iterator iterator=list.iterator();
        while(iterator.hasNext()){
            Music currentMusic=(Music) iterator.next();
            System.out.println(currentMusic.getSongTitle());
            System.out.println(currentMusic.getArtist());
        }
    }
}
```

Zadatak 5-2. Primeniti *Command* obrazac na komandu butovanje (*Booting*) računara.



Zadatak 5-3. Primeniti odgovarajući obrazac na sistem za daljinsko upravljanje svetlom i ventilacijom u kući.



Korak 1: Počnimo sa svetlom prijemnika i ventilatorom:

```

public class Light {
    public void turnOn(){
        System.out.println("Light is on.");
    }
    public void turnOff(){
        System.out.println("Light is off.");
    }
}

public class Fan {
    void start(){
        System.out.println("Fan started...");
    }
    void stop(){
        System.out.println("Fan stopped.");
    }
}

```

Korak 2: Interfejsi i konkretne implementacije komandi:

```

public interface HomeCommand {
    public void execute();
}

public class TurnOnLight implements HomeCommand {
    public Light receiver;
    public Light getReceiver() {
        return receiver;
    }
    public void setReceiver(Light receiver) {
        this.receiver = receiver;
    }
    public TurnOnLight(Light receiver) {
        this.receiver = receiver;
    }
    public void execute() {
        System.out.println("\nTurning on light.");
        receiver.turnOn();
    }
}

public class TurnOffLight implements HomeCommand {
    public Light receiver;
    public Light getReceiver() {
        return receiver;
    }
    public void setReceiver(Light receiver) {
        this.receiver = receiver;
    }
    public TurnOffLight(Light receiver) {
        this.receiver = receiver;
    }
    public void execute() {
        System.out.println("\nTurning off light.");
        receiver.turnOff();
    }
}

public class StartFan implements HomeCommand {
    public Fan receiver;
    public Fan getReceiver() {
        return receiver;
    }
    public void setReceiver(Fan receiver) {
        this.receiver = receiver;
    }
    public StartFan(Fan receiver) {
        this.receiver = receiver;
    }
    public void execute() {
        System.out.println("\nStarting Fan.");
        receiver.start();
    }
}

```

```
public class StopFan implements HomeCommand {
    public Fan receiver;
    public Fan getReceiver() {
        return receiver;
    }
    public void setReceiver(Fan receiver) {
        this.receiver = receiver;
    }
    public StopFan(Fan receiver) {
        this.receiver = receiver;
    }
    public void execute() {
        System.out.println("\nStopping Fan.");
        receiver.stop();
    }
}
```

Korak 3: Klasa *Invoker*:

```
public class HomeAutomationRemote {
    private HomeCommand command;
    public HomeCommand getCommand() {
        return command;
    }
    public void setCommand(HomeCommand command) {
        this.command = command;
    }
    public void buttonPressed() {
        command.execute();
    }
}
```

Korak 4: Implementacija jednog scenarija klijenta:

```
public class Client {
    public static void main(String[] args) {
        Light livingRoomLight = new Light(); //receiver 1
        Fan livingRoomFan= new Fan(); // receiver 2
        Light bedRoomLight = new Light(); //receiver 3
        Fan bedRoomFan = new Fan(); //receiver 4

        HomeAutomationRemote remote = new HomeAutomationRemote();
        //invoker

        remote.setCommand(new TurnOnLight(livingRoomLight));
        remote.buttonPressed();

        remote.setCommand(new TurnOffLight(bedRoomLight));
        remote.buttonPressed();

        remote.setCommand(new StartFan(livingRoomFan));
        remote.buttonPressed();

        remote.setCommand(new StopFan(livingRoomFan));
        remote.buttonPressed();
    }
}
```

```

        remote.setCommand(new StartFan(bedRoomFan));
        remote.buttonPressed();

        remote.setCommand(new StopFan(bedRoomFan));
        remote.buttonPressed();
    }
}

```

Output:

Turning on light.
Light is on.

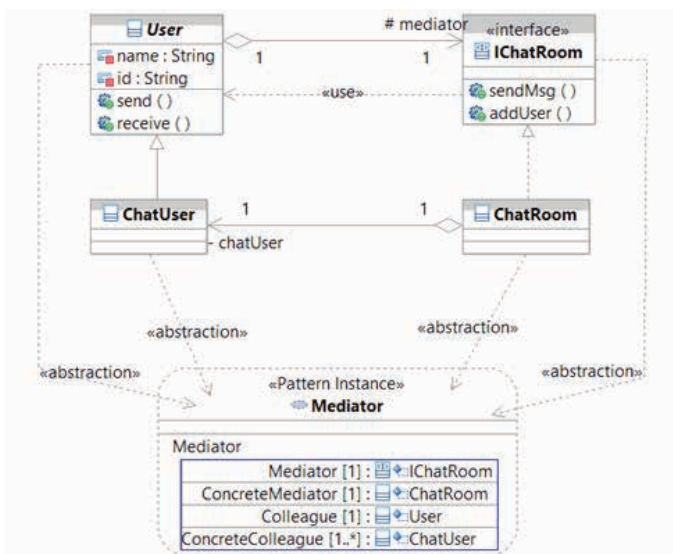
Turning off light.
Light is off.

Starting Fan.
Fan started...

Stopping Fan.
Fan stopped.

Starting Fan.
Fan started...

Stopping Fan.
Fan stopped.

Zadatak 5-4. Primeniti *Mediator* obrazac na aplikaciju za četovanje.

```
public abstract class User {
    protected IChatRoom mediator;
    private String id;
    private String name;

    //getters and setters
    public IChatRoom getMediator() {
        return mediator;
    }
    public void setMediator(IChatRoom mediator) {
        this.mediator = mediator;
    }
    public String getId() {
        return id;
    }
    public void setId(String id) {
        this.id = id;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
}

//Constructor from fields
public User(IChatRoom mediator, String id, String name) {
    super();
    this.id = id;
    this.mediator = mediator;
    this.name = name;
}

//abstract methods send and receive
public abstract void send(String userId, String msg);
public abstract void receive(String msg);
}

public class ChatUser extends User {
    public ChatUser(IChatRoom mediator, String id, String name) {
        super(mediator, id, name);
    }
    @Override
    public void receive(String msg) {
        System.out.println(this.getName()+" (Primljena poruka): "+msg);
    }
    @Override
    public void send(String userId, String msg) {
        System.out.println(this.getName()+" (Poslata poruka): "+msg);
        getMediator().sendMsg(msg, userId);
    }
}
```

```

public interface IChatRoom {
    public void sendMsg(String msg, String userId);
    public void addUser(User user);
    public void notify(User colleague);
}

import java.util.*;
public class ChatRoom implements IChatRoom {
    private ChatUser chatUser;
    public ChatUser getChatUser() {
        return chatUser;
    }
    public void setChatUser(ChatUser chatUser) {
        this.chatUser = chatUser;
    }
    private Map<String, User> usersMap=new HashMap<String, User>();
    public void sendMsg(String msg, String userId) {
        User u=usersMap.get(userId);
        u.receive(msg);
    }
    public void addUser(User user) {
        this.usersMap.put(user.getId(), user);
    }
    public void notify(User colleague) {
        if (colleague instanceof ChatUser) {

        }
    }
}

public class ChatRoomMain {
    public static void main(String[] args) {
        IChatRoom chatroom=new ChatRoom();
        User user1=new ChatUser(chatroom, "1", "Aleksa");
        User user2=new ChatUser(chatroom, "2", "Petar");
        User user3=new ChatUser(chatroom, "3", "Novi korisnik");
        chatroom.addUser(user1);
        chatroom.addUser(user2);
        chatroom.addUser(user3);
        user1.send("2", "Zdravo Petre");
        user2.send("1", "Hej Aleksa!");
        user3.send("2", "Zdravo Petre, ja sam novi korisnik!");
    }
}

```

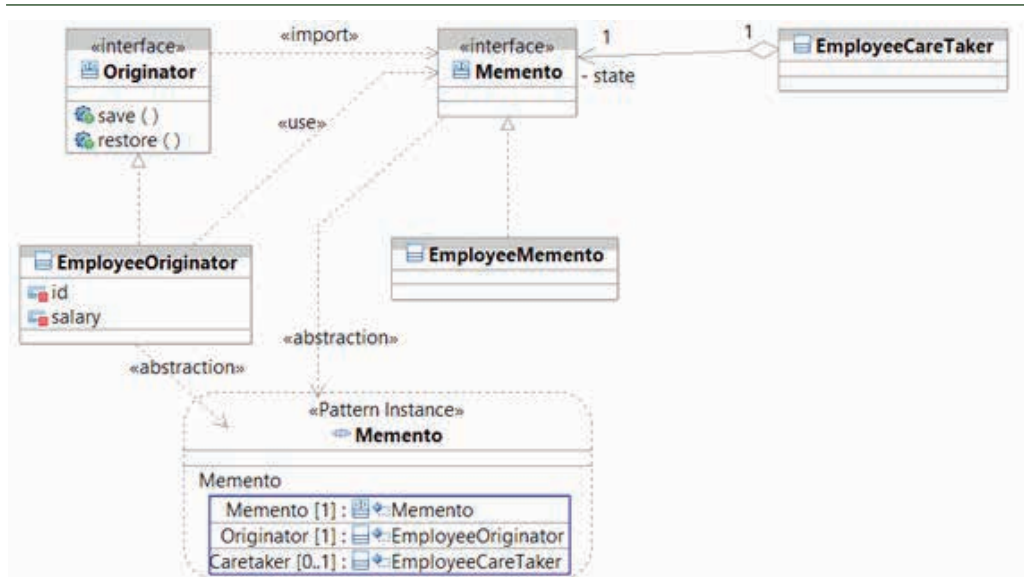
Izlaz:

Aleksa (Poslata poruka): Zdravo Petre
 Petar (Primljena poruka): Zdravo Petre

Petar (Poslata poruka): Hej Aleksa!
 Aleksa (Primljena poruka): Hej Aleksa!

Novi korisnik (Poslata poruka): Zdravo Petre, ja sam novi korisnik!
 Petar (Primljena poruka): Zdravo Petre, ja sam novi korisnik!

Zadatak 5-5. Primeniti Memento obrazac na sledeći scenario:



```
public interface Memento {
    float getSalary();
}
```

```
public class EmployeeMemento implements Memento {
    private float salary;
    public EmployeeMemento(float salary) {
        super();
        this.salary = salary;
    }
    public float getSalary() {
        return salary;
    }
    public void setSalary(float salary) {
        this.salary = salary;
    }
}
```

```
public interface Originator {
    public Memento save();
    public void restore(Memento memento);
    public void setSalary(float salary);
}
```

```

public class EmployeeOriginator implements Originator {
    private int id;
    private float salary;
    public EmployeeOriginator(int id, float salary) {
        super();
        this.id = id;
        this.salary = salary;
    }
    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
    public float getSalary() {
        return salary;
    }
    public void setSalary(float salary) {
        this.salary = salary;
    }
    public Memento save() {
        return new EmployeeMemento(this.salary);
    }
    public void restore(Memento memento) {
        this.salary=memento.getSalary();
    }
    @Override
    public String toString() {
        return "EmployeeOriginator [id = "+id+", salary = "+salary+"]";
    }
}

import java.util.Stack;
public class EmployeeCareTaker {
    private Memento state;
    public Memento getState() {
        return state;
    }
    public void setState(Memento state) {
        this.state = state;
    }
    public Stack<Memento> stack=new Stack<Memento>();
    public void addMemento(Memento m) {
        stack.push(m);
    }
    public Memento getMemento(){
        return stack.pop();
    }
}

```

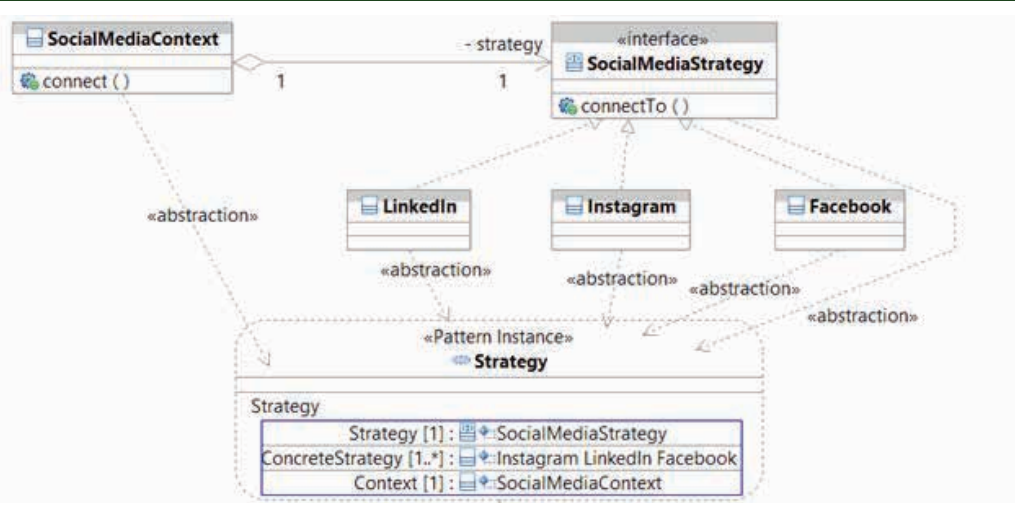
```
public class Client {  
    public static void main(String[] args) {  
        EmployeeCareTaker caretaker = new EmployeeCareTaker();  
        Originator originator=new EmployeeOriginator(25, 82000);  
  
        originator.setSalary(85000);  
        System.out.println(originator);  
        caretaker.addMemento(originator.save());  
  
        originator.setSalary(95000);  
        caretaker.addMemento(originator.save());  
  
        originator.setSalary(100000);  
        System.out.println(originator);  
        originator.restore(caretaker.getMemento());  
        System.out.println(originator);  
    }  
}
```

Izlaz:

```
EmployeeOriginator [id = 25, salary = 85000.0]  
EmployeeOriginator [id = 25, salary = 100000.0]  
EmployeeOriginator [id = 25, salary = 95000.0]
```

Zadatak 5-6. Kreirati aplikaciju koja omogućava korisniku da se poveže sa svojim prijateljima putem SM (Social Media) platformi i to preko Instagram-a, Facebook-a i LinkedIn-a.

Korisnik će proslediti ime prijatelja i željenu platformu, a aplikacija će ga transparentno povezati. U slučaju da korisnik bude želeo da proširi broj SM platformi, dodati ih bez degradiranja početnog dizajna aplikacije.



```
public interface SocialMediaStrategy {
    public void connectTo(String friendName);
}

public class Instagram implements SocialMediaStrategy {
    public void connectTo(String friendName) {
        System.out.println("Connecting with "+friendName+" via Instagram!");
    }
}

public class Facebook implements SocialMediaStrategy {
    public void connectTo(String friendName) {
        System.out.println("Connecting with "+friendName+" via Facebook!");
    }
}

public class LinkedIn implements SocialMediaStrategy {
    public void connectTo(String friendName) {
        System.out.println("Connecting with "+friendName+" via LinkedIn!");
    }
}
```

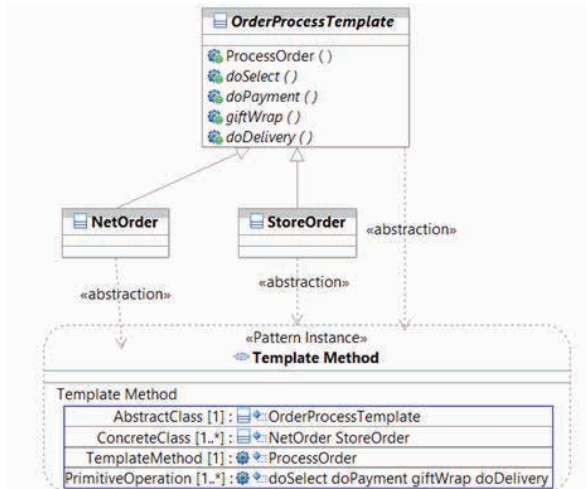
```
public class SocialMediaContext {
    private SocialMediaStrategy strategy;
    public SocialMediaStrategy getStrategy() {
        return strategy;
    }
    public void setStrategy(SocialMediaStrategy strategy) {
        this.strategy = strategy;
    }
    public void connect(String name){
        strategy.connectTo(name);
    }
}

public class Client {
    public static void main(String[] args) {
        SocialMediaContext context=new SocialMediaContext();
        context.setStrategy(new Facebook());
        context.connect("Marina");
        context.setStrategy(new LinkedIn());
        context.connect("Bojana");
        context.setStrategy(new Instagram());
        context.connect("Marina");
    }
}
```

Izlaz:

```
Connecting with Marina via Facebook!
Connecting with Bojana via LinkedIn!
Connecting with Marina via Instagram!
```

Zadatak 5-7. Primeniti *Template method* obrazac na različite načine poručivanja robe, a koji prate istu proceduru poručivanja.



```

public abstract class OrderProcessTemplate {
    public boolean isGift;
    public final void processOrder(boolean isGift) {
        doSelect();
        doPayment();
        if(isGift){
            giftWrap();
        }
        doDelivery();
    }

    public abstract void doSelect();
    public abstract void doPayment();
    public final void giftWrap(){
        try {
            System.out.println("Gift wrap successful.");
        }
        catch (Exception e)
        {
            System.out.println("Gift wrap unsuccessful.");
        }
    }
    public abstract void doDelivery();
}

```

```

public class NetOrder extends OrderProcessTemplate {
    public void doSelect() {
        System.out.println("Item added to online shopping cart.");
    }
}

```

```
        System.out.println("Get gift wrap preference.");
        System.out.println("Get delivery address.");
    }
    public void doPayment() {
        System.out.println("Online payment through NetBanking, card or
        PayPal.");
    }
    public void doDelivery() {
        System.out.println("Ship the item through post to delivery address.");
    }
}

public class StoreOrder extends OrderProcessTemplate {
    public void doSelect() {
        System.out.println("Product selection: Customer chooses the
        item from shelf.");
    }
    public void doPayment() {
        System.out.println("Payment: Customer pays at counter through
        cash/POS.");
    }
    public void doDelivery() {
        System.out.println("Delivery: Item delivered to in delivery
        counter.");
    }
}

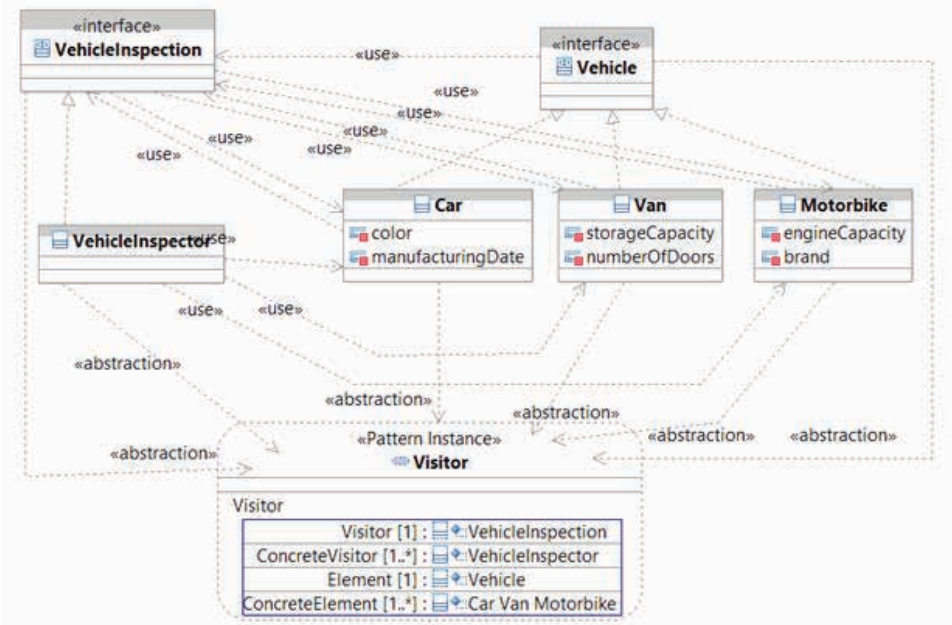
public class Client {
    public static void main(String[] args) {
        OrderProcessTemplate netOrder= new NetOrder();
        netOrder.processOrder(true);
        System.out.println();
        OrderProcessTemplate storeOrder=new StoreOrder();
        storeOrder.processOrder(true);
    }
}
```

Rezultat:

Item added to online shopping cart.
Get gift wrap preference.
Get delivery address.
Online payment through NetBanking, card or PayPal.
Gift wrap successful.
Ship the item through post to delivery address.

Product selection: Customer chooses the item from shelf.
Payment: Customer pays at counter through cash/POS.
Gift wrap successful.
Delivery: Item delivered to in delivery counter.

Zadatak 5-8. Primeniti obrazac na aplikaciju za servisiranje vozila. Svako vozilo (*Car*, *Van*, *Motorbike*) je posećeno od strane inspektora (*VehicleInspector*) koji vrši inspekciju (*VehicleInspection*) i izračunava servisnu naknadu za svako vozilo i ukupne troškove servisiranja.



```
public interface VehicleInspection {
    public int visit(Car car);
    public int visit(Van van);
    public int visit(Motorbike motorbike);
}

public interface Vehicle {
    public int accept(VehicleInspection vehicleInspection);
}

public class Car implements Vehicle {
    private String color;
    private int manufactureDate;

    public int accept(VehicleInspection vehicleInspection) {
        return vehicleInspection.visit(this);
    }

    //getters and setters
    public String getColor() {
        return color;
    }
}
```

```
        public void setColor(String color) {
            this.color = color;
        }
        public int getManufactureDate() {
            return manufactureDate;
        }
        public void setManufactureDate(int manufactureDate) {
            this.manufactureDate = manufactureDate;
        }
        //creating a constructor of the class
        public Car(String color, int manufactureDate) {
            super();
            this.color = color;
            this.manufactureDate = manufactureDate;
        }
        public Car() {
            super();
        }
    }

    public class Motorbike implements Vehicle {
        private int engineCapacity;
        private String brand;

        public int accept(VehicleInspection vehicleInspection) {
            return vehicleInspection.visit(this);
        }

        public int getEngineCapacity() {
            return engineCapacity;
        }

        public void setEngineCapacity(int engineCapacity) {
            this.engineCapacity = engineCapacity;
        }

        public String getBrand() {
            return brand;
        }

        public void setBrand(String brand) {
            this.brand = brand;
        }

        public Motorbike() {
            super();
        }

        public Motorbike(String brand, int engineCapacity) {
            super();
            this.brand = brand;
            this.engineCapacity = engineCapacity;
        }
    }
}
```

```
public class Van implements Vehicle {
    private int storageCapacity;
    private int numberOfDoors;

    public int accept(VehicleInspection vehicleInspection) {
        return vehicleInspection.visit(this);
    }

    public int getStorageCapacity() {
        return storageCapacity;
    }

    public void setStorageCapacity(int storageCapacity) {
        this.storageCapacity = storageCapacity;
    }

    public int getNumberOfDoors() {
        return numberOfDoors;
    }

    public void setNumberOfDoors(int numberOfDoors) {
        this.numberOfDoors = numberOfDoors;
    }

    public Van() {
        super();
    }

    public Van(int numberOfDoors, int storageCapacity) {
        super();
        this.numberOfDoors = numberOfDoors;
        this.storageCapacity = storageCapacity;
    }
}

public class VehicleInspector implements VehicleInspection {
    public int visit(Car car) {
        int serviceCharge=0;
        if(car.getColor()=="Black") {
            serviceCharge+=100;
        } else {
            serviceCharge+=50;
        }
        System.out.println("Service Charge for Car: "+serviceCharge);
        return serviceCharge;
    }

    public int visit(Van van) {
        int serviceCharge=0;
        if (van.getNumberOfDoors()>4){
            serviceCharge+=500;
        }else {
            serviceCharge+=100;
        }
    }
}
```

```
        System.out.println("Service Charge for Van: "+serviceCharge);
        return serviceCharge;
    }

    public int visit(Motorbike motorbike) {
        int serviceCharge=0;
        if(motorbike.getEngineCapacity()>=200){
            serviceCharge+=200;
        }else{
            serviceCharge+=50;
        }
        System.out.println("Service Charge for motorbike is " +
            serviceCharge);
        return serviceCharge;
    }
}

public class VehicleService {
    private static int calculateTotal(Vehicle[] vehicles){
        VehicleInspection inspector = new VehicleInspector();
        int total=0;
        for (Vehicle vehicle:vehicles){
            total = total + vehicle.accept(inspector);
        }
        return total;
    }

    public static void main(String[] args) {
        Vehicle[] vehicles=new Vehicle[]{
            new Car("Black", 2020),
            new Van(5000,6),
            new Motorbike("TVS",100)
        };
        int totalCost=calculateTotal(vehicles);
        System.out.println("Total Service Sharge: "+totalCost);
    }
}
```

Izlaz:

```
Service Charge for Car: 100
Service Charge for Van: 500
Service Charge for motorbike is 50
Total Service Sharge: 650
```



```
        public Handler() {
            super();
        }
    }

    public class SpamHandler extends Handler {
        public void setNextChain(Handler nextChain){
            this.chain=nextChain;
        }

        public void forwardMail(Mail mailObj) {
            if(mailObj.getSubject().equalsIgnoreCase(Handler.SPAM_MAIL)){
                System.out.println("Mail Deleted.");
            }
            else {
                this.chain.forwardMail(mailObj);
            }
        }

        public SpamHandler(SpamHandler aSuccessor) {
            super(aSuccessor);
        }

        public SpamHandler() {
            super();
        }
    }

    public class FanHandler extends Handler {
        @Override
        public void setNextChain(Handler nextChain) {
            this.chain=nextChain;
        }

        public void forwardMail(Mail mailObj) {
            if (mailObj.getSubject().equalsIgnoreCase(Handler.FAN_MAIL)){
                System.out.println("Fowarding Mail to CEO");
            }
            else {
                this.chain.forwardMail(mailObj);
            }
        }

        public FanHandler(FanHandler aSuccessor) {
            super(aSuccessor);
        }

        public FanHandler() {
            super();
        }
    }

    public class ComplaintHandler extends Handler {
        @Override
        public void forwardMail(Mail mailObj) {
            if(mailObj.getSubject().equalsIgnoreCase(Handler.COMPLAINT_MAIL)){
```

```

        System.out.println("Fowarding Email to Legal Department");
    }else {
        this.chain.forwardMail(mailObj);
    }
}
@Override
public void setNextChain(Handler nextChain) {
    this.chain=nextChain;
}
public ComplaintHandler(ComplaintHandler aSuccessor) {
    super(aSuccessor);
}
public ComplaintHandler() {
    super();
}
}

public class MailChain {
    private Handler chain;
    public MailChain(){
        this.chain=new SpamHandler();
        Handler fanHandlerObj=new FanHandler();
        Handler complHandlerObj=new ComplaintHandler();
        chain.setNextChain(fanHandlerObj);
        fanHandlerObj.setNextChain(complHandlerObj);
    }
    public static void main(String[] args) {
        MailChain mailChainObj=new MailChain();
        mailChainObj.chain.forwardMail(new Mail("SPAM_MAIL"));
        mailChainObj.chain.forwardMail(new Mail("FAN_MAIL"));
        mailChainObj.chain.forwardMail(new Mail("COMPLAINT_MAIL"));
    }
}

```

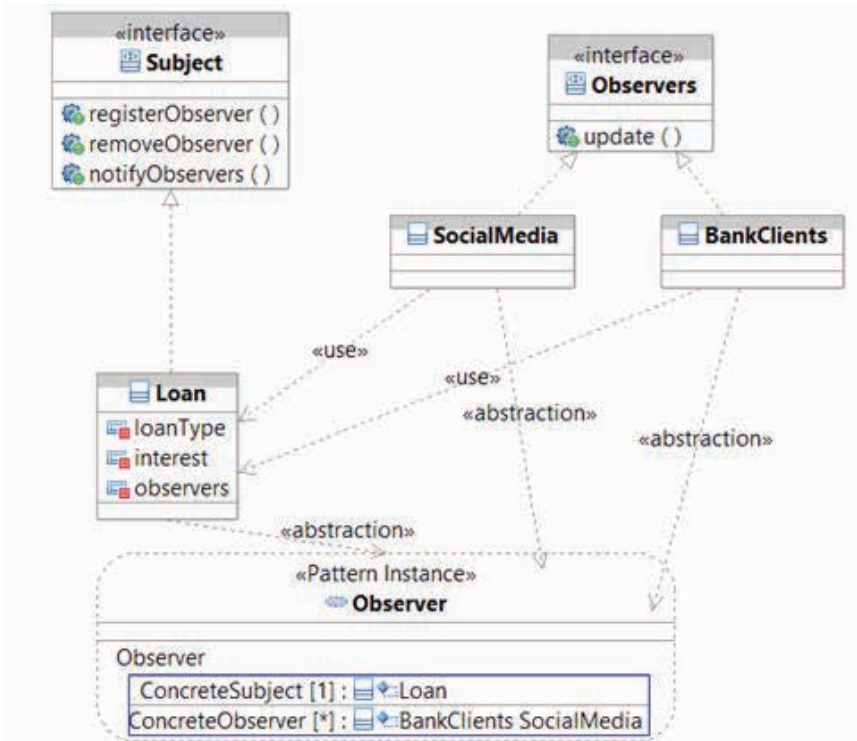
Izlaz:

```

Mail Deleted.
Fowarding Mail to CEO
Fowarding Email to Legal Department

```

Zadatak 5-10. Kreirati aplikaciju za banku koja će redovno obavještavati svoje klijente (*BankClients*) i društvene medije (*SocialMedia*) o promeni kamate (*interest*) na pozajmljena sredstva.



```

public interface Subject {
    public void registerObserver(Observers observer);
    public void removeObserver(Observers observer);
    public void notifyObservers();
}

public interface Observers {
    public void update(float interest);
}

import java.util.ArrayList;

public class Loan implements Subject {
    private ArrayList<Observers> observers=new ArrayList<Observers>();
    private String loanType;
    private float interest;

    //constructors
    public Loan() {

```

```

        super();
    }
    public Loan(String loanType, float interest) {
        super();
        this.interest = interest;
        this.loanType = loanType;
    }

    //getters and setters
    public String getLoanType() {
        return loanType;
    }
    public void setLoanType(String loanType) {
        this.loanType = loanType;
    }
    public float getInterest() {
        return interest;
    }
    public void setInterest(float interest) {
        this.interest = interest;
        notifyObservers();
    }

    @Override
    public void notifyObservers() {
        System.out.println("\nNotifying Observers on change in Loan
        interest rate.");
        for (Observers ob: observers){
            ob.update(this.interest);
        }
    }
    @Override
    public void registerObserver(Observers observer) {
        observers.add(observer);
    }
    @Override
    public void removeObserver(Observers observer) {
        observers.remove(observer);
    }
}

public class BankClients implements Observers {
    @Override
    public void update(float interest) {
        System.out.println("Bank Clients: Interest Rate has been
        updated. The new Rate is: "+interest+"!");
    }
}

public class SocialMedia implements Observers {
    @Override
    public void update(float interest) {

```

```
        System.out.println("Social Media: Interest Rate updated, new  
Rate is: "+interest+"!");  
    }  
}  
  
public class ObserverTest {  
    public static void main(String[] args) {  
        BankClients clients = new BankClients();  
        SocialMedia onlineMedia = new SocialMedia();  
  
        Loan personalLoan = new Loan("Personal Loan",12.5f);  
        personalLoan.registerObserver(clients);  
        personalLoan.registerObserver(onlineMedia);  
        personalLoan.setInterest(7.5f);  
        personalLoan.setInterest(3.5f);  
    }  
}
```

Izlaz:

Notifying Observers on change in Loan interest rate.

Bank Clients: Interest Rate has been updated. The new Rate is: 7.5!

Social Media: Interest Rate updated, new Rate is: 7.5!

Notifying Observers on change in Loan interest rate.

Bank Clients: Interest Rate has been updated. The new Rate is: 3.5!

Social Media: Interest Rate updated, new Rate is: 3.5!

CIP - Каталогизација у публикацији
Народна библиотека Србије, Београд

004.41(075.8)

ЊЕГУШ, Ангелина, 1971-

Obrasci projektovanja softvera : software design patterns / Angelina Njeguš. - 1. izd. - Beograd : Univerzitet Singidunum, 2023 (Beograd : Caligraph). - VIII, 289 str. : ilustr. ; 24 cm

Na vrhu nasl. str.: Tehnički fakultet. - Tiraž 1.000. - Bibliografija: str. 226-227.

ISBN 978-86-7912-796-9

а) Софтвер -- Пројектовање -- Задачи

COBISS.SR-ID 121126409

© 2023

All rights reserved. No part of this work covered by the copyright herein may be reproduced, transmitted, stored or used in any form or by any means graphic, electronic, or mechanical, including but not limited to photocopying, recording, scanning, digitizing, taping, Web distribution, information networks, or information storage and retrieval systems, without the prior written permission of the publisher.



OBRASCI PROJEKTOVANJA SOFTVERA

Angelina Njeguš

