



# SE201 - UVOD U SOFTVERSKO INŽENJERSTVO

## Softverski procesi

Lekcija 02

PRIRUČNIK ZA STUDENTE

# SE201 - UVOD U SOFTVERSKO INŽENJERSTVO

## Lekcija 02

### ***SOFTVERSKI PROCESI***

- ▼ Softverski procesi
- ▼ Poglavlje 1: Softverski proces
- ▼ Poglavlje 2: Modeli softverskog procesa
- ▼ Poglavlje 3: Osnovne aktivnosti procesa
- ▼ Poglavlje 4: Razvoj softvera u uslovima stalnih promena
- ▼ Poglavlje 5: Rational ujedinjeni proces (RUP)
- ▼ Poglavlje 6: Specijalizovani modeli
- ▼ Poglavlje 7: Izbor modela procesa razvoja softvera
- ▼ Poglavlje 8: Pokazna vežba
- ▼ Poglavlje 9: Individualna vežba
- ▼ Zaključak

Copyright © 2017 – UNIVERZITET METROPOLITAN, Beograd. Sva prava zadržana. Bez prethodne pismene dozvole od strane Univerziteta METROPOLITAN zabranjena je reprodukcija, transfer, distribucija ili memorisanje nekog dela ili čitavih sadržaja ovog dokumenta., kopiranjem, snimanjem, elektronskim putem, skeniranjem ili na bilo koji drugi način.

Copyright © 2017 BELGRADE METROPOLITAN UNIVERSITY. All rights reserved. No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise, without the prior written permission of Belgrade Metropolitan University.

## ✓ Uvod

### UVOD

*Softverski proces čini skup aktivnosti u kojima se razvija softver.*

Cilj ove lekcije da vas uvede u pojam „softverski proces“, tj. proces razvoja softvera i skup aktivnosti koje čine taj proces.

**Ključna pitanja:**

- Objasniti koncept softverskih proces i modela softverskih proces.
- Koja su tri opšta modela softverskog procesa i kada se oni koriste?
- Koje su osnovne aktivnosti inženjerstva utvrđivanja zahteva za softverom, za testiranjem i evolucijom?
- Zašto procesi treba d abudu tako organizovani da mogu da se prilagođavaju promenama zahteva i projektnog rešenja softvera?
- Kako Rational Unified Proces integriše dobru inženjersku praksu radi kreiranja prilagodljivih softverskih procesa

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

## ✓ Poglavlje 1

### Softverski proces

## OSNOVNE AKTIVNOSTI SOFTVERSKOG PROCESA

*Softverski proces je skup povezanih aktivnosti koji vodi proizvodnji softverskog proizvoda.*

Softverski proces je skup povezanih aktivnosti koji vodi proizvodnji softverskog proizvoda. Ove aktivnosti mogu da dovedu do razvoja potpuno novog softvera u nekom od standardnih programskih jezika (npr. Java ili C), ili do usavršavanja nekog postojećeg softvera, što je najčešći slučaj kod poslovnih aplikacija. Stvarni softverski procesi su mešavina sekvenci tehničkih, kolaborativnih i upravljačkih aktivnosti sa ukupnim ciljem da specificiraju, projektuju, implementiraju i testiraju neki softverski sistem. Softverski inženjeri u svom radu koriste različite softverske alate radi rada sa različitim tipovima dokumenata i uređivanja velike količine detaljnih informacija koji se stvara u projektu razvoja nekog velikog softverskog sistema.

Najčešće se softverski procesi sadrže sledeće četiri osnovne aktivnosti koje čine osnovu softverskog inženjerstva:

1. **Specifikacija softvera:** Definiše funkcionalnost softvera i ograničenja na rad softvera.
2. **Projektovanje (dizajn) i razvoj (implementacija) softvera:** Definiše kako softver ostvaruje svoju specifikaciju i kako se on pravi.
3. **Provera (validacija) softvera:** Softver se proverava da bi se utvrdilo da li zadovoljava potrebe i očekivanja korisnika.
4. **Evolucija softvera:** Softver mora da ima svoju evoluciju (stalni razvoj i prilagođavanje) da bi zadovoljio nove zahteve korisnika.

Ove osnovne aktivnosti mogu da imaju svoje **pod-aktivnosti** (npr. provera zahteva, dizajn arhitekture, jedinični testovi i dr.). Postoje i aktivnosti koje podržavaju osnovne aktivnosti procesa, kao što je izrada tehničke dokumentacije, i definisanje konfiguracije softvera.

Pored navođenja aktivnosti, jedan softverski proces se opisuje i navođenjem drugih faktora, kao što su:

1. *Proizvodi koji su rezultat aktivnosti procesa.*
2. *Uloge (eng. Roles) koje definišu odgovornosti ljudi koji učestvuju u procesu.*
3. *Uslovi koje neka aktivnost mora da zadovolji pre nego što počne sa radom (preduslovi), a da bi završila sa radom (izlazni uslovi).*

## OSNOVNI TIPOVI SOFTVERSKOG PROCESA

*Softverski procesi se mogu podeliti na 1) procese vođene planom, i na 2) agilne procese.*

Softverski procesi su složeni procesi, kao i svi intelektualni i kreativni procesi u kojima učestvuju ljudi koji donose odluke. Zbog toga, oni zavise od ljudi koji rade u organizacijama i odražavaju specifične karakteristike tih organizacija. Zato se softverski procesi razliku od organizacije do organizacije.

Na procese utiče i tip softvera. Ako se radi o softveru koji upravlja tzv. kritičnim sistemima, onda on mora da sledi vrlo formalizovanu proceduru. S druge strane, kako se poslovne aplikacije razvijaju u uslovima promenljivih zahteva korisnika, to su procesi za njihov razvoj manje formalizovani, te se lakše prilagođavaju promenama da bi bili efektivniji. Zbog toga, softverski procesi mogu podeliti i na:

1. procese vođene planom, i na
2. agilne procese.

Procesi vođeni planom su procesi kod kojih su sve aktivnosti unapred planirane i napredak u razvoju softvera se određuje stepenom ostvarivanja tog plana.

Agilni procesi su procesi kod kojih se planiranje radi postupno (inkrementalno) kako bi se procesi lakše prilagodili promenljivim zahtevima korisnika.

Obe vrste procesa se koriste, zavisno od vrste softvera, a mogu se i kombinovati. Izazov za softverske inženjere je da primene procese koji najbolje odgovaraju uslovima njihove organizacije, jer idealnih (i opšte prihvaćenih procesa) nema.

Procesi se mogu poboljšati primenom odgovarajućih *standarda*, pri čemu se smanjuju specifične razlike između organizacija. Ovo dovodi da smanjivanja vremena obuke, bolje komunikacije i povećanje ekonomičnosti automatizacije procesa. Standardizacija je važan prvi korak uvođenja novih metoda softverskog inženjerstva i tehnika, i dobra praksa softverskog inženjerstva.

## RAZVOJ SOFTVERA (VIDEO)

*Georgia Tech video o razvoju softvera*

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

## VIDEO PREDAVANJE ZA OBJEKAT "SOFTVERSKI PROCES"

*Trajanje video snimka: 11min 19sek*

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

## ▼ Poglavlje 2

# Modeli softverskog procesa

## TRI OPŠTA MODEL A SOFTVERSKIH PROCESA

*Model softverskog procesa je uprošćeno predstavljanje nekog softverskog procesa.*

Model softverskog procesa je uprošćeno predstavljanje nekog softverskog procesa. Svaki model procesa predstavlja neki proces iz neke posebne perspektive, te zbog toga, obezbeđuje samo delimične informacije o procesu, tj. informacije važne za tu perspektivu (npr. funkciju softvera). Zato se najpre daju vrlo uopšteni modeli softverskih procesa (tzv. „paradigme procesa“) koji se prikazuju iz perspektive arhitekture procesa. Oni prikazuju tok procesa sa aktivnostima, ali ne prikazuju i detalje o svakoj aktivnosti. Zbog toga, **ovi opšti modeli predstavljaju samo apstrakcije procesa (uprošćeni prikaz) razvoja softvera**. Njihova namena je da ukažu na različite pristupe u razvoju softvera. Oni su samo osnova za razvoj detaljnijih modela procesa.

Najčešće se navode sledeća tri opšta modela softverskih procesa:

1. Model vodopada (engl., **the waterfall model**): On prikazuje osnovne aktivnosti procesa: specifikacija, razvoj, provera (validacija) i evolucija. Ove aktivnosti su prikazane kao posebne faze procesa, kao što su: specifikacija zahteva, projektovanje (dizajn) softvera, primena (implementacija), testiranje i dr. koje se jedna za drugom realizuju.
1. Inkrementalni (postepeni) razvoj: On povezuje aktivnosti specifikacije, razvoja, i provere, kao niz serija verzija (inkremenata) softvera, pri čemu svaka verzija dodaje određenu funkcionalnost na prethodnu verziju.
2. Softversko inženjerstvo zasnovano na višestrukoj upotrebljivosti (eng. **Reuse-oriented software engineering**): Ovaj pristup se oslanja na korišćenje komponenata softvera koje se mogu višestruko koristiti. Ovim procesom se integrišu postojeće softverske komponente u sistem, umesto da se razvijaju nove.

Ovi modeli se često kombinuju jer se mogu međusobno dopunjavati, što je važno naročito kod razvoja velikih softverskih sistema. Za velike sisteme, moraju biti poznati zahtevi korisnika, definisani na formalni način i na osnovu toga, da se postavi arhitektura softvera (primena modela vodopada). Međutim, manji, podsistemi, se mogu razvijati i na drugačiji način. Neki, dobro definisani delovi, mogu se razvijati primenom modela vodopada. Drugi, koje je teško unapred precizno definisati, kao što je na primer korisnički interfejs, razvijaju se na inkrementalni način.

# SOFTVERSKI PROCESI (VIDEO)

## Model softverskog procesa - video klip 1

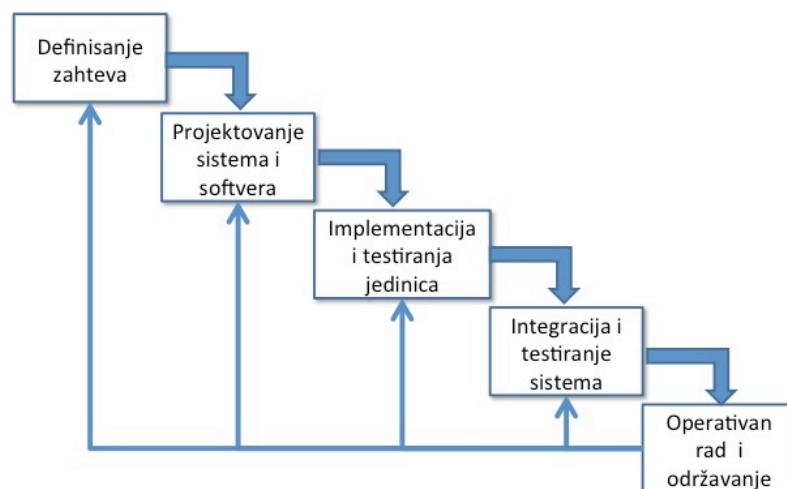
**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

### ▼ 2.1 Model vodopada

#### FAZE RAZVOJA U MODELU VODOPADA

*Model vodopada je planski vođeni proces jer se ceo proces mora planirati i odrediti termini za sve aktivnosti procesa, pre nego što počne njegovo izvršenje*

Model vodopada je primer planski vođenog procesa jer se ceo proces mora planirati i odrediti termini za sve aktivnosti procesa, pre nego što počne njegovo izvršenje (slika 1.). Glavne faze modela vodopada su direktno povezane sa osnovnim aktivnostima razvoja softvera:



Slika 2.1.1 Model vodopada [1.2]

1. Analiza i definisanje zahteva: Definišu se servisi sistema, ograničenja i ciljevi, definisani uz konsultaciju sa korisnicima sistema. Detaljnije se opisuju kao sistemska specifikacija.
2. Projektovanje sistema i softvera : Proces projektovanja sistema raspoređuje zahteve svim komponentama sistema i uspostavljanje celokupne arhitekture sistema, kao i utvrđivanje i opisivanje osnovnih apstrakcija softverskog sistema i njihove međuzavisnosti (relacije).
3. Implementacija i testiranje jedinica: U ovoj fazi je projektno rešenje softvera realizovano skupom programa ili programskih jedinica. Testiranje jedinica utvrđuje da li svaka jedinica ostvaruje svoju planiranu funkciju.

4. **Integracija i testiranje sistema:** Sve programske jedinice u ovoj fazi se integrišu u sistem i testiraju se kao kompletan sistem radi provere da li softver zadovoljava postavljene zahteve, tj. ostvaruje svoje funkcije i performanse. Posle ovog testiranja, softverski sistem se isporučuje kupcu.

5. **Operativni rad i održavanje:** Ovo je obično najduža faza životnog ciklusa softvera. Sistem je instaliran i pušten u operativni rad, tj. u upotrebu. Održavanje obuhvata ispravljanje grešaka koje nisu otkrivene u ranijim fazama životnog ciklusa, poboljšavanja primene programskih jedinica i poboljšanje usluga softverskog sistema u skladu sa novopostavljenim zahtevima.

## KADA KORISTITI MODEL VODOPADA?

*Model vodopada se koristi samo u slučajevima kada su svi zahtevi dobro definisani, jasni i stabilni.*

Na kraju svake faze dobija se jedan ili više dokumenata koje neko mora da zvanično odobri, što je uslov za početak sledeće faze životnog ciklusa softverskog sistema. U praksi, radi ubrzanja procesa (što je uvek važan zahtev), faze se delimično preklapaju, te se specifikacija vrši i u toku kajnijih faza. Često se u fazi kodiranja vrše dorade projektnog rešenja softvera. Softverski proces nije linearan, već ima povratne sprege između faza procesa. Zbog toga, se vrši i naknadna dorada već urađenih dokumenata.

Kako proizvodnja i odobravanje dokumenata košta, to i svaka iteracija (vraćanje posla na neku od prethodnih faza) košta, jer zahteva ponekad, i značajan dodatni rad. Zbog toga, obično se posle nekoliko iteracija vrši „zamrzavanje“ urađenog posla na razvoju, kao na primer, izrada specifikacije, i kreće se na narednu fazu procesa razvoja softvera. Preostali problemi se ostavljaju za kasnije rešavanje, ili se ignorišu ili se programski naknadno rešavaju. Prevremeno zamrzavanje zahteva može dovesti do toga da sistem ne radi ono što korisnik od njega očekuje. Takođe, to može da dovede do loše strukturisanog sistema, jer se problemi rešavaju raznim ad-hoc programerskim korekcijama.

Model vodopada je u saglasnosti sa drugim inženjerskim modelima procesa i sa praksom da se posle svake faze dobija odgovarajuća dokumentacija. Ovo čini proces vidljivim od strane menadžera, jer oni mogu da ga nadziru i kontrolišu da li je u skladu sa planom. Međutim, nedostatak ovog pristupa je u nefleksibilosti procesa, jer se teško prilagođava promenama zahteva korisnika, što se u praksi često dešava.

Po pravilu, model vodopada se koristi samo u slučajevima kada su svi zahtevi dobro definisani, jasni i stabilni (ne menjaju se za vreme razvoja softvera). I pored nedostataka, model vodopada je i dalje u čestoj upotrebi, jer se dobro uklapa modele upravljanja projektima.

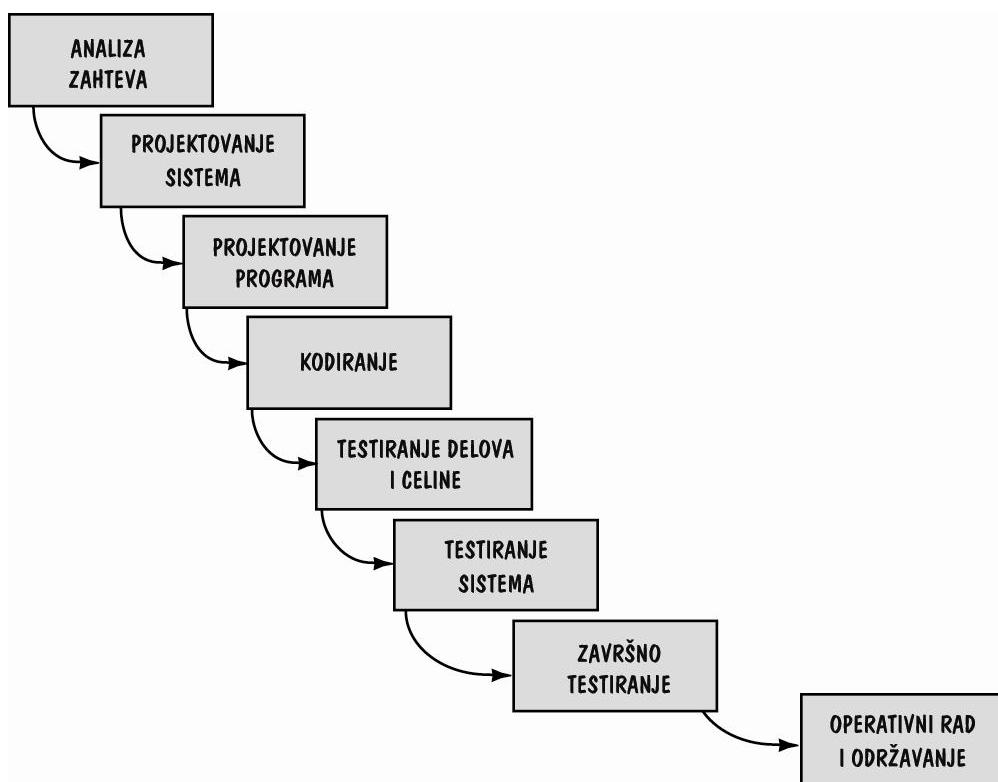
Jedna posebna varijanta model vodopada je *formalan sistem razvoja*. U njemu, postoji matematički model specifikacije sistema, koji se matematičkim transformacijama, prebacuje iz faze u fazu, zadržavajući konzistentnost sistema sve do dobijanja izvršnog koda. Pod pretpostavkom da je matematički model ispravno postavljen, primena formalnog procesa razvoja, dolazi se do softvera koji 100% konzistentan sa definisanim zahtevima. Na ovaj način se obično razvija softver koji mora da garantuje bezbednost, pouzdanost i zadovoljenje zahteva sigurnosti.

## PRIMENA KOD KRITIČNIH SOFTVERSKIH SISTEMA

*Izbor pravog softverskog procesa je od ključne važnosti za operativni rad i održavanje.*

Smatra se da metod vodopada je najbolje koristiti u slučaju razvoja velikih softverskih sistema i tzv. kritičnih sistema, koji moraju da budu vrlo pouzdan. Na primer, jedan od takvih sistema je sistem protiv blokiranja kočnica u automobilima.

Ovo je sigurnosno kritičan sistem, tako da zahteva naprednu analizu pre implementacije. Svakako je potreban planski usmereni pristup razvoju sa pažljivo analiziranim zahtevima koji nisu podložni promenama. Model vodopada je stoga najprikladniji pristup za korišćenje, uz dodatak formalnih transformacija između različitih faza razvoja.



Slika 2.1.2 Faze razvoja po modelu vodopada sistema protiv blokiranja kočnica u automobilim

## MODEL VODOPADA (VIDEO)

*Model vodopada - video klips 1*

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

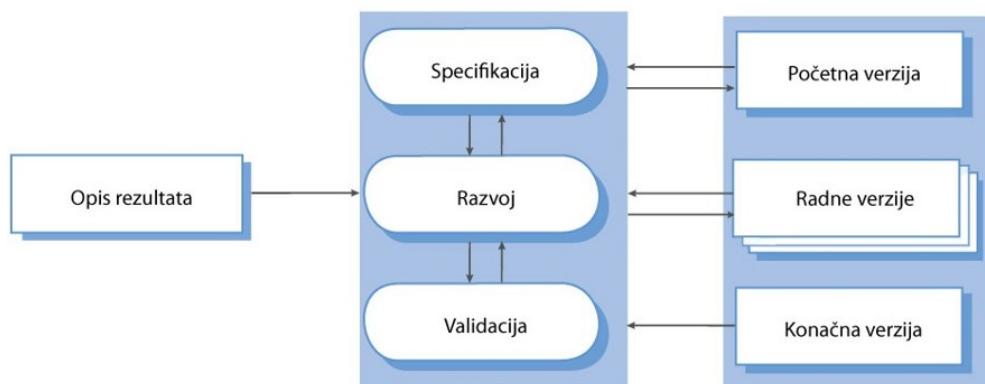
## ✓ 2.2 Model inkrementalnog razvoja

### KONCEPT INKREMENTALNOG RAZVOJA SOFTVERA

*Inkrementalni razvoj se zasniva na postupku postepenog razvoja novih inkremenata, tj. verzija softvera, i zamenom prethodno razvijenih verzija.*

Inkrementalni razvoj se zasniva na postupku po kome se razvije neka početna implementacija softvera, koja se pokazuje korisniku. Na osnovu njegovih primedbi, napravi sa nova verzija softvera, i tako preko više verzija, dolazi se do konačne verzije, koja predstavlja razvijeni softver.

Inkrementalni razvoj, koji čini osnov agilnih pristupa u razvoju softvera, je bolji od modela vodopada za najveći broj poslovnih aplikacija, e-poslovanje i personalnih sistema. Inkrementalni razvoj odražava način kako mi često rešavamo probleme. Mi retko pravimo detaljan plan za budućnost, već idemo u budućnost pristupom korak-po-korak. Pri tome se često vraćamo nazad, da bi ispravili uočene greške. Inkrementalni razvoj softvera je jeftiniji i lakši za rad kada se često traže promene u softveru tokom njegovog razvoja.



Slika 2.2.1 Model inkrementalnog razvoja softvera [1.2]

Svaki inkrement, tj. nova verzija softvera, sadrži neke od potrebnih funkcija (tj. funkcionalnost) koje traži korisnik. Prve verzije obično sadrže najvažnije funkcije, ili najhitnije zahtevane funkcije. Ovo omogućava da korisnik softvera može da u ranim fazama razvoja softvera oceni da li softver obezbeđuje funkcije koje se od njega zahtevaju. Ako se utvrdi da to softver ne obezbeđuje, trenutna verzija softvera se onda menja, a nova funkcija se ostavlja za neku kasniju verziju, tj. inkrement.

### ŠTA SADRŽI JEDAN INKREMENT?

*Inkrement sadrži novi deo softvera koji zadovoljava novu grupu zahteva koji su dodeljeni inkrementu, i tako razvijen inkrement se integriše sa prethodno razvijenim softverom*

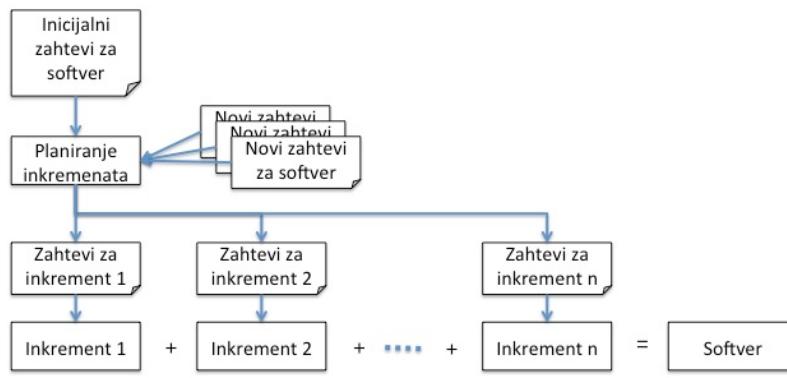
Kao i kod linearног, tj. sekвencijalног modela (modela vodopada), i kod inkrementalног razvoja najpre se napravi lista zahteva koje softverski sistem treba da zadovolji. Razlika je samo u tome, што kod inkrementalнog razvoja, ti zahtevi se ne primenjuju u razvoju svi odjedanput već se podele po fazama razvoja softvera. U svakoj fazi se primenjuje samo skup zahteva dodeljen toj fazi. Svaka faza se realizuje određenim inkrementom (dodatkom) koji predstavlja novi deo softvera koji se dodaje prethodno razvijenom softveru.

Svaki inkrement, znači, sadrži svoj skup zahteva koje treba da primeni, tj. da se razvije softverski inkrement koji primenjuju te zahteve. Zato se svaki inkrement detaljno analizira, i softver koji se razvija u okviru jednog inkrementa prolazi kroz sve aktivnosti sekвencijalнog modela: projektovanje, razvoj (implementacija), testiranje, integracija se prethodno razvijenim inkrementima, tj. Softverskim sistemom, njegovo testiranje i evolucija.

Za razvoj svakog inkrementa se planira jedno fiksno vreme (npr. dve nedelje). Ako se za to vreme ne primene svi planirani zahtevi za taj inkrement, oni se onda izostavljaju iz inkrementa razvijenog softvera i takav, reducirani inkrement se integriše sa prethodno razvijenim softverom. Neprimenjeni deo zahteva se ostavlja za naredni inkrement.

Ako se desi, da su svi planirani zahtevi realizovani pre planiranog vremena razvoja inkrementa, onda se dodaje deo zahteva planiranih za naredni inkrement, i tako proširen inkrement se razvija i integriše sa do tada razvijenim softverskim sistemom. Kupac softvera onda može da analizira tako dobijenu novu verziju softvera i da da neke nove primedbe i zahteve, koji se onda ubacuju u listu zahteva za naredne inkrente.

Da rezimiramo. **Inkrement** obezbeđuje novi deo softvera koji zadovoljava novu grupu zahteva. Sabiranjem tako razvijenih softverskih inkremenata, dobija se konačana softverski sistem (slika 2)



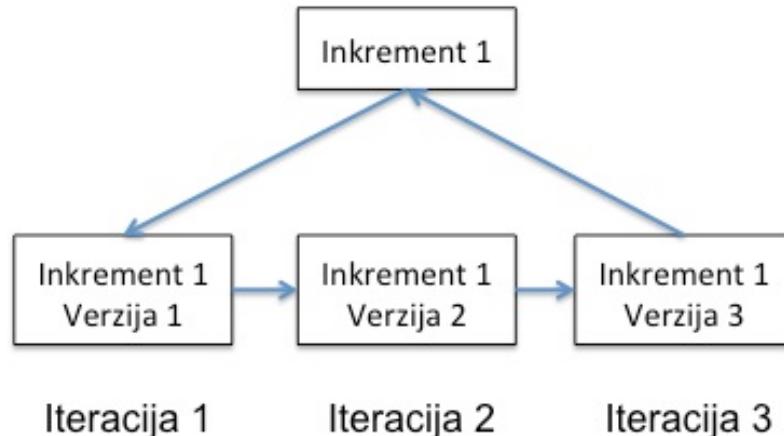
Slika 2.2.2 Inkrementalni razvoj softvera [1.1]

## INKREMENTALNI I ITERATIVNI RAZVOJ SOFTVERA

*Inkrementalni razvoj preko više verzija, vrši poboljšanje softvera. Svaki softverski inkrement se može i iterativno razvijati, pri čemu svaka verzija poboljšava.*

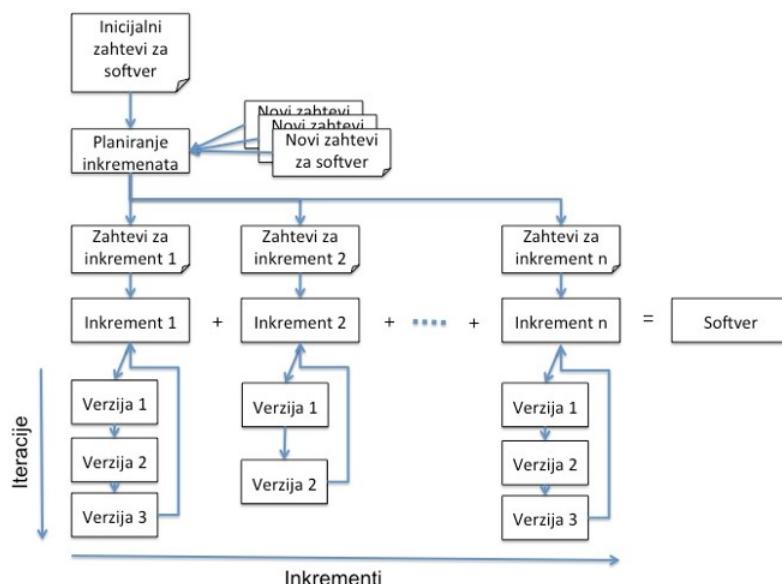
**Iteracija** znači ponavljanje. **Iterativni razvoj** znači da se proizvod poboljšava ponavljanjem nekih aktivnosti u procesu razvoja da bi se izvršila njegova poboljšanja. Iterativni razvoj se

može primeniti i u slučaju razvoja inkrementa. Svaki inkrement se razvija u više verzija, pre čemu svaka naredna verzija prestavlja poboljšanje prethodne verzije (slika 3).



Slika 2.2.3 Iterativni razvoj jednog softverskog inkrementa [1.1]

Korišćenje inkremenata i iteracija su bitni delovi više poznatih inkrementalnih metoda razvoja softvera (npr. Scrum, RUP). Slika 4 prikazuje princip njihovog kombinovanja.



Slika 2.2.4 Inkrementalni i iterativni razvoj jednog softvera[1.1]

## DOBRE I LOŠE STRANE PRIMENE MODELA INKREMENTALNOG RAZVOJA

*Inkrementalni razvoj obezbeđuje niže troškove razvoja, mišljenje korisnika i brži razvoj softvera. Nedostaci su u nevidljivosti procesa i postepeno urušavanje sa dodavanjem novih inkrementa.*

Inkrementalni razvoj, u odnosu na model vodopada, obezbeđuje sledeće prednosti:

1. Niži troškovi realizacije zahteva korisnika. Količina analiza i dokumenata koja treba da se ponovo uradi je znatno manja nego što je u slučaju primene modela vodopada.

2. Lakše je obezbediti mišljenje korisnika na rezultat razvoja, jer on može da komentariše demonstraciju radne verzije softvera i da vidi da li su njegovi zahtevi primjenjeni. Korisnici se teško snalaze u dokumentaciji, te ne mogu na osnovu nje da ocenjuju da li su njihovi zahtevi zadovoljeni. Tek kada vide kako softver radi, mogu da komentarišu, a to inkrementalni razvoj obezbeđuje.
3. Brža isporuka i instalacija softvera kod korisnika, iako nema sve tražene funkcionalnosti (funkcije). Korisnici mogu softver da koriste ranije, iako bez svih traženih funkcija, nego što je to slučaj kod modela vodopada.

Inkrementalni razvoj se danas primenjuje u različitim oblicima pri razvoju aplikacija. Može se primeniti i u obliku planom vođenog procesa, kada se inkrementi (softverske verzije) unapred planiraju..

Kada se primenjuje u obliku agilnog razvoja softvera, rani inkrementi (verzije softvera) su identifikovani, a razvoj kasnijih inkremenata zavisi od napretka u razvoju, kao i od prioriteta koje korisnik nameće.

Iz perspektive upravljanja, inkrementalni pristup razvoju softvera ima dva problema:

1. *Proces nije vidljiv.* Menadžeri žele da imaju regularnost u dobijanju rezultata da bi ocenjivali napredak u radu. Međutim, kako se sistem brzo razvija, štedi sa na izradi dokumentacije za svaku verziju softvera.
2. *Struktura sistema ima tendenciju urušavanja sa dodavanjem novih inkremenata* (verzija). Bez ulaganja dodatnog novca i vremena da se sistem strukturno unapredi, regularne promene sistema vode ka njegovom postepenom urušavanju. Ubacivanje novih promena u sistemu, njegovo održavanje vremenom postaje skupo i teško.

U slučaju velikih sistema, neophodna je stabilnost arhitekture sistema i jasna odgovornost različitih timova koji rade na delovima sistema, a u odnosu na arhitekturu. To se mora unapred planirati, umesto inkrementalno razvijati.

## DVA SISTEMA KOJI SE RAZVIJAJU INKREMENTALNIM RAZVOJEM

*Model opšteg softverskog procesa koji najviše odgovara za upotrebu i za razvoj sledećih sistema*

**Model opšteg softverskog procesa koji najviše odgovara za upotrebu i za razvoj sledećih sistema**

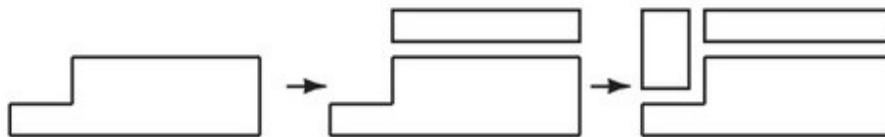
### 1. Održavanje sistema virtualne realnosti

Ovo je sistem u kome će se zahtevi menjati i postojaće opsežne komponente korisničkog interfejsa. Inkrementalni razvoj sa, možda, nekim prototipom UI-a je najprikladniji model. Može se koristiti i agilni proces razvoja.

### 2. Interaktivni sistem za planiranje putovanja koji omogućava korisnicima planiranje svojih putovanja za minimalnim uticajem promena

Ovaj sistem predstavlja sistem sa složenim korisničkim interfejsom koji mora biti stabilan i pouzdan. Inkrementalni razvoj je najprikladniji jer će podržati sve promene u zahtevima sistema pri čemu bi troškovi bili znatno manji pri svakoj izmeni.

### INKREMENTALNI RAZVOJ



Slika 2.2.5 Vizuelizacija načina razvoja inkrementalnim modelom

## ✓ 2.3 Model razvoja upotrebom komponenata

### RAZVOJ ZASNOVAN NA PONOVNOJ UPOTREBI KOMPONENTA

*Razvoj zasnovan na ponovnoj upotrebi komponenata zasniva se na velikoj bazi višestruko upotrebljivih softverskih komponenti koje se biraju i integrišu u sistem koji se kreira.*

U većini softverskih projekata, prisutna je neka softverska komponenta koja je ranije razvijena. Ovo se obično dešava neformalno kada ljudi koji rade na projektu poznaju dizajn ili kod koji je sličan onome koji je tražen. Oni ga nalaze, modifikuju kao što je zahtevano i objedinjuju ga u sistem.

Razvoj zasnovan na ponovnoj upotrebi komponenata zasniva se na velikoj bazi ponovno korišćenih softverskih komponenti kojima se može pristupiti, kao i nekom integrativnom okviru za ove komponente. Ponekad ove komponente su sistemi (COTS ili Commercial-off-the-shelf sistemi) koji mogu biti korišćeni da obezbede posebnu funkcionalnost kao što je formatiranje teksta, numerički proračuni, itd.

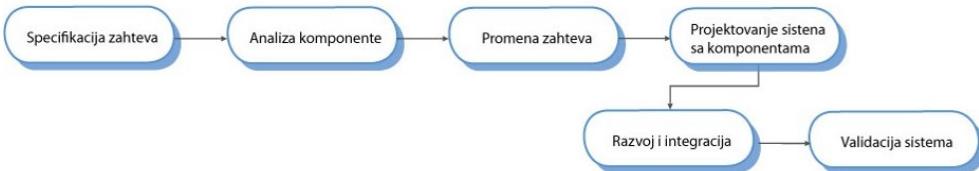
Dok su faza specifikacije početnih zahteva i faza validacije uporedive sa drugim procesima, međufaze u ovom procesu su različite. Ove faze su::

1. Analiza komponenti- U skladu sa datom specifikacijom zahteva, sprovedena je pretraga za neophodnim komponentama. Obično, ne postoji potpuna podesivost i komponente koje mogu biti korišćene obezbeđuju samo deo zahtevane funkcionalnosti.
2. Modifikacija zahteva- Tokom ove faze, analizirani su zahtevi korišćenjem informacija o komponentama koje su pronađene. Oni su zatim modifikovani da budu u skladu sa dostupnim komponentama. Kada su modifikacije nemoguće, aktivnost analize komponenti može biti ponovno urađena da bi se pronašla alternativna rešenja.
3. Projektovanje sistema sa komponentama- Tokom ove faze, projektovan je strukturni okvir (framework) sistema ili je korišćen postojeći okvir. Projektanti uzimaju u obzir komponente koje mogu da se ponovno koriste i organizuju okvir da bude prilagođen njima.

4. **Razvoj i integracija**- Razvijen je softver koji ne može biti kupljen i komponente i COTS sistemi su integrirani da stvore sistem. Integracija sistema, u ovom modelu, pre može biti deo procesa razvoja nego odvojena aktivnost.

## PREDNOSTI I NEDOSTACI RAZVOJA UPOTREBOM KOMPONENTA

*Razvoj zasnovan na ponovnoj upotrebi komponenata smanjuje obim posla u razvoju softvera, što smanjuje cenu i vreme razvoja, ali ne uspeva uvek da zadovolji sve zahteve klijenata.*



Slika 2.3.1 Model razvoja softvera korišćenjem višestruko upotrebljivih komponenti [1.2]

Model razvoja zasnovan na ponovnoj upotrebi komponenata ima očiglednu prednost da smanjuje količinu softvera koji treba da bude razvijen i takođe smanjuje troškove i rizike. To obično vodi ka bržoj isporuci softvera. Međutim, kompromisi zahteva su neizbežni i ovo može voditi ka sistemu koji neće u potpunosti ispuniti stvarne potrebe korisnika.

Postoje **tri tipa komponenti** koje se koriste:

1. **Veb servisi** koji se razvijaju u skladu sa standardima za servise i koji se mogu aktivirati iz udaljene lokacije.
2. **Kolekcije objekata** koji su razvijeni kao paketi radi kasnije integrisanja u sisteme, a u okviru komponentinskih okvira, kao što su .NET i J2EE.
3. **Samostalni softverski sistemi** koji se konfigurišu za upotrebu u određenom okruženju.

Razvoj zasnovan na ponovnoj upotrebi komponenata ima jasnou prednost u količini softvera koji treba razviti, jer je ona znatno smanjena, što se odražava na niže troškove i kraće vreme razvoja. Međutim, softverski sistem zbog upotrebe unapred definisanih komponenti, neminovno ne zadovoljava sve potrebe i zahteve korisnika. Takođe, gubi se i kontrola nad evolucijom softvera, sa upotrebom novih verzija komponenata, jer su one razvijene najčešće u drugim organizacijama.

## PRIMER RAZVOJA SISTEMA ZA RAČUNOVODSTVO UNIVERZITETA

*Izbor pravog softverskog procesa je bitna odluka u softverskom inženjerstvu.*

**Sistem za računovodstvo univerziteta koji treba da zameni postojeći sistem.**

Ovo je sistem čiji su zahtevi prilično poznati i koji će se koristiti u okruženju u kombinaciji sa puno drugih sistema, kao što je **podsistem upravljanja privilegija za korišćenje sistema**. Jasno je da ovaj sistem predstavlja samo deo celokupnog univerziteta jer obuhvata deo koji se odnosi na računovodstvo, pa prema tome, pristup koji zasniva na ponovnoj upotrebi komponenata može biti odgovarajući za ovo. Na taj način komponente celokupnog sistema mogu u hijerarhiji stajati nezavisno što ih čini upotrebljivim i u ostalim sličnim sistemima. Obično pri tazvoju softvera, inženjeri softvera teže da minimiziraju vreme i napore razvoja softvera tako što koriste već ranije razvijene komponente, koje se ponavljaju u različitim softverskim proizvodima. To mogu biti softverske komponente koje je njihova firma ranije razvila, ili koje preuzmu sa Interneta, kao komponente sa otvorenim kodom (ređe se komponente kupuju). U razmatranom slučaju ovde, postoji puno razvijenih softvera za vođenje računovodstvenih aktivnosti, te je jeftinije i brže koristiti neki od postojećih sistema kao softversku komponentu, koju treba integrisati sa informacionim sistemom univerziteta. Naravno, izbor odgovarajuće komponente i njen odgovarajuće integrisanje sa celim informacionim sistemom su dva zadatka koja se moraju rešiti.

## ✓ Poglavlje 3

### Osnovne aktivnosti procesa

## ČETIRI OSNOVNE AKTIVNOSTI SOFTVERSKIH PROCESA

*Četiri osnovne aktivnosti softverskih procesa su: specifikacija, razvoj, validacija i evolucija*

Stvarni softverski procesi su mešavina sekvenci tehničkih, kolaborativnih i upravljačkih aktivnosti sa ukupnim ciljem da specificiraju, projektuju, implementiraju i testiraju neki softverski sistem. Softverski inženjeri u svom radu koriste različite softverske alate radi rada sa različitim tipovima dokumenata i uređivanja velike količine detaljnih informacija koji se stvara u projektu razvoja nekog velikog softverskog sistema.

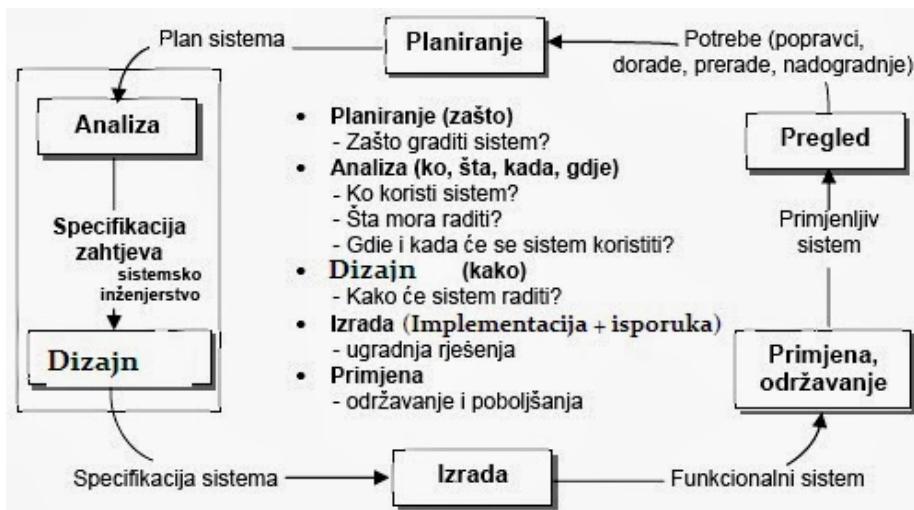
**Četiri osnovne aktivnosti softverskih procesa** (slika 1) su :

1. **Specifikacija softvera** (inženjering zahteva)
2. **Razvoj softvera** (projektovanje i programiranje)
3. **Validacija softvera** (testiranje) i
4. **Evolucija softvera** (održavanje i usavršavanje)

One se različito organizuju u različitim procesima razvoja softvera. U modelu vodopada, aktivnosti su organizovane redno (jedna posle druge), dok su pri inkrementalnom razvoju one izmešane. Kako se te aktivnosti realizuju, to zavisi od tipa projekta, ljudi, i organizacione strukture. Pri primeni agilne metodologije poznate kao Ekstremno programiranje, specifikacija se piše na karticama. Testovi se izvršavaju i razvijaju pre samog programa. Evolucija softvera može da obuhvati značajnu promenu strukture softvera. Na slici 2 prikazan je proces razvoja na nešto drugačiji način.



Slika 3.1.1 Četiri osnovne faze procesa razvoja softvera [1.1]



Slika 3.1.2 Jeden od načina prikazivanja procesa razvoja softvera

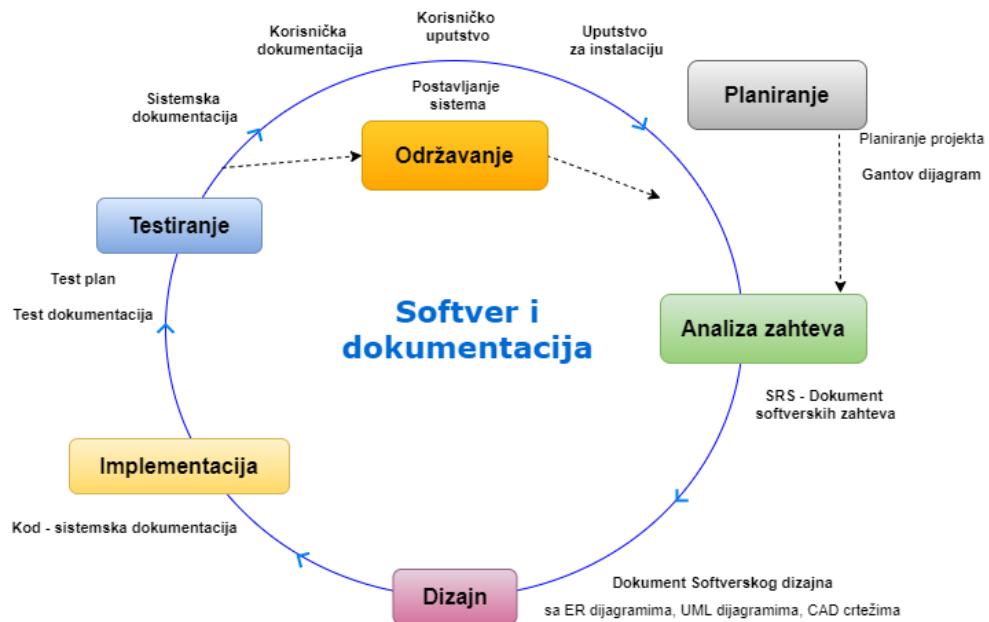
## PRIKAZ AKTIVNOSTI SOFTVERSKOG PROCESA

### *Osnovne aktivnosti softverskih procesa.*

Proces razvoja softvrea se može prikazati na različite načine. Na slici 3 je prikazan jedan primer prikaza. Međutim, svi oni u osnovi sadrže osnovane aktivnosti (ili faze) procesa razvoja softvera (koji su prikazani na slici 1).

Termin projektovanje softvera (**software design**) ponekad se kod nas naziva i dazajnom softvera, mada to ne bi trebalo da bude u upotrebi u kontekstu projektovanja proizvoda. Za inženjerstvo je odavno odomaćen termin "projektovanje" nečega, a ne dizajn. Dizajn se više koristi u primijenjenoj umetnosti, npr. industrijski dizajn, grafički dizajn, modni dizajn.... U inženjerstvu se koriste termini: projektovanje zgrade, mosta, mašine.... Zato je i prirodno da se kaže "projektovanje softvera", jer je softversko inženjerstvo jedna vrsta inženjerstva a ne umetnosti.

U narednim poglavljima ćemo ukratko opisati ove četiri osnovne aktivnosti.



Slika 3.1.3 Jeden od načina prikazivanja procesa razvoja softvera, tj. softverskog procesa

## FAZE SOFTVERSKIH PROCESA (VIDEO)

### Aktivnosti procesa -Video klip 1

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

## VIDEO PREDAVANJE ZA OBJEKAT "OSNOVNE AKTIVNOSTI PROCESA"

*Trajanje video snimka: 11min 57sek*

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

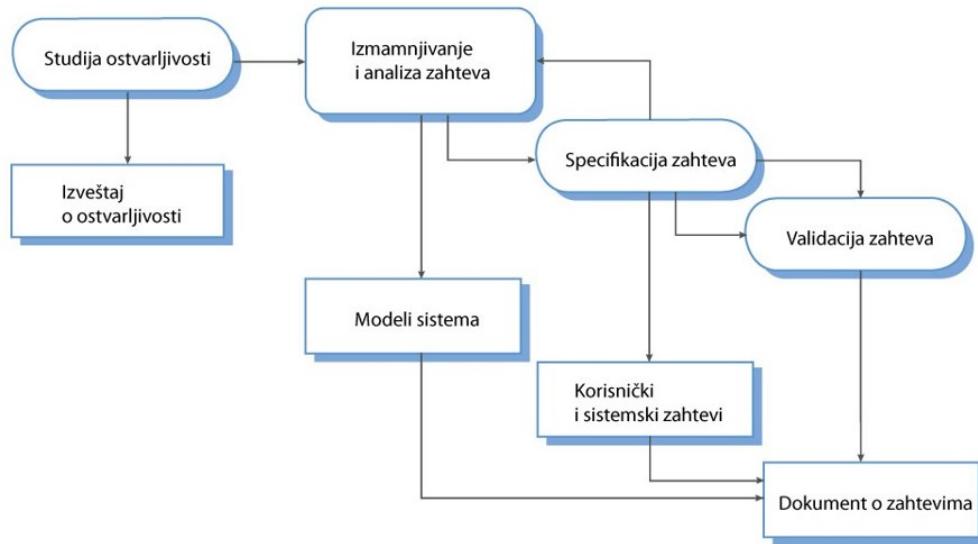
### ✓ 3.1 Specifikacija softvera

## INŽENJERING ZAHTEVA

*Inženjering zahteva je faza softverskog procesa, sa četiri aktivnosti: izrada studije izvorljivosti, prikupljanje potreba i njihova analiza, specifikacija zahteva i njihova validacija (potvrđivanje).*

Specifikacija softvera treba da ustanovi koji servisi su zahtevani od strane sistema i ograničenja rada i razvoja sistema. Ova aktivnost se često zove inženjering zahteva. Inženjering zahteva je posebno kritična faza softverskog procesa zato što greške u ovoj fazi neizbežno vode kasnijim problemima u projektovanju i implementaciji sistema.

Ovaj proces vodi ka stvaranju dokumenta zahteva koji je specifikacija za sistem. Zahtevi su u ovom dokumentu obično predstavljeni u dva nivoa detalja. Krajnjim korisnicima i mušterijama potreban je visoki nivo zahteva; inženjerima koji razvijaju sistem potrebna je još detaljnija specifikacija sistema.



Slika 3.2.1 Proces inženjeringa zahteva [1.2]

## ČETIRI GLAVNE AKTIVNOSTI PROCESA INŽENJERINGA ZAHTEVA

*Glavne aktivnosti procesa inženjeringa zahteva su: studija izvodljivosti, izvođenje i analiza zahteva, specifikacija zahteva i validacija zahteva.*

Postoje četiri glavne aktivnosti ili faze procesa inženjeringa zahteva:

1. Izrada studije izvodljivosti- Procena je napravljena u skladu sa tim da li identifikovane potrebe korisnika mogu biti zadovoljene korišćenjem postojećeg softvera i hardverskih tehnologija. Na osnovu studije se donosi odluka da li je predloženi sistem troškovno-efektivan iz poslovne tačke gledišta i da li može biti razvijen poštujući postojeća ograničenja budžeta. Studija izvodljivosti trebalo bi da bude relativno jeftina i treba da bude brzo urađena. Rezultat studije izvodljivosti treba da pruži informaciju koja će odlučiti da li ići napred sa mnogo detaljnijom analizom.
2. Izvođenje i analiza zahteva- Ovo je proces izvođenja zahteva sistema preko posmatranja postojećih sistema, diskusije sa potencijalnim korisnicima, analize zahteva, itd. Ovo može obuhvatiti razvoj jednog ili više različitih modela sistema i prototipova. Ovo pomaže analitičarima u razumevanju sistema za koji treba da se uradi specifikacija.
3. Specifikacija zahteva- Specifikacija zahteva je aktivnost prevođenja informacija skupljenih tokom analize u dokument koji definiše skup zahteva. U ovaj dokument mogu biti uključena dva tipa zahteva. Korisnički zahtevi su abstraktni iskazi zahteva

sistema za mušteriju i krajnjeg korisnika; sistemski zahtevi su mnogo detaljniji opis funkcionalnosti koja treba da bude obezbeđena.

4. Validacija zahteva- Ova aktivnost proverava zahteve za realizmom, konzistencijom i potpunošću. Tokom ovog procesa, greške u dokumentaciji o zahtevima se neizbežno otkrivaju. Oni zbog toga moraju biti modifikovani da bi se korigovali ovi problemi.

Naravno, aktivnosti u procesu zahteva nisu jednostavno tretirane nekom strogom sekvencom.

## INŽENJERSTVO ZAHTEVA (VIDEO)

*Specifikacija softvera - Inženjerstvo zahteva (video)*

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

### ▼ 3.2 Projektovanje softvera i programiranje

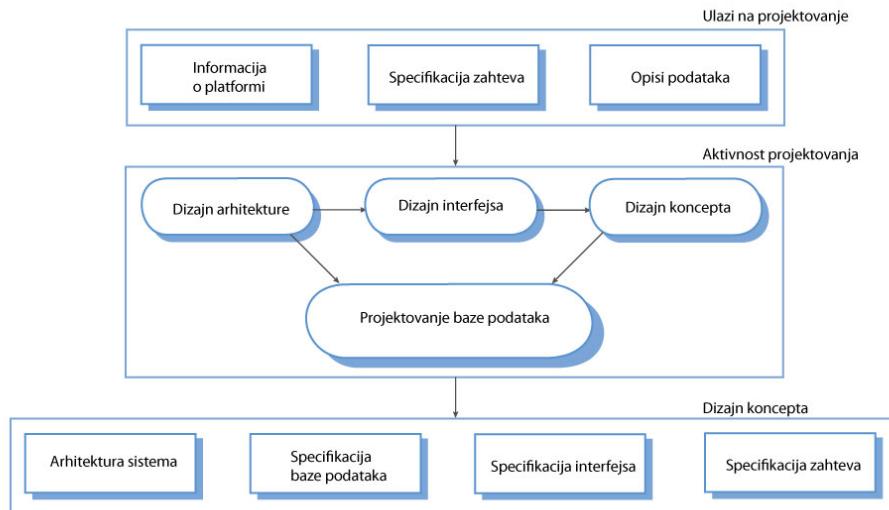
#### PROJEKTOVANJE SOFTVERA

*Projektovanje softvera je opis strukture softvera koji treba da bude razvijen, podataka koji su deo sistema, interfejsa između komponenti sistema, i poneka korišćenih algoritama.*

Implementaciona faza razvoja softvera je proces pretvaranja specifikacije sistema u izvršni sistem. Ona uvek obuhvata procese projektovanja i programiranja softvera, ali ako je koriščen evolucionarni pristup razvoja, on može takođe obuhvatiti prečišćavanje specifikacije softvera.

Projektovanje softvera je opis strukture softvera koji treba da bude razvijen, podataka koji su deo sistema, interfejsa između komponenti sistema, i ponekad korišćenih algoritama. Projektanti ne stižu do konačnog projekta trenutno, već razvijaju projekat kroz različite verzije.Proces projektovanja obuhvata dodavanje formalnosti i detalja dok se razvija projektno rešenje, sa čestim vraćanjem unazad da bi se korigovale prethodne verzije.

Na slici je prikazan jedan apstraktan model ovog procesa koji prikazuje ulaze u proces projektovanja, aktivnosti procesa, i proces projektovanja, aktivnosti procesa, i dokumenta koji su rezultat ovog procesa.



Slika 3.3.1 Opšti model procesa projektovanja softvera [1.2]

Dijagram na slici 1 ukazuje da se sve faze procesa projektovanja redno izvršavaju, što nije tako u praksi, jer su te aktivnosti izmešane. Povratne sprege koje povezuju dve faze izazivaju dodatan rad na promeni projektnog rešenja, jer je to neminovnost u svim projektima razvoja.

## ČETIRI AKTIVNOSTI PROCESA PROJEKTOVANJA

*Četiri osnovne aktivnosti procesa projektovanja softvera su: projektovanje arhitekture, projektovanje interfejsa, projektovanje komponenata i projektovanje baze podataka.*

Najveći broj softvera komunicira sa drugim softverskim sistemima, kao što su: operativni sistemi, baze podataka, softver srednjeg sloja, i drugi aplikacioni sistemi. Sve to čini „softversku platformu“, tj. okruženje u kojoj softver radi. Informacija o softverskoj platformi je značajan ulaz u proces projektovanja, jer na osnovu toga, projektanti treba da odluče kako na najbolji način da integrišu svoj softver sa softverskim okruženjem.

Specifikacija zahteva opis funkcionalnosti softvera koju softver mora da obezbedi, kao i zahteve za performansama i zahteve za povezivanjem. Ako sistem obrađuje podatke, onda se podaci moraju opisati u specifikaciji platforme. U suprotnom, opis podataka bi trebalo da se obezbedi na ulazu u proces projektovanja, kako bi se definisala organizacija podataka u sistemu.

Slika 1 prikazuje četiri aktivnosti koje su deo procesa projektovanja informacionih sistema:

1. Projektovanje arhitekture, gde se utvrđuje ukupna struktura sistema, glavne komponente (podsistemi ili moduli), njihove veze, i način njihovog raspoređivanja.
2. Projektovanje interfejsa, kada se definišu interfejsi između komponenti sistema. Kada se definišu interfejsi, svaka komponenta može nezavisno i istovremeno da se razvija.

3. **Projektovanje komponenata**, kada se svaka komponenta sistema projektuje. To može biti i jednostavni iskaz o očekivanoj funkcionalnosti koja se mora primeniti, pri čemu se specifično projektno rešenje ostavlja da uradi programer. Može biti i lista promena koje se moraju izvršiti na komponenti ili može biti detaljno projektovani model. Projektovani model može se iskoristiti i za automatsku generaciju implementacije, tj. koda.
4. **Projektovanje baze podataka**, kada se projektuje struktura podataka i definiše kako se ona predstavlja u bazi podataka. Rad zavisi i od toga da li se radi nova baza, ili se menja neka postojeća baza podataka.

## PROGRAMIRANJE

*Programiranje, kao jedna individualizirana aktivnost koja najčešće ne sledi neki opšti model procesa.*

Slika 1 pokazuje i niz izlaznih rezultata aktivnosti procesa projektovanja koji se dosta razlikuju od sistema do sistema. Za *kritičke sisteme*, dokumentacija mora da definiše vrlo precizno i tačno opis sistema. Pri razvoju sistema na bazi modela, ovi izlazi su uglavnom u obliku dijagrama. U slučaju primene agilnih metoda, posebna dokumentacija i ne mora da postoji, jer se dokumentuje samo izvorni kod.

Programiranje, kao jedna individualizirana aktivnost koja najčešće ne sledi neki opšti model procesa. Neki programeri počinju rad prvo sa komponentama, koje razumeju, te ih i prve razviju. A onda prelaze na manje poznate komponente. Nasuprot ovome, drugi programeri ostavljaju poznate komponente za kraj, jer znaju kako će ih razviti. Neki programeri vole da prvo definisu podatke u procesu, a onda razvijaju proces na bazi definisanih podataka.

Najčešće, programeri vrše određeno testiranje koda koji su razvili. Tim procesom tzv. dibaginga (engl. **debugging**), tj. otklanjanja grešake iz koda. Testiranje defekata i dibaging su dva različita procesa. Testiranje utvrđuje postojanje defekata. Dibaging se usmerava da locira mesto greške i na ispravljanje grešaka u kodu.

Pre otklanjanja grešaka, programeri koriste alate za otklanjanje grešaka (**debugging**), jer oni omogućavaju praćenje trenutnih vrednosti promenljivih u programu pri kretanju od jedne do druge linije u kodu.

## PROJEKTOVANJE SOFTVERA I NJEGOVE FAZE (VIDEO)

*Razvoj softvera - video klip 1*

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

## ✓ 3.3 Validacija softvera

### ŠTA JE VALIDACIJA SOFTVERA?

*Validacija (potvrđivanje) softvera, ili tačnije verifikacija i validacija, treba da pokaže da li sistem zadovoljava svoju specifikaciju i očekivanje korisnika.*

Validacija (potvrđivanje) softvera treba da pokaže da li sistem zadovoljava svoju specifikaciju i očekivanje korisnika sistema. Zato, koristi se testiranje sistema sa simuliranim test podacima. Validacija može da obuhvati i procese provere (inspekcija i recenzija) svake faze softverskog procesa, počev od zahteva korisnika, pa do razvoja programa. Zbog dominantne aktivnosti testiranja, najveći troškovi validacije se vrše za vreme i posle implementacije sistema.



Slika 3.4.1 Proces testiranja [1.2]

Ako se i moduli i pod-sistemi smatraju jedinicama (komponentama softvera), onda se proces validacije može podeliti u tri osnovne faze:

1. *Razvojno testiranje:* Komponente sistema se posebno testiraju od strane onih koji su ih i razvili, u toku procesa njihovog razvoja. Svaka komponenta se posebno testira.

Komponenta može biti vrlo jednostavna jedinica, koja obezbeđuje samo jednu funkciju, ali može biti i grupa ovakvih entiteta (npr. paket klasa). U ovoj fazi koriste se alati za automatsko testiranje, kao što je JUnit.

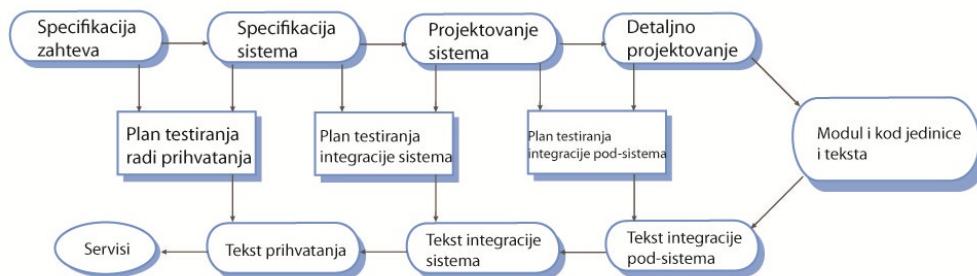
1. *Sistemsko testiranje:* Komponente sistema su integrisane i čine jedinstveni sistem. Testiranje sistema se vrši da bi se našle greške usled neodgovarajuće interakcije između komponenata i problema sa njihovim interfejsima. Testiranje sistema takođe treba da pokaže da li sistem ostvaruje sve funkcionalne i nefunkcionalne zahteve (npr. performanse). Kod velikih sistema (kao što je ovaj prikazan na slici), ovaj proces testiranja sistema se može izvršavati u fazama, jer se prvo testiraju integrisane jedinice u module, moduli u podsisteme, a na kraju, u ceo sistem.

2. *Test prihvatanja:* Ovo je krajnja faza procesa testiranja, pre njegovog prihvatanja za operativnu upotrebu. Sistem se testira korišćenjem podataka koje je obezedio korisnik (kupac). Test prihvatanja može da ukaže greške koje potiču od neadekvatnih zahteva na sistemskom nivou, što testiranje sa simuliranim podacima nisu mogla da pokažu. Testiranje prihvatanja treba da pokaže da sistem zadovoljava sve zahteve korisnika i ostvaruje zahtevane performanse.

## PLAN TESTIRANJA

*U stvarnosti, razvoj komponenata i proces testiranja su izmešani. Plan testiranja povezuje aktivnosti testiranja i razvoja.*

U stvarnosti, razvoj komponenata i proces testiranja su izmešani. Programeri rade sopstvene testove sa svojim podacima i inkrementalno testiraju kod u toku njegovog razvoja. To je sa ekonomskog stanovišta ispravan pristup, jer programer najbolje poznaje komponentu koju je razvio, te je najbolja osoba i da generiše slučajeve (scenarije) za testiranja. U ekstremnom programiranju, testovi se razvijaju zajedno sa zahtevima, i pre nego što razvoj i počne. Ovo omogućava testerima i programerima da bolje razumeju zahteve i da obezbede da nema kašnjenja pri testiranju. Pri primeni softverskih procesa koji su vođeni planom (slika 2 ), testiranje se vrši u skladu sa planovima testiranja. Nezavistan tim testera pripreme te planove, a na osnovu dokumentacije sa specifikacijom sistema i projektnom rešenju. Slika 2 pokazuje kako planovi testiranja povezuju aktivnosti testiranja i razvoja



Slika 3.4.2 Faze testiranja u procesu razvoja vođenim planom [1.2]

## ALFA I BETA TESTIRANJE

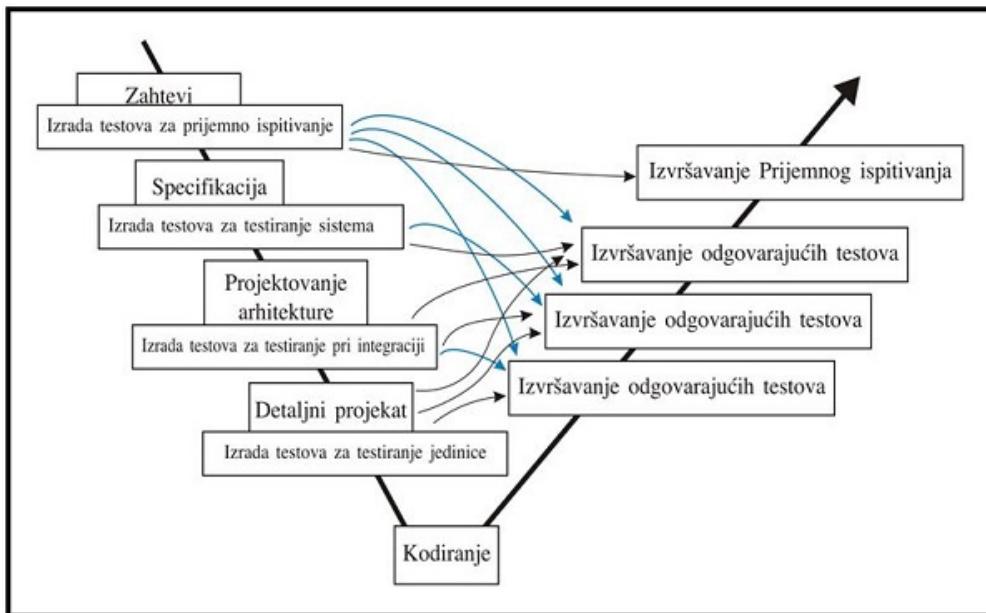
*Alfa testiranje vrši razvojni tim, sam, ili sa naručiocem softvera (kao test prihvatanja). Beta testiranje se vrši kod grupe odabralih korisnika sistema u njihovom radnom okruženju.*

Testovi prihvatanja se često nazivaju i „alfa testovima“. Alfa testiranje traje sve dok se naručilac softvera i razvojni tim sistema ne slože da razvijeni sistem zadovoljava postavljene zahteve.

U slučaju softverskih proizvoda (koji se prodaju na tržištu), koristi se i proces testiranja koji se naziva „beta testiranje“. Beta testiranje sistema se vrši kod grupe odabralih korisnika sistema, tj. u njihovom radnom okruženju. Oni šalju izveštaj razvojnog timu sistema. Na ovaj način se utvrđuju greške koje se javljaju u stvarnim uslovima rada sistema. Na osnovu ovih izveštaja, razvojni tim vrši promene u sistemu i proizvodi novu verziju ili za novo beta testiranje, ili za prodaju na tržištu.

## VEZA TESTOVA SA PROJEKTNOM DOKUMENTACIJOM SOFTVERA

*Prikaz faza testiranja paralelno sa fazama razvoja softvera.*



Slika 3.4.3 Prikaz paralelnog razvoja testiranja preklopjenih sa fazama razvoja softvera

## VERIFIKACIJA I VALIDACIJA SOFTVERA (VIDEO)

*Validacija softvera - video klip 1*

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

## ✓ 3.4 Evolucija softvera

### ŠTA JE EVOLUCIJA SOFTVERA?

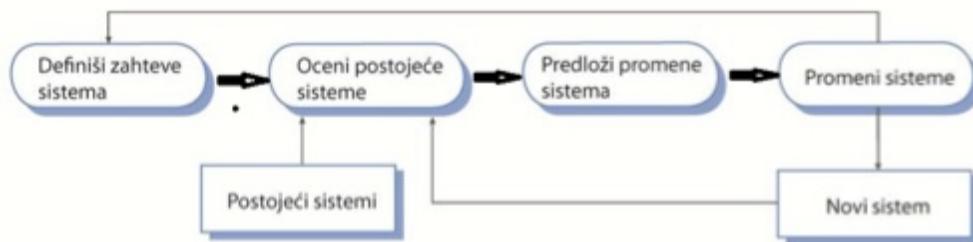
*Softversko inženjerstvo je evolucionarni proces gde se softver kontinualno menja tokom njegovog životnog veka da bi mogao da odgovori na promene zahteva naručioca.*

Fleksibilnost softverskih sistema je jedan od glavnih razloga zašto je sve više i više softvera objedinjeno u velike, kompleksne sisteme. Jednom kada je doneta odluka da se napravi hardver, vrlo je skupo praviti promene u projektovanom hardveru. Međutim, za softver promene mogu biti napravljene u bilo kom trenutku tokom ili nakon razvoja sistema. Ove promene mogu biti vrlo skupe, ali još uvek mnogo jeftinije u poređenju sa odgovarajućim promenama u hardveru sistema.

Istorijski gledano, uvek je postojala linija razgraničenja između procesa softverskog razvoja i procesa evolucije (održavanja) softvera. Razvoj softvera se odnosio na kreativnu aktivnost

gde je razvijan softverski sistem od početnog koncepta preko radnog sistema. Održavanje softvera je proces promene sistema jednom kada je sistem krenuo sa radom.

Mada su troškovi održavanja softvera često nekoliko puta veći od početnih troškova razvoja, procesi održavanja se smatraju manje izazovnim nego razvoj originalnog softvera. Ova linija razgraničenja postala je skoro nebitna. Malo softverskih sistema su kompletno novi sistemi i ima mnogo više smisla razmatrati razvoj i održavanje kao kontinuum. Pre nego dva odvojena procesa, mnogo je više realno razmatrati softverski inženjering kao evolucionarni proces gde se softver kontinualno menja tokom njegovog životnog veka da bi mogao da odgovori na promene zahteva i potreba korisnika. Ovaj evolucionarni proces je prikazan na slici.



Slika 3.5.1 Evolucija softvera [1.2]

## ODRŽAVANJE SOFTVERA (VIDEO)

### *Evolucija softvera - video klip 2*

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

## ▼ Poglavlje 4

# Razvoj softvera u uslovima stalnih promena

## KAKO RAZVIJATI SOFTVER U USLOVIMA STALNIH PROMENA?

*Kako se u praksi često menjaju zahtevi u toku razvoja softvera, primenjuju se dva pristupa u razvoju: izrada prototipa softvera, radi provere zahteva.*

Kod svih projekata razvoja velikih sistema, promene zahteva su neminovnost, jer se menjaju i uslovi kod naručioca sistema. Zato je vrlo važno da se, bez obzira na primjenjen model softverskog procesa, model procesa menja u skladu sa promjenjenim zahtevima.

Promjenjeni zahtevi obično uzrokuju ponavljanje nekih aktivnosti u razvoju, što povećava troškove razvoja. Koriste se dva pristupa smanjivanja ovih troškova:

1. **Izbegavanje promena**, u slučajevima kada softverski proces sadrži aktivnosti u kojima se očekuju promene, pre nego što se zahteva značajniji ponovni rad. Na primer, pre nego što se pribegne promeni sistema koje zahtevaju veće troškove, razvije se poseban prototip pomoću koga se naručiocu pokazuje kako bi sistem radio, kada se izvrše njegove promene. Tek kada se naručilac složi sa očekivanim rezultatom, pristupa se aktivnostima kojima se menja stvarni sistem, radi ostvarivanja usaglašenih rezultata.
2. **Tolerisanje promena**, neka forma inkrementalnog razvoja. Predložene promene se onda realizuju na inkrementu koji još nije razvijen. Na taj način, samo mali deo sistema (inkrement) je zahvaćen promenom, a ne i ceo sistem kada je proces tako projektovan da se promene mogu obavljati sa relativno niskim troškovima. U ovom slučaju, najčešće se primenjuje.

Imajući ovo u vidu, u praksi se primenjuju dva pristupa razvoju softvera u uslovima čestih promena zahteva:

1. **Izrada prototipa sistema**, kojim se prototip sistema ili njegov deo brzo razvija radi provere zahteva naručioca i radi provere ostvarljivosti nekih projektnih rešenja. Promene sistema se odlažu sve dok se te promene ne verifikuju na prototipu. Na taj način se smanjuje broj zahteva za promenama posle isporuke softverskog sistema.
2. **Inkrementalna isporuka**, kojim se naručiocu isporučuju inkrementi sistema radi komentarisanja i eksperimentisanja. To obuhvata i izbegavanje (prevremenih) promena i toleranciju promena. Time se izbegava prerano opredeljavanje da se nešto

u sistemu menja, dok se to ne proveri na inkrementu. Takođe, umesto da promene obuhvate ceo sistem, one se odnose samo na inkrement, što je znatno jeftinije.

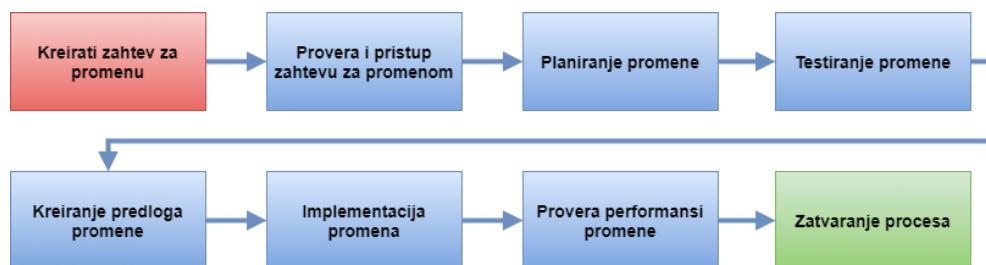
## PRIMER - SISTEM ZA ONLINE GLEDANJE FILMOVA I SERIJA

### *Sistem za online gledanje filmova i serija.*

Krajnji cilj sistema je da omogući korisniku pretraživanje i gledanje filmova i serija, ocenjivanje istih i dodavanje u listu omiljenih, kao i mogućnost korisnika da šalje zahteve administratoru sistema za postavljanje novog filma ili serije. Takođe sistem treba da obezbedi administratorima osnovne CRUD operacije nad filmovima i serijama. Sistem takođe prati najgledanije filmove/serije i postavlja ih na početnu stranicu. Pretraga se vrši po različitim kategorijama, kao što su naslov filma, godina izdavanja, žanr, rating itd.

Ako bi za ovaj sistem primenili inkrementalni model, jer nemamo precizno definisane zahteve omogućili bismo da se sistem koristi pre nego što su implementirane sve funkcionalnosti, prva faza razvoja bi trebalo da omogući CRUD funkcionalnost za filmove, serije i korisnike aplikacije, kao i pretragu i prikaz filmova i serija. Nove manje prioritetne funkcionalnosti bi se dodavale sa svakim novim inkrementom, kao što su na primer ocenjivanje filmova/serija, stavljvanje u listu omiljenih, komentarisenje, praćenje popularnosti.

Ovako projektovan sistem bio bi u stanju da odgovori na promene koje nastaju usled izmene u zahtevima korisnika.



Slika 4.1.1 Proces upravljanje promenama

## ▼ 4.1 Primena prototipa softvera

### ŠTA JE PROTOTIP SOFTVERA?

*Prototip je početna verzija softverskog sistema koja se upotrebljava radi pokazivanja koncepata, probe projektnih opcija, i boljeg razumevanja problema.*

Prototip je početna verzija softverskog sistema koja se upotrebljava radi pokazivanja koncepata, probe projektnih opcija, i boljeg razumevanja problema i njegovih mogućih rešenja. Brz, iterativan razvoj prototipa sa niskim troškovima, je neophodan, kako bi korisnici

softverskog sistema mogli da eksperimentišu sa prototipom u ranim fazama softverskog procesa.

Softverski prototip se koristi u procesu razvoja softvera da bi se olakšala primena zahtevane promene:

1. *U procesu inženjeringu zahteva*, prototip može da pomogne da se izmame i potvrde sistemski zahtevi.
2. *U procesu projektovanja sistema*, prototip se koristi radi ispitivanja određenog softverskog rešenja i radi podrške projektovanog korisničkog interfejsa.

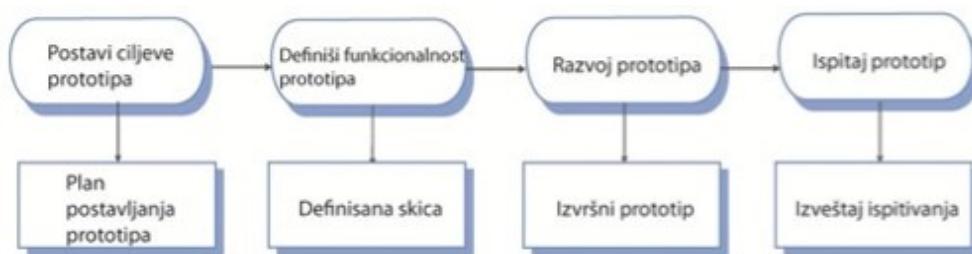
Prototipovi sistema pomažu korisnicima da vide koliko dobro sistem podržava njihov rad. Tada mogu da im se javе nove ideje za zahteve, a i da utvrde područja u kojima je softver slab ili jak. Tada oni predlažu nove zahteve. Pored toga, primena prototipova može da ukaže na greške i nedostatke u zahtevima koje su predloženi. Funkcija definisana u dokumentaciji, sama za sebe, može izgledati da je prikladna. Međutim, ista funkcija, kada se kombinuje sa drugim funkcijama, može se pokazati kao neprihvatljivom ili nekorektnom. U tom slučaju se menja specifikacija sistema kako bi bolje odražavala promene zahteva.

Prototip sistema se može koristiti u fazi projektovanja sistema radi sprovođenja eksperimenata i radi ocenjivanja ostvarljivosti predloženog projektnog rešenja. Primena brzih metoda razvoja radi provere korisničkog interfejsa, na primer, je često u upotrebi.

## PROCES RAZVOJA PROTOTIPIA SOFTVERA

*Proces razvoja prototipa određuje aktivnosti njegovog razvoja u skladu sa postavljenim ciljem. Ključno pitanje je šta prototip treba da obuhvati, a šta ne.*

Model procesa razvoja prototipa je prikazan da slici. Ciljevi razvoja prototipa bi trebalo da se jasno definišu na početku procesa. To može biti provera projektnog rešenje korisničkog interfejsa, ili validacija funkcionalnih zahteva sistema, ili razvoj sistema za proveru ostvarljivosti aplikacije. Isti prototip se može koristiti za više ciljeva. Bez jasnih ciljeva, može se postaviti pitanje namene prototipa.



Slika 4.2.1 Proces razvoja prototipa softvera [1.2]

Pri razvoju prototipa, ključno pitanje je šta on treba da obuhvati, a šta ne treba da bude u njemu. Radi minimizacije troškova, prototip treba oslobođiti nepotrebnih funkcija. Obično se izostavljaju i zahtevi vezani za performanse, kao što je brzina sistema i korišćenje

memorije. I otklanjanje grešaka može da bude izostavljeno, ako je cilj, na primer, provera dizajna korisničkog interfejsa.

Krajnja faza procesa je evaluacija prototipa, koja se vrši na bazi postavljenih ciljeva prototipa. Vremenom, korišćenjem sistema, korisnici otkrivaju greške u zahtevima i ispuštene zahteve.

## PROTOTIP NIJE SOFTVER ZA ISPORUKU

*Ponekad, menadžment vrši pritisak da razvojni tim pretvori prototip u konačno rešenje, da bi se uštedelo na vremenu i novcu, što nije najčešće prihvatljivo.*

Jedan od problema rada sa prototipom je što se on ne koristi isto kao i konačan sistem. Ako je prototip spor, korisnici će ga koristiti drugačije (idu skraćenicama) nego što će kasnije koristiti konačan sistem, koji je brži. To može da dovede do ispuštanja iz vida (i provere) nekih svojstava softvera u fazi ispitivanja prototipa, a koji se manifestuju pri radu konačnog sistema.

Ponekad, menadžment vrši pritisak da razvojni tim pretvori protoip u konačno rešenje, da bi se uštedelo na vremu i novcu, što nije najčešće prihvatljivo:

1. Najčešće je nemoguće podesiti prototip tako da zadovoljava nefunkcionalne zahteve, kao što su performanse, bezbednost, robusnost i pouzdanost, jer se na njih ne obraća pažnja pri razvoju prototipa.
2. Brze promene u toku razvoja, neophodno dovode do prototipa bez dokumentacije. Postoji samo kod. To nije dovoljno dobro za kasnije održavanje sistema.
3. Promene urađene na prototipu, najčešće kasnije negativno utiču na strukturu sistema. Sistem postaje težak i skup za održavanje.
4. Pri razvoju prototipa obično se ne koriste standardi kvaliteta organizacije.

Prototipovi ne moraju uvek da budu u vidu izvršnog koda. Mogu biti i samo na papiru, kao na primer, u slučaju provere dizajna korisničkog interfejsa, što zнатно ubrzava proces i smanjuje cenu. I interakcija sa korisnikom, može da bude simulirana a ne stvarna, tako da odgovore korisnika prima čovek, a ne sistem.

## KADA I KAKO KORISTITI PROTOTIPOVE?

*Prototipovi se najčešće koriste da bi se proverili i verifikovali zahtevi ili da bise testirala tehnologija koja će se koristiti pri razvoju softvera.*

Ovde ćemo rezimirano do sada izloženo. Najčešće se prototipovi koriste u dva slučaja pri razvoju softvera:

1. **Utvrđivanje zahteva korisnika:** Pokazivanjem prototipa softvera korisniku, daje mu se prilika da potvrdi da li je to što očekuje ili da dopuni svoje zahteve. Tada se često utvrđuju suprostavljeni i nejasni zahtevi.

**2. Procenjivanje tehničkih i arhitektonskih rizika:** Pri projektovanju softvera se najčešće koriste projektni šabloni. Međutim, kod složenih softvera njihova primena nije uvek moguća. Primena prototipa je neophodna da bi se testirala i postavljena arhitektura softverskog sistema. U slučaju primene novih tehnologija, na ovaj način se proverava njena upotrebljivost u slučaju softvera koji se razvija.

Pri razvoju softvera, prototipovi se koriste u dva oblika kao:

- a) model procesa i kao
- b) tehničko rešenje, tj. tehnologija.

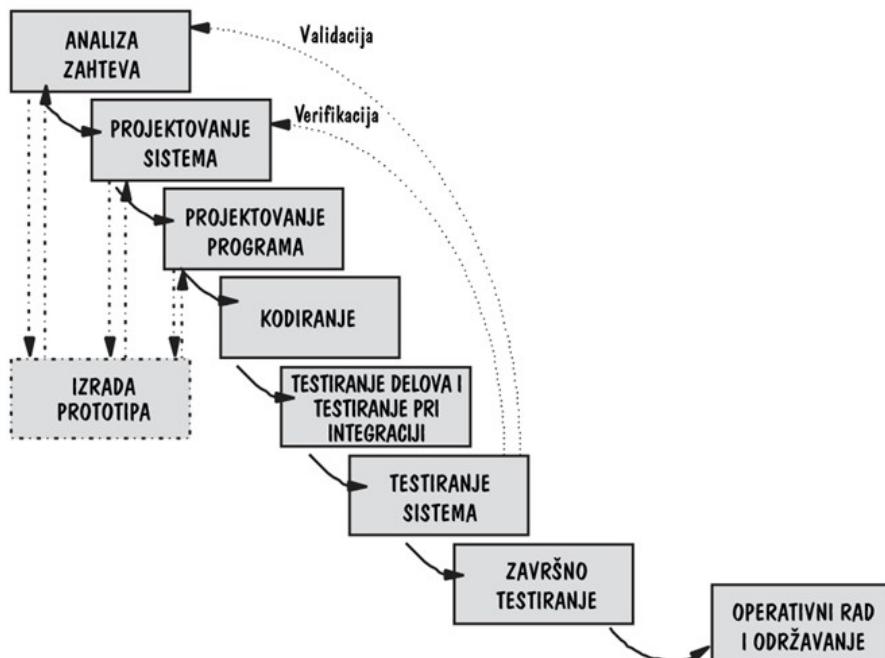
Ako se prototip koristi radi modelovanja procesa, onda se za svaku aktivnost procesa razvoja softvera koriste prototipovi, kako bi se proverilo da li se dobija ono što se želi. Kada se to dobije, prelazi se na sledeću aktivnost, itd. Na kraju, posle poslednje aktivnosti, vrši se isporuka softvera razvijenog na ovakav način.

Ako se, pak, prototip koristi da bi testirala primenjena tehnologija, onda se on koristi unutar nekog od modela procesa. RUP i spiralni model razvoja koriste prototipove unutar svog modela procesa razvoja softvera. Kada se prototipom izvrši provera tehnologije, on se odbacije, i za razvoj softvera se ne koristi više prototip, već se razvija softver za korisnika.

Prototipovi se retko koriste kao modeli procesa, jer su retki slučajevi da su zahtevi potpuno nepoznati ili da su postavljenje arhitekture toliko rizične i nepoznate da ih je potrebno proveriti. Mnogo češće se prototipovi koriste radi provere zahteva i tehnologija.

## PRIMENA PROTOTIPOVA U FAZAMA RAZVOJA SOFTVERA

### *Primena prototipa softvera*



Slika 4.2.2 Izrada prototipa za podršku fazama razvoja softvera

Kreiranje prototipova za sistem podrazumeva:

Throwaway Prototyping

- podrazumeva kreiranje prototipa koji će u dogledno vreme biti odbačen
- nakon preliminarne faze zahteva, konstruiše se jednostavniji radni model sistema radi ilustracije korisniku
- izrada radnog modela različitih delova sistema u ranoj fazi razvoja
- može da se uradi veoma brzo
- testira se i User Interface

Evolutionary Prototyping (Breadboard prototyping)

- izgradnja vrlo robusnog prototipa na strukturiran način i njegovo konstantno usavršavanje
- gradi se samo na osnovu onih zahteva koji su dobro razmotreni i prihvaćeni
- dodavanje novih funkcionalnosti, evaluacija kroz različita operativna okruženja

## EVOLOUCIJA PROTOTIPOVA (VIDEO)

*Primena prototipa softvera - Video klip 1*

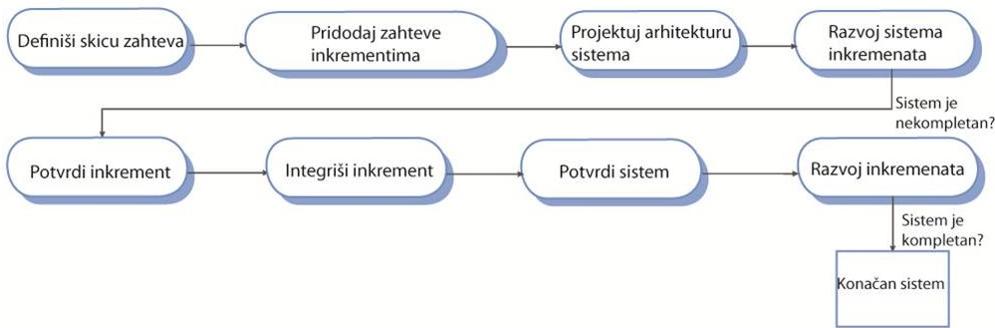
**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

## ✓ 4.2 Inkrementalna isporuka softvera

ŠTA JE INKREMENTALNA ISPORUKA SOFTVERA?

*Inkrement je razvijeni deo sistema kojim se proširuje prethodno razvijeni softver i onda isporučuje kupcu.*

Inkrementalna isporuka softvera je pristup u razvoju softvera koja omogućava da se kupcima softvera isporučuju razvijeni delovi sistema (inkrementi), koji se onda instaliraju i puštaju u operativnu upotrebu. Na taj način, kupci utvrđuju servise koje treba da im obezbedi sistem. Oni određuju prioritetne servise, a koji su im najmanje važni. Onda se definiše određeni broj inkremenata, tako da svaki od njih obezbeđuje određen podskup funkcija sistema. Prvo se razvijaju inkrementi (moduli) sa najvećim prioritetom.



Slika 4.3.1 Inkrementalna isporuka softvera [1.2]

Promene zahteva se prihvataju za planirane, a još nerazvijene inkremente, ali se ne prihvataju zahtevi koji se odnose na tekući inkrement. Zato njegov razvoj mora detaljno da se isplanira.

## PREDNOSTI I NEDOSTACI INKREMENTALNE ISPORUKE SOFTVERA

*Inkrementalni razvoj nudi niz prednosti: kupac ranije dobija softver za prioritetne funkcije i dozvoljava lako definisanje novih zahteva.*

Inkrementalna isporuka softvera nudi niz prednosti:

1. Kupci mogu da koriste prve inkremeante kao prototipove i steknu iskustvo, koje koriste pri definisanju zahteva za kasnije inkremente. Za razliku od prototipova, to su delovi stvarnog sistema, te nema potreba da ponovo uče njegovo korišćenje, kao u slučaju korišćenja prototipova.
2. Kupci ne moraju da čekaju da se razvije ceo sistem da bi im bio isporučen, te ranije dobijaju korist od korišćenja delova sistema. Obično prvi inkrement repava njihove najvažnije zahteve.
3. Proces omogućava relativno lako ubacivanje novih promena u sistem.
4. Kako se prvo isporučuju inkremani sa funkcijama najvećeg prioriteta, to oni i dobijaju i najdetaljnije testiranje, te taj, najvažniji deo sistema onda im najmanje zadaje probleme.

Međutim, inkrementalna isporuka softvera ima i nekih nedostataka:

1. Najveći broj sistema zahtevaju ispunjenje osnovnih funkcija, a koje obezbeđuju pojedini delovi sistema. Kako zahtevi nisu detaljno definisani sve dok se jedan inkrement ne primeni, teško je da se utvrde osnovne, zajedničke funkcije koje će biti realizovane od strane svih inkremenata.
2. Otežan je iterativni razvoj kada se razvija novi sistem. Korisnici žele celu funkcionalnost starog sistema i često nisu voljni da eksperimentišu sa nekompletnim novim sistemom. Zato, teško je dobiti korisne odzive korisnika sistema.
3. Srž iterativnog razvoja je u paralelizmu definisanja specifikacije i razvoja softvera. Međutim, to je u suprotnosti sa praksom ugovaranja i plaćanja kada se ugovara i plaća kompletan sistem. Kod inkrementalnog pristupa, nema specifikacije kompletogn sistema, sve dok se specifikacija ne pripremi i za poslednji inkrement.

To zahteva novu formu ugovora, koje najčešće ne odgovaraju velikim državnim agencijama.

## VELIKI SISTEMI I INKREMENTALNA ISPORUKA SOFTVERA

*Kod nekih velikih softverskoh sistema, inkrementalni pristup razvoju nije i najbolji pristup.*

Kod nekih velikih softverskoh sistema, inkrementalni pristup razvoju nije i najbolji pristup. To je slučaj kada se sistem razvija na više lokacija, ili u slučaju pojedinih ugrađenih sistema kod kojih softver zavisi od razvoja hardvera, i u slučaju kritičkih sistema, kod kojih se svi zahtevi moraju analizirati jedinstveno radi provere interakcija, a u cilju provere bezbednosti i sigurnosti sistema. U ovim slučajevima, treba koristiti proces u kome se koristi iterativno razvijen prototip sistema kao platforma za eksperimentisanje sa sistemskim zahtevima i projektnim rešenjima. Na osnovu tako steklenog iskustva, utvrđuju se konačni zahtevi sistema.

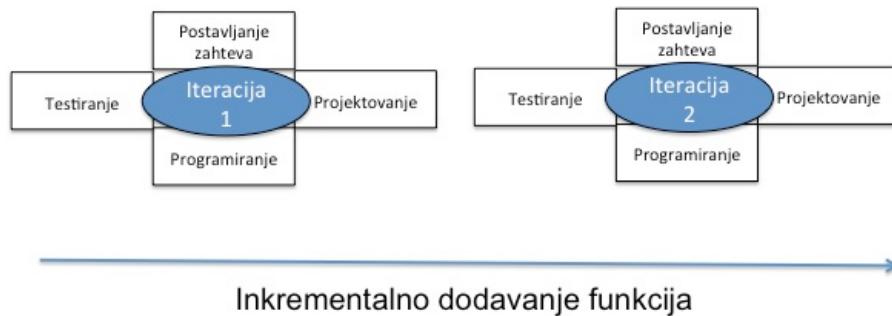
### ▼ 4.3 Agilni model procesa razvoja softvera

## OSNOVNE VREDNOSTI AGILNOG MODELA

*Inkrementalni i iterativni proces se primenjuje primenom specifičnih principa agilnosti: važan je tim, softver koji radi, saradnja sa kupcem i reagovanje na promene.*

Agilni model procesa razvoja softvera se oslanja na inkrementalan i iterativan metod razvoja softvera. Agilni model ima za cilj da omogući: razvoj pouzdanog softvera, i to brzo, eliminisanje nepotrebnih aktivnosti .

U agilnom modelu procesa softver se inkrementalno razvija. Svaki inkrement dodaje novu funkcionalnost softvera. Pri razvoju svakog inkrementa softvera, iterativno se primenjuju uobičajene aktivnosti razvoja softvera: postavljanje zahteva, projektovanje softverskog rešenja, programiranje i testiranje softverskog inkrementa (slika 1). Pri ovome, ne daje se poseban fokus ni na jednu od ovih aktivnosti. One se realizuju delimično i paralelno i uz iterativno ponavljanje, od strane tima programera.



Slika 4.4.1 Agilni model procesa razvoja softvera [1.2]

Agilni model razvoja softvera se oslanja na sledeće vrednosti:

1. *Programeri i njihove interakcije su važniji od procesa i alata.* Važan je složan tim koji sarađuje, te je vrlo važno da se formira.
2. *Softver koji rad je važniji od kompletnosti dokumentacije.* Dokumentacija je važna, ali ako je previše, postaje skupa a teško je redovno ažurirati. Dovoljno je formirati manje detaljnu dokumentaciju.
3. *Saradnja sa kupcem je važnija nego ugovaranje.* Važni su komentari kupca i njihov rad sa timom programera. Na taj način se definišu buduće iteracije.
4. *Reagovanje na promene je važnije od postavljenog plana.* Planovi moraju da se fleksibilno menjaju. Primjenjuje se planiranje na nivoa: detaljan nedeljni plan, plan iteracije, ukupan plan

## PRINCIPI AGILNIH PROCESA

*Vrednosti agilnih procesa se postižu primenom 12 principa agilnih procesa.*

Pri primeni agilnog modela, postavljaju se sledeća pitanja: *Koliko planiranja? Do koje mere treba slediti planiran proces? U kom obimu treba pripremiti dokumentaciju analize i projektovanja kao i generisanisanih modela?* Potrebno je onoliko dokumentacije i procesa koliko je nephodno da bi se zadovoljile navedene vrednosti agilnih modela. Te vrednosti se ostvaruju primenom sledećih principa agilnosti:

- Najveći prioritet se daje zadovoljenju kupca što bržom isporukom softvera, i to stalno u toku procesa njegovog razvoja.
- Prihvataju se promene zahteva u zadnjem momentu. Agilni procesi prihvataju promene radi obezbeđenja konkurentnosti, prednosti kupcu.
- Treba proizvoditi novi softver koji radi u što kraćim vremenskim intervalima, od par nedelja do par meseci.
- Poslovni ljudi i programeri treba da svakodnevno rade zajedno tokom celog trajanja projekta.
- U projekat uključiti motivisane pojedince. Treba im dati potrebno okruženje i podršku koju traže, kao i ukazati im poverenje da će posao uraditi.

- Najefikasniji i najefektniji način razmene informacija unutar tima je direktni razgovor njegovih članova.
- Mere napretka u razvoju je softver koji radi.
- Agilni procesi promovišu održivi razvoj.
- Poklanja se stalna pažnja postizanju tehničke izvršnosti i agilnosti u postizanju dobrog projektnog rešenja.
- Teži se jednostavnim rešenjima, koji zadovoljavaju zahteve korisnika.
- Najbolja arhitektura, specifikacija zahteva, i projektno rešenje se dobija od samoorganizovanog tima. Tim deli odgovornost.
- Povremeno, tim analizira svoj rad analizirajući kako da bude efektniji, te se međusobno usaglašava.

## AGILNI MODELI SOFTVERSKEH PROCESA (VIDEO)

### *Agilni modeli softverskih procesa - video klip 1*

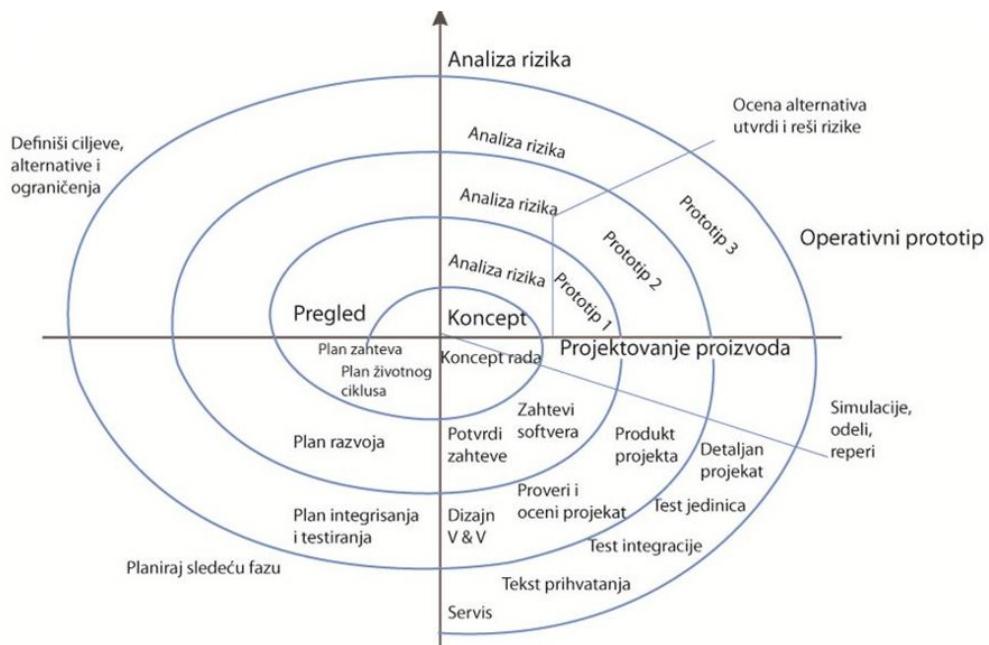
**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

### ▼ 4.4 Spiralni model razvoja

#### ŠTA JE SPIRALNI MODEL RAZVOJA?

*Softverski proces je predstavljen kao spirala pre nego sekvenca aktivnosti. Svaka petlja u spiralnom modelu predstavlja fazu procesa.*

Spiralni model softverskog procesa (pričetan na slici 1) prvobitno je bio predložen od Boehma (1998) i sada je naširoko poznat. Proces je predstavljen kao spirala pre nego sekvenca aktivnosti sa povratnom spregom. *Svaka petlja u spiralnom modelu predstavlja fazu procesa.* Prema tome, krajnja unutrašnja petlja može se odnositi na izvodljivost sistema, sledeća petlja na definiciju zahteva sistema, sledeća petlja na projektovanje sistema, itd



Slika 4.5.1 Spiralni model razvoja softvera [1.2]

## ČETIRI SEKCIJE PETLJE U SPIRALI MODELI

Važna razlika između spiralnog modela i drugih modela softverskih procesa je eksplicitno razmatranje rizika u spiralnom modelu.

Svaka petlja u spirali je podeljena u četiri sekcije kao što je prikazano na slici 3.7. Sekcije spiralnog modela su:

- Definisanje ciljeva** – Definisani su specifični ciljevi za ovu fazu projekta. Identifikovana su ograničenja procesa i proizvoda i skiciran je detaljni plan upravljanja. Identifikovani su rizici. Mogu biti planirane alternativne strategije zavisno od rizika.
- Određivanje i redukcija rizika** – Za svaki od identifikovanih projekata rizika, izvedena je detaljna analiza. Da bi redukovali rizik uvedeni su koraci. Na primer, ako postoji rizik za koji su zahtevi nezadovoljavajući, može biti razvijen prototip sistema.
- Razvoj i validacija** – Nakon proračuna rizika, izabran je razvojni model za sistem. Na primer, ako su rizici korisničkog interfejsa dominantni, odgovarajući razvojni model može biti evolucionarni prototip. Ako su rizici sigurnosti dominantni, može biti pogodan razvoj baziran na formalnoj transformaciji, itd. Model vodopada može biti najpogodniji razvojni model ako je glavni identifikovani rizik integracija pod-sistema.
- Planiranje** – Projekat je razmotren i doneta je odluka da li nastaviti sa narednom petljom spirale. Ako je doneta odluka da se nastavi, crtaju se planovi za narednu fazu projekta.

Važna razlika između spiralnog modela i drugih modela softverskih procesa je eksplicitno razmatranje rizika u spiralnom modelu. Neformalno, rizik je jednostavno nešto što može da krene pogrešno. Na primer, ako je intencija da koristimo nove programske jezike, rizik je da su postojeći kompjajleri nepouzdani ili da ne proizvode dovoljno efikasan objektni kod. Rezultat

rizika u projektnim problemima kao što su raspored i prekoračenje troškova je vrlo važna aktivnost upravljanja projektom.

Ciklus spirale počinje sa razradom ciljeva kao što su performanse, funkcionalnost, itd. Zatim su nabrojani alternativni načini dostizanja ovih ciljeva i ograničenja zadata od strane svake alternative. Svaka alternativa je ocenjena nasuprot svakom cilju. Ovo obično rezultira identifikacijom izvora rizika projekta. Sledeći korak je proračunavanje ovih rizika preko aktivnosti kao što su detaljna analiza, simulacija, itd. Jednom kada su rizici ocenjeni, izvršen je razvoj i ovo je praćeno planiranjem aktivnosti za narednu fazu procesa.

## SVOJSTVA SPIRALNOG MODELAA

*Spiralni model je pogodan za velike i složene projekte, ali je skup i složen za korišćenje.*

Primena spiralnog modela je odgovarajuća za slučaj velikih i složenih projekata, koje prate veliki razici, jer se u spiralnom modelu posebna pažnja poklanja analizi rizika.

Dobro svojstvo spiralnog modela je što *omogućava ikrementalnu isporuku* softvera i što u sebi *uključuje i tehnike korišćenja prototipova*, kako bi se smanjili rizici pri razvoju softvera. Ustvari, *spiralni model podržava korišćenje i inkrementalnog i sekvenčnog modela razvoja, kao i upotrebu modela sa prototipovima*. Ova kombinacija pristupa, čini ovaj model vrlo moćnim.

Međutim, *složenost modela mu je istovremeno i slabost*, jer je složen za upravljanje i njegovu primenu prate visoki troškovi korišćenja, pre svega zbog analiza rizika i korišćenja prototipova u svakoj petlji. Zato, *nije pogodan za primenu u malim i srednjim projektima razvoja softvera ili u slučajevima gde je agilnost procesa vrlo bitna*.

## SPIRALNI MODEL SOFTVERSKOG PROCESA (VIDEO)

*Spiralni model razvoja - Video klip 1*

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

## ▼ Poglavlje 5

# Rational ujedinjeni proces (RUP)

## RUP MODEL

*RUP model se bavi analizom rizika i podržava razvoj koji primenjuje slučajeve korišćenja. Posebnu pažnju posvećuje arhitekturi softvera.*

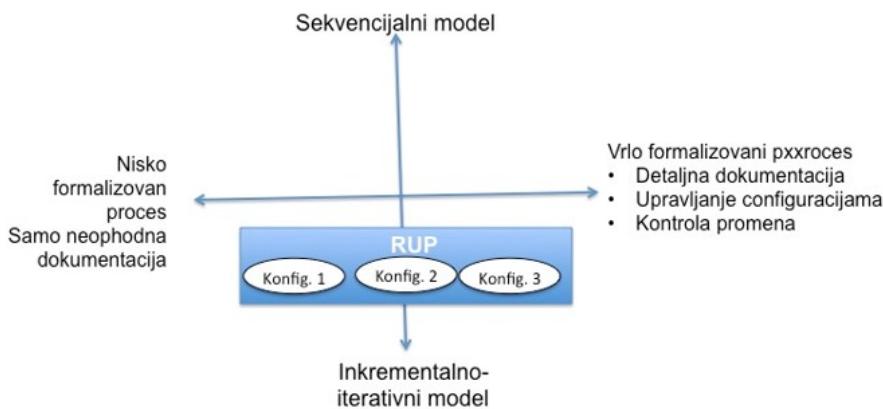
Rational Unified Process ili RUP je model procesa koji se bazira na inkrementalnom i iterativnom proces razvoja softvera. RUP model se bavi analizom rizika i podržava razvoj koji primenjuje slučajeve korišćenja. Posebnu pažnju posvećuje arhitekturi softvera i ona je u centru njegove pažnje.

RUP je takođe i okvir za modeliranje procesa. Omogućuje prilagođavanje procesa potrebama korisnika (process customization) i omogućava definisanje procesa. Na taj način, omogućava dobijanje različitih konfiguracija procesa. Te specifične konfiguracije omogućavaju:

- Podršku razvojnim timovima različite veličine (male, srednje i velike)
- Primenu vrlo formalizovanog ili malo formalizovanog procesa razvoja softvera, tj. metoda razvoja softvera.

Da bi jasnije pozicionirali RUP u odnosu na bitne parametre koje klasificiše različite pristupe u postavljanju procesa razvoja softvera, koristićemo dve ose sa po dve ekstremne vrednosti tih parametara (slika 1).

Na vertikalnoj osi prikazan je metod isporuke razvijenog softvera. Na vrhu je ekstremni slučaj primene metoda vodopada, a na dnu je drugi ekstrem – primena inkrementalnog i iterativnog načina isporuke, tj. razvoja softvera. Na horizontalnoj osi se prikazuje mera formalizacije procesa razvoja softvera (novo proizvedene i korišćene dokumentacije, upravljanja konfiguracijama i kontrole promena u softveru). Na desnoj strani se daje najveći stepen korišćenja projektne i druge dokumentacije, metoda konfigurisanja softvera i kontrole promena u softveru. Na drugom, levom kraju je dat drugi ekstremni slučaj: vrlo malo dokumentacije i velika fleksibilnost u definisanju procesa razvoja.



Slika 5.1 Konfiguracije RUP-a [1.2]

## FAZE SOFTVERSKOG PROCESA U RUP MODELU

*RUP model ima četiri faze softverskog procesa u kome su ove faze povezane sa aktivnostima, koje su su povezane u skladu sa poslom, a ne tehnikom.*

Rational ujedinjeni proces (engl. **The Rational Unified Process** – RUP) je primer modernog modela procesa koji primenjuje UML (engl. **Unified Modelling Language**) za modeliranje procesa. Kao model hibridnog procesa, on sadrži elemente sadržane u svim opštim modelima procesa, i predstavlja dobru praksu pripreme specifikacije i projektnog rešenja softvera, kao i podrške primene prototipova i inkrementalne isporuke softvera.

RUP opisuje proces iz **tri perspektive**:

1. Dinamička perspektiva – pokazuje faze model tokom vremena.
2. Statička perspektiva – prikazuje aktivnosti procesa
3. Perspektiva prakse – sugerše dobre prakse upotrebe modela

Najčešće se primenjuje kombinovanje statičke i dinamičke perspektive u vidu jednog jedinstvenog dijagrama.

RUP model ima **četiri faze softverskog procesa**. Za razliku od modela vodopada, u kome su ove faze povezane sa aktivnostima, kod RUP modela, ove faze su povezane u skladu sa poslom, a ne tehnikom (slika). Naredna faza počinje kada se prethodna završi, ali u okviru svake faze postaje više iteracija. Te faze su:

1. **Početak** : Cilj početne faze je postavljanje poslovnog scenarija (slučaja) sistema, tj. određivanja šta sistem treba da radi. Utvrđuju se spoljni akteri (ljudi i sistemi) koji su interakciji sa sistemom i definišu se te interakcije. *Vrši se procena efekata rada sistema na poslovanje i procena rizika.* Ako su efekti mali, može da dođe i do zaustavljanja projekta.
2. **Razrađivanje** (elaboracija): Cilj ove faze je razvoj razumevanja domena problema, postavljanje arhitektonskog okvira sistema (konceptualno rešenje arhitekture), razvoj plana projekta, i utvrđivanje ključnih rizika projekta. Na kraju ove faze, dobija se

model zahteva sistema u vidu skupa UML slučajeva primene, opis arhitekture i razvijen plan projekta.

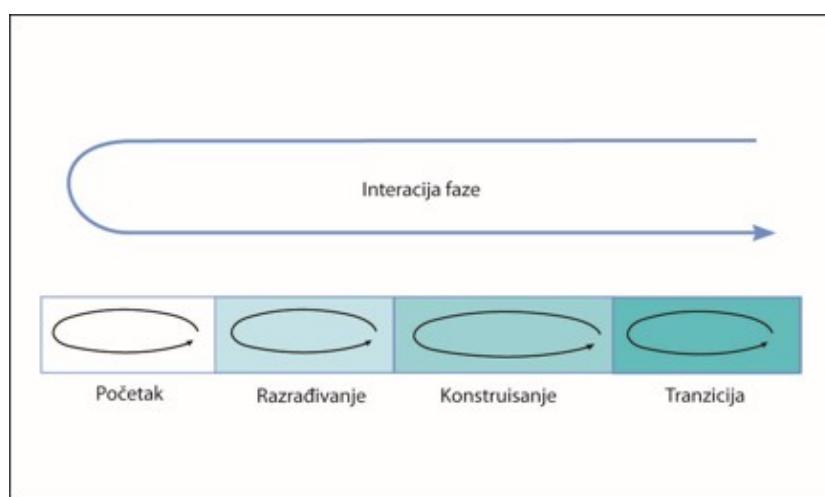
3. **Konstrukcija:** Faza konstrukcije obuhvata projektovanje sistema, programiranje i testiranje. Delovi sistema se paralelno razvijaju u ovoj fazi. Na kraju ove faze, sistem je u radnom stanju, zajedno sa pratećom dokumentacijom, te je spreman za isporuku korisnicima.

4. **Tranzicija:** Konačna faza RUP-a se bavi prenosom sistema iz razvojnog okruženja u korisničko okruženje, i stavlja ga u rad u stvarnom okruženju. Na kraju ove faze, sistem je opremljen kompletnom softverskom dokumentacijom i ispravno radi u operativnom (radnom) okruženju.

## RADNI TOKOVI U RUP MODELU

*Statički pogled na RUP se fokusira na aktivnosti razvojnog procesa. One se nazivaju i radnim tokovima.*

Iteracije u RUP-u se javljaju unutar svake faze, ali i na nivou celog procesa (vraćanje na prethodne faze).



Slika 5.2 Iteracije u RUP modelu [1.2]

Statički pogled na RUP se fokusira na aktivnosti razvojnog procesa. One se nazivaju i radnim tokovima (eng. **workflows**). Postoji šest ključnih radnih tokova procesa i tri radna toka podrške (videti Tabelu 1). Kako je RUP projektovan primenom UML, to je i opis ovih radnih tokova izvršen u vidu odgovarajućih UML modela.

Radni tok	Opis
Modelovanje poslovanja	Poslovni procesi se modeluju primenom UML slučajeva upotrebe (UML use cases)
Zahtevi	Utvrđeni su akteri koji su interakciji sa sistemom i slučajevi upotrebe su razvijeni radi modelovanja zahteva sistema.
Analiza i projektovanja	Kreira se i dokumentuje model projektovanja upotrebom modela arhitektura, modela komponenti, objektnih modela i sekvenčnih modela
Implementacija	Komponente sistema su primenjene i strukturisane u podsistemima. Automatska generacija koda iz projektnog modela ubrzavaju proces.
Testiranje	Testiranje je iterativni proces koji se sprovodi zajedno sa implementacijom. Testiranje sistema se vrši posle završetka implementacije.
Instaliranje	Napravljena je konačna verzija proizvoda, podeljena je korisnicima i instalirana na mestima upotrebe.
Konfigurisanje i upravljanje promenama	Ovaj radni tok upravlja promenama sistema.
Upravljanje projektima	Ovaj tok rada upravlja razvojem sistema.
Okruženje	Ovaj tok rada omogućava da tim za razvoj softvera koristi odgovarajuće softverske alate.

Slika 5.3 Radni tokovi u RUP-u

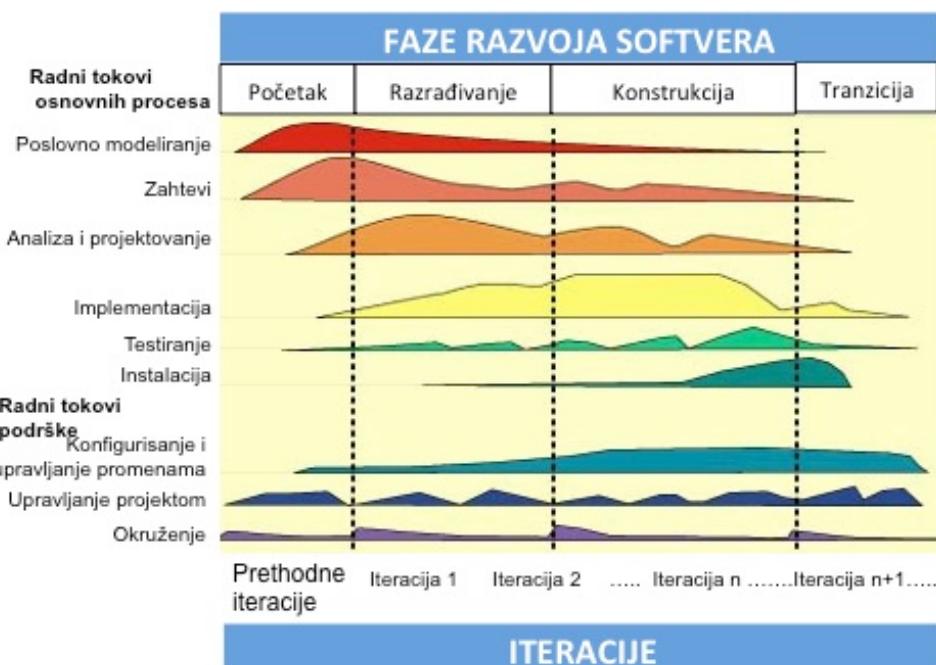
## AKTIVNOSTI RADNIH TOKOVA PROCESA RAZVOJA

*Aktivnosti radnih tokova procesa razvoja, a i procesa podrške, prisutne se u svim fazama razvoja softvera i izvršavaju se sa različitim intenzitetom.*

Na slici 4 su prikazani radni tokovi osnovnih procesa i radni tokovi podrške tokom četiri faze razvoja softvera. Radne tokove osnovnih procesa čini šest radnih tokova, a tri radna toka čine radne tokove podrške. Aktivnosti ovih devet radnih tokova se realizuju u svim fazama, sa različitim intenzitetom. Na primer, u početnoj fazi, najintenzivniji je rad na poslovnom modelovanju (slučajevi upotrebe, scenariji) i na definisanju zahteva. U fazi razrađivanja, naintenzivnije aktivnosti su na detaljnijoj specifikaciji zahteva i na analizi sistema i projektovanju. U fazi konstrukcije softvera, najintenzivniji je rad na implementaciji projektnog rešenja, tj. na programiranju u testiranju. U fazi tranzicije najviše se radi na završnom i korisničkom testiranju.

Aktivnosti radnih tokova podrške se odvijaju tokom celog procesa različitim intenzitetom.

U svima fazama razvoja softvera, kreiraju, se primenom više iteracija, različite verzije softverskih jedinica i konfiguracije celog softverskog sistema.



Slika 5.4 Radni tokovi u RUP modelu [1.2]

## NAJBOLJA PRAKSA U KORIŠĆENJU RADNIH TOKOVA RUP-A

*Najveća inovacija RUP-a je odvajanje faza i radnih tokova, kao i prepoznavanje raspoređivanja (instalacije) softvera u radno okruženje korisnika.*

U principu, svi radni tokovi mogu biti aktivni u svim fazama procesa razvoja. Neki su aktivniji u početnim fazama, neki u završnim fazama. Preporučuje se šest osnovnih najboljih praksi:

1. *Iterativni razvoj softvera:* Planirati inkrementne sisteme na osnovu prioriteta kupca.
2. *Uređivanje zahteva:* Jasno definisati zahteve kupaca i beležite promene tih zahteva. Analizirati posledice usvajanja novih zahteva na sistem, pre nego što se prihvate.
3. *Upotrebiti arhitekturu baziranu na komponentama:* Strukturisati arhitekturu sa komponentama.
4. *Vizuelno modelovati softver:* Upotrebiti UML modele za predstavljanje statičkih i dinamičkih pogleda na softver.
5. *Proveriti kvalitet softvera:* Obezbediti proveru da li softver zadovoljava standarde kvaliteta organizacije.
6. *Kontrolisati promene u softveru:* Uređivati promene u softveru upotrebom sistema za upravljanje promena sistema i alata i procedure upravljanja konfiguracijom.

RUP nije pogodan za sve sisteme, kao što su na primer, ugrađeni sistemi. Najveća inovacija RUP-a je odvajanje faza i radnih tokova, kao i prepoznavanje raspoređivanja (instalacije) softvera u radno okruženje korisnika kao deo procesa. Faze su dinamičke i imaju svoje ciljeve. Radni tokovi su statički i predstavljaju tehničke aktivnosti koje nisu povezane sa

pojedinačnom fazom, već se mogu upotrebiti za vreme celog razvoja radi ostvarivanja ciljeva svake faze.

RUP je okvir koji vam omogućava da izaberete model procesa koji najviše odgovara specifičnostima softvera koji razvijate, jer možete birati stepen formalizma procesa.

## RATIONAL UNIFIED PROCESS - RUP (VIDEO)

*RUP - Video klip 1*

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

## VIDEO PREDAVANJE ZA OBJEKAT "RATIONAL UJEDINJENI PROCES (RUP)"

*Trajanje video snimka: 28min 8sek*

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

## ▼ Poglavlje 6

# Specijalizovani modeli

## ŠTA SU SPECIJALIZOVANI MODELI?

*Specijalizovani modeli neke specifičnosti vezane za upotrebu specijalnih tehnika i metoda kojim se rešavaju njihovi specifični problemi.*

Specijalizovani modeli se razvijaju za specifične vrste projekte razvoja softvera, tj. za specifične vrste softvera. Specijalizovani modeli imaju karakteristike procesno-orientisanih modela za razvoj softvera, ali imaju i neke specifičnosti vezane za upotrebu specijalnih tehnika i metoda kojim se rešavaju njihovi specifični problemi.

Zavisno od njihovih specifičnosti, specijalizovani model se mogu klasifikovati u nekoliko grupa, kao na primer:

1. Modeli sa komponentama
2. Formalni metodi
3. Razvoj u skladu sa aspektom korišćenja softvera

## METODI SA KOMPONENTAMA

*Metodi sa komponentama koriste unapred razvijene softverske komponente čijom se integracijom kreira softverski sistem.*

Modeli sa komponentama se koriste u slučajevima kada se razvoj softvere zasniva na upotrebi komponenti čijim povezivanje se kreira željeni softverski sistem. Softverske komponente su najčešće ranije razvijene ili kupljene kako bi se koristile kada bude potrebe za njima pri razvoju softvera. Te komponente su razvijene s namerom da se višestruko koriste, tj. koriste u različitim projektima razvoja softvera. Svaka komponenta se razvija sa određenom funkcionalnošću, tako da se ona koristi ako je ta funkcionalnost potrebna u softvreskom sistemu koji se razvija.

Ako ne postoje ranije razvijene komponente sa potrebnom funkcionalnošću, onda se razvijaju nove sa tim funkcionalnostima, i sa namerom da budu tako razvijene da mogu da se kasnije ponovo koriste u drugim projektima razvoja softvera.

Arhitektura softverskog sistema koji se razvija uz pomoć komponenata je orijentisana na integraciju softverskih komponenata. Pri tome, važnu ulogu imaju interfejsi svake komponente koje definišu poruke koje mogu da primaju i šalju komponente, tj. sadrže metode koje mogu da pozivaju i koje izvršavaju pojedine funkcije za koje su razvijene komponente.

Testiranje softvera koji koristi komponente se vrši analizom poruka poruka koje one razmenjuju, a u skladu sa njihovim interfejsima.

Primena softverskih komponenti u razvoju softvera se može objasniti analogijom sa primenom Lego kockica. Svaka komponenta je lego kockica. Kombinacijom potrebnih lego kockica, razvija se traženo rešenje, tj. softverski sistem koji čine odabrane softverske komponente.

## FORMALNI METODI

*Formalni metodi koriste formalne matematičke specifikacije za definisanje zahteva i omogućavaju automatsko projektovanje, generisanje koda i testiranja*

Formalni metodi upotrebljavaju formalne matematičke specifikacije. Zahtevi se definišu matematičkim jednačinama sa precizno definianim rečnikom, sintaksom i semantikom. Verifikacija (provera) zahteva, projektovanje softvera i generisanje koda se vrši potpuno automatski. Takođe, i sprovodenje testova je takođe automatizovano.

Dobro svojstvo formalnih metoda je što su nemogući dvosmisleni zahtevi, kao što je to moguće kod neformalnih metoda.

Formalni metodi su pogodni za razvoj softvera za rad u realnom vremenu, koji obično imaju kritične ciljeve (misije), kao što su to sistemi koji rade u kontrolama letenja i dr. Međutim, primena formalnih metoda zahteva puno vremena, te zahtevaju velike troškove u primeni. Takođe, zahteva od programera puno specifičnog znanja, koje obično nedostaje. Iz ovih razloga, retko se koriste.

## ASPEKTNO ORIJENTISANI RAZVOJ

*Aspektno orijentisani razvoj obezbeđuje metodologiju za specificiranje, projektovanje, i razvoj ovih aspekata (Aspects).*

Postoje softverski sistemi koji zahteva neku specifičnu funkciju, ili komponentu, ili uslugu. O tome se mora posebno voditi računa prilikom razvoja ovakvog softvera. Mora se voditi računa o tom **aspektu** softverskog sistema. Te specifičnosti su najčešće nefunkcionalni zahtevi vezani za bezbednost, keširanje ili upravljanje transakcijama.

Aspektno orijentisani razvoj definiše ove specifičnosti kao aspekte - **Aspects**. **Aspektosko orijentisani razvoj obezbeđuje metodologiju za specificiranje, projektovanje, i razvoj ovih aspekata (Aspects)**. Kako su oni izolovani, tj. posebno razvijani i višestruko korišćeni, održavani, mogu se uklapati u sistem .

## ▼ Poglavlje 7

# Izbor modela procesa razvoja softvera

## KRITERIJUMI IZBORA MODELA PROCESA

*Izbor modela procesa zavisi od učestalosti promena zahteva, složenosti arhitekture softvera i veličine projekta.*

Da bi se razvio dobar softver, potrebno je primeniti odgovarajući proces razvoja softvera. Postoji više različitih modela procesa razvoja softvara. Koji izabrati?

Ne postoji jedno univerzalno rešenje, tj. jedan proces koji je optimalan za sve vrste softvera. Na softver inženjerima je zadatak da na osnovu analize specifičnosti softvera koji treba da razviju, i poznavanja specifičnosti svih poznatih modela procesa, izaberu jedan koji najviše odgovara softveru koji treba da razviju. Taj proces treba da obezbedi da projektni tim dobije:

- odgovarajući softver, tj. softver koji zadovoljava potrebe kupca, tj. korisnika,
- koji ima potreban nivo kvaliteta,
- koji dobija u okviru planiranog budžeta i u planiranom roku.

Svaki model procesa je odgovarajući za određene scenarije i vrste softvera. Da bi organizacija koja se bavi razvojem softvera mogla da izabere pravi proces, potrebno je da definiše svoje kriterijume za izbor modela procesa koji najviše odgovara tipu softvera koji treba da se razvije.

Na osnovu iskustva, uspostavljene su izvesne preporuke za izbor procesa razvoja softvera. Jedan od jednostavnijih metoda izbora modela procesa uzima u obzir tri faktora:

1. Brzinu promena zahteva
2. Složenost arhitekture softvera
3. Veličina projekta izražena u broju potrebnih čovek-dana, veličini tima).

Na slici 1 je prikazan primer postavljenih kriterijuma za izbor modela procesa u jednoj organizaciji.

Brzina promene zahteva	Složenost arhitekture	Veličina projekta	Preporučeni model procesa
Velika	visoka/srednja	srednji/veliki	RUP sa nisko formalizovanim procesom
Mala	visoka/srednja	srednji/veliki	RUP sa srednje formalizovanim procesom
Velika	visoka/srednja	mali	Agilni modle
Velika	Niska	mali/srednji	Agilni model
Velika	Niska	veliki	RUP sa nisko formalizovanim procesom

Slika 7.1 Primer kriterijuma za izbor modela procesa [1.2]

## IZBOR MODELAA PROCESA I BUDUĆI SOFTVERSKI MODULI

*Koncept razvoja budućih softverskih modula ostaje isti, bez obzira od izabranog modela procesa razvoja.*

Postavlja se pitanje: Da li izbor modela procesa razvoja softvera utiče na module softvera koje ćemo u budućnosti razvijati? Da bi odgovorili na ovo pitanje, najpre ćemo naglasiti da se opšte aktivnosti razvoja softvera (specifikacija zahteva, projektovanje softvera, programiranje, testiranje, instalacija i evolucija) postoje u svim modelima procesa. Modeli procesa ove aktivnosti realizuju na svoj, specifični način, ali one postoje. Modeli procesa određuju tok aktivnosti, organizaciju razvojnog tima, uloge i odgovornosti njegovih članova, i granularnost (veličinu) softverskog dela koji se razvija. Iz ovde iznetog, može se zaključiti da su budući softverski moduli (i primenjene aktivnosti razvoja i metodi) nezavisni od izabranog modela procesa. Mogu se razlikovati redosledi izvršenja razvojnih aktivnosti, njihov tok, odgovornosti i uloge članova tima, kao i veličina softverske jedinice koja se razvija, ali se sam koncept razvoja softvera ne menja.

Na primer, aktivnost *Specifikacija zahteva*, uvek je prisutna. U sekvensijalnom modelu vodopada, ona se u celosti završi pre nego što se pređe u sledeću fazu razvoja. U slučaju RUP modela, specifikacija zahteva se odvija u svim fazama. Slično je sa aktivnošću projektovanja softvera i sa drugim.

U slučaku agilnih modela procesa, tj. agilnih metoda razvoja, u svakoj iteraciji se primenjuju sve glavne aktivnosti razvoja softvera (specifikacija zahteva, projektovanje softvera, programiranje i testiranje), samo što je uklonjena oštra granica među njima, te se one ne odvijaju određenim redosledom. Projektni tim iterativno radi na razvoju jednog inkrementa, bez posebnog naglaska na neku aktivnost razvoja softvera i bez posebno definisanog redosleda odvijanja tih aktivnosti. Umesto linearног, primenjuju nelinearni model odvijanja aktivnosti procesa, tako da u uskoj međusobnoj kolaboraciji, obavljaju neophodne razvojne aktivnosti, manje-više simultano.

Znači, koncept razvoja softvera baziran na navedenim osnovnim aktivnostima ostaje isti, samo se menjaju načini primene tih aktivnosti. Izbor modela procesa utiče na tok aktivnosti i na poseban naglasak na neke od njih.

## PREGLED PREPORUKA ZA UPOTREBU UOBIČAJENIH MODEL A PROCESA

*Model procesa se bira u odnosi na vrstu softvera koji se razvija.  
Međutim, osnovne aktivnosti razvoja su u svima njima prisutne, samo na različite načine.*

U sledećoj tabeli, daje se pregled osnovnih svojstava i preporuka za upotrebu uobičajenih modela procesa.

Model procesa	Osnovna svojstva i preporuke za primenu
Model vodopada	Stistematski i sekvensijalni prist prozusu. Nije dobar kada se zahtevi često menjaju. I kada kupac želi da učestvuje u razvoju. Retko se koristi jer se traže više evolutivni modeli procesa razvoja softvera.
Interaktivni/ inkrementalni	Pruža osnovu za evolutivne modele procese (npr. Agilni, RUP). Obezbeđuje dodavanje novih inkremenata, koji se iterativno poboljšavaju.
Prototipovi	Koriste se da bi se testirao izabrani koncept ili proces. Retko se koristi kao za modelovanje celog procesa. Najčešće se koristi u okviru drugih modela (npr. RUP, spiralni)
Spiralni	Voden je analizom rizika. Primjenjuje više iteracija, tj. petlji koje predstavljaju fazu razvoja. Dosta koristi prototipove za analizu rizika. Primjenjuje se kod velikih i složenih projekata.
Agilni	Pogodan za manje projekte. Primjenjuje postavljene vrednosti i principe. Primjenjuje timski rad, saradnju sa kupcem, kontinualnu isporuku softvera, i reaguje na promene. Obezbeđuje brz, inkrementalni i iterativni razvoj, sa niskim troškovima režije.
RUP	Voden je rizicima i slučajevima korišćenja. Arhitektura je na centralnom mestu. Kao okvir za procese, omogućava primenu različitih stepene formalizacije procesa. Koristi 4 faze i za svaku se utvrđuje rizik. Naglasak je na postavljanju slučajeva upotrebe, projektovanje i arhitekturu. Prilagođiv potrebama projektu. Nije pogodan za ugradene sisteme.
Specijalizovani	Primenjuju specifične metode ili tehnike za odredene probleme. Model sa komponentama se koristi za uobičajene sisteme. Formalni metodi se koriste za osetljive sisteme (npr. navigacija aviona). Aspektno orijentisan razvoj izouljuje određeni aspekt softvera i upravlja razvojem u odnosu na njega.

Slika 7.2 Tabela-2 Pregled najčešćih modela procesa i preporuke za njihov izbor [1.2]

## IZBOR MODEL A SOFTVERSKOG PROCESA (VIDEO)

*Izbor modela procesa razvoja softvera - video klip 1*

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

## KLASIČNE GREŠKE PRI IZBORU SOFTVERSKOG PROCESA (VIDEO)

*Izbor modela procesa razvoja softvera - video klip 7*

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

## ✓ Poglavlje 8

### Pokazna vežba

#### IZBOR MODELA SOFTVERSKOG PROCESA - SISTEM RESTORANA (20 MINUTA)

*Na osnovu studije slučaja iz lekcije 1, predlaže se model softverskog procesa korišćenjem RUP metodologije.*

RUP model sistemu poslovanja restorana koji je predložen u prethodnoj lekciji treba da obezbedi da se razvije od biznis modela (prikazan kroz viziju sistema u lekciji 1) do modela analize i dizajna.

Korišćenjem RUP procesa, primenjuju se sledeće faze u razvoju:

1. **Početak** - inicijalna evaluacija se sprovodi u cilju utvrđivanja da li je projekat vredan pažnje, kreira se biznis model, definisan kroz viziju sistema i prikaz komponenti u prethodnoj lekciji. Poslovni plan je detaljno definisan, a treba definisati i procenu troškova razvoja i rasporeda. Postignut je dogovor o obimu projekta sa svim zainteresovanim stranama.
2. **Razrađivanje** - u toku razrade, vrši se detaljnija evaluacija, kreira se plan razvoja i ublažavaju ključni rizici. Razvojni tim piše 80% svih slučajeva korišćenja, stvara sistemsku arhitekturu i plan razvoja.
3. **Konstrukcija** - u toku izgradnje, stvara se softverski sistem - kod je napisan i testiran. Tim takođe testira rezultujući softver. Ključni izlaz ove faze je operativni softver
4. **Tranzicija** - u fazi tranzicije, softver se izdaje krajnjem korisniku. Aplikacija se šalje na store. Kada se prijava prihvati, projekat je formalno objavljen, a tim nastavlja da ga održava.

RUP model se bavi analizom rizika i podržava razvoj koji primenjuje slučajeve korišćenja. Posebnu pažnju posvećuje arhitekturi softvera i ona je u centru njegove pažnje. Koristi Unified Modeling Language (UML) kao osnovu za razvoj proizvoda. Kao i RUP, UML sadrži oznake koje svaki programer razume i može primeniti za modeliranje unutar životnog ciklusa softverskog inženjerstva.

Kroz naredne lekcije, studija slučaja će biti obrađena poštujući sva pravila RUP modela od faze 2 do faze 4.

# RUP MODEL SISTEMA RESTORANA - FAZA 1 - POČETAK

*Metode i postupci primjenjeni u fazi 1 sistema restorana.*

Tokom razvoja sistema restorana, vodiće se evidencija o specifikaciji softvera kroz različita poglavlja i opise, što će u svakoj lekciji biti prikazano. Svaka promena dokumenta treba da se evidentira kroz log promena kao što je definicano na slici 1

## PROMENE I VERZIJE

Dokument	Vezije	Datum	Autor
Specifikacija softvera	1.0	05.09.2018	P.Petrović
Specifikacija softvera	1.1	11.09.2018	M.Marković

Slika 8.1 Log promena specifikacije softvera [Izvor: Autor]

Da bi projekat nesmetano mogao da se razvija definisaćemo komunikacioni plan projekta i prikazati sva dokumenta relevantna za projekat, sa definisanom frekvencijom promene (Slika 2)

## KOMUNIKACIONI PLAN PROJEKTA

U ovom poglavlju daje se pregled tabele za komunikaciju u okviru projekta koja se koristi da biste identifikovali komunikacijske dokumente potrebne za vaš projekat, primaocu dokumenta, lica odgovorna za kreiranje i ažuriranje dokumenta i informacije o tome koliko često treba dokumente ažurirati.

Primer komunikacionog plana projekta:

Dokument	Zadužena osoba	Frekvencija promene
Specifikacija softvera	P. Petrović, M. Marković	nedeljno

## STRUKTURA TIMA

Ovo poglavlje Identificuje ključne uloge članova tima i normalne obrasce komunikacije između uloga. Po potrebi možete kreirati dijagram ili tabelu kako biste ilustrirali komunikacijske odnose.

## CILJEVI TIMA

- Dobra komunikacija
- Kvalitet praćenja i kontrole projekta

## TEHNIČKE KARAKTERISTIKE

U sledećoj tabeli se prikazuju timski ciljevi, vođstvo tima i timske uloge.

Naziv tima	Ciljevi	Lider u timu
Dizajn	Kreiranje interfejsa visokog nivoa	A.Savić
Projektanti	Kreiranje detaljnog plana projekta sa preciznom dokumentacijom	M. Marković
Programeri	Kreiranje aplikacija po unapred definisanim zahtevima	M. Mitić

Slika 8.2 Komunikacioni plan projekta

# RUP MODEL SISTEMA RESTORANA - FAZA 1 - BIZNIS MODEL

*Biznis model sistema restorana.*

Biznis model - restoran		Datum:	Verzija
		11.09.2018.	1.1
<b>Korisnici</b> Gost restorana Konobar Supervizor Kuvar	<b>Ključne aktivnosti</b> Priprema hrane. Nabavka Skladištenje namirnica Kontora i revizija naručivanja i isporuke hrane. Naručivanje hrane. Plaćanje. Kontrola procesa.	<b>Relacije između korisnika</b> Supervizor ima sve opcije ostalih korisnika: gosta, konobara i kuvara.	<b>Finansijski segment</b> Faza 1 - Početak - 20% ukunog budžeta Faza 2 - Razrađivanje - 30% ukunog budžeta Faza 3 - Konstrukcija - 40% ukunog budžeta Faza 4 - Tranzicija - 10% ukunog budžeta
	<b>Ključne komponente</b> Restoran Kuhinja Administracija Skladište Isporuka i prijem Banka Dobavljači		

Slika 8.3 - Biznis model restorana [Izvor: Autor]

## IZBOR MODELA SOFTVERSKOG PROCESA - BANKARSKI SISTEM (15 MINUTA)

*Hibrid modela softverskog procesa je pogodan za bankarski sistem*

Tradicionalno se softver u finansijskom sektoru razvija primenom **modela vodopada**. Dobro definisane faze razvoja i predvidljivost privlače kompanije koje se bave finansijama da izaberu baš ovakav model. Povećana interakcija krajnjih korisnika sa finansijskim sistemima dovodi do **češćih promena** u zahtevima, kao i do potrebe za sve **bržom isporukom** softvera zbog sve veće konkurenkcije na tržištu.

U ovakvoj situaciji moguće rešenje je kombinovanje tradicionalnog modela vodopada sa **inkrementalnim modelima**. Na taj način se dobija **hibridni** model softverskog procesa. Delovi sistema koji su najpodložniji promenama i koji najviše interaguju sa krajnjim korisnikom se razvijaju inkrementalno, a ostatak sistema modelom vodopada.

U našem primeru jednostavnog bankarskog sistema, softver za bankomat bi se razvijao modelom vodopada, a mobilna aplikacija inkrementalno. Pored toga, model vodopada je najpogodniji za centralni bankarski sistem. Softver za bankomat ima jasno definisane zahteve koji se retko menjaju, a pored toga je potrebno da softver bude veoma bezbedan i pouzdan. Mobilna aplikacija mora dosta češće da se prilagođava novim zahtevima korisnika, kao i novim trendovima na tržištu.

# IZBOR MODELA SOFTVERSKOG PROCESA ZA SISTEM ZA ONLINE IZNAJMLJIVANJE HOTELSKOG SMEŠTAJA (15 MINUTA)

*Sistem za online iznajmljivanje hotelskog smeštaja*

## **Opis softverskog sistema:**

Sistem omogućava rezervaciju hotelskog smeštaja (soba, apartmana, noćenja sa doručkom, polupansion, punansion i sl.), iznajmljivanje vikendica, vila, bungalova, odnosno rezervaciju kompletne ponude fizičkih lica koja postavljaju svoju ponudu.

Sistem treba da poseduje:

- mogućnost rezervacije smeštaja uz mogućnost online plaćanja,
- formu koja izračunava dostupnost smeštaja i cene ponuđenog smeštaja u skladu sa zahtevima klijenta,
- kontrolu ponuđenog smeštaja preko admin panela posebno za fizička lica koja postavljaju neku ponudu, a posebno za administratora celokupnog sistema.

## **Model softverskog procesa:**

Model koji će se koristiti prilikom izrade ovog softverskog sistema je model vodopada. Zahtevi su jasno definisani što omogućava da koraci idu sekvencijalno.

Prvi korak je analiziranje kao i definisanje svrhe ovog softvera, određivanje stepena izvodljivosti realizacije definisanih zahteva, kao i jasno definisanje samih zahteva. Nakon toga sledi projektovanje sistema i softvera u kome se zahtevi raspoređuju svim komponentama sistema i uspostavlja se arhitektura sistema. Zatim sledi implementacija softvera odnosno njegovih programskih jedinica. Paralelno sa implementacijom razvijaju se i testovi svih programskih jedinica da bi se utvrdilo da li svaka programska jedinica ispunjava odgovarajuće funkcije. Nakon toga sledi korak u kome se sve programske jedinice integrišu u jednu i onda se vrši završno testiranje. Ukoliko je testiranje uspešno proteklo, sistem se isporučuje kupcu. Poslednja faza u izradi softverskog sistema je operativni rad i održavanje. Pod ovim pojmovima se podrazumeva rad na poboljšanju softvera nakon njegovog plasiranja. Takođe ova faza služi i za otkrivanje nekih grešaka koje nismo uspeli da otkrijemo tokom faza u kojima smo testirali softver.

## ALATI ZA RAZVOJ SOFTVERA (VIDEO)

*Model softverskog procesa - video klip 3*

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

## ▼ Poglavlje 9

### Individualna vežba

#### ZADACI ZA INDIVIDUALNI RAD - ZADACI OD 1 DO 3

*Definisanje modela softverskog procesa za predložene sisteme.*

**Zadatak 1** (10 minuta)

Pored RUP, da li postoji još neki model softverskog procesa koji bi bio pogodan za razvoj sistema za restoran? Koji od obrađenih modela softverskog procesa nikako nije pogodan za ovakav sistem?

**Zadatak 2** (15 minuta)

Na osnovu pokaznog primera sistema banke, pored tradicionalnog modela vodopada koji se koristi za razvoj centralnog sistema banke, a koji je pogodan i za softver bankomata, koji konkretni modeli bi mogli da se upotrebe za razvoj mobilne aplikacije banke? Obrazložiti odgovor i opisati sve aktivnosti softverskog procesa.

**Zadatak 3:** (60 minuta)

Predložite i obrazložite koji model opštег softverskog procesa najviše odgovara za upotrebu, za razvoj sledećih sistema:

- Sistem za blokiranje automobila prilikom pokušaja krađe.
- Sistem za praćenje treninga i ishrane sportista.
- Sistema za online učenje.
- Interaktivni sistem za planiranje putovanja koji omogućava korisnicima planiranje svojih putovanja sa minimalnim uticajem promena.

Definišite i opišite zadatke svake od aktivnosti koja treba da se primeni u modelu koji ste predložili.

#### ZADACI ZA INDIVIDUALNI RAD - ZADACI OD 4 DO 6

*Definisanje aktivnosti softverskog procesa za predložene sisteme.*

**Zadatak 4** (20 minuta)

Za sistem automatskog pilota aviona, definišite komponente sistema i nakon toga, primenom modela vodopada, opišite i definišite zadatke svake od aktivnosti navedenog softverskog procesa.

**Zadatak 5** (20 minuta)

Primenom agilnog modela, definisati aktivnosti softverskog procesa za prostorno planiranje primenom sistema virtualne (augmented) realnosti koja u zavisnosti od dimenzija prostorije može da vrši različite vrste uređivanja enterijera. ali i planiranje izgradnje na određenoj površini. Navedite alternativni model za razvoj navedenog sistema.

## ✓ Zaključak

### POUKE OVE LEKCIJE

*Softver ne nastaje trenutno, već procesom koji obuhvata niz aktivnosti, kojim se obezbeđuje ispunjenje postavljenih zahteva.*

1. Softverski procesi su aktivnosti koje se realizuju pri razvoju softverskog sistema. Modeli softverskih procesa su apstraktna predstavljanja ovih procesa.
2. Opšti modeli procesa opisuju organizaciju softverskih procesa. Primeri ovih opštih modela su: model vodopada, inkrementalni razvoj i razvoj korišćenjem višestruko upotrebljivih komponenata.
3. Inženjerstvo zahteva je proces razvoja specifikacije softvera. Specifikacije povezuju potrebe kupaca i razvojni tim softvera.
4. Procesi projektovanja i implementacije vrše transformaciju zahteva u izvršni softverski sistem. Sistematski metodi projektovanja mogu se upotrebiti za deo ovih transformacija.
5. Evolucija softvera se javlja zbog promena softverskog sistema tokom svog rada, a zbog zadovoljavanja novih zahteva korisnika. Promene su stalne i softver se mora menjati da bi ostao koristan.
6. Procesi treba da sadrže i aktivnosti koje se bave promenama. To može da obuhvati fazu izrade prototipa, koja može da pomogne u izbegavanju donošenja loših odluka u vezi zahteva i projektnog rešenja. Procesi mogu biti strukturisani za iterativni razvoj i isporuku, tako da se promene mogu realizovati bez ugrožavanja celine sistema.
7. Rational ujedinjeni proces (RUP) je moderan opšti model procesa koji je organizovan u fazama (početak, razrađivanje, konstrukcija i tranzicija), i aktivnostima (zahtevi, analiza, projektovanje i dr) u ovim fazama.

### LITERATURA

*Preporučena literatura*

#### **1. Obavezna literatura:**

1. Onlajn nastavni materijal na predmetu SE201 Uvod u softverski inženjering, školska 2020/21, univerzitet Metropolitan
2. Ian Sommerville, Software Engineering, Tenth Edition, Pearson Education Inc., 2016.

#### **2. Dopunska literatura:**

1. T.C.Lethbridge, R. Lagariere - Object-Oriented Software Engineering - Practical Software Development using UML and Java - 2005
2. B. Bruegge, A. Dutoit, Object-Oriented Software Engineering – Using OML, Patterns, and Java, Thirth Edition, Prentice Hall, 2010

3. P. Stevens, Using UML – Software Engineering with Objects and Components, Second Edition, Assison-Wesley, Pearson Eduction, 2006
4. R. Pressman, Software Engineering – A Practioner's Approach, Seventh Edition, McGraw Hill Higher Education, 2010
5. Partha Kuchana, Software Architecture Design patterns in Java
6. SWEBOK V3.0, Guide to the Software Engineering Body of Knowledge, Editores: Pierree Burque i Richard E. Fairley, IEEE Computer Society, 2014 ([www.swebok.org](http://www.swebok.org))

### **3. Veb lokacije :**

1. <http://www.uml.org/>
2. <http://www.netobjectives.com/resources/books/design-patterns-explained>



## SE201 - UVOD U SOFTVERSKO INŽENJERSTVO

Projektovanje arhitekture  
softverskog sistema

Lekcija 07

PRIRUČNIK ZA STUDENTE

# SE201 - UVOD U SOFTVERSKO INŽENJERSTVO

## Lekcija 07

### ***PROJEKTOVANJE ARHITEKTURE SOFTVERSKOG SISTEMA***

- ▼ Projektovanje arhitekture softverskog sistema
- ▼ Poglavlje 1: Arhitektura sistema
- ▼ Poglavlje 2: Odluke u vezi arhitekture
- ▼ Poglavlje 3: Projektovanje arhitekture sistema
- ▼ Poglavlje 4: Arhitektonski šabloni
- ▼ Poglavlje 5: Arhitekture aplikacije
- ▼ Poglavlje 6: Projektna dokumentacija
- ▼ Poglavlje 7: Preporuke za dobro projektno rešenje
- ▼ Poglavlje 8: Pokazna vežba
- ▼ Poglavlje 9: Individualna vežba
- ▼ Zaključak

Copyright © 2017 – UNIVERZITET METROPOLITAN, Beograd. Sva prava zadržana. Bez prethodne pismene dozvole od strane Univerziteta METROPOLITAN zabranjena je reprodukcija, transfer, distribucija ili memorisanje nekog dela ili čitavih sadržaja ovog dokumenta., kopiranjem, snimanjem, elektronskim putem, skeniranjem ili na bilo koji drugi način.

Copyright © 2017 BELGRADE METROPOLITAN UNIVERSITY. All rights reserved. No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise, without the prior written permission of Belgrade Metropolitan University.

## ✓ Uvod

# UVOD

### *Cilj ove lekcije*

Cilj ove lekcije jeste da vas upozna sa konceptom arhitekture softvera i projektnim rešenjem arhitekture. Ova lekcija vam omogućava da:

- razumete zašto je projektno rešenje arhitekture softvera važno;
- razumete odluke koje treba doneti o arhitekturi sistema za vreme procesa projektovanja arhitekture;
- se upoznate sa idejom arhitektonskih mustri/šablonu (eng., patterns), tj. oprobanog načina organizovanja arhitekture sistema, a koje se mogu upotrebiti u projektima sistema;
- saznote arhitektonske mustre softvera koje se često koriste kod primenjenih sistema različitog tipa, uključujući sisteme za obradu transakcija i sisteme za obradu jezika.

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

## ✓ Poglavlje 1

### Arhitektura sistema

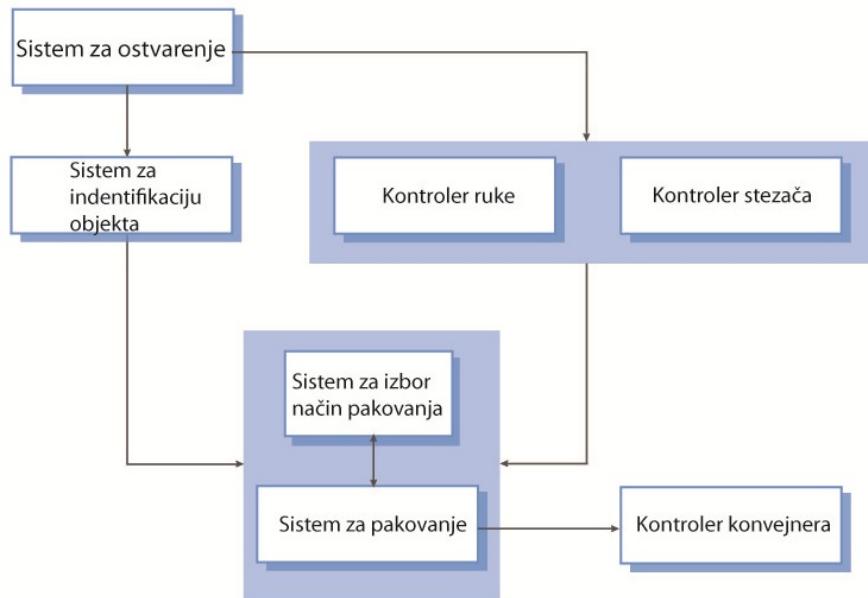
#### ŠTA JE ARHITEKTURA SOFTVERA?

*Arhitektura softvera ukazuje nam kako bi sistem trebalo da bude organizovan i prikazuje ukupnu strukturu sistema.*

Arhitektura softvera ukazuje nam kako bi sistem trebalo da bude organizovan i prikazuje ukupnu strukturu sistema. Projektovanje arhitekture je prva faza procesa projektovanja softvera. To je kritična veza između inženjerstva zahteva i projektovanja softvera, jer utvrđuje glavne strukturne komponente u sistemu i veze između njih. Na izlazu iz procesa projektovanja arhitekture se dobija model arhitekture koji opisuje kako je sistem organizovan kao skup komunikacionih komponenti.

I kod agilnih procesa se smatra da je u ranim fazama procesa razvoja neophodno da se definiše ukupna arhitektura sistema. Inkrementalni razvoj arhitekture ne vodi uspehu, a promene u arhitekturi sistema su skupe. Zato je važno na početku projektovanja sistema da se postavi kvalitetna arhitektura softverskog sistema koja se kasnije neće menjati. Komponente je lako menjati, ali arhitekturu – nije.

Na slici 1 prikazana je arhitektura jednog robotizovanog sistema za pakovanje. Sistem pakuje različite objekte. Ima komponentu koja mu obezbeđuje da vidi objekt da bi ga uzeo iz konvejera, utvrdio tip objekta i izabrao prvi način pakovanja. Sistem onda prebacuje objekt iz konvejera za isporuku, u konvejer za pakovanje. On stavlja upakovane objekte na drugi konvejer



Slika 1.1 Arhitektura robotizovanog sistema za pakovanje. [1.2]

## POVOLJNOSTI POSTAVLJANJA DOBRE ARHITEKTURE

*Dobra arhitektura omogućava dobru komunikaciju sa drugim sistemima i korisnicima, odgovarajuću analizu sistema i višestruku upotrebljivost i u drugim softverima.*

U idealnom slučaju, specifikacija sistema ne bi trebalo da sadrži informacije o projektnom rešenju softvera. U praksi, sem kod vrlo malih sistema, to se ne može ostvariti. Najčešće je potrebno da se već na početku izvrši dekompozicija arhitekture budućeg sistema da bi se dobila strukturisana i organizovala specifikacija. Zato, već u fazi utvrđivanja zahteva, može se predložiti apstraktna arhitektura sistema, da bi se povezala svojstva sistema sa najvećim i najbitnijim komponentama sistema. Tako predstavljena specifikacija funkcija i zahteva, se onda diskutuje sa akterima sistema.

Projektovanje arhitekture softvera primenjuje dva nivoa apstrakcije:

1. **Arhitektura u malom** – odnosi se na arhitekturu posebnih programa, tj. na strukturu i komponente samog programa.
2. **Arhitektura u velikom** – odnosi se na arhitekturu složenog organizacijskog sistema koji uključuje druge sisteme, programe i programske komponente. Ovo je u stvari arhitektura distribuiranih sistema, što se na Univerzitetu Metropolitan izučava na posebnom predmetu.

Arhitektura softvera je važna jer utiče na performanse, robusnost, distributivnost i održivost sistema. Dok komponente sistema

ostvaruju funkcionalne zahteve sistema, nefunkcionalni zahtevi najviše zavise od arhitekture sistema, jer ona određuje organizaciju tih komponenti i njihovu međusobnu komunikaciju.

Posedovanje projektnog rešenja arhitekture softvera obezbeđuje sledeću povoljnost:

1. *Komunikacija sa akterima sistema:* Arhitektura daje pregled celine sistema i dobro je sredstvo za komunikaciju sa različitim akterima sistema.
2. *Analiza sistema:* Izrada arhitekture sistema u ranim fazama razvoja zahteva sprovođenje odgovarajućih analiza na nivou sistema. Zato, odluke vezane za projektno rešenje arhitekture imaju dubok efekat na to da li će sistem zadovoljiti kritičke zahteve, kao što su performanse, pouzdanost i održivost.
3. *Višestruka upotrebljivost:* Model arhitekture sistema je kompaktan, upravljiv opis kako je sistem organizovan i kako komponente rade zajedno. Arhitektura sistema je često ista za sisteme sa sličnim zahtevima, te se mogu koristiti više puta, kod drugih sličnih sistema.

## KORIŠĆENJE MODELA ARHITEKTURE SISTEMA

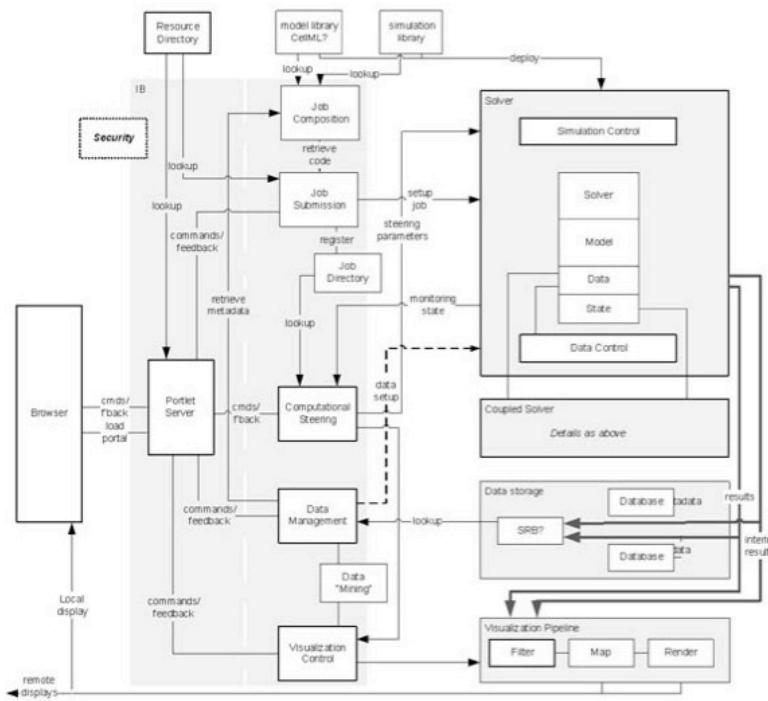
*Model arhitekture služi za olakšavanje diskusije o arhitekturi sistema ili za dokumentovanje arhitekture sistema radi razumevanja i daljeg razvoja sistema.*

Arhitektura sistema se često modeluje u vidu jednostavnih blok dijagrama, kao na slici 2 , gde svaki pravougaonik predstavlja jednu komponentu. Pravougaonici unutar pravougaonika ukazuju da komponenta ima pod-komponente. Strelice pokazuju tok podataka i upravljačkih signala. Ovako urađeni blok dijagram predstavljaju strukturu sistema koju ljudi iz različitih disciplina, a koji su na neki način deo procesa razvoja sistema, razumeju.

Postoji dva načina korišćenja modela arhitekture nekog programa:

1. *Način za olakšavanje diskusije o projektnom rešenju arhitekture:* Koristi se u razgovorima sa akterima sistema i pri planiranju projekta, jer ne sadrži sve detalje. Model sadrži ključne komponente koje treba razviti, što je dovoljno za planiranje njihovog razvoja.
2. *Način dokumentovanja arhitekture koju treba projektovati:* Cilj ovog načina korišćenja modela je da se proizvede kompletan model sistema kako bi se pokazale različite komponente sistema, njihovi interfejsi, i njihove veze. Detaljan opis arhitekture je potreban radi razumevanja i daljeg razvoja sistema.

Neki osporavaju korisnost i isplativost primene drugog načina modelovanja arhitekture.



Slika 1.2 Primer jednog projektnog rešenja arhitekture [1.2]

## ŠTA JE ARHITEKTURA SOFTVERA I ZAŠTO JE VAŽNA? (VIDEO)

*Intervju sa profesorom Nenadom Medvidovićem, SAD*

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

## ŠTA JE ARHITEKTURA? (VIDEO)

*GeorgiaTech prezentacija o arhitekturi*

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

## ODGOVOR NA KVIZ (VIDEO)

*Georgia Tech predavanje - Odgovor na kviz pitanje*

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

# VIDEO PREDAVANJE ZA OBJEKAT "ARHITEKTURA SISTEMA"

*Trajanje video snimka: 24min 33sek*

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

## ▼ Poglavlje 2

### Odluke u vezi arhitekture

#### PITANJA NA KOJE ARHITEKTA SISTEMA TREBA DA NAĐE ODGOVORE

*Projektovanje arhitekture softvera je kreativan proces u kome se projektuje organizacija sistema koja treba da zadovolji funkcionalne i ne-funkcionalne zahteve sistema.*

Projektovanje arhitekture softvera je kreativan proces u kome se projektuje organizacija sistema koja treba da zadovolji funkcionalne i ne-funkcionalne zahteve sistema. Aktivnosti ovog procesa zavise od tipa sistema koji se razvija, od iskustva arhitekte sistema i od specifičnih zahteva sistema. To je u stvari proces donošenja odluka.

Arhitekte sistema treba da uzmu u razmatranje sledeća fundamentalna pitanja o sistemu:

1. Da li postoji opšta arhitektura sistema koja se može koristiti kao uzor za sistem koji treba da se projektuje?
2. Kako će sistem da se podeli po procesorima?
3. Koje arhitektonske mustre ili stilovi se mogu koristiti?
4. Koji će se fundamentalni pristup strukturiranju sistema primeniti?
5. Kako će se komponente strukture dalje podeliti u pod-komponente?
6. Koja će se strategija kontrole rada komponenti upotrebiti?
7. Koja je organizacija arhitekture najbolja za obezbeđenje nefunkcionalnih zahteva sistema?
8. Kako će se vrednovati projektno rešenje arhitekture?
9. Kako će se dokumentovati arhitektura sistema?

Iako je svaki softverski sistem jedinstven, sistemi u istom domenu primene često imaju slične arhitekture, u saglasnosti sa osnovnim konceptima domena. Zato se treba videti koje klase sistema i aplikacije imaju ta zajednička svojstva, i da se na osnovu toga doneše odluka koja se primenjena arhitektura može ponovo upotrebiti.

U slučaju ugrađenih sistema, ili sistema na personalnim računarima, obično se koristi samo jedan procesor, te se ne može projektovati distributivna arhitektura sistema, kao što se mora raditi kod velikih sistema, koji koriste više procesora. Odluka o primeni distribuirane arhitekture sistema je od ključne važnosti jer utiče na performanse i pouzdanost sistema.

Projektno rešenje arhitekture sistema može se zasnovati na nekoj posebnom šablonu arhitekture. Arhitektonski šabloni (engl., **architectural pattern**) su opisi organizacije sistema, kao što su arhitektura klijent-server, ili višeslojna arhitektura. Arhitektonski šabloni sadrže

bitna arhitektonska rešenja koja su upotrebljena kod različitih sistema. Preporučljivo je da se njihova eventualna upotreba razmotri na početku projektovanja arhitekture sistema.

## NEFUNKCIONALNI ZAHTEVI I ARHITEKTURA SISTEMA

*Pri projektovanju arhitekture, pojedini zahtevi vode ka konfliktnoj situaciju prilikom projektovanja arhitektura*

Pri izboru stila arhitekture i strukture sistema, treba voditi računa o nefunkcionalnim zahtevima sistema:

1. *Performanse*: Ako su performanse bitne za sistem, onda arhitektura treba da obezbedi lokalizaciju kritičnih operacija unutra malog broja komponenti, u okviru istog računara. To znači da je bolje koristiti veliku umesto male komponente da bi se smanjila komunikacija između komponenti, koja odnosi dosta vremena.
2. *Bezbednost*: Ako je bezbednost bitna, višeslojna arhitektura je dobro rešenje, da bi se najkritičnije vrednosti stavile pod kontrolu u unutrašnjim slojevima sistema, sa visokom dozom zaštite.
3. *Zaštita*: Ako je zaštita od značaja, onda arhitektura treba da obezbedi da se zaštićene operacije obavljaju u samo jednoj komponenti ili u vrlo malom broju komponenti. To smanjuje troškove zaštite i primenu sistema zaštite koji mogu da isključe sistem u slučaju nekog kvara.
4. *Raspoloživost*: Ako je raspoloživost kritičan zahtev za sistema, onda arhitektura treba da bude tako projektovana da uključuje redundantne (ponovljive) komponente koje mogu jedna-drugu da zamene u slučaju da jedna otkaže.
5. *Lakoća održavanja*: Ako je lakoća održavanja kritičan zahtev, arhitekturu sistema treba projektovati sa malim, samodovoljnim komponentama koje se lako zamjenjuju. Proizvodnja podataka se odvaja od korisnika a izbegava se deljivost struktura podataka.

Pri projektovanju arhitekture, pojedini zahtevi vode ka konfliktnoj situaciju prilikom projektovanja arhitektura. Na primer, velike komponente poboljšavaju performanse, a male olakšavaju održavanje. Ako su oba zahteva važna, mora se onda tražiti kompromisno rešenje. Ponekad, za pojedine delove sistema se onda, koriste različiti stilovi arhitekture.

Ocena arhitekture se može dobiti i njenim upoređenjem sa referentnim arhitekturama (ranije primjenjenim i provereno dobrom) ili sa opštim arhitektonskim šablonima (uzorima). Konačnu ocenu arhitekture daće ocena rada celog sistema, i zadovoljenje svih funkcionalnih i nefunkcionalnih zahteva.

## ODLUKE U FAZI KREIRANJE ARHITEKTURE (VIDEO)

*Georgia*

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

## EVOLUCIJA ARHITEKTURE (VIDEO)

*Georgia Tech predavanje - Evolucija arhitekture*

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

## DEGRADACIJA ARHITEKTURE (VIDEO)

*Georgia Tech predavanje - Degradacija arhitekture*

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

## POPRAVKA ARHITEKTURE (VIDEO)

*Georgia Tech predavanje - Popravka arhitekture*

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

## ✓ Poglavlje 3

# Projektovanje arhitekture sistema

## POGLEDI NA ARHITEKTURU SISTEMA

*Svaki model pokazuje samo jedan pogled ili perspektivu sistema. Obično se koriste logički, procesni, razvojni i fizički pogled na arhitekturu sistema.*

Svaki model pokazuje samo jedan pogled ili perspektivu sistema. Korisno je predstaviti više pogleda softverske arhitekture kada se različite informacije o sistemu traže u različitim vremenima (fazama razvoja). Krutchen (1995) je postavio svoj 4+1 pogled modela arhitekture softvera, sa četiri fundamentalna arhitektonska pogleda, uz pomoć slučajeva korišćenja i scenarija:

1. **Logičan pogled** – pokazuje ključne apstrakcije sistema kao objekte ili klase objekata. Zgodan je za povezivanje sistemskih zahteva sa ovim entitetima sistema.
2. **Procesni pogled** – pokazuje kako u fazi rada, sistem radi korišćenjem povezanih procesa. Koristan je za ocenu nefunkcionalnih karakteristika, kao što su performanse i raspoloživost.
3. **Razvojni pogled** – pokazuje dekompoziciju softvera radi njegovog razvoja, tj. prikazuje komponente koje su implementirane od strane razvojnog tima. Ovaj pogled je značajan za softverske menadžere i programere.
4. **Fizički pogled** – prikazuje hardverske i softverske komponente i njihovu distribuciju po procesorima sistema. Ovaj pogled je zgodan za planiranje razvoja i instaliranja sistema.

Pored ovih pogleda, koristi se i tzv. **koncepcijski pogled**, kao jedan apstraktan pogled na sistem koji može biti osnova za dekompoziciju uopštenih zahteva na detaljnije specifikacije zahteva. To pomaže inženjerima da odluče oko komponenti i njihovoј višestrukoj upotrebi, i da predstave jednu liniju proizvoda umesto jednog sistema.

U praksi, koncepcijski pogledi se uvek razvijaju u toku procesa razvoja da bi se olakšalo donošenje odluka u vezi arhitekture. Oni olakšavaju komunikaciju sa akterima sistema. Ostali pogledi se razvijaju prema potrebi.

Pored neformalnih oznaka i načina prikazivanja arhitekture softvera, može se za te svrhe koristiti i UML, mada je on primereniji za faze projektovanja i implementacije sistema.

Inženjeri softvera koji primenjuju agilne metode osporavaju potrebu razvoja detaljne projektne dokumentacije, jer smatraju da se ona slabo koristi, a da njena priprema traži i vreme i novac. Međutim, nije sporno da treba razviti poglede na arhitekturu radi komunikacije sa drugim akterima sistema, bez potrebe da se pravi detaljna dokumentacija. Međutim,

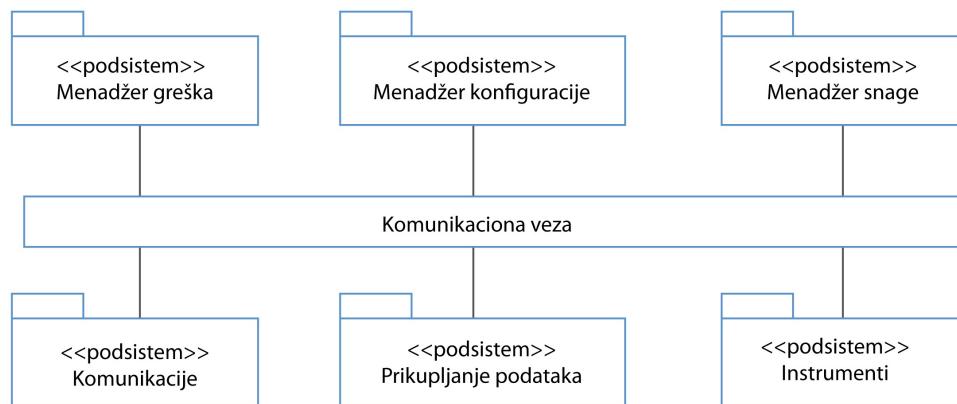
detaljna dokumentacija je potrebna kod kritičnih sistema, jer ona omogućava odobrenje projekta od strane regulacionih vlasti.

## ARHITEKTURA SOFTVERA – PRIMER

*Projektovanje arhitekture softvera je određivanje njegovih glavnih komponenti koje čine sistem i njihove međusobne interakcije.*

Kada se odrede interakcije sistema koji se projektuje, i sistema i njegovom okruženju, prostupa se projektovanju arhitekture sistema. To znači da treba odrediti njegove glavne komponente koje čine sistem i njihove međusobne interakcije. Pri projektovanju arhitekture, koristi se neka od mustri za projektovanje arhitekture, te se one realizuje ili u vidi slojevite arhitekture, ili u vidu klijent-server modela, ili se primenjuje neka druga mustra.

Najapstraktniji nivo prikazivanja projekta arhitekture softvera stanice za prikupljanje vremenskih prilika je prikazan na slici1. Podsistemi sistema emituju poruke preko zajedničke komunikacione veze (magistrale). Svaki podsistem osluškuje poruke ove infrastrukture i preuzima poruke koje su mu namenjene.



Slika 3.1 Prikaz uprošćene arhitekture softvera stanice za prikupljanje vremenskih prilika [1.2]

## AKTIVNOSTI PROJEKTOVANJA SISTEMA

*Projektovanje sistema definiše: ciljeve projektovanja, određuje arhitekturu softvera i njegove granične uslove*

**Model analize**, dobijen analizom zahteva i mogućnosti, sadrži

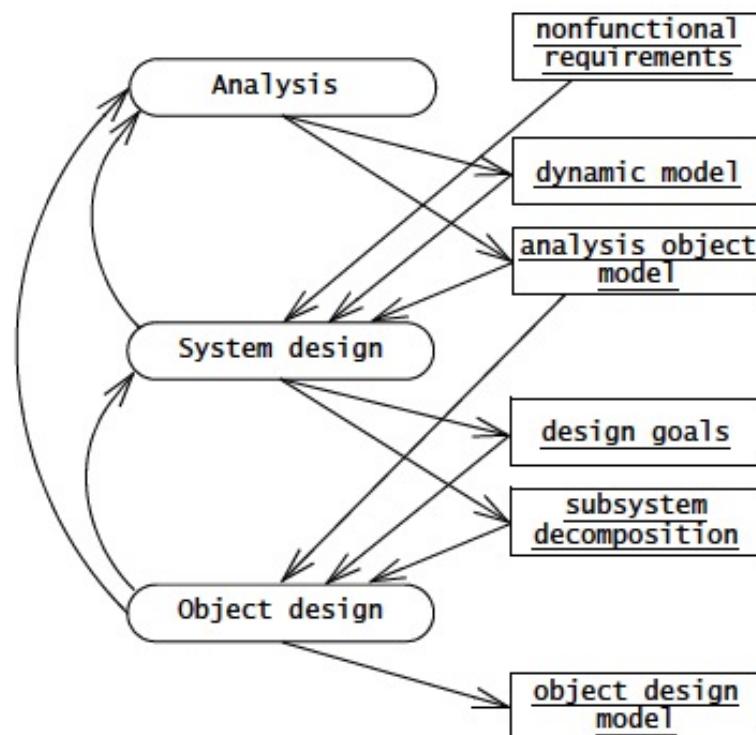
- skup nefunkcionalnih zahteva i ograničenja,
- model slučajeva korišćenja
- model objekata (Boundary, Control i Entity)
- sekvencijalni dijagram za svaki slučaj korišćenja, koji pokazuje redosled interakcije utvrđenih objekata.

Model analize je polazna osnova za projektovanje softverskog sistema. Model analize ne sadrži informaciju o unutrašnjoj strukturi sistema, njegovu konfiguraciju, a ni informaciju kako bi sistem realizovao postavljene zahteve.

**Projektovanje softverskog sistema** treba da dovede do sledećih rezultata:

- **ciljeve projektovanja**, koji opisuju kvalitet koji projektanti mora da ostvare,
- **arhitekturu softvera**, koja pokazuje dekompoziciju sistema na podsisteme, sa njihovim vezama i odgovornostima, i njihovo mapiranje u hardver, kao i definisanje toka kontrole, pristupa i skladišenja podataka,
- **granične slučajeve korišćenja**, koji opisuju konfiguraciju, uključenje, isključenje i rad sa izuzecima.

Na slici 2 je prikazan dijagram aktivnosti projektovanja sistema, i njegovu povezanost sa ostalim aktivnostima softverskog inženjerstva.



Slika 3.2 Aktivnosti projektovanja sistema [1.2]

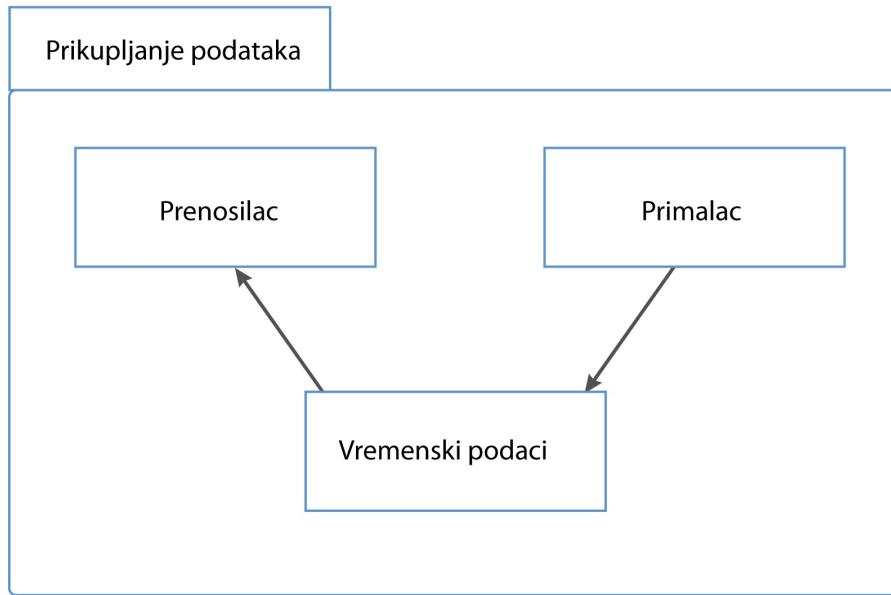
## PODSISTEMI I NJIHOVA KOMUNIKACIJA

*Kada komunikacioni sistem prima neku kontrolnu komandu, komandu dobija svaki od podistema, te svaki od njih samostalno reaguje na najbrži način*

Kada komunikacioni sistem prima neku kontrolnu komandu, kao na primer, instrukcija za isključenje, komandu dobija svaki od podistema, te svaki od njih se isključuje samostalno na najbrži način.

Kako sistem koji šalje poruku ne mora da zna adresu podsistema kome šalje poruku, ova arhitektura sistema lako podržava različite konfiguracije sistema podsistema sistema.

Na slici 3 prikazana je **arhitektura podsistema za prikupljanje podataka**, koji je deo cele arhitekture sistema na slici 1 . Pošiljalac i primalac, kao objekti, upravljaju komunikacije i objekt VremenskiPodaci sadrži informaciju prikupljenu od instrumenata i koja se prenosi informacionim sistemom za praćenje vremena. Ovakva struktura odgovara mustri proizvođač-korisnik.

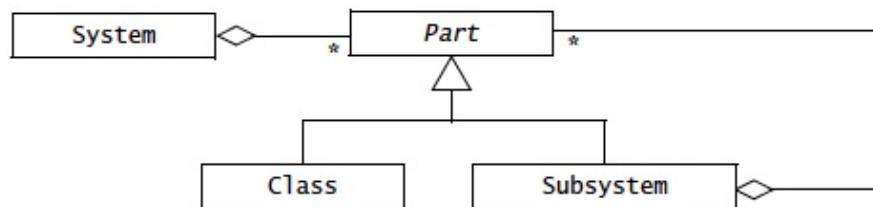


Slika 3.3 Arhitektura sistema za prikupljanje vremenskih podataka [1.2]

## PODSISTEMI I KLASE

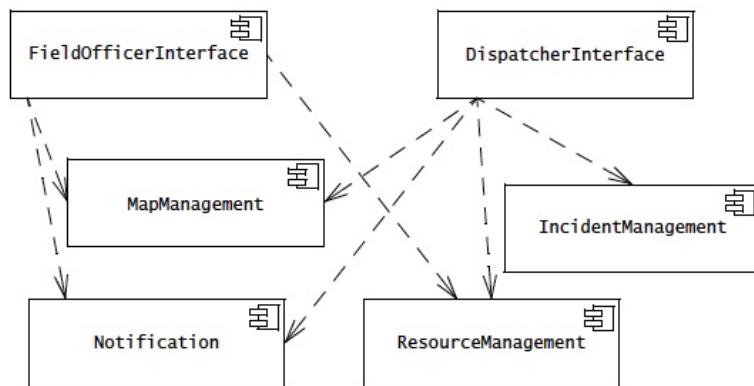
*Podsistemi su zamjenjivi deo sistema sa jasno definisanim interfejsima.  
Podsistemi sadrži klase.*

**Podsistemi** je zamjenjivi deo sistema sa dobro definisanim interfejsima koji učauruju stanja i ponašanja. Podelom sistema na podsisteme, omogućava se relativno nezavisno njihov paralelni razvoj od strane različitih članova tima za razvoj. Dekompozicija sistema nije samo ograničena na podsisteme, jer se i ovi mogu dalje dekomponovati (slika 4).



Slika 3.4 Dekompozicija sistema [1.2]

Na slici 5 prikazana je dekompozicija sistema za upravljanje incidentima.



Slika 3.5 Primer dekompozicije sistema za upravljanje rizicima [1.2]

Java realizuje podsisteme primenom paketa (packages)

## SERVISI

*Servis je skup operacija koji dele neku zajedničku svrhu, a interfejsi podistema sadrže spisak servisa.*

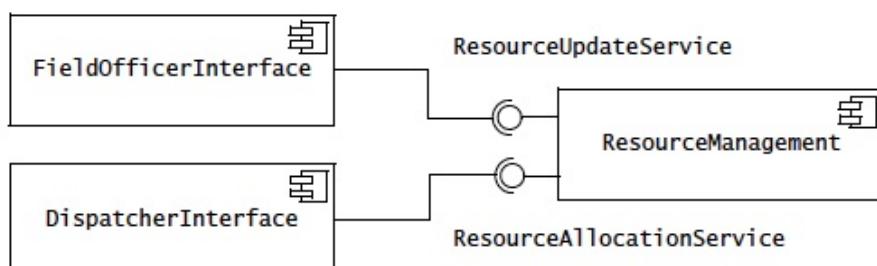
Karakteristično je za podsisteme je da daju određene servise drugim podsistemima.

**Servis** je skup operacija koje dele neku zajedničku svrhu. **Interfejs** podistema sadrži spisak operacija koje klase čine jedan podistem. On sadrži nazine operacija, njihove parametre, njihove tipove, i tip povratne vrednosti.

**Projektovanjem sistema** određuju se servisi podistema. **Projektovanje objekata** (Object Design) se fokusira na programski interfejs aplikacije, tj. **API-Application Programmer Interface**, koji proširuje i dopunjuje interfejs podistema.

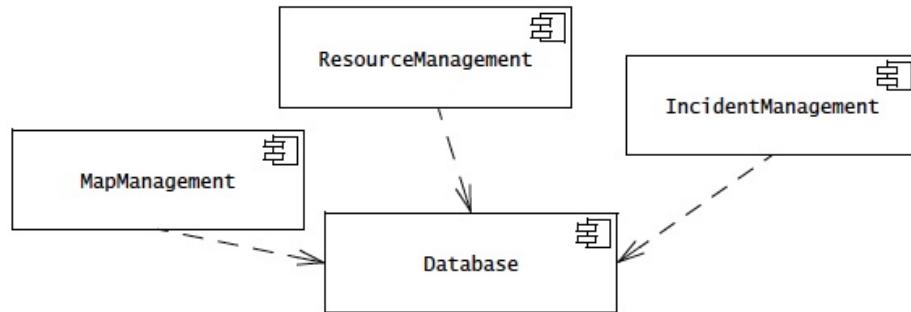
Za povezivanje interfejsa, UML koristi tzv. **konektori** (**assembly connectors**). Nazajau se i soketima, tj. **ball and socket connectors** (slika 6). Na slići 7 je prikazan primer podistema za rad a bazom podataka, sa svojim servisima, koji je dodat podistemima na slići 7

**Coupling** (spojka) je broj zavisnosti (**dependencies**) između dva podistema. Podistemi sa jakom povezanošću (coupling) se teže menjaju, jer promena kod jednog utiče i na drugi. Kod slabo povezanih podistem (loosely coupled) imaju veliki stepen nezavisnosti..



Slika 3.6 Servisi podistema [1.2]

Tokom projektovanja, odlučeno je da se podsistemima na slici 5 podistem koji bi ih povezivao sa bazom podataka (slika 7)



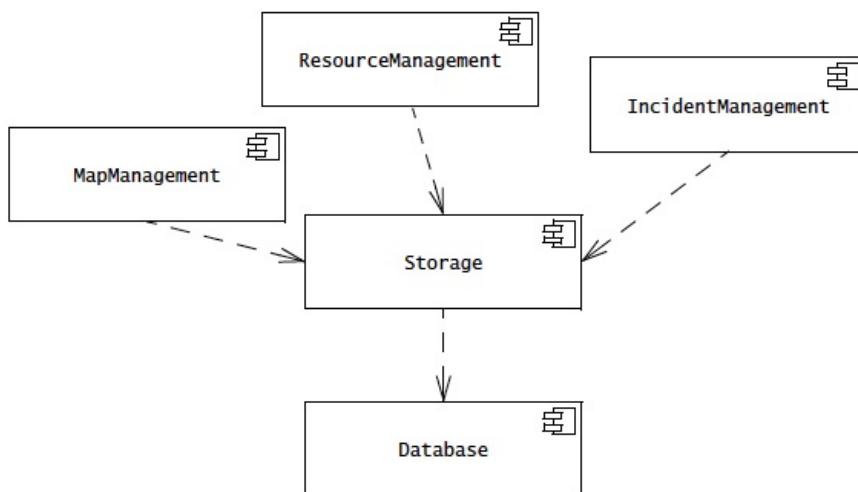
Slika 3.7 Podsistemi sa slike 5, prošireni sa podistemom za rad sa bazom podataka [1.2]

## SPAJANJE I KOHEZIJA

*Potrebno je naći balans između veza podistema (coupling) i kohezija podistema.*

Da bi podistemi bili što nezavisniji od proizvođača sistema baza podataka (**DBMS - Database Management System**), može se između njih i sistema baze podataka, umetnuti novi podistem koji svoje servise prema podistemima ne menja, ali menja samo interfejs prema podistemu baze podataka, tj. koristi onaj koji je specifičan za određeni sistem baza podataka, tj. koji zavisi od njegovog proizvođača.

Na slici 8 prikazan je taj slučaj, te podistem Storage obavlja funkciju posrednika između podistema koji koriste usluge podistema rada sa bazom podataka i samog sistema baze podataka.



Slika 3.8 Podistem Storage je neutralni posrednik između podistema i podistema sistema baze podataka. [1.2]

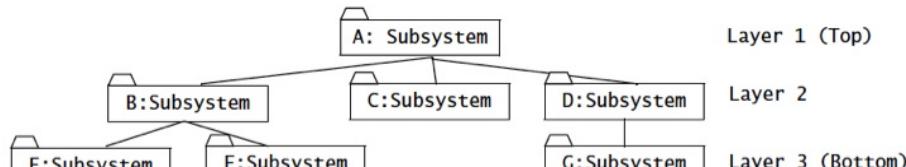
**Kohezija (Cohesion)** je broj zavisnosti unutar jednog podistema. Cilj projektovanja je dobijanje podistema sa visokom kohezijom.

Potrebno je nači balans između veza podistema (coupling) i kohezija podistema. Ako podistemi imaju visoku kohiziju, obično imaju nisku međusobnu povezanost (mali coupling). To dovodi do povećanja broja podistema, ali i do povećanja ukupnog broja interfejsa među podistemima. Smatra se da je kompromis imati 7 +/- 2 koncepta na istom nivou apstrakcije. To znači da ako ima više od 9 podistema, tj. da jedan podistem mora da ima više od 9 servisa, preporuka je da se izvrši revizija podistema, da bi se taj broj veza i servisa smanjio.

## PRIMENA SLOJEVA

*Jedan sloj predstavlja grupu podistema koji daju povezane servise, pri čemu koriste i servise sleva ispod njih*

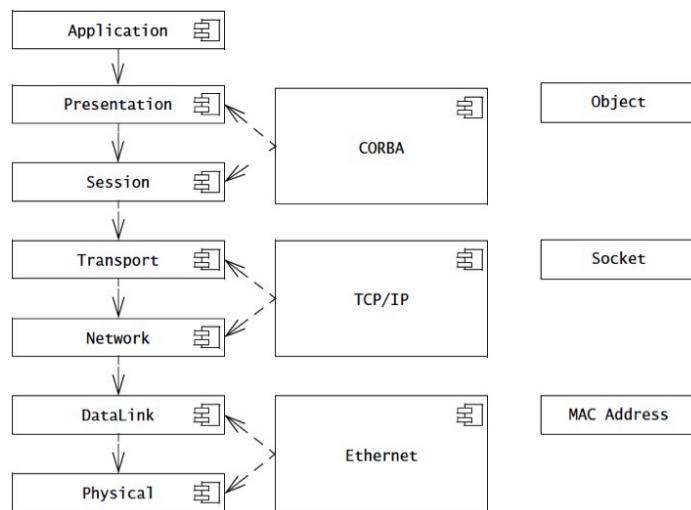
**Hijerarhijska dekompozicija** sistema dovodi do **slojevite arhitekture** sistema (slika 9). Jedan **sloj** predstavlja grupu podistema koji daju povezane servise, pri čemu koriste i servise sleva ispod njih i nema saznanja o slojevima iznad njega. Slojevi se redaju tako, da zavise samo od slojeva ispod njih, a najniži sloj ne zavisi ni od jednog sloja. Sloj na vrhu ne daje servise nijednom drugom



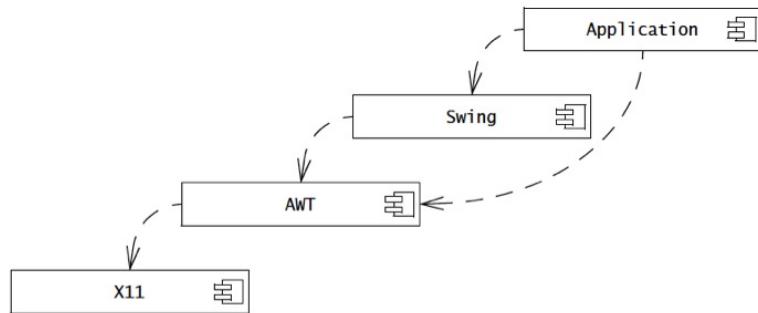
Slika 3.9 Slojevita arhitektura sistem [1.2]

Na slici 10 dat je primer slojevite arhitekture sistema koji obezbeđuje servise za rad distribuiranih sistema na Internetu

Na slici 11 prikazan je primer otvorene slojevite arhitekture. Ona dozvoljava da viši sloj preskače neki niži sloj i pristupa direktno nekom još nižem sloju. Na taj način se može ubrzati rad softverskog sistema, ali se gubi na fleksibilnosti i nezavisnosti slojeva.



Slika 3.10 Primer slojevite arhitekture sistema za komunikaciju distribuiranih sistema preko Interneta [1.2]



Slika 3.11 Primer otvorene slojevite arhitekture [1.2]

## DVE VRSTE ARHITEKTURE (VIDEO)

*Georgia Tech predavanje - Dve vrste arhitekture*

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

## ▼ Poglavlje 4

### Arhitektonski šabloni

#### ŠTA SU ARHITEKTONSKI ŠABLONI?

*Arhitektonski šabloni opisuju organizaciju sistema koja se pokazala uspešnom u ranijim sistemima*

Primena arhitektonskih šablona je način predstavljanja, deljenja i zajedničke upotrebe znanja o softverskim sistemima. Šabloni su apstraktni opisi dobre prakse, koja je testirana i oprobana u različitim sistemima i okruženjima. Arhitektonski šabloni opisuju organizaciju sistema koja se pokazala uspešnom u ranijim sistemima. Svaki šablon ima svoju oblast primene, kao i slabosti i dobre osobine. To isto treba da bude dokumentovano za svaki šablon.

#### VIDEO PREDAVANJE ZA OBJEKAT "ARHITEKTONSKI ŠABLONI"

*Trajanje video snimka: 27min 44sek*

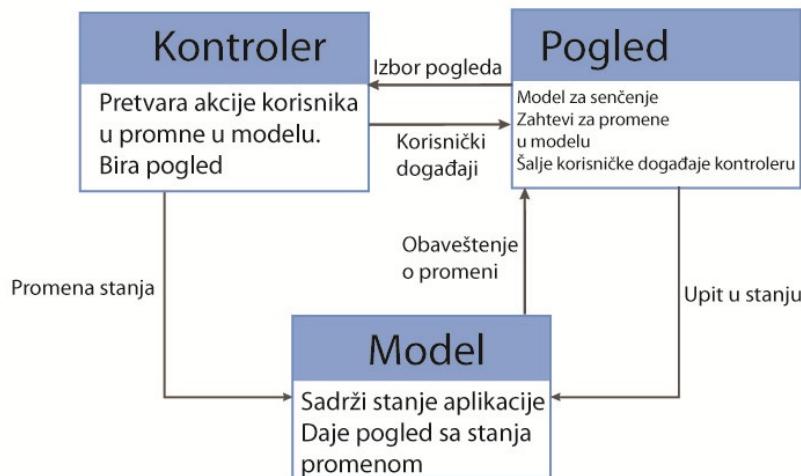
**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

#### ▼ 4.1 MVC arhitektura

#### MVC ILI MODEL-POGLED-KONTROLER ŠABLON

*Arhitektonski šabloni opisuju organizaciju sistema koja se pokazala uspešnom u ranijim sistemima*

Radi ilustracije, na slici 1 je prikazan dobro poznati šablon **Model-Pogled-Kontroler** (engl. Model-View-Controller), ili MVC šablon koja se koristi kod mnogih veb sistema.

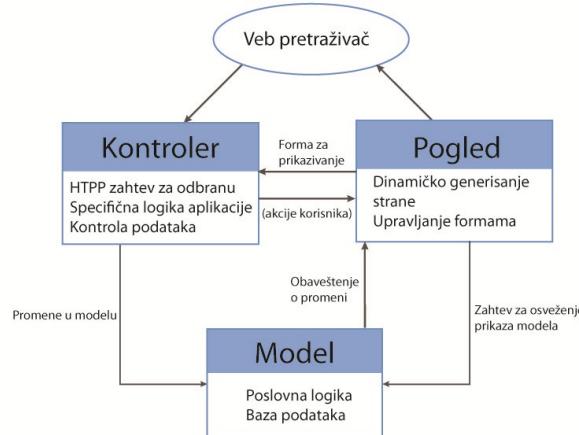


Slika 4.1.1 Koncepcijski pogled na MVC šablon [1.2]

Pored grafičkog modela, opis šablonu obezbeđuje njeno ime, opis upotrebe i primer tipa sistema u kome je šablon primenjen (Tabela 1.). Takođe, daje se i preporuka gde se šablon može koristiti, i navode se njegove prednosti i nedostaci..

Ime	MVC
Opis	Odvaja prezentaciju i interakciju od podataka sistema. Sistem se deli na tri logične komponente koje su u interakciji. Komponenta Model upravlja podacima sistema i operacijama nad podacima. Komponenta Pogled (View) definije i upravlja prezentacijom podataka koje korisnik vidi. Komponenta Kontroler (Controller) upravlja interakcijom korisnika i prenosi te interakcije na Pogled i Model
Primer	Veb aplikacija prikazana na slici 3
Kada se koristi	Upotrebljava se kada postoji više načina pogleda i interakcije sa podacima. Takođe, upotrebljava se i kada budući zahtevi za interakcijom i za prezentacijom po podatkovima nisu poznati unapred.
Prednosti	Omogućava promenu podataka nezavisno o njihove prezentacije, i suprotno. Omogućava da se isti podaci predstave na različite načine, a da se pri promeni podataka ta promena vidi na svim različitim prezentacijama.
Nedostaci	Može zahtevati dodatni kod i uvećava složenost koda i u slučaju primene kod jednostavnih modela.

Slika 4.1.2 Tabela-1 Opis MVC šablonu [1.2]



Slika 4.1.3 Izvršni pogled na MVC šablon [1.2]

## MVC (MODEL-VIEW-CONTROL) ARHITEKTURA

*MVC arhitektura svrstava sve podsistema u tri kategorije: Model, View i Controller*

MVC arhitektura (slika 4) ima podsisteme svrstane u tri tipa:

- **model podsistem** . održava domensko znanje
- **view podsistem** - prikazuje model korisniku
- **controler podsistemi**- upravljaju redosledom interakcija sa korisnikom.

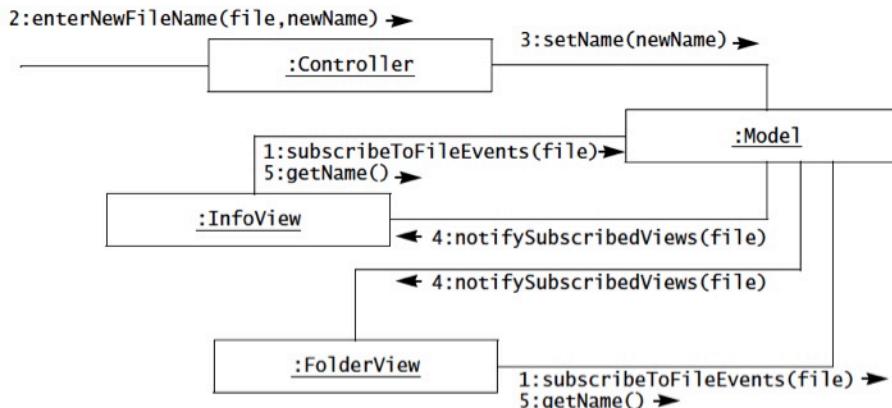
Model podsistem je tako razvijen da ne zavisi od view i controller podsistema. Promene svog stanja prenosi na view podsistem primenom subscribe/notify protokola

MVC je specijalni slučaj repozitorijuma gde Model primjenjuje centralnu strukturu podataka, a controlni objekti kontrolisu tok kontrolnih poruka.

Na slici 5 prikazan je primer komunikacionog dijagrama sistema koji primjenjuje MVC arhitekturu, a koji prikazuje sekvence događaja.



Slika 4.1.4 MVC arhitektura [1.2]



Slika 4.1.5 Primer toka događaja u sistemu sa MVC arhitekturom. [1.2]

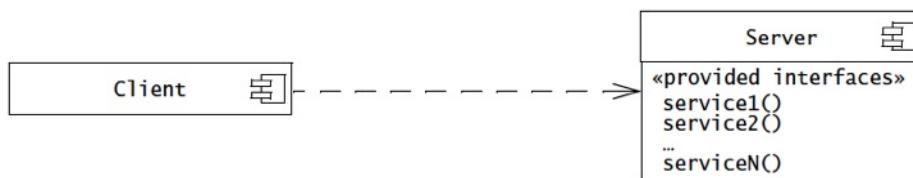
## ❖ 4.2 Klijent-server arhitektura

### KLIJENT/SERVER ARHITEKTURA

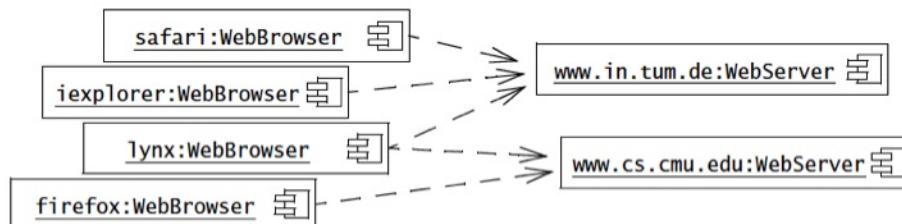
*Serverski podsistem obezbeđuje servise klijentima, tj. drugim podsistemima.*

Kod primene **Klijent/Server arhitekture**, serverski podsistem obezbeđuje servise instancama drugih podsistema, koji se nazivaju klijentima, koji su odgovorni za interakciju sa korisnikom. Zahtev za servisom je obično iniciran sa udaljenom procedurom pozivanja (RPC) ili primenom zajedničkih objektnih brokera (npr, CORBA, Java RMI ili HTTP). Kontrolni tok kod klijenata i servisa je nezavisan sem u slučaju sinhronizacije upravljanja zahtevima ili prijemom rezultata.

Klijent/Server arhitektura je dobra kod distribuiranih sistema koji rade sa velikom količinom podataka. (slika 1,2)



Slika 4.2.1 Klijent-Server arhitektura [1.2]



Slika 4.2.2 Primer Veb klijent-server arhitekture. [1.2]

### ŠABLON KLIJENT-SERVER ARHITEKTURE

*Serveri ne moraju da znaju identitet klijenata, dok klijenti mogu da znaju nazive servera i njihovih servisa. Klijenti koristi servise upotrebom poziva udaljenih procedura kao što je HTP*

Šablon klijent-server arhitekture je organizovan u vidu servisa i odgovarajućih servera, kao i klijenata koje pristupaju sistemu radi korišćenja tih servisa (Tabela 1 ).

Ime	Klijent-server arhitekture
Opis	Kod klijent-serverskih arhitektura funkcionalnost sistema je organizovana da pruža servise, gde svaki servis obezbeđuje poseban (softverski) server. Klijenti su korisnici ovih servisa.
Primer	Slika 2 prikazuje primer video/DVD biblioteke organizovane u vidu klijent-sever arhitekture.
Kada se upotrebljava	Upotrebljava se kada se podacima u bati podataka pristupa sa raznih lokacija. Kako se serveri mogu umnožavati, može se se broj servera povećavati u skladu sa opterećenjem sistema.
Prednosti	Glavna prednost ovog modela je mogućnost distribuiranja servera na mreži. Opšta funkcionalnost (npr. štampanje) se može nuditi svim klijentima, a ne mora da se na taj način nude svi servisi svima.
Nedostaci	Svaki servis je jedinstvena tačka otkaza, te može biti cilj napada na servise ili na server. Performanse je teško predvideti jer zavise od mreže i od sistema. Mogu se javiti i problemi upravljanja, ako su serveri pod kontrolom različitih organizacija.

Slika 4.2.3 Tabela-1: Opis šablona klijent-server arhitekture [1.2]

Glavne komponente klijent-server modela su:

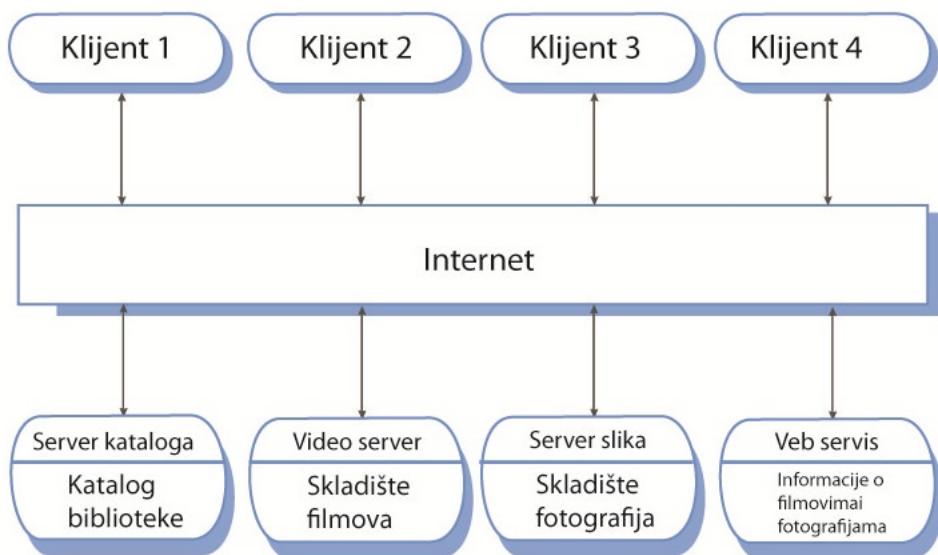
1. Skup servera koji nude servise drugim komponenata (pr., servisa za štampanje, fajl serveri, kompjuteri serveri).
2. Skup klijenata koji pozivaju servise koje nude servere.
3. Mreža koja dozvoljava klijentima da pristupe servisima. Najčešće se klijent-server arhitektura realizuje u vidu distribuiranog sistema na mreži sa Internet protokolom.

Serveri ne moraju da znaju identitet klijenata, dok klijent mogu da znaju nazive servera i njihovih servisa. Klijenti koristi servise upotrebom poziva udaljenih procedura (engl. **remote procedure calls**-RPC), kao što je HTP. Klijent šalje poziv i čeka odgovor servera. Glavna prednost klijent-server modela je njegova distribuirana arhitektura. Mogu se koristiti i distribuirani procesori. U sistem se lako, po potrebi, mogu dodavati novi serveri, bez isključivanja sistema.

## PRIMER KLIJENT-SERVER ARHITEKTURE

*Glavna prednost klijent-server modela je njegova distribuirana arhitektura*

Slika 4 prikazuje primer sistema koji koristi klijent-server arhitekturu. To je višekorisnički veb sistem koji obezbeđuje biblioteku file-ova i fotografija. Video server radi kompresije i dekompresije raznih formata video zapisa (file-ova), da bi se ubrzao njih transfer mrežom. različitih upita i podržava prodaju fotografija..



Slika 4.2.4 Primer klijent-server arhitekture [1.2]

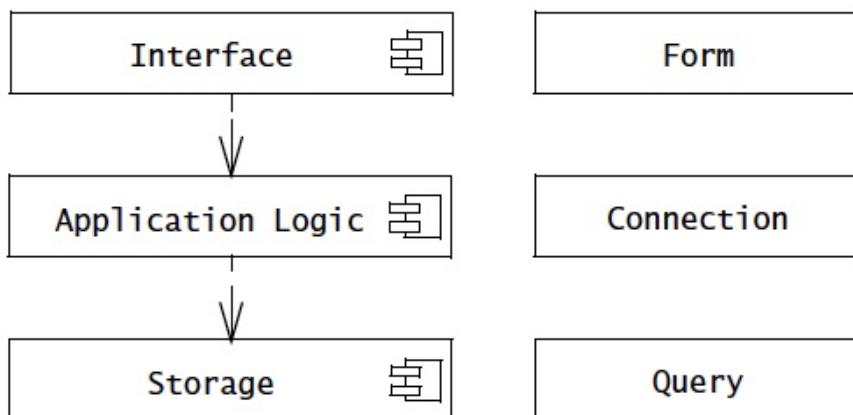
## ❖ 4.3 Slojevita arhitektura

### TROSLOJNA I ČETVOROSLOJNA ARHITEKTURA

*Tri sloja: interfejs (Boundary klase), sloj logike (Comtrol klase) i memorijski sloj (Entity klase)*

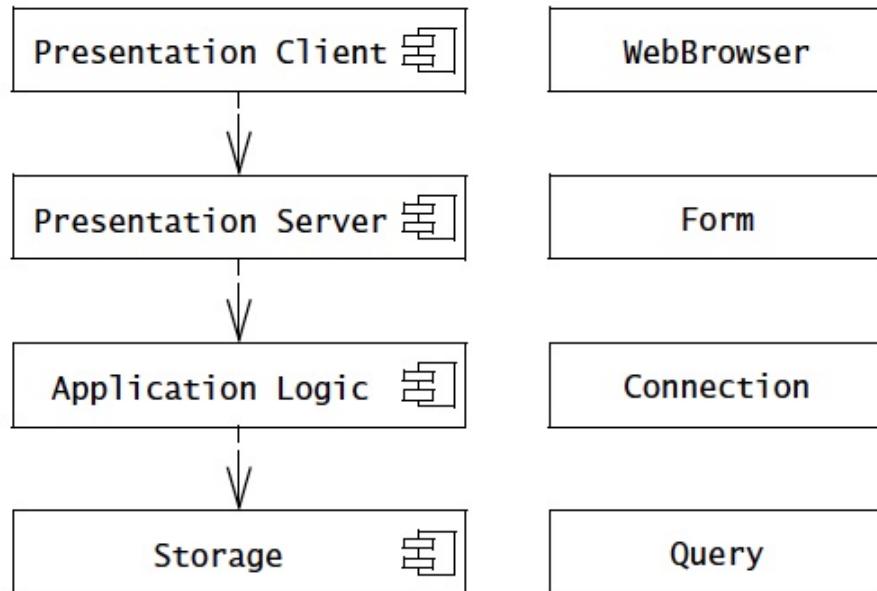
Troslojna arhitektura organizuje podsisteme u tri sloja (slika 1):

- **interfejs sloj** - uključuje sve granične objekte (Boundary)
- **sloj logike aplikacije** - obuhvata sve kontrolne i entitetske objekte (Control i Entity) koji vrše obradu podataka, proveru pravila, i slanje obaveštenja u skladu sa potrebama aplikacije
- **memorijski sloj** - memorije, pretražuje i manipuliše sa trajno zapisanim podacima.



Slika 4.3.1 Three-teer (troslojna) arhitektura [1.2]

**Četvoroslojna arhitektura** (slika 2) je troslojna arhitektura u kojoj je interfejs sloj podeljen na interfejs na klijentu i interfejs na serveru (slika 2).



Slika 4.3.2 Four-teer (četvoroslojna) arhitektura [1.2]

## ARHITEKTURA IZRAŽENA U FUNKCIONALNIM SLOJEVIMA

*Slojevita arhitektura obezbeđuje odvajanje i nezavisnost komponenata softvera primenom funkcionalnih slojeva. Svaki sloj zavisi od uređaja i servisa koje nudi sloj ispod njega*

**Slojevita arhitektura** obezbeđuje odvajanje i nezavisnost komponenata softvera primenom funkcionalnih slojeva (Tabela 3). Svaki sloj zavisi od uređaja i servisa koje nudi sloj ispod njega.

Ime	Slojevita arhitektura
Opis	Organizuje sistem u vidu slojeva koji se odnose na funkcionalnost specifičnu za svaki sloj. Sloj obezbeđuje servise sloju iznad njega, tako da najniži sloj obezbeđuje ključne servise koji se najčešće koriste u svim slojevima.
Primer	Model slojevite arhitekture sistema koji obezbeđuje zajedničko korišćenje dokumenata koji se drže u različitim bibliotekama, kao što je prikazano na slici 2.
Kada se upotrebljava	Kada se dodaju novi uređaji i servisi, sa novom funkcionalnošću, preko postojećih sistema, Kada se razvoj radi preko nekoliko timova pri čemu je svaki tim odgovoran za razvoj funkcionalnosti jednog sloja. Kada postoji zahtev za primenu višeslojne bezbednosti.
Prednosti	Omogućava zamenu celog sloja, uz uslov da se time na menja interfejs. Ponovljive funkcije (npr. autentikacija) se mogu obezrediti u svakom sloju radi povećanje nezavisnosti od sistema.
Nedostaci	U praksi, teško je ostvariti jasnu podelu između slojeva a najviši sloj često ima komunikaciju sa nižim slojevima, a ne samo sa slojem ispod njega. Performanse mogu biti problem jer se u svakom sloju mora da vrši interpretacija zahteva za servisom , tj. obrada u svakom sloju.

Slika 4.3.3 Tabela-1:Opis šabloni slojevite arhitekture [1.2]

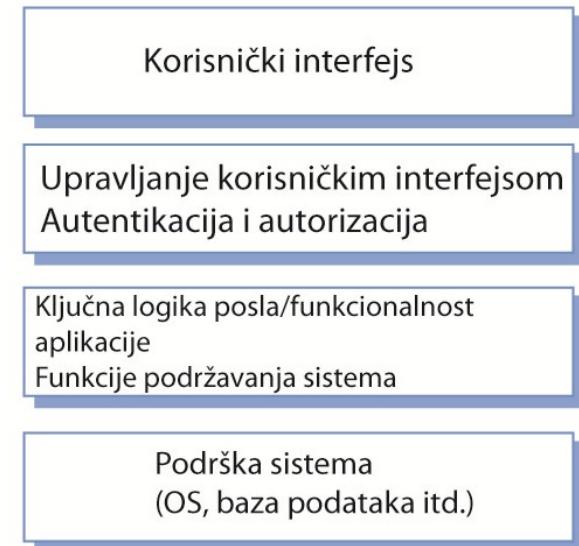
Primena slojevite arhitekture softverskog sistema omogućava njegov inkrementalni (postepeni) razvoj. Čim se razvije jedan sloj, njegovi servisi se mogu koristiti, nezavisno od kasnijih servisa novih slojeva, u nadgradnji sistema. Arhitektura se lako menja i prenosi. Važno je obezrediti da se interfejsi ne menjaju jedan sloj se može zameniti drugim, ekvivalentnim slojem (obavlja istu funkciju, ali na drugi način). Ako se promeni funkcionalnost jednog sloja, to se održava samo na sloj iznad njega.

Slojeviti sistemi lokalizuju zavisnost od maštine (npr. operativnog sistema, tipa baze podataka) samo na unutrašnje (donje) slojeve. To olakšava kreiranje multi-platformske implementacije neke aplikacije. U tom slučaju, za svaku mašinu se menjaju samo ti unutrašnji slojevi

## PRIMERI ARHITEKTURE SA SLOJEVIMA

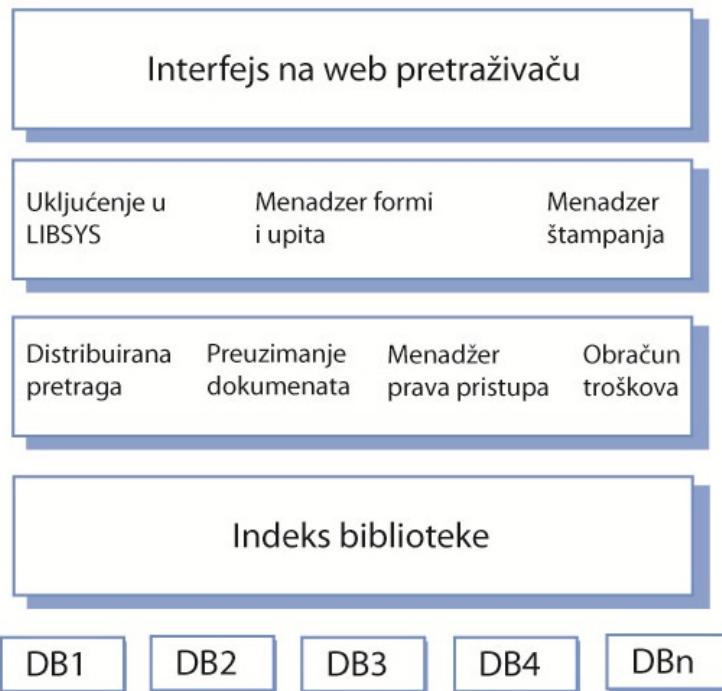
*Svaki sloj obezbeđuje novu funkcionalnost, a oslanja se na funkcionalnost slojeva ispod njega.*

Na slici 4 prikazan je primer arhitektura sistema sa četiri sloja. Najniži sloj obezbeđuje podršku podržavajućeg sistema, kao na primer, određenog sistema baze podataka (DBMS) ili određenog operativnog sistema. Sledeći sloj je aplikacioni sloj koji sadrži komponente koje obezbeđuju funkcionalnost aplikacije i korisničke komponente, koje koriste druge komponente aplikacije. Treći sloj obezbeđuje upravljanje korisničkim interfejsom, kao i autentikaciju (identifikaciju) i autorizaciju (ovlašćenje) korisnika. Četvrti, najviši sloj, implementira korisnički interfejs. Ovi slojevi se mogu dalje deliti na dva ili više slojeva.



Slika 4.3.4 Grafički prikaz slojevite arhitekture softverskog sistema [1.2]

Na slici 5 prikazan je primer primene šabloni višeslojne arhitekture bibliotečkog softverskog sistema, LIBSYS, koji omogućava elektronski pristup zaštićenom materijalu iz grupe univerzitetskih biblioteka. On koristi pet slojeva, pri čemu je najniži sloj zavistan od sistema baze podataka koji koristi svaka biblioteka.



Slika 4.3.5 Arhitektura LYBSYS sistema [1.2]

## ✓ 4.4 Arhitektura sa skladištem podataka

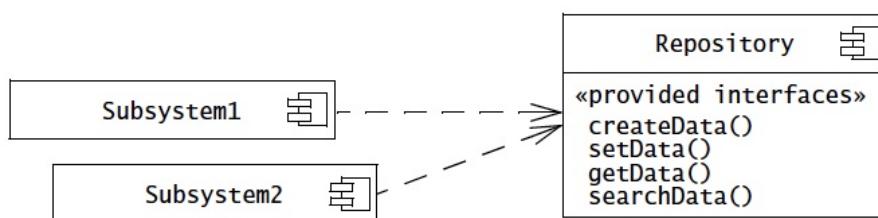
### REPOZITORIJUM - ZAJEDNIČKO SKLADIŠTE PODATAKA

*Repozitorijum obezbeđuje pristup svih podsistema jedinstvenoj strukturi podataka.*

**Arhitektura softvera** uključuje dekompoziciju sistema, globalni tok kontrole, rad sa graničnim uslovima, i protokole komunikacije između podsistema.

Primenom različitih arhitektonskih stilova, olakšan je i ubrzani rad projektanata, jer pri izboru određenog arhitektonskog stila, oni preuzimaju sva svojstva izabrane arhitekture, te nije potrebno da to samo definišu

U slučaju arhitektonskog stila **Repozitorijum**, podsistemi pristupaju jednom jedinstvenoj strukturi podataka, koji se naziva **centralni repozitorijum** (skladište podataka). (slika 1).



Slika 4.4.1 Repozitorijum, kao centralno skladište trajnih podataka [1.2]

Repozitorijumi su tipični za aplikacije koji koriste sisteme baza podataka, Centralizacijom podataka, lakše se kontroliše pristup podacima i uređuje istovremeni rad korisnika sa podacima. Takođe, lakše se kontroliše integritet povezanih podsistema.

Repozitorijume treba koristiti kod aplikacija koje se stalno menjaju, a imaju složene zadatke obrade podataka. Kada je centralni repozitorijum dobro definisan, lako se mogu dodavati ili menjati servisi koje obezbeđuju podsistemi. Galvni nedostatak primene repozitorijama je opasnost da oni postanu "usko grlo" jer svi podsistemi koriste isti repozitorijum.

Ako je veza repozitorijuma sa sistemima jaka, onda se oni teže menjaju, jer te promene zahtevaju i promene u repozitorijumu, a to pak, zahteva promene i u ostalim sistemima.

### ARHITEKTURA KORIŠĆENJA ZAJEDNIČKOG SKLADIŠTA PODATAKA

*Organizacija alata oko zajedničke baze podataka je način da se poveća efikasnost rada sa podacima, jer nema prenosa podataka sa jednog na druge module*

Šablon za projektovanje pod nazivom „Arhitektura sa skladištem podataka“ određuje skup komunicirajućih komponenti koje koriste i dele isto skladište podataka (slika 2 ).

Ime	Skladište podataka
Opis	Svi podaci sistema se upravljaju u centralnom skladištu kome mogu prići sve komponente sistema. Komponente ne komuniciraju direktno sa podacima, već preko skladišta (softverskog sloja koji upravlja podacima u bazi podataka).
Primer	Slika 2 prikazuje primer IDE sistema u kome sve komponente koriste skladište sistema sa informacijama o projektnom rešenju. Svaki softverski alat generiše informaciju koja je onda raspoloživa drugim alatima.
Kada se upotrebljava	Ova se mustra koristi u slučaju sistema sa velikom količinom informacija koje se moraju uskladištiti za dugi niz godina. Može se koristiti i kod sistema vođenim podacima u kojima smeštaj nekog podatka u skladište može da pokrene neku akciju ili da aktivira neki alat.
Prednosti	Komponente mogu biti nezavisne, tj. ne moraju da znaju postojanje drugih komponenti. Promene koje je izvršila jedna komponenta (u podacima), šire se svim drugim komponentama. Svi podaci se konsistentno upravljaju (npr. bekap se radi istovremeno) kao da su na istom mestu.
Nedostaci	Skladište je jedinstvena tačka otkaza jer njenim otkazivanje, staje ceo sistem. U slučaju gусте komunikacije, performanse sistema mogu biti lošije. Teško je primeniti distribuisane baze skladišta podataka.

Slika 4.4.2 Opis šablon sa zajedničkim skladištem podataka [1.2]

Najveći broj sistema koji se koriste veliku količinu podataka organizuju oko velike zajedničke baze podataka, tj. skladišta (eng. **repository**). Ovaj šablon projektovanja sistema je pogodna za takve slučajeve u kojima jedne komponente pune bazu podataka, a druga grupa ih koristi. Tako rade informacioni sistemi za upravljanje poslovanjem organizacija, CAD sistemi i interaktivni sistemi za razvoj softvera.

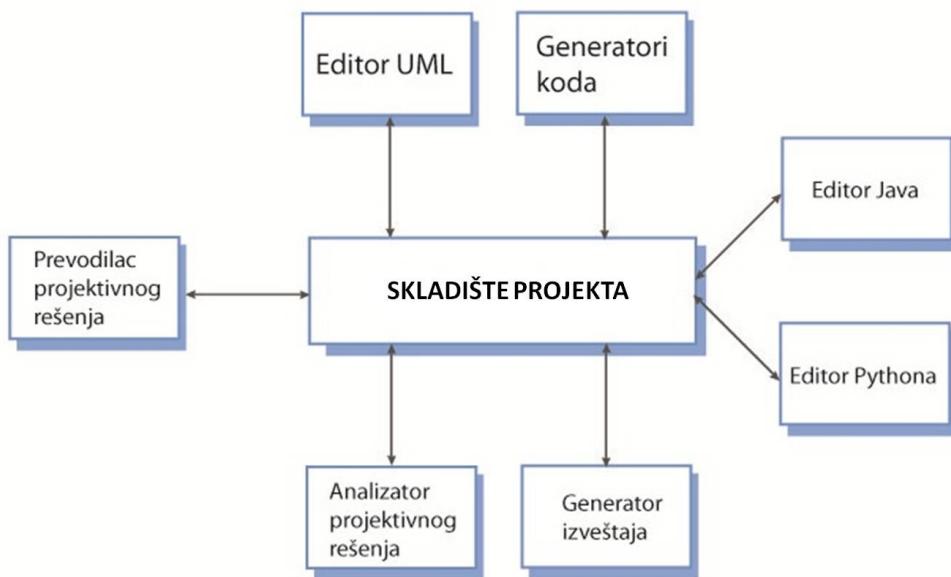
## PRIMER ARHITEKTURE SA SKLADIŠTEM - IDE SISTEM

*Kontrakcijom svih podataka na jednom mestu dobija se na efikasnosti sistema*

Na slici 3 je prikazan jedan IDE sistem (Integrated Development Environment) koji koristi zajedničko skladište podataka.

Organizacija alata oko zajedničke baze podataka je način da se poveća efikasnost rada sa podacima, jer nema prenosa podataka sa jednog na druge module. Kako sve komponente koriste isti model podataka, otežano je dodavanje novih komponenata jer, po pravilu, one imaju svoje specifične modele podataka.

U sistemu na slici 3, skladište je pasivno, jer rad sa podacima zavisi od rada komponenata. To su „pasivna“ skladišta, za razliku od „aktivnih“, koji mogu da podstaknu aktivnost komponenata, kada se dogodi neki unapred definisan događaj..



Slika 4.4.3 Arhitektura sa skladištem (repozitorijem) jedno IDE sistem [1.2]

## ✓ 4.5 Arhitektura cevi i filtra

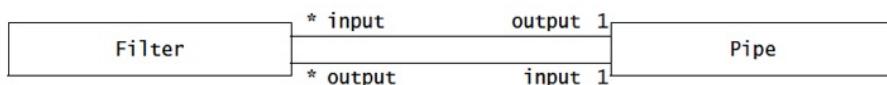
### ŠABLON ARHITEKTURE "CEVI I FILTRA"

*Prima veći broj ulaza i šalje rezultate podsistemima u vidu skupa izlaza*

Sistem sa **arhitekturom cevi i fitra** prima veći broj ulaza i šalje rezultate podsistemima u vidu skupa izlaza. Podsistemi se nazivaju "filtrima" a veze (asocijacije) između podstistema "cevima". Svaki filter zna samo format podataka i sadržaj samo na svojim ulaznim cевима, i ne zna filtre (podsisteme) koji su ih proizveli. Filtri rade istovremeno, a sinhronizacija se obavlja preko cevi.

Najpoznatija primena ove arhitekture je kod Unixshell

**Arhitektura cevi i filtra** je pogodna za sisteme koje vrše transformaciju tokova podataka bez intervencije korisnika. Nije pogodna za sisteme koji imaju složenije interakcije među podsistemima, kao što su informacioni sistemi upravljanja (IMS) ili interaktivni sistemi.



Slika 4.5.1 Arhitektura cevi i filtra. [1.2]

Model cevi i filtra je model izvršne organizacije sistema proces funkcionalnih transformacija koristeći svoje ulaze, proizvodi izlaze. *Podaci teku od jedne do druge transformacije po određenom redosledu. Svaki korak obrade se predstavlja kao jedna transformacija.* Ulazni podaci teku kroz ove transformacije sve dok se ne dobije željeni izlaz. **Transformacije se**

izvršavaju redno i/ili paralelno. Podaci se obrađuju u svakoj transformaciji, pojedinačnom, ili paketnom obradom.

Naziv „cev“ vodi poreklo od Unix operativnog sistema, gde se „cevi“ koriste za povezivanje procesa. Termin „filter“ se koristi za transformacije, jer one „filtriraju“ podatke polazeći od njihovog ulaznog niza. Kada su transformacije sekvencijalne, ovaj model postaje sekvencijalni model za paketnu obradu podataka. Ovu arhitekturu koriste i ugrađeni sistemi tako da se svaki proces može paralelno da odvija (paralelne „cevi“).

## KARAKTERISTIKE ŠABLONA ARHITEKTURE "CEVI I FILTRA"

*Model cevi i filtra je model izvršne organizacije sistema procesa funkcionalnih transformacija koji, koristeći svoje ulaze, proizvodi izlaze.*

Tabela 1 na slici 2 prikazuje karakteristike ovog modela

Ime	Arhitektura cevi i filtera
Opis	Obrada podataka u sistemu je organizovana tako da svaka komponenta (filter) je posebna i vrši jedan tip transformacije podataka. Podaci teku (kao u cevima) od jedne do druge komponente radi obrade podataka.
Primer	Slika 2 prikazuje jedan sistem sa cevi i filterom koji se koristi ko obrade računa za naplatu.
Kada se upotrebljava	Često se koristi kod sistema za obradu podataka (i paketne i transakcione obrade) u kojima se ulazi obrađuju u vrše različitih faza da bi se generisao željeni izlaz.
Prednosti	Lako se razume i podržava ponovnu upotrebu transformacija. Radni tok obrade se uklapa u mnoge poslovne procese. Evolucija dodavanja transformacija je vrlo jednostavna,
Nedostaci	Format transfera podataka treba da bude utvrđen sporazumno među transformacija koje komuniciraju. Svaka transformacija obrađuje ulaz i izlaz u željeni oblik. Ovo povećava opterećenje sistema i može da spreči ponovnu upotrebu funkcionalnih transformacija koje upotrebljavaju nekompatibilne strukture podataka.

Slika 4.5.2 Tabela-1 Opis šablona arhitekture cevi i filtra [1.2]

## PRIMER ARHITEKTURE "CEVI I FILTRA"

*Arhitektura cevi i filtra je dobra za sisteme sa paketnom obradom podataka*

Slika 3 prikazuje primer aplikacije sa paketnom obradom. Sistem šalje račune kupcima. Jedanput nedeljno se šalju zahtevi za plaćanjem, a oni kada se naplate, onda se šalju računi o potvrđenoj uplati. Kupcima koji nisu platili svoje račune, šalju se opomene. Ovim modelom se teško rade interaktivni sistemi jer oni traže nizove podataka za obradu. Grafički korisnički interfejsi interaktivnih sistema, međutim, imaju složenije U/I formate i upravljačku strategiju.



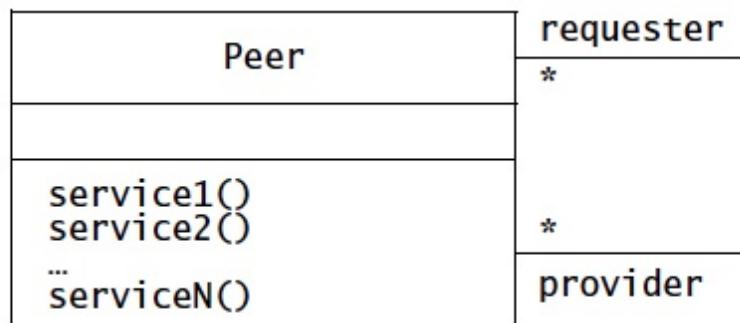
Slika 4.5.3 Primer arhitekture cevi i filtra [1.2]

## ✓ 4.6 Peer-to-peer arhitektura

### PEER-TO-PEER ARHITEKTONSKI ŠABLON

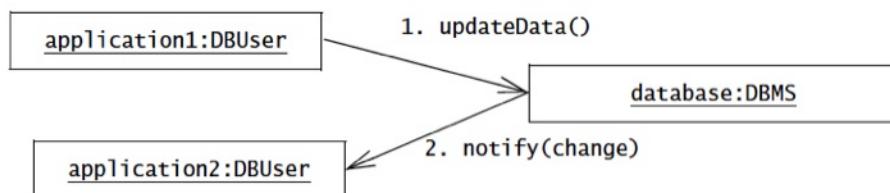
*Svi podsistemi mogu biti i serveri i klijenti*

**Peer-to-peer arhitektura** (slika 1) je generalizacija klijent-server arhitekture u kojoj svih podsistemi mogu biti i klijenti i serveri. Kontrolni tok unutar svakog podsistema je nezavisan od drugih, sem kod sinhronizacije zahteva.



Slika 4.6.1 Peer-to-peer arhitektura [1.2]

**Callback** su operacije koje su privremene i prilagođene specifčnoj svrsi. Primer na slici 2 pokazuje slučaj DBUser menja podatak u bazi podataka, a DBMS obaveštava drugog korisnika da je podatak izmenjen.



Slika 4.6.2 Primer za callback [1.2]

## ❖ 4.7 Servisno-orientisana arhitektura

### ŠABLON SERVISNO-ORIJENTISANE ARHITEKTURA

*Servisno-orientisana arhitektura organizuje aplikaciju kao kolekciju servisa koji komuniciraju jedan sa drugim preko dobro-definisanih interfejsa. Ovi servisi se nazivaju Veb servisima.*

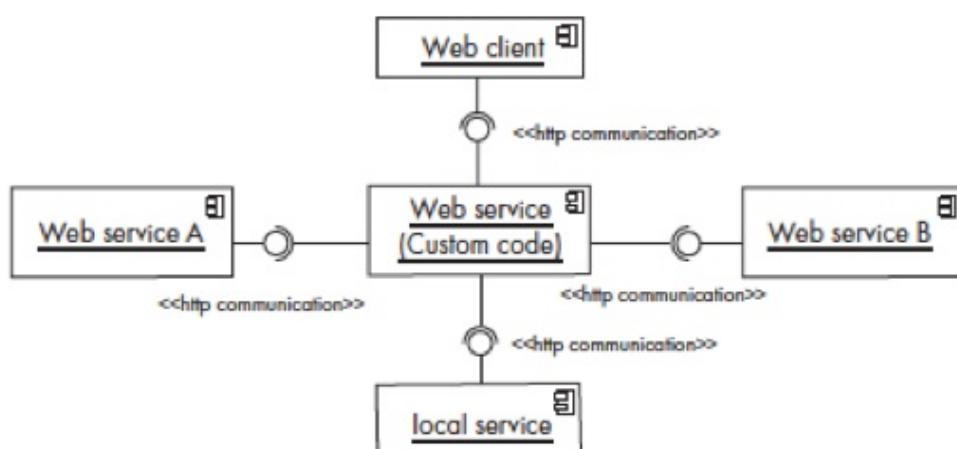
Servisno-orientisana arhitektura organizuje aplikaciju kao kolekciju servisa koji komuniciraju jedan sa drugim preko dobro definisanih interfejsa. Ovi servisi se nazivaju Veb servisima.

Veb servis je aplikacija koja je dostupna preko interneta a koja može da bude integrisana sa drugim veb servisima i na taj način, zajedno, čine jednu Veb aplikaciju. Da bi se veb servis koristio, morate da mu pošaljete zahtev tačno formatiran u HTTP u njegov HTTP server. U ovom slučaju operacija se obavlja mimo veb pretraživača, u pozadini. Server izvršava servisnu aplikaciju i vraća rezultat u vidu dokumenta, tipično, u strukturisanom jeziku, kao što je XML. Ovo je pokazano na slici 1.

Klijentski program koji razvijamo, dobija informaciju od Veb servisa koji obezbeđuje neka kompanija preko Interneta. To izaziva slanje

Klijentski program koristi protokole Veb servisa da bi pristupio lokalnom serveru koji radi u istoj kompaniji.

Ključni aspekt arhitektura Veb servisa je da svi podsistemi međusobno komuniciraju primenom otvorenih Wew standarda. Sve komponente sistema su povezane preko Interneta, bez obzira gde se nalazile. Zato, Veb servis može da se koristi od strane mnogih aplikacija se mnogih lokacija.



Slika 4.7.1 Servisno-orientisana arhitektura [1.2]

## ZAŠTO KORISTITI VEB SERVISE?

*Moraju se zadovoljiti dva zahteva: zaštita korisnika i informacija, kao i pouzdanost u radu.*

Veb servisi mogu da izvrše široki opseg zadataka, počev od obrade jednostavnih zahteva, do vrlo složenih poslovnih obrada. Organizacije mogu da koriste Veb servise da bi automatizovale i poboljšale svoje operacije. Na primer, aplikacije elektronskog poslovanja mogu koristiti veb servise da:

- pristupe bazama podataka o proizvodima svojih dobavljača,
- obrađuju kreditne kartice upotrebom Veb servisa banaka,
- organizuju isporuku upotrebom Veb servisa transportne organizacije.

Najveći izazov za inženjere softvera koji razvijaju Veb servise je sigurnost. Servis koji nudite, otvara vaše poslovne aplikacije i podatke vašim udaljenim klijentima. Zato, mora da se posebna pažnja posveti zaštiti i korisnika servisa i poslovnih informacija. Drugi važan aspekt je pouzdanost u radu, tj. raspoloživost i skalabilnost Veb servisa. Dve poznate platforme: J2EE i .NET možete koristiti da bi razvili aplikacije sa Veb servisima. To su veliki horizontalni radni okviri koji obezbeđuju, pored mnogih drugih stvari, zahtevanu funkcionalnost za razvoj interoperabilnih servisa. Obe platforme obezbeđuju fleksibilne, sigurne modele i druge mehanizme za podršku skalabilnosti i pouzdanosti.

Veb servisi nam omogućuju da zadovoljimo sledeće principe projektovanja softvera:

1. *Podeli i vladaj:* Aplikaciju čine nezavisni podsistemi koji su distribuisani i raspoloživi na Internetu.
2. *Povećati koheziju:* Veb servisi primenjuju slojevitu arhitekturu.
3. *Smanjiti međuzavisnost:* Veb aplikacije su labavo povezane.
4. *Povećati abstrakciju:* Klijenti servisa neznaju detalje oko načina implementacije veb servisa
5. *Povećati višestruku upotrebljivost komponenata.* Veb servisi su realizovani kao komponente za višestruku upotrebu
6. *Povećaj ponovnu upotrebljivost:* Veb servisi su po prirodi ponovo upotrebljivi.
7. *Predvideti zastarelost:* Kada zastari postojeća tehnologija Veb servisa, može se primeniti nova. Međutim, korisnici to neće osetiti jer je način komunikacije ostao nepromenjen.
8. *Portabilnost:* Veb servisi se mogu izmeniti na raznim računarim koji podržavaju Veb standarde.
9. *Lako testiranje:* Svaki servis ili usluga može se nezavisno testira.
10. *Defanzivno projektovanje:* Aplikacije pisane od različitih programera mogu da koriste isti servis.

## ✓ 4.8 Arhitektura sistema orijentisanog na poruke

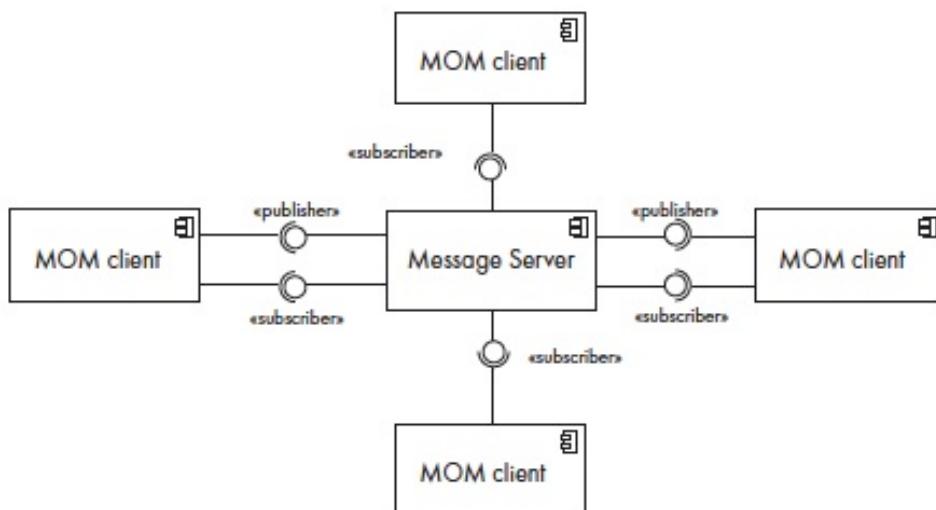
### MOM ARHITEKTURA

*MOM arhitektura je zasnovana na ideji da sistem radi na osnovu međusobne komunikacije podistema koji ga čine.*

**Arhitektura sistema koji se zasniva na porukama** je poznata i kao MOM (Message-Oriented Middleware). Ona je zasnovana na ideji da sistem radi na osnovu međusobne komunikacije podistema koji ga čine. Srž sistema je sistem za razmenu poruka između aplikacija. Podistemi u komunikaciji samo treba da znaju format poruke , a ne moraju da ništa da znaju o podistemu koji šalje ili prima poruku. Pored toga, podistemi ne moraju biti raspoloživi u isto vreme.

Slika 1 prikazuje osnovnu strukturu sistema koji se zasiva na porukama. Poruka je samostalni paket sa podacima aplikacije, zajedno sa podacima o rutiranju na mreži koje koristi sistem za prenos poruka. Poruke se šalju preko virtuelnih kanala koji se nazvaju - **teme (topics)**. Softverska komponenta koja šalje poruku je - **izdavač (publisher)** a komponenta koja prima poruke je **preplatnik (subscriber)**. Poruka poslata jednoj temi (topic), distribuira se svim preplatnicima (subscribers) te teme.

Aplikacija može da izabere da ignoriše poruku koju prima, ili da reaguje na nju. Na primer, slanjem odgovora u kome iznosi neki zahtev. Razmena poruka je vođena primenom dva principa upravljanja porukama: Prvo, poruke su potpuno asinhronne, što znači da nije definisano vreme kada će preplatnik dobiti ili koristiti poruku. Drugo, primenjuje se pouzdan mehanizam distribucije poruka koji garantuje da je poruka poslata samo jedanput.



Slika 4.8.1 Arhitektura sistema zasnovanog na porukama [1.2]

## PRIMENA MOM ARHITEKTURE

*Efikasnost sistema zavisi od sistema za isporuku poruka. JMS zadovoljava te zahteve.*

Mobilni telefoni predstavljaju primer jednog jednostavnog sistema koji se zasniva na porukama. Drugi primer, složenijeg sistema je sistem koji omogućava firmi koja obavlja prodaju proizvoda raznih dobavljača. Kada kompanija proda neki proizvod, ona šalje poruke odgovarajućoj temi. Kompanija koja obavlja funkciju skladišta tog proizvoda, kao pretplatnik te teme, dobija tu poruku i odmah pokreće proces isporuke proizvoda. Istovremeno, prodajno odeljenje će primiti istu poruku i koristiće je za izradu statističkog izveštaja o trendovima prodaje. Računovodstvo će obračunato maržu (zaradu) dobavljaču na osnovu poruke koju je dobro. Na kraju, kao pretplatnik, i fabrika koja je proizvela taj proizvod, dobija istu poruku, i na osnovu toga pravi plan dalje proizvodnje.

Efikasnost sistema zavisi od sistema za isporuku poruka. Pri projektovanju ovog sistema, moguće je primeniti dva pristupa:

1. Upotreba centralizovane arhitekture u kojoj sve poruke idu sa servera za slanje poruka svim pretplatnicima.
2. Upotreba decentralizovane arhitekture u kojoj sloj mreže dobija zadatak distribucije poruka klijentima.

JMS (**Java Mesaging System**) je API koji sadrži sva potrebna svojstva za razvoj i primenu aplikacija zasnovanih na porukama. JMS aplikacije su prenosive i mogu se koristiti u različitim aplikacijama koje komuniciraju primenom servisa različitih provajdera sistema za prenos poruka.

Arhitektura zadaka komponenta se nezavisno testira da zadovoljava sledeće principe projektovanja:

1. *Podeli i osvoji:* Komponente su izolovane komponente, nezavisno razvijene i distribuirane na mreži.
2. *Smanjiti međuzavisnost (coupling):* Komponente su u labavoj vezi jer dele samo format poruke.
3. *Povećati apstrakciju:* Ceo sistem je zasnovan samo na formatu poruke, te unutrašnja arhitektura komponenata nije bitna.
4. *Povećati vešestruku upotrebljivost:* Komponente se mogu koristiti u različitim kontekstima u meru u kojoj je format poruke prilagodljiv.
5. *Povećati ponovo korišćenje:* Komponente se mogu koristiti i kod novog sistema ako novi sistem koristi isti format poruka.
6. *Projektovanje za fleksibilnost:* Jednostavnim promenama komponenata (dodati, izbaciti, zameniti), sistem se lako može menjati, u skladu sa novom funkcionalnošću.
7. *Projektovanje za testiranje:* Svaka komponenta se nezavisno testira.
8. *Defanzivno projektovanje:* Sve primljene poruke se proveravaju pre korišćenja i ignorisu se one koje se ne mogu pravilno prevesti.

## WHAT IS MVC? SIMPLE EXPLANATION (VIDEO)

*Trajanje: 10,39 minuta*

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

## PEER TO PEER ARCHITECTURE (VIDEO)

*Trajanje: 2,17 minuta*

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

## ▼ Poglavlje 5

### Arhitekture aplikacije

## ARHITEKTURE APLIKACIONIH SISTEMA

*Aplikacioni sistemi su softverski sistemi koji se koriste radi zadovoljenja poslovnih ili organizacionih potreba neke firme.*

Aplikacioni sistemi su softverski sistemi koji se koriste radi zadovoljenja poslovnih ili organizacionih potreba neke firme. Firme u istom poslovnom sektoru imaju slične potrebe, te i aplikacioni sistemi u tim sektorima imaju sličnu funkcionalnost, tj. skup funkcija koje podržavaju. Ovi sistemi imaju softverske arhitekture koje odražavaju strukturu i organizaciju premereno organizacijama u koje obavljaju slične funkcije.

Aplikacione arhitekture sadrže glavne karakteristike određene klase softverskih sistema. Zato, postoje proizvodi koji imaju „standardnu“ arhitekturu za određenu klasu biznisa, a za potrebe svake organizacije u toj klasi, ti softverski sistemi imaju mogućnost prilagođavanja. Na taj način, firme ne moraju da razvijaju svoje, nove sisteme, već mogu da kupe odgovarajući standardni sistem i da ga onda prilagode svojim potrebama. Po pravilu, to bi trebalo da bude jeftinije, brže i pouzdano rešenje. Na primer, na ovaj način se najčešće danas primenjuju sistemi sa upravljanje poslovanjem organizacija, koji se nazivaju ERP sistemima ([Enterprise Resource Planning](#)). Najveći proizvođači tih sistema su danas SAP i Oracle.

Na tržištu se nude gotovi paketi (softverski sistemi) za pojedine uobičajene poslovne funkcije, tj. specijalizovane aplikacije za određene specifične poslove. Ovi sistemi se u konkretnoj situaciji konfigurišu i prilagođavaju konkretnim potrebama korisnika sistema. U tome mogu učestvovati specijalizovani spoljni konsultanti, koji predlažu najbolji način primene određenog opštег softverskog sistema.

## KORIŠĆENJE MODELA ARHITEKTURA APLIKACIJA

*Postoje više načina korišćenja modela arhitekture aplikacije.*

Ako ste projektant softvera možete koristiti model arhitektura aplikacije na različite načine:

1. *Kao početnu tačku procesa projektovanja arhitekture:* Ako nedovoljno poznajete vrstu aplikacije koju treba da razvijete, možete uzeti, kao početni uzor, arhitekturu opšte aplikacije za tu vrstu poslovanja, i onda prilagođavate tu arhitekturu potrebama aplikacije koju razvijate.
2. *Kao referencu za upoređenje:* Pošto ste razvili arhitekturu sistema koji projektujete, želite da uporedite vaše rešenje sa arhitekturom opštih aplikacija u tom domenu.

3. *Kao način da organizujete rad projektnog tima:* Imajući u vidu standardnu arhitekturu aplikacija u određenom domenu, možete raspodeliti zadatke razvoja arhitektura vašeg sistema po uobičajenim komponentama koje će i vaša arhitektura imati.
4. *Kao sredstvo za ocenu komponenti radi višestruke upotrebljivosti:* Pri projektovanju komponente sistema, želite da je napravite tako da se može koristiti i u drugim sistemima, tj. da se višestruko koristi. Zato je korisno da se ona upoređuje sa komponentama opštih sistema da bi ste videli da li ona ima svojstva koja su slična svojstvima komponentama opštih sistema. Ako ima, to je dobar indikator da se i vaša komponenta može koristiti i u drugim aplikacijama.
5. *Kao rečnik termina koji se koristi kod razgovora o aplikacijama određenog tipa:* Kod diskusija o specifičnim aplikacijama ili kod upoređivanja aplikacija istog tipa, možete koristiti koncepte opštih arhitektura kada govorite o aplikacijama.

## ZAJEDNIČKE ARHITEKTURE TIPOVA APLIKACIJA

*Aplikacije istog mogu se opisati zajedničkom arhitekturom koju sve aplikacije određenog tipa imaju.*

U praksi se koristi veliki broj različitih aplikacija (softverskih sistema razvijenih za određene poslove). Međutim, te aplikacije se u stvari ne razlikuju tako mnogo kao što na prvi pogled to izgleda. Imaju mnoge zajedničke elemente. Zato, one se mogu opisati zajedničkom arhitekturom koju sve aplikacije određenog tipa imaju. Radi ilustracije, ovde se navode dve takve arhitekture:

**1. Aplikacije za transakcione obrade:** To su aplikacije u kojima centralno mesto zauzima baza podataka. Aplikacija obrađuje zahtev za upis ili dobijanje informacije dobijenu od korisnika, u skladu sa tim, vrši promene u bazi podataka, odnosno isporučuje traženu informaciju korisniku (posle određene obrade, tj. pripreme izveštaja). Ovi tipovi aplikacija su najčešće primjenjeni kod interaktivnih poslovnih sistema. One su tako organizovane da akcija korisnika ne može da utiče na akcije drugih korisnika, a integritet baze podataka je uvek očuvan. Primeri takvi sistema su: interaktivni bankarski sistemi, sistemi e-poslovanja, informacioni sistemi organizacija, ili sistemi za rezervacije.

**2. Sistemi za obradu jezika:** U ovim sistemima korisnik se izražava primenom nekog formalnog jezika (npr u Javi). Sistem obrađuje ovaj jezik i pretvara ga u neki unutrašnju format (oblik) i onda interpretira (predstavlja) to kao svoje unutrašnje predstavljanje korisničkog iskaza. Na primer, tako rade tzv. kompjajleri, tj. prevodioци programskih jezika, koji prevode programe viših programskih jezika u unutrašnji mašinski kod računara. Pored toga, ovaj tip sistema može da prevodi komande namenjene sistemima baza podataka (npr. SQL) ili jezika za markiranja (npr. XML).

## VIDEO PREDAVANJE ZA OBJEKAT "ARHITEKTURE APLIKACIJE"

*Trajanje video snimka: 17min 56sek*

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

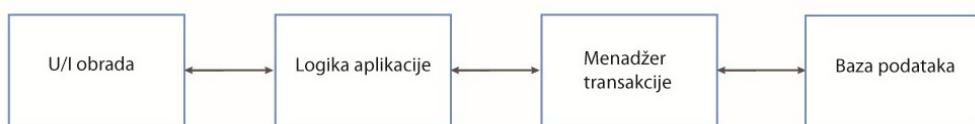
## ❖ 5.1 Sistemi za transakcionu obradu

### TRANSAKCIJE I STRUKTURA TRANSAKCIIONIH SISTEMA

*Sistemi za transakcionu se projektuju da obezbede obradu zahteva korisnika za informacijama iz baze podataka, ili zahteve za unos novih podataka u bazu podataka*

**Sistemi za transakcionu obradu** (engl. **Transaction processing**- TP) se projektuju da obezbede obradu zahteva korisnika za informacijama iz baze podataka, ili zahteve za unos novih podataka u bazu podataka. Transakcija u sistemu baza podataka je niz operacija koji se tretira kao jedna jedinica (atomska jedinica) koja se mora izvršiti u celini pre nego što se stanje u bazi podataka trajno promeni (tj. upisani podaci postaju memorisani). To obezbeđuje integritet baze podataka, jer i u slučaju nekog poremećaja u toku procesa upisivanja novih podataka ili čitanja postojećih, ne može doći do nedefinisanih stanja baze. Transakcija (definisan niz operacija) mora se u celosti izvršiti, ili se poništavaju izvršene operacije niza i sistem se postavlja na početak, tj. u stanje u kome je bio pre početka izvršenja prve operacije u transakciji. Sistemi za transakcionu obradu su najčešće u formi interaktivnih sistema u kojima korisnici postavljaju asinhronne zahteve za određene servise sistema. .

Na slici 1 prikazana je koncepcijska arhitektura jedne TP aplikacije. Najpre korisnika postavlja zahtev preko U/I komponente sistema. Taj zahtev se onda obrađuje primenom odgovarajuće logike aplikacije. Kreira se transakcija kojom rukovodi komponenta „Menadžer transakcija“. To je najčešće komponenta sistema za upravljanje bazama podataka (engl. DBMS - **Database Management Systems**). Kada menadžer transakcija utvrdi da je transakcija u celosti završena, on šalje signal aplikaciji da je obrada zahteva završena



Slika 5.1.1 Struktura sistema za transakcionu obradu

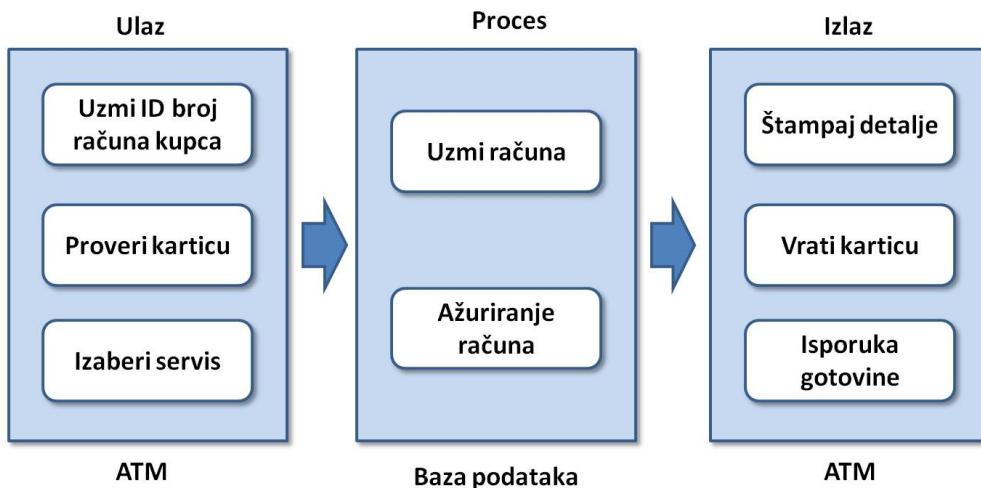
### PRIMER TRANSAKCIJONOG SISTEMA – ATM SISTEM

*Transakcija postaje kompletirana i konačno izvršena tek kada su sve operacije predviđene za realizaciju neke funkcije urađene bez problema i prekida.*

Sistemi za transakcionu obradu često se projektuju po šablonu arhitekture „cevi i filtera“, te sistem ima komponente koje su odgovorne za ulaz zahteva i podataka, obradu podataka i

izlaz traženih rezultata. Jedan primer transakcionog sistema je softverski sistem bankomata (engl. ATM – **Automated Teller Machine**). Na slici 2 prikazana je arhitektura ATM sistema. Ulazne i izlazne komponente su primenjene softverski u okviru bankomata (ATM) a obradu podataka računa vrši bankarski sistema za upravljanje bazama podataka.

Transakcija postaje kompletirana i konačno izvršena tek kada su sve operacije predviđene za realizaciju neke funkcije urađene bez problema i prekida. Tek tada se promeni stanje na računu štediše, ako je povukao ili prebacio novac na neki drugi račun.



Slika 5.1.2 Arhitektura ATM softverskog sistema

## ARHITEKTURA INFORMACIONIH SISTEMA

*Svi sistemi koji koriste interakciju sa zajedničkom bazom podataka su tzv. transakcionalni informacioni sistemi.*

Svi sistemi koji koriste interakciju sa zajedničkom bazom podataka su tzv. transakcionalni informacioni sistemi. Informacioni sistem dozvoljava kontrolisan pristup velikoj bazi informacija, kao što je katalog bibliotečkih jedinica, red vožnje letova, ili elektronski zdravstveni kartoni pacijenata u nekoj bolnici. Najčešće, informacioni sistemi su i veb sistemi, jer obezbeđuju pristup preko veb pretraživača.

Na slici 3 prikazan je jedan vrlo uopšteni model jednog informacionog sistema. Sistem koristi slojevitu arhitekturu, u kojoj sloj na vrhu obezbeđuje interfejs sa korisnicima, a najniži sloj, sa bazom podataka sistema. Komunikacioni sloj obezbeđuje sve izlaze i izlaze iz korisničkog interfejsa, a sloj za prikupljanje informacija koristi specifičnu logiku aplikacije za pristup bazi i za upisivanje novih podataka u njoj.



Slika 5.1.3 Slojevita arhitektura informacionog sistema

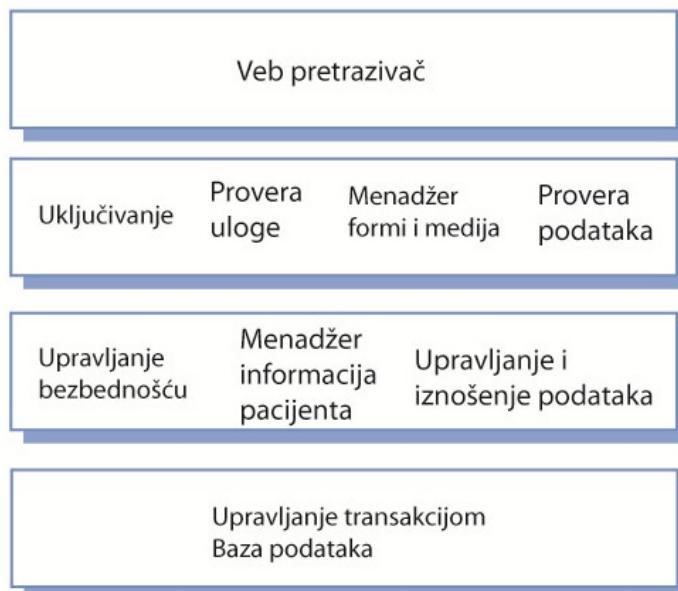
## PRIMER INFORMACIONOG SISTEMA: MHC-PMS SISTEM

*Sistem održava i upravlja informacijama o pacijentima sa mentalnim problemima*

Jedan primer informacionog sistema (MHC-PMS) sa slojevitom arhitekturom prikazan je na slici 4 . Sistem održava i upravlja informacijama o pacijentima sa mentalnim problemima:

1. Sloj na vrhu je odgovoran za korisnički interfejs, koji je preko veb pretraživača.
2. Drugi sloj obezbeđuje funkcionalnost korisničkom interfejsu. Sadrži komponente za uključenje korisnika i koje proveravaju da li korisnik koristi samo dozvoljene operacije, u skladu sa svojom ulogom. Ovaj sloj upravlja formama na kojima se predstavljaju informacije korisnicima, a komponenta za validaciju podataka proverava konzistentnost informacija.
3. Treći sloj primenjuje funkcionalnost sistema jer ima komponenta koje obezbeđuju bezbednost sistema, kreiranje i održavanje informacija o pacijentima, a daje na izlazu podatke o pacijentima za prebacivanje u druge baze podataka, kao i komponente koje vrše generisanje izveštaja o radu sistema.
4. Najniži sloj, sadrži sistem za upravljanje bazom podataka (DBMS) koji obezbeđuje upravljanje transakcijama i trajno skladište podataka

Sloj koji obezbeđuje upravljanje podacima, može da koristi ne samo jednu, već više servera sa sistemima za upravljanje podacima, zavisno od potrebama aplikacije. To dovodi do efikasnog upravljanja velikim brojem transakcija i baza sa velikim brojem podataka.



Slika 5.1.4 Arhitektura MHC-PMS sistema

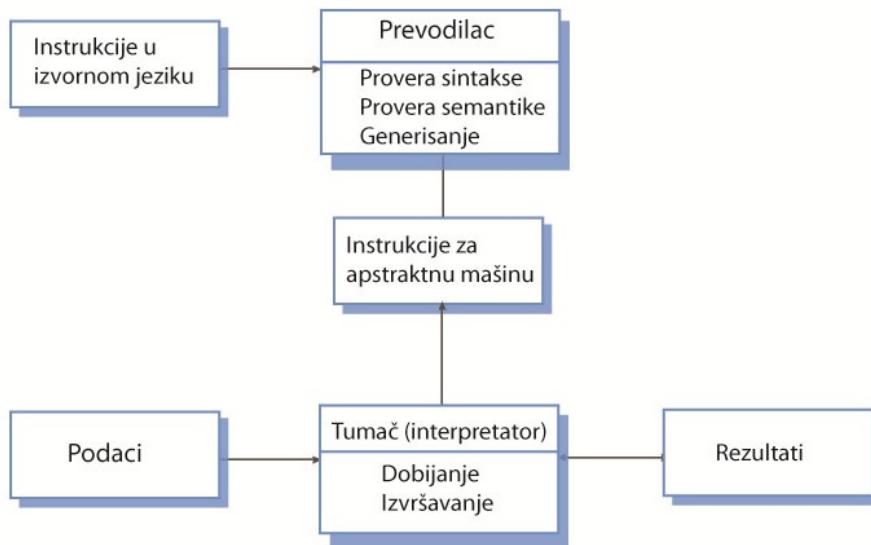
## 5.2 Sistemi za obradu jezika

### ARHITEKTURA SISTEMA ZA OBRADU JEZIKA

*Sistemi za obradu jezika prevode neki prirodan ili veštački jezik u neki drugi način predstavljanja tog jezika*

Sistemi za obradu jezika prevode neki prirodan ili veštački jezik u neku drugi način predstavljanja tog jezika. U slučaju programskega jezika, dobija se mašinski kod koji omogućava računarsko izvršenje programske instrukcije. Drugi jezici u obliku XML forme podataka mogu se prevesti u komande za rad sa bazom podataka ili u neku drugu XML formu predstavljanja podataka. U slučaju prirodnih jezika, prevodenje može da dovede do prevoda u drugom prirodnom jeziku (npr., sa engleskog u nemački).

Slika 1 prikazuje jednu od mogućih arhitektura sistema za obradu jednog programskega jezika. Instrukcije pisane u izvornom programskom jeziku se prevode u izvršne instrukcije apstraktne mašine. Te instrukcije se onda obrađuju od strane jedne komponente koja priprema instrukciju za izvršenje, uzimajući i podatke o izvršnom okruženju. Izlaz iz ovog procesa su prevedene programske instrukcije sa ulaznim podacima.



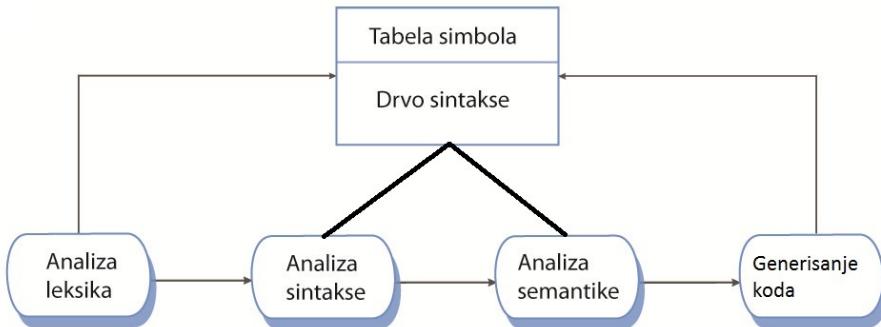
Slika 5.2.1 Arhitektura sistema za obradu jezika [1.2]

## PRIMER: ARHITEKTURA KOMPAJLERA

*U slučaju kompjajlera (programskih prevodioca), interpreter je hardver koji obrađuje mašinske instrukcije, a apstraktna mašina je stvarni procesor.*

U slučaju kompjajlera (programskih prevodioca), interpreter je hardver koji obrađuje mašinske instrukcije, a apstraktna mašina je stvarni procesor. Kod dinamičkih tipova jezika, kao što je Python, interpreter može biti softverska komponenta. Na slici 2 prikazana je opšta arhitektura prevodioca programskih jezika (kompjajlera), koja sadrži sledeće komponente:

1. *Leksički analizator*, koji prevodi ulazni jezički slog i prevodi ga u neku unutrašnju formu.
2. *Simbolička tabela*, koja sadrži informacije o imenima entiteta (promenljive, imena klase, imena objekata i dr.) a koji su sadržani u tekstu koji treba da se prevede (tj. u izrazima programskog jezika).
3. *Sintaksni analizator*, koji proverava sintaksu programskog jezika koji se prevodi, koristeći definisanu gramatiku jezika i generišući sintaksno drvo.
4. *Sintaksno drvo*, koje predstavlja unutrašnju strukturu programa koji se prevodi.
5. *Semantički analizator*, koji koristi informacije preuzete sa sintaksnog drveta i simboličke tabele i proverava semantičku ispravnost teksta ulaznog jezika.
6. *Generator koda*, koji „ide“ po sintaksnom drvetu i generiše apstraktni mašinski kod.



Slika 5.2.2 Arhitektura kompjajlera izrađena po šablonu cevi i filtra [1.2]

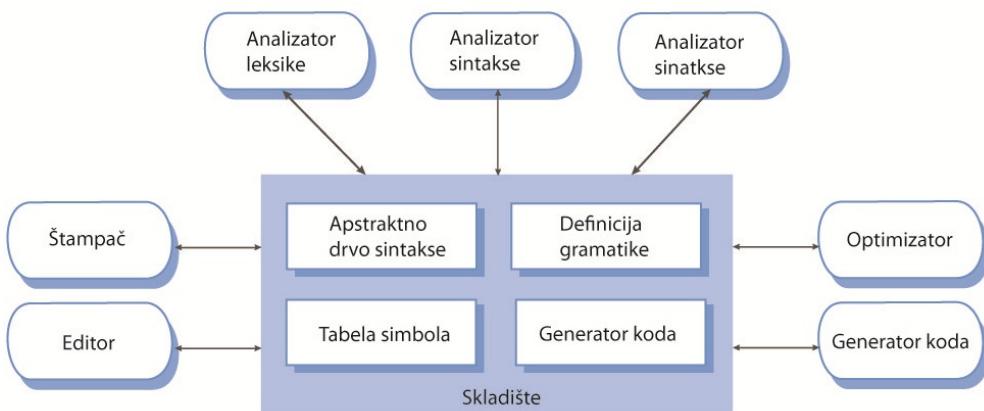
## PRIMER: PREVOĐENJE PRIRODNOG JEZIKA

*Arhitektura sistema za prevođenje programskog jezika sadrži skladište, tj. arhitekturu koja je urađena po arhitektonskom šablonu sa skladиштем*

Pored navedenih, mogu se koristiti i dodatne komponente koje mogu da analiziraju i prevode sintaksno drvo radi povećanje efikasnosti i izbacivanja redundantnosti iz generisanog mašinskog koda.

U slučaju sistema za prevođene prirodnih jezika, koriste se i druge komponente, kao što je rečnik, a generisan kod je ulazni tekst koji je preveden u drugi jezik.

Ako se kompjajler nalazi kao deo nečega softverskog sistema za razvoj softvera, onda je efikasnija nešto drugačija arhitektura, koja bolje podržava interaktivnost nego ona na slici 2. Na slici 3 prikazana je arhitektura sistema za prevođenje programskog jezika koja sadrži skladište (engl., **repository**), tj. arhitekturu koja je urađena po arhitektonskom šablonu sa skladijštem.



Slika 5.2.3 Arhitektura sa skladištem sistema za prevođenje programskog jezika [1.2]

## THE DIFFERENCE BETWEEN SOFTWARE ARCHITECTURE AND SOFTWARE DESIGN SOFTWARE ARCHITECT CA (VIDEO)

*Trajanje: 4 minuta*

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

## COMPLEX SYSTEMS DESIGN: 14 EVENT DRIVEN ARCHITECTURE (VIDEO)

*Trajanje: 7,38 minuta*

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

## ▼ Poglavlje 6

# Projektna dokumentacija

## ZAŠTO DOKUMENTACIJA? KOME JE NAMENJENA DOKUMENTACIJA?

*Dokumentacija dovodi do boljeg projektnog rešenja i poboljšava komunikaciju sa njegovim korisnicima.*

Projektna dokumentacija ima dve svrhe:

1. Omogućava projektantnu ili projektnom timu da donosi dobre odluke u vezi projektnog rešenja, jer omogućava dodatno razmišljanje o onome što je projektovano.
2. Omogućava komunikaciju o projektnom rešenju sa drugima.

Pri izradi dokumentacije mogu se uočiti slabosti u projektnom rešenju. Pri recenziji rešenja, takođe se može doći do poboljšanja. Inženjeri softvera često izbegavaju izradu dokumentacije, ili je rade kada je projekat na kraju. To je velika greška, jer se kasnije, mogu otkriti manjkavosti u projektnom rešenju. Zamislite da neko gradi zgradu ili proizvodi automobil bez dokumentacije? U svim oblastima inženjerstva, tehnička dokumentacija je preduslova za uspešan rad i realizaciju projekta. Tako treba da bude i u softverskom inženjerstvu. Kao mlada inženjerska disciplina, i ona mora da poštuje pravila i iskustva drugih inženjerskih disciplina.

Početi programiranje aplikacije, bez prethodno pripremljene dokumentacije, može dovesti do nefleksibilnosti sistema i do prevelike složenosti softverskog sistema.

Pri pisanju dokumentacije, morate znati kome se obraćate dokumentacijom. Projektnu dokumentaciju će koristi sledeće tri gupe ljudi:

- **Programeri**, koji treba da primene projektno rešenje sistema
- **Inženjeri softvera** koji će u budućnosti menjati sistem
- **Inženjeri softvera** koji razvijaju podsistem koje, preko interfejsa, treba da bude povezan sa vašim sistemom.

Zavisno od osoba kojima je namenjen dokument, ili deo dokumenta, projektant odlučuje koje informacije da uključi u dokumentaciju.

Važno je da se ne daje samo gotovo, urađeno projektno rešenje. Pored toga, važno je navesti razloge zbog kojih je pripremljeno ovakvo rešenje. Čitalac će onda bolje shvatiti projektno rešenje. Recenzenti će lakše proveriti da li su donete dobre odluke projektanata, a inženjeri održavanja će lakše moći da predlože usavršavanja projektnog rešenja.

## SADRŽAJ PROJEKTNOG DOKUMENTA

*Dokument se piše u skladu ciljnom grupom čitalaca, i treba da ima i odgovarajuće informacije a i odgovarajuću dužinu.*

Preporučuje se da dokument sadrži sledeće informacije:

1. **Svrha:** Navedite koji sistem ili deo sistema opisuje dokument. Dajte reference ka zahtevima koji su primenjeni.
2. **Opšti prioriteti:** Navedite prioritete koje ste koristili pri izradi projektnog rešenja.
3. **Sažet opis projektnog rešenja:** Dajte sažeti opis projektnog rešenja da bi ga čitaoc brzo razumeo. Dijagrami u tome mogu biti od velike koristi.
4. **Glavna projektna pitanja:** Iznesite važna pitanja na koje ste nalazili odgovore. Navedite i moguće alternative koje ste razmatrali, i dajte razloge za odluke koje ste doneli.
5. **Detalji projektnog rešenja:** Navedite sve informacije koje čitalac treba da zna, a koje do sada niste dali. Ovde možete uključiti detaljne opise protokola za komunikaciju između klijenta i servera, opis strukture podataka i algoritama, kao i upotrebu različitih API.

Generalno, kada pišete projektni dokument, gledajte da ne bude ni isuviše kratak, ali ni isuviše dug. Primena dokumenta odgovarajuće dužine omogućava uspešno prenošene informacije onima kojima je dokument namenjen. Ne zaboravite da ne treba davati informacije koje čitaoc već zna te ih nikada neće čitati izbegnite davanje informacija koje čitalac može lako dobiti iz drugih izvora.

Obratite pažnju na sledeće preporuke:

- Izbegnite opis informacije koja je nepotrebna, jer je poznata, vešt tom programeru ili projektantu.
- Izbegnite da pišete detalje u projektnom dokumentu, koje je bolje pisati u vidu komentara u programskom kodu.
- Izbegnite da pišete o detaljima koji se mogu automatki dobiti iz samog programskega koda. Kao što je lista javnih metoda.

Kako se kod često menja, zato je potrebno te komentare pisati u samom kodu, da ne bi kasnije morale izmene vršiti na dva mesta: u samom kodu, ali i u projektnom dokumentu.

## SOFTWARE ARCHITECTURE DOCUMENT (VIDEO)

*Trajanje 12 minuta*

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

## TRADITIONAL DESIGN DOCUMENTATION (VIDEO)

*Trajanje: 1,22 minuta*

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

## ▼ Poglavlje 7

# Preporuke za dobro projektno rešenje

## IZAZOVI I RIZICI PROJEKTOVANJA

*Preporuke za pripremu dobrog projektnog rešenja arhitekture sistema*

1. Kao i modelovanje, i projektovanje je **veština koja traži odgovarajuće iskustvo**. Pri projektovanju, morate da ocenjujete različite alternative. Da bi to uspešno radili, morate da imate odgovarajuće znanje o njima da bi ih pravilno ocenili i mogli da ocenite i posledice vaše odluke. **Rešenje:** Ne pokušavajte da projektujete veliki sistem dok ne steknete odgovarajuće iskustvo u projektima razvoja softvera. Aktivno proučavajte projektna rešenja drugih sistema, i onih za koje se smatra da su dobra, ali i onih za koje se smatar da su loša.
2. Slabo projektno rešenje će dovesti **do većih troškova održavanja**. Na primer, sistemi sa visokom međuzavisnošću a niskom kohezijom svojih podsistema, biće teški za promenu, imaće mnogo nedostataka, i doveće do bržeg pada performansi sistema pri korišćenju. **Rešenje:** Primenite principe dobrog projektovanja. Koristite tehnikе modelovanja i primenite šablona projektovanja. Zadržite recenzije projektnog rešenja tako da i drugi mogu da razumeju vaše projektno rešenje i da razumeju vaše odluke.
3. **Obezbeđenje da neki softverski sistem ostane dobar tokom celog svog životnog veka**, zahteva stalni napor. Projektno rešenje, tokom korišćenja, počinje da pokazuje slabosti, i sistem gubi željena svojstva, i kao posledica stalnog dodavanja novih funkcija i promenama u softveru. Oni koji menjaju softver, često nisu ni razumeli dobro projektno rešenje softvera.. **Rešenje:** Pripremite vaše projektno rešenje da bude što fleksibilnije, tako da može da lako prihvate buduće promene i proširenja. Obezbedite da projektna dokumentacija bude upotrebljiva i sa odgovarajućim nivoom detalja, tako da oni koji održavaju softver, mogu da je uspešno koriste. Obezbedite pažljivo upravljanje promenama i sprovođenje recenzija svih promena zahteva i projektnog rešenja. Ako primena novih zahteva ugrožava integritet arhitekture, onda sprovedite reinženjering sistema da bi ga osposobili da uspešno prihvati neophodne promene, a pre svega njegovu arhitekturu.

## ISSUES WITH DESIGN (VIDEO)

*Trajanje: 1 minut*

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

# OO SYSTEMS ANALYSIS AND DESIGN - ESSENTIALS OF DESIGN

*Trajanje: 30:42 minuta*

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

## ✓ Poglavlje 8

### Pokazna vežba

#### UVOD U VEŽBE

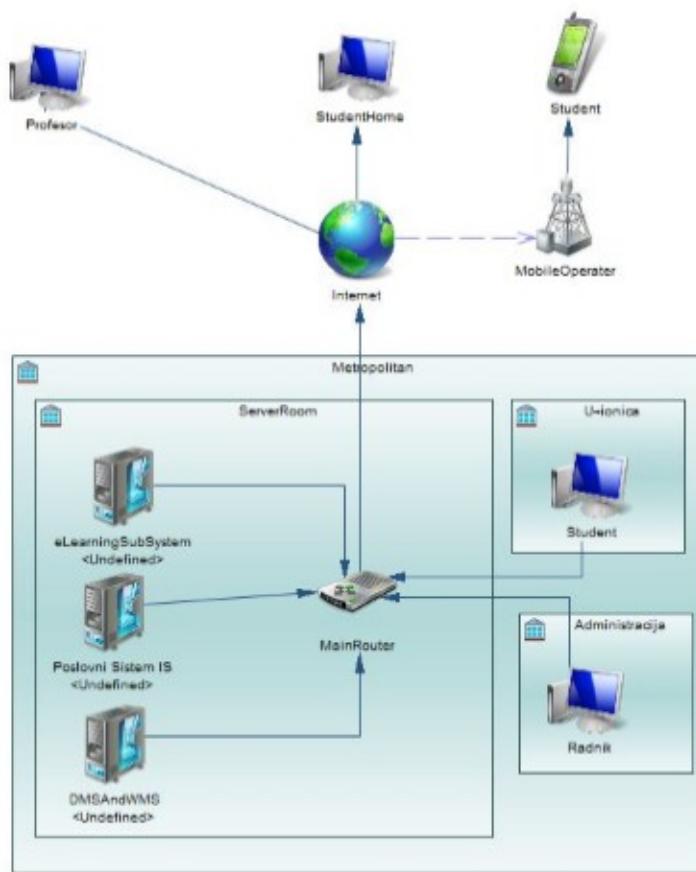
##### *Cilj i opis vežbi*

Ova vežba demonstrira izradu i upotrebu infrastrukturnog dijagrama i dijagrama arhitekture. Kompletna izrada dijagrama i njihova upotreba demonstrirana je na video snimku koji se nalazi u okviru dodatnog materijala lekcije.

Pored primera prikazanih na video snimku u dodatnom materijalu je moguće preuzeti i Power Designer model dijagrama arhitekture i infrastrukturni dijagram.

#### INFRASTRUKTURNI I DIJAGRAM ARHITEKTURE (15 MINUTA)

*Dijagram arhitekture trebalo bi da se prikaže na jednoj strani*



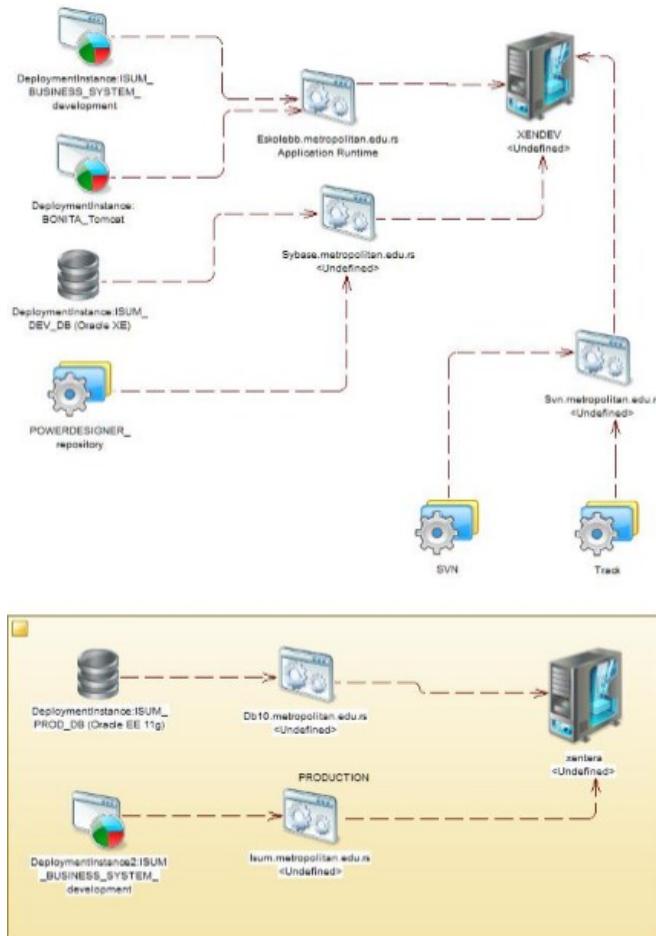
Slika 8.1.1 Primer jedne arhitekture [Izvor: Autor]

Sistem prikazan na slici 1 prikazuje arhitekturu institucionalnog sistema Univerziteta Metropolitan. Pored jasno naznačenih elemenata da sistem sadrži deo koji koriste profesori i studenti putem svojih računara, obezbeđena je i mobilna aplikacija, namenjena studentima.

Sistem prikazuje i unutrašnju infrastrukturu servera koji obezbeđuju funkcionisanje ovakvog sistema.

## INFRASTRUKTURNI I DIJAGRAM ARHITEKTURE - NASTAVAK

### Dijagram arhitekture - primer 2



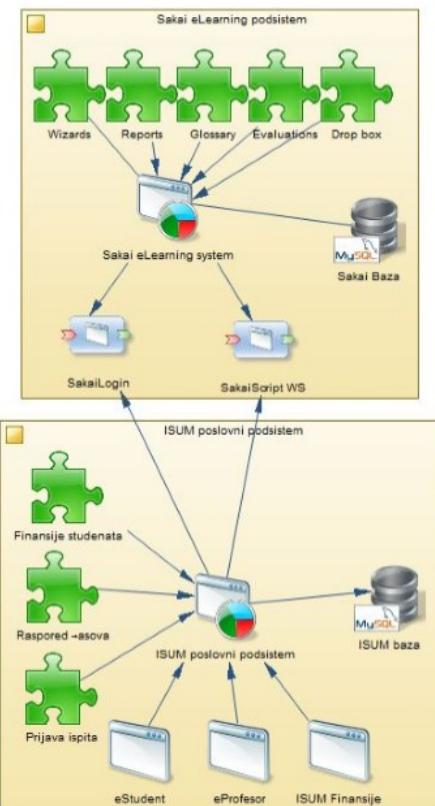
Slika 8.1.2 Primer arhitekture [Izvor: Autor]

Dijagram na slici 2 prikazuje detaljnije interne elemente arhitekture sistema za upravljanje poslovanjem jedne ustanove.

Na slici su navedeni repozitorijumi za čuvanje dokumentacije i PoweDesigner repository.

## INFRASTRUKTURNI I ARHITEKTONSKI DIJAGRAMI (15 MINUTA)

*Primer dijagrama arhitekture i infrastrukturnih dijagrama dizajniranih korišćenjem Power Designer alata*



Slika 8.1.3 Primer arhitektonskog dijagrama sistema [Izvor: Autor]

Slika 3 prikazuje internu strukturu informacionog i eLearning sistema sa detaljnijim komponentama koje ga čine. Arhitekture i njihove baze su definisane posebnim slikama, ali je prikazana i interakcija dva sistema u cilju ostvarivanja zajedničkog cilja.

## NAPSTER (VIDEO)

*Georgia Tech predavanje - Napster*

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

## SKYPE (VIDEO)

*Georgia Tech predavanje - Skype*

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

## ZAVRŠNA PORUKA (VIDEO)

*Georgia Tech predavanje - Završna poruka*

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

- ✓ 8.1 Pokazni primer 1 - Dijagram arhitekture restorana

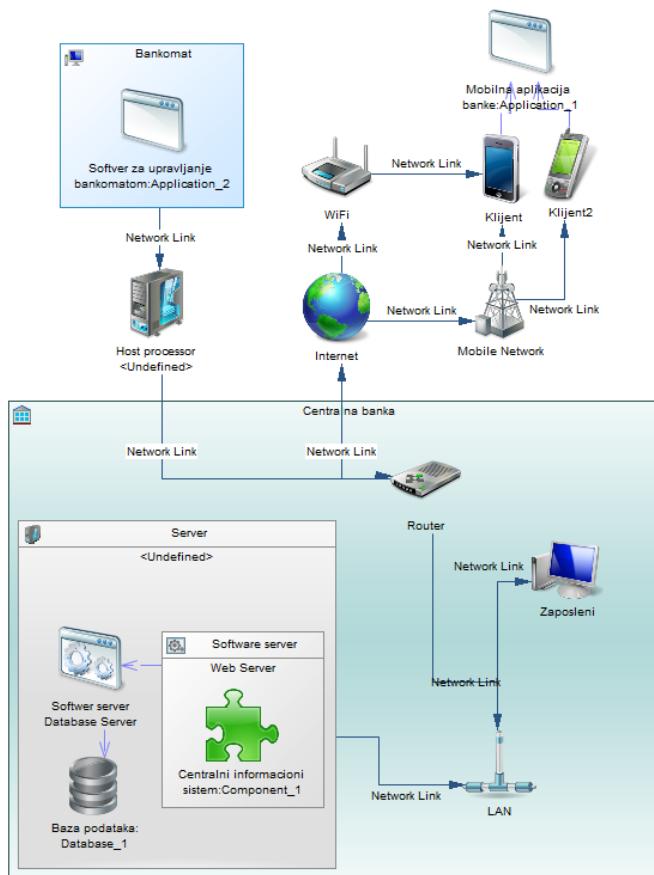
## ARHITEKTURA SOFTVERSKOG SISTEMA BANKE (20 MINUTA)

*Infrastrukturni dijagram arhitekture bankarskog sistema*

Kako je reč o bankarskom sistemu, gde je centralni deo sistema obrada transakcija, pogodan izbor za arhitekturu aplikacija koje čine sistem jeste arhitektura **aplikacija za transakciju obradu**. Pored toga, komponente sistema imaju svoje specifičnosti koje se mogu predstaviti odgovarajućim arhitekturama.

Mobilna aplikacija za klijente banke ima **klijent-server** arhitekturu. Deo koji se izvršava na samom telefonu predstavlja klijenta, dok server banke obrađuje zahteve, tj. transakcije i predstavlja serverski deo. Softver koji upravlja bankomatom predstavlja primer **transakcione aplikacije** koje obično koriste arhitekturu "filtera i cevi". Centralni bankarski sistem koji upravlja podacima u bazi i omogućava nadgledanje transakcija od strane zaposlenih predstavlja primer arhitekture **transakcionih informacionih sistema**.

Na dijagramu se može videti infrastrukturni dijagram arhitekture koji sadrži i 3 osnovne komponente sistema. Logički banka ima dva servera jedan koji upravlja bazom podataka, a drugi koji pokreće centralni informacioni sistem banke. Radnici banke se preko lokalne mreže povezuju na sistem i imaju uvid u transakcije. Banka klijentima pruža pristup sistemu preko Interneta. Bankomat obično ima posebnu mrežu, koja može da koristi infrastrukturu ISP (Internet Service Provider), ali i ne mora. Dodatno, potreban je poseban server (na koji je povezano više bankomata različitih banaka) koji će da obradi zahtev bankomata i odredi kojoj banci šalje podatke.



Slika 8.2.1 Primer arhitekture jednostavnog sistema banke [Izvor: Autor]

## ✓ 8.2 Pokazni primer 2 - Dijagram arhitekture banke

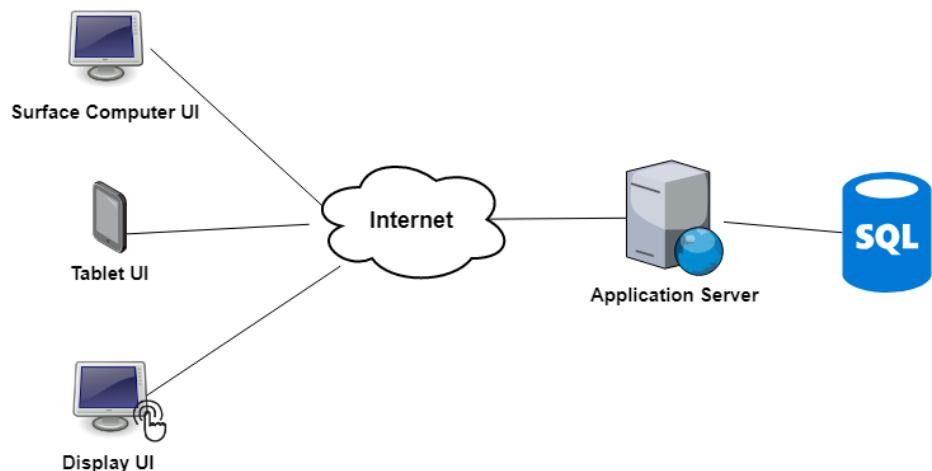
### ARHITEKTURA SOFTVERSKOG SISTEMA RESTORANA (20 MINUTA)

#### *Infrastrukturni dijagram arhitekture restorana*

Kako je reč o sistemu restorana gde je centralni deo obrada podataka prikupljena od različitih klijenata, pravi izbor arhitekture jeste klijent-server arhitektura.

Centralni deo sistema se nalazi na aplikacionom serveru, koji u komunikaciji sa bazom podataka obrađuje i skladišti sve neophodne podatke prikupljene od različitih klijenata.

Nivoom privilegija obezbeđeno je da svaki klijent pristupa delu aplikacije koji je njemu namenjen u skladu sa zahtevima definisanim u prethodnim lekcijama. Po potrebi u aplikaciji može biti definisan mod za offline rad, ali je podrazumeva da internet ili lokalna mreža nisu u prekidu duže od 1 minuta.



Slika 8.3.1 Klijent-server arhitektura [Izvor: Autor]

## ▼ Poglavlje 9

### Individualna vežba

## ZADACI ZA INDIVIDUALNI RAD - ZADACI OD 1 DO 3

### *Definisanje arhitekture sistema*

#### **Zadatak 1** (20 minuta)

Definisati arhitekturu sistema za rezervaciju bioskopskih karata. Nacrtati osnovne komponente korišćenjem PowerDesigner-a. Da li neka od standardnih arhitektura može da se primeni na ovaj problem?

#### **Zadatak 2** (20 minuta)

Definisati arhitekturu sistema za kontrolu i monitoring železničkog saobraćaja. Nacrtati osnovne komponente korišćenjem PowerDesigner-a. Da li neka od standardnih arhitektura može da se primeni na ovaj problem?

#### **Zadatak 3** (20 minuta)

Definisati arhitekturu softvera za automatsko prepoznavanje bolesti kože sa medicinskih fotografija. Nacrtati osnovne komponente korišćenjem PowerDesigner-a. Da li neka od standardnih arhitektura može da se primeni na ovaj problem?

#### **Zadatak 4** (30 minuta)

Nacrtajte dijagrame koji pokazuju koncepcijski i procesni pogled na arhitekturu sledećih sistema: a) Sistema za automatsku prodaju železničkih karata na železničkoj stanici, b) Kompjuterski upravljan sistem za video konferencije, a koji dozvoljava video, audio, i računarskih podataka dostupnim učesnicima video konferencije, c) robotizovani čistač podova. On mora da ima senzore za izbegavanje zidova i drugih prepreka.

## ZADACI ZA INDIVIDUALNI RAD - ZADACI OD 5 DO 6

### *Rešavanjem ovih zadataka, student dobija bolje razumevanje problema projektovanja arhitekture*

#### **Zadatak 5** (20 minuta)

Korišćenjem PowerDesigner-a nacrtajte arhitekturu sistema za prodaju muzike preko Interneta koju predlažete. Koji arhitektonski šabloni su osnova za njihovu arhitekturu?

#### **Zadatak 6** (20 minuta)

Definisati arhitekturu sistema za kontrolu i monitoring automatskog pilota. Nacrtati osnovne komponente korišćenjem PowerDesigner-a. Da li neka od standardnih arhitektura može da se primeni na ovaj problem?

## ✓ Zaključak

### POUKE LEKCIJE

#### *Sumiranje stečenih znanja*

1. Arhitektura softvera je opis organizacije nekog softvera. Od njegove arhitekture zavise svojstva sistema, kao što su performanse, bezbednost i raspoloživost.
2. Odluke projektovanja arhitekture obuhvataju odluka o tipu aplikacije, distributivosti sistema, arhitektonskom stilu koji će se koristiti, i o načinima na kojima će se dokumentovati i vrednovati arhitektura
3. Arhitekture softvera se mogu dokumentovati koristeći nekoliko različitih pogleda na softver, kao što su: koncepcijski pogled, logički pogled, procesni pogled, razvojni pogled i fizički pogled.
4. Arhitektonski šabloni su sredstvo višestrukog korišćenja opštih arhitektura sistema. One opisuju arhitekturu, objašnjavaju kada mogu da budu upotrebljavane, i diskutuju prednosti i nedostatke koje nude.
5. Česti arhitektonski šabloni su: Model-pogled-kontroler, Slojevita arhitektura, Skladište, Klijent-server, i Cev i filter.
6. Opšti modeli arhitektura aplikacionih sistema pomažu razumevanje rada aplikacija, upoređenje sa drugim aplikacijama istog tipa, odobravanje projektnog rešenja informacionog sistema, i višestruku upotrebljivost komponenta.
7. Sistemi za obradu transakcija su interaktivni sistemi koji omogućuju udaljeni pristup informacijama u bazi podataka i njihove promene od strane svojih korisnika. Primeri transakcionih obradnih sistema su informacioni sistemi i sistemi za upravljanje resursima.
8. Sistemi za obradu jezika se upotrebljavaju za prevodenje teksta iz jednog jezika u drugi i za prevodenje instrukcija definisanih ulaznim jezikom. To može biti prevodilac i neka apstraktna mašina koja izvršava generisan jezik.

### LITERATURA

#### *Literatura koje se preporučuje studentima*

##### **1. Obavezna literatura:**

1. Onlajn nastavni materijal na predmetu SE201 Uvod u softverski inženjeriranje, školska 2017/18, univerzitet Metropolitan
2. Ian Sommerville, Software Engineering, Tenth Edition, Pearson Education Inc., 2016.

##### **2. Dopunska literatura:**

1. T.C.Lethbridge, R. Lagariere - Object-Oriented Software Engineering - Practical Software Development using UML and Java - 2005

2. B. Bruegge, A. Dutoit, Object-Oriented Software Engineering – Using OML, Patterns, and Java, Thirth Edition, Prentice Hall, 2010
3. P. Stevens, Using UML – Software Engineering with Objects and Components, Second Edition, Assison-Wesley, Pearson Eduction, 2006
4. R. Pressman, Software Engineering – A Practioner's Approach, Seventh Edition, McGraw Hill Higher Education, 2010
5. Partha Kuchana, Software Architecture Design patterns in Java

**Veb lokacije :**

1. <http://www.uml.org/>
2. <http://www.netobjectives.com/resources/books/design-patterns-explained>



## SE201 - UVOD U SOFTVERSKO INŽENJERSTVO

Projektovanje softvera i šabloni  
projektovanja kreiranja objekata

Lekcija 08

PRIRUČNIK ZA STUDENTE

# SE201 - UVOD U SOFTVERSKO INŽENJERSTVO

## Lekcija 08

### *PROJEKTOVANJE SOFTVERA I ŠABLONI PROJEKTOVANJA KREIRANJA OBJEKATA*

- ✓ Projektovanje softvera i šabloni projektovanja kreiranja objekata
- ✓ Poglavlje 1: Proces projektovanja objektno-orientisanog softvera primenom UML
- ✓ Poglavlje 2: Utvrđivanje svojstava klasa
- ✓ Poglavlje 3: Specifikacija interfejsa
- ✓ Poglavlje 4: Šabloni projektovanja i njihove vrste
- ✓ Poglavlje 5: Šablon Singleton
- ✓ Poglavlje 6: Šablon Builder
- ✓ Poglavlje 7: Šablon Abstract Factory (Apstraktna fabrika)
- ✓ Poglavlje 8: Pokazna vežba
- ✓ Poglavlje 9: Individualna vežba
- ✓ Zaključak

Copyright © 2017 – UNIVERZITET METROPOLITAN, Beograd. Sva prava zadržana. Bez prethodne pismene dozvole od strane Univerziteta METROPOLITAN zabranjena je reprodukcija, transfer, distribucija ili memorisanje nekog dela ili čitavih sadržaja ovog dokumenta., kopiranjem, snimanjem, elektronskim putem, skeniranjem ili na bilo koji drugi način.

Copyright © 2017 BELGRADE METROPOLITAN UNIVERSITY. All rights reserved. No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise, without the prior written permission of Belgrade Metropolitan University.

## ✓ Uvod

# UVOD

### *Cilj lekcije*

Cilj ove lekcije da vas upozna sa osnovama projektovanja objektno-orientisanog softvera primenom UML i da ukaže na važne aspekte implementacije dobijenog projektnog rešenja . Ova lekcija vam omogućava da:

- razumete najvažnije aktivnosti nekog uopštenog objektno-orientisanog procesa projektovanja;
- razumete različite modele koji se koriste za dokumentovanje nekog objektno-orientisanog projektnog rešenja;
- znate ideju projektnih šablona i kako se one koriste za višestruku upotrebu znanja i iskustva u projektovanju;

Da bi projektovanje softvera bilo brže, uspešnije i kvalitetno, ono se najčešće oslanja na prethodna iskustva. Ta iskustva se stalno prikupljaju i predstavljaju u vidu tzv. šablonu za projektovanje (engl. design patterns). U ovoj lekciji biće izloženi sledeći šabloni kreirani:

- Singleton,
- Builder i
- Abstract Factory (Apstraktna fabrika),

Kako svaki šablon ima naziv koji je originalno dat na engleskom jeziku, i kao takav opšte poznat u svetu, to se ovde ne daje prevod ovih originalnih naziva, te se u tekstu koriste originalni nazivi na engleskom. To isto važi i za sve klase i interfejse koji se pominju pri objašnjavanju šablonu za projektovanje

NAPOMENA:

Zbog obimnosti s jedne strane, i zbog održavanja jedinstvenosti nastavne jedinice, s druge strane, ova vrlo obimna lekcija se predaje tri nedelje, te na tradicionalnoj nastavi ima 3 x (3+1+3) časova, tj. ukupno 9 časova predavanja, 3 časa pokaznih vežbi i 9 časova individualnih vežbi.. **Ova predavanja i vežbe se izvode 7., 8. i 9. nedelji u semestru.**

## UVODNI VIDEO

*Trajanje video snimka: 3min 3sek*

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

## ▼ Poglavlje 1

# Proces projektovanja objektno-orientisanog softvera primenom UML

## PROCES PROJEKTOVANJA SOFTVERA

*Definisati kontekst i spoljne interakcije sa sistemom, projektovati arhitekturu softvera, utvrditi glavne objekte sistema, razviti projektne modele sistema...*

Pri projektovanju softverskog sistema, koje se kreće od projektovanja koncepta, pa do detaljnog projektovanja, potrebno je uraditi sledeće:

1. Razumeti i definisati kontekst (problem i okruženje) i spoljne interakcije sa sistemom.
2. Projektovati arhitekturu softvera.
3. Utvrditi glavne objekte sistema.
4. Razviti projektne modele sistema.
5. Specificirati sve interfejse objekata.

Iako su ovi uobičajeni zadaci projektanta softvera, oni ne moraju da predstavljaju i aktivnosti jednog sekvensijalnog procesa razvoja softvera. Proces razvoja softvera je obično složeniji, jer sadrži i povratne sprege, tj. putanje kretanja posla, a i odvijanje i paralelnih, a ne samo sekvensijalnih (rednih) aktivnosti. Na primer, kada počinjete projektovanje nekog sistema, vi počinjete od nekih ideja, razvijate predloge rešenja koje detaljnije razrađujete kada vam neke informacije postaju dostupne. Kasnije u toku procesa projekta, vi vidite da morate da se vratite nekoliko koraka nazad i da nešto promenite, i onda da ponovite te korake.

Ako bi taj proces rada opisali nekim dijagramom sa svim aktivnostima procesa, on sigurno ne bi izgledao tako jednostavno, kao pet redno povezanih aktivnosti. Pored toga što svaka glavna aktivnost se može raščlaniti na više drugih, dijagram bi pokazao i više ovih povratnih puteva rada, a i rad paralelnih ili čak, međusobno spregnutih aktivnosti (paralelne aktivnosti treba koristiti svuda gde je moguće, jer se tim skraćuje vreme projektovanja i razvoja sistema).

U prethodnim lekcijama, objasnili smo šta je potrebno raditi u prve dve navedene aktivnosti procesa projektovaja (1 i 2). U četvrtoj lekciji (**Inženjerstvo zahteva**) objašnjen je značaj razumevanje zahteva korisnika i sistema, kao i interakcije sa okruženjem. U petoj lekciji (**arhitektura sistema**) je izložena aktivnost projektovanja arhitekture sistema, te se to neće ponavljati u ovoj lekciji. U trećoj lekciji (Modelovanje sistema) dat je postupak utvrđivanja glavnih, apstraktnih objekata podeljenih u tri osnove kategorije Boundary, Control i Entity, što je osnova za dalje projektovanje klasa sistema.

U toku procesa projektovanja se ovi objekti dalje razrađuju, dele ili dodaju. Međutim, najvažniji doprinos aktivnosti projektovanja je što omogućava definisanje svojstava svih klasa objekata, tj. određivanje njihovih atributa i operacija, kao i potrebnih interfejsa objekata. To će i biti fokus ove lekcije, kao i primena šablona projektovanja.

## ODNOS SOFTVERA I OKRUŽENJA - MODEL KONTEKSTA

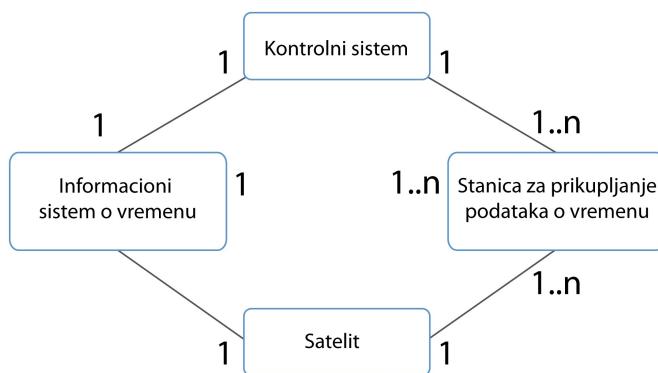
*Model konteksta sistema sadrži asocijacije između sistema, koji se projektuje, i okruženja, tj. drugih sistema sa kojima sistem treba da bude u nekoj vezi.*

Prva stvar koja se treba uraditi prilikom projektovanja softverskog sistema je da se razume odnos između softvera koji se projektuje i spoljnog okruženja. Ovo je bitno za donošenje odluke o tome kako da se obezbedi zahtevana funkcionalnost sistema i kako da se napravi struktura sistema za njegovu komunikaciju sa okruženjem. Razumevanje konteksta u kome radi sistema omogućava i određivanje granica sistema (tj. šta on „pokriva“ a šta ne). Na osnovu uspostavljenih granica sistema, odlučujete koja svojstva sistem treba da ima, kao i svojstva drugih, povezanih sistema.

Analiziraćemo **primer sistema za praćenje stanja vremenskih prilika** koji koristi više distribuiranih stanica za merenje vremenskih parametara (temperatura vazduha, pritisak, vlažnost, brzina vetra i dr...). Pri projektovanju ovog sistema treba da odlučite i kako da raspodelite (distribuirate) funkcionalnost sistema između kontrolnog sistema i stanica za merenje vremenskih parametara, kao i unutar ugrađenog računarskog sistema u svakoj stanici. Veze između okruženja i sistema opisuju sa primenom dva modela:

1. **Model konteksta sistema** je strukturni model koji pokazuje druge sisteme u okruženju sistema koji se razvija.
2. **Model interakcija** je dinamički model koji pokazuje kako sistem komunicira sa okruženjem pri svom radu.

**Model konteksta sistema** sadrži asocijacije između sistema koji se projektuje, i okruženja, tj. drugih sistema sa kojima sistem treba da bude u nekoj vezi. Asocijacije pokazuju da ima neke veze između entiteta koje su u asocijaciji. Na primer, na slici 1 su date asocijacije (veze) između informacionog sistema za praćenje vremenskih prilika, satelitskog sistem i kontrolnog sistema. To su sistemi u vezi koji predstavljaju okruženje svake stanice za praćenje vremenskih prilika. Vidi se da prikazano okruženje čine jedan informacioni sistem, jedan satelitski sistem, jedan kontrolni sistem, a više stanica za praćenje vremenskih prilika. Znači, model konteksta pokazuje samo strukturu okruženja sistema koga projektujemo i razvijamo.



Slika 1.1 Kontekst sistema za stanice za praćenje vremenskih prilika [1.2]

## MODEL INTERAKCIJE

*Model interakcija daje dodatne informacije, jer ukazuje kako sistem koji projektujemo komunicira sa okruženjem, tj. sa drugim sistemima u svom okruženju*

**Model interakcija** daje dodatne informacije, jer ukazuje kako sistem koji projektujemo komunicira sa okruženjem, tj. sa drugim sistemima u svom okruženju (a na koje ukazuje model konteksta). Jedan od UML model interakcija je UML dijagrama korišćenja (engl. **use case**), koji opisuju po jednu interakciju sa sistemom koji projektujemo. Simbol aktera označava ili neki drugi sistem ili čoveka koji koristi naš sistem. Na slici 3 je prikazan dijagram korišćenja koji pokazuju slučajevе korišćenja stanica za praćenje vremenskih prilika, tj. sistema koji se projektuje. Svaka elipsa simbolično označava po jedan slučaj korišćenja, koji se bliže opisuje tekstualnom informacijom u vidu scenarija, kao na slici 2.

Sistem	Stanica za praćenje vremenskih prilika
Slučaj korišćenja	Izveštaj o vremenu
Akteri	Informacioni sistem o vremenu, Stanica za praćenje vremenskih prilika
Opis	Stanica za praćenje vremenskih prilika šalje sumarni izveštaj sa vremenskim podacima koji su prikupljeni od instrumenata u toku određenog vremenskog intervala koje definiše informacioni sistem. Ti podaci predstavljaju maksimalne, minimalne i srednje vrednosti temperature vazduha, vazdušnog pritiska, brzine vetra, ukupan nivo padavina i pravac vetra u intervalima od pet minuta.
Stimulans	Informacioni sistem o vremenu sadrži link za komunikaciju sa satelitom koji je u vezi sa stanicama za praćenje vremenskih prilika zahteva prikupljene podatke.
Odgovor	Sumarni prikaz podataka poslatih informacionom sistemu o vremenu.
Komentari	Od stanica za praćenje vremenskih prilika obično treba da na svaki sat pošalju podatke o vremenu, ali ta frekvencija slanja može da bude različita po stanicama, a može i da se menja u budućnosti.

Slika 1.2 Opis slučaja korišćenja na primeru stanice za praćenje vremenskih prilika [1.2]

Slika 1.3 Slučajevi korišćenja stanica za praćenje vremenskih prilika [1.2]

## UVOD U PROJEKTOVANJE SOFTVERA (VIDEO)

*Georgia Tech predavanje - Uvod u projektovanje softvera*

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

## DESIGN - GEORGIA TECH - SOFTWARE DEVELOPMENT PROCESS (VIDEO)

*Trajanje: 1,19 minuta*

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

## VIDEO PREDAVANJE ZA OBJEKAT "PROCES PROJEKTOVANJA OBJEKTNO-ORIJENTISANOG SOFTVERA PRIMENOM UML"

*Trajanje video snimka: 26min 18sek*

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

## ▼ Poglavlje 2

# Utvrđivanje svojstava klasa

## MODELI SISTEMA

*Modeli projektovanja, ili modeli sistema, prikazuju objekte ili klase objekata sistema. Oni prikazuju i asocijacije (veze) između ovih entiteta (objekata ili klasa)*

**Modeli projektovanja**, ili modeli sistema, prikazuju objekte ili klase objekata sistema. Oni prikazuju i asocijacije (veze) između ovih entiteta (objekata ili klasa). Ovi modeli su svojevrsni „most“ između aktivnosti utvrđivanja zahteva i aktivnosti implementacije sistema. Oni bi trebalo da budu dovoljno apstraktni kako bi sakrili nepotrebne detalje, ali bi trebalo da uključe dovoljno detalja potrebnih programerima za njihov rad.

Da bi se ovi oprečni zahtevi uspešno zadovoljili, najčešće se koriste više modela sistema sa različitim nivoom apstrakcije (uopštavanja). Zato, kritična odluka u projektovanju sistema je odluka o nivou detalja koje model sistema, ili dela sistema treba da sadrži.

Modeli zavise i od vrste sistema koji se projektuje. Na primer, ako je u pitanju sistem za sekvencijalnu (rednu) obradu podataka, model se razlikuje od modela za slučaj nekog ugrađenog računarskog sistema. UML nudi 13 različitih tipova modela, ali u praksi se najčešće koristi samo nekoliko. Korišćenje manjih broja modela smanjuje troškove projektovanja, kao i potrebno vreme.

Pri primeni UML u projektovanju sistema, obično se razvija dve vrste modela projektovanja (sistema):

**Strukturni modeli**, koji opisuju statičku strukturu sistema upotreboom klase objekata i njihove veze (relacije). Važne veze su i one koje se naseleđuju (od superklasa) ili koje se uopštavaju i prebacuju na superklase, kao i veze tipa „upotrebljava/upotrebljen od“ i kompozitne (složene) veze.

**Dinamički modeli**, koji opisuju dinamičku strukturu sistema i pokazuju interakcije između objekata sistema. Te interakcije najčešće obuhvataju: niz zahteva za servisima koje generišu objekti i promene stanja objekata (npr. vrednosti njihovih atributa) do kojih dolazi zbog dejstva interakcija, tj. poruka koje neki objekt dobija od drugog objekta.

Od ovih modela, najviše se koriste:

- **dijagram klasa** koji prikazuje klase nekog od podsistema softverskog sistema koji se projektuje
- **sekvencijalni model**, koji pokazuje redosled interakcija između objekata sistema

- **model stanja**, koji prikazuje stanja i njihovu promenu usled dejstva pojedinih događaja kod objekata

Prvi je strukturni model, a druga dva su dinamički modeli.

## DETALJNO PROJEKTOVANJE OBJEKATA SISTEMA

*Primena UML slučajeva korišćenja sistema pomaže da se utvrde ne samo objekti već i operacije u sistemu koje su neophodne za njegov rad*

Kada ste odredili glavne objekte koji čine strukturi softverskog sistema (tj. njegovu arhitekturu), treba da dalje detaljnije projektujete ove objekte. Primena UML slučajeva korišćenja sistema pomaže da se utvrde ne samo objekti već i operacije u sistemu koje su neophodne za njegov rad. Na primer, iz opisa slučaja upotrebe stanice za praćenje vremenskih prilika na nekoj lokaciji, jasno je koji su instrumenti potrebni toj stanici da bi mogla da obezbedi tražene podatke o vremenu.

Na osnovu utvrđenih neophodnih glavnih objekata koje sistem treba da ima, sada je mogućno početi sa utvrđivanjem klasa objekata koje softverski sistem treba da obezbedi. Za to utvrđivanje klase, mogu se koristiti nekoliko tehnika:

- *Upotreba analiza gramatike prirodnog jezika* u kome je dat tekstualni opis sistema. Objekti i njihovi atributi su imenice, a operacije ili servisi su glagoli u tom tekstu.
- *Upotreba opštijih pojmove za pojedine reči* u tekstualnom opisu sistema:
- opipljivi entiteti (stvari) u aplikacionom domenu, kao što je avion
- uloge (engl. **roles**), kao što su menadžer ili doktor,
- događaji (engl. **events**), kao što su zahtevi,
- instrukcije, kao što su sastanci
- lokacije, kao što su kancelarije
- organizacione jedinice, kao što su kompanije, itd.

*Upotreba analize scenarija* datih u okviru slučajeva upotrebe. Svaki scenario se analizira da bi se utvrdili zahtevani objekti, atributi, i operacije.

Upotrebom ovih tehnika, kao i izvora kao što su dokumenti koji opisuju zahteve za sistem, informacije prikupljene diskusijama sa korisnicima, i analizom postojećih sistema dobijaju se potrebne informacije.

U lekciji br. 6 (Modelovanje i analiza sistema) dali smo način kako identifikovati apstraktne objekte i kako ih razvrstati po kategorijama: Boundary, Control i Entity. Tada smo napomenuli da apstraktne klase nemaju ni atributte ni operacije, u fazi analize. Međutim, u fazi projektovanja sistema, moraju se naći veze (asocijacije među klasama, moraju se odrediti atributi i operacije klase). U ovom poglavlju ćemo videti kako se to radi.

## UTVRĐIVANJE ASOCIJACIJA IZMEĐU OBJEKATA

*Utvrđuju se uglavno analizom glagola u tekstu scenarija*

**Asocijacije** između objekata se mogu utvrditi analizom tekstova scenarija datih u okviru slučajeva korišćenja. Od posebnog značaja su fraze kao što su: "ima", "deo je", "upravlja", izveštava ", podstaknute sa..." "je zadržan u ..." "govori nekome", uključuje")

Svaka asocijacija treba da ima svoj naziv, a na svakom kraju bi trebalo da budu upisane uloge.

Ovo su preporuke za utvrđivanja asocijacija objekata (klasa):

- analiziraj rečenice sa glagolima.
- Precizno imenuj asocijaciju i uloge na krajevima
- Upotrebi kvalifikatore što češće, da bi identifikovao prostor imena i ključne atribute.
- Ukloni bilo koju asocijaciju koja je dobijena iz druge asocijacije.
- ne obračajte pažnu na kardinalnost sve dok se ne stabilišu asocijacije
- Isuviše mnogo asocijacija čini model nečitkovim.

**Asocijacije sa agregacijom** označavaju pripadnost objekta nekom drugom objektu. One se lako nalaze u tekstu koji opisuje neku strukturu i šta pripada toj strukturi. Ako niste sigurni u agregaciju, možete, bar na početku, koristiti običnu asocijaciju tipa "one to many".

## UTVRĐIVANJE ATRIBUTA

*Atributi objekta se utvrđuju analizom teksta scenarija, podvlačenjem imenica.*

**Atributi** su svojsva objekata (slika 1) i treba ih specificirati samo ako su od značaja za sistem. Ostale atribute možete ignorisati. Na primer, informacioni sistem univerziteta ne koristi informaciju o broju pasoša studenta, te taj podatak ne bi trebalo da nude autribut objekta Student. Međutim, za neki drugi sistem, na primer, informacioni sistem turističke agencije, taj podatak može biti potreban, te ga tada i treba koristiti kao atribut.

EmergencyReport
emergencyType:{fire,traffic,other}
location:String
description:String

Slika 2.1 Atributi obajekta[1.2]

Ne smatraju se atributima koji za vrednost imaju neke druge objekte, Zato se prvo definišu asocijacije, pa tek onda atributi.

Atributi objekta se utvrđuju analizom teksta scenarija, podvlačenjem **imenica**. U slučaju Entity objekata, svaka informacija ili podatak koja mora da se trajno čuva, tj. stavi u trajnu memoriju, treba da bude atribut Entity objekta.

Atributi su najnestabilniji deo objekta. Nije problem da neke uključimo i pri kraju porještovanja.

Preporuke za utvrđivanje atributa:

1. Ispitaj posesivne rečenice u scenarijima
2. Predtsavit memorisan stanje kao atribut Entity objekta
3. Opiši svaki atribut.
4. Ne predstavi objekat kao atribut. Koristi za to asocijaciju ka tom objektu.
5. Ne gubi vreme na detaljima sve dok struktura objekta ne postane stabilna.

**Entity objekti** najčešće imaju karakteristike koje ih identificuju od strane aktera sa kojima komuniciraju. Jedan od atributa objekta bi trebalo da bude identifikator po kome mu se akteri obraćaju. Na primer, za studenta, to je broj indeksa.

## UTVRĐIVANJE OPERACIJA PRIMENOM SEKVENCIJALNIH DIJAGRAMA

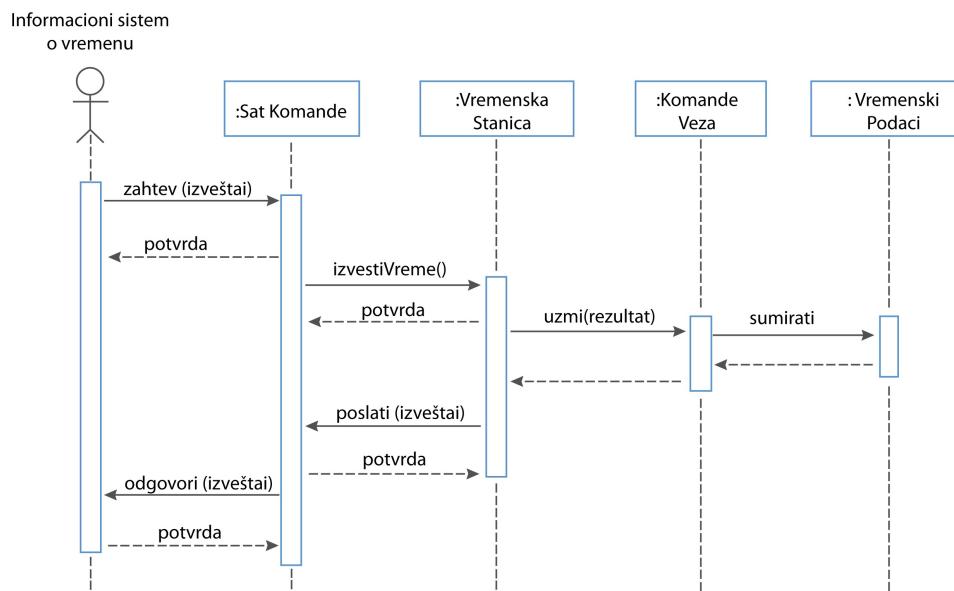
*Sekvencijalni dijagrami se koriste za modelovanje kombinovanog ponašanja grupe objekata.*

U fazi projektovanja najčešće se definišu **glavne klase podsistema**, te njihovi dijagrami klasa ne sadrže sve klase koje će sistem imati na kraju razvoja. Drugi deo klasa se definišu u fazi implementacije, tj. kada programeri pišu kod i određuju vrlo detaljno projektno rešenje svakog pod sistema.

**Sekvencijalni dijagrami** se pravi za svaku značajniju interakciju. Kada napravite modul slučaja korišćenja, trebalo bi da napravite za njega i sekvencijalni dijagram (model). Na slici 2 je prikazan sekvencijalni dijagram koji pokazuje interakcije između objekata stanice za prikupljanje podataka o vremenskim prilikama kada stanica primi zahtev za slanje prikupljenih podataka. Sekvencijalni dijagram se čita odozdo ka dole (duž vremenske ose). Sekvencijalni dijagrami se koriste za modelovanje kombinovanog ponašanja grupe objekata.

**Svaka poruka određuje operaciju primajućeg objekta, a parametri poruke, mogu biti atributi primajućeg objekta.**

Objekat *SatKomunikacija* osluškuje i prima poruke spolnjih sistema, dekodira te poruke i pokreće rad stanice, koju predstavlja objekat *VremenskiPodaci*. Ova dva objekta mogu da rade paralelno (istovremeno).



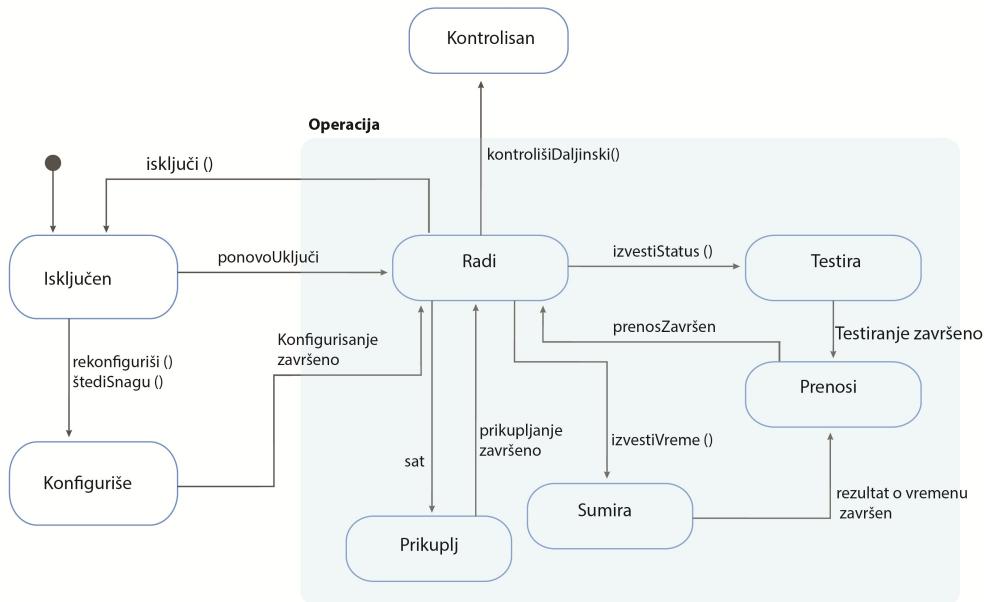
Slika 2.2 Sekvencijalni dijagram koji opisuje interakcije neophodne za prikupljanje podataka[1.2]

## DIJAGRAM STANJA

*Dijagram stanja ne pokazuje, kao sekvencijalni dijagram, ko šalje poruke, već samo poruke koje objekt prima, i stanja u koja ulazi kao njegovu reakciju*

Međutim, potrebno je i videti kako koji objekt menja stanje (vrednost svojih atributa) kada dobije neku poruku, tj. kada se javi neki događaj koji pobuđuje neku njegovu operaciju. Za tu svrhu se koristi **dijagram stanja objekta**. Na slici 3 prikazan je primer dijagrama stanja za slučaj stanice za prikupljanje podataka o vremenskim prilikama (ili kraće, stanica) koji pokazuje kako stanica, kao objekt sistema, reaguje na različite servise, tj. događaje (poruke) u svom okruženju. Na dijagramu se vide stanja u koji dolazi objekt koji predstavlja stanicu, kada dobije navedene poruke. Dijagram stanja ne pokazuje, kao sekvencijalni dijagram, ko šalje poruke, već samo poruke koje objekt prima, i stanja u koja ulazi kao njegovu reakciju na primljene poruke. Do promene stanja dolazi zato što ove poruke aktiviraju bar jednu operaciju (metod) objekta koji vrši promenu vrednosti bar jednog atributa objekta.

Dijagrami stanja su korisni za uopštenje (apstraktne) modele sistema ili za opisivanje rada nekog objekta. Obično se ne koriste za sve objekte sistema. Objekti koji su jednostavni, ne moraju se opisivati dijagramima stanja.



Slika 2.3 Dijagram stanja objekata koji predstavlja stancu za prikupljanje podataka o vremenskim prilikama[1.2]

## UTVRĐIVANJE PONAŠANJE OBEKATA USLOVLJENO NJIHOVIM STANJEM

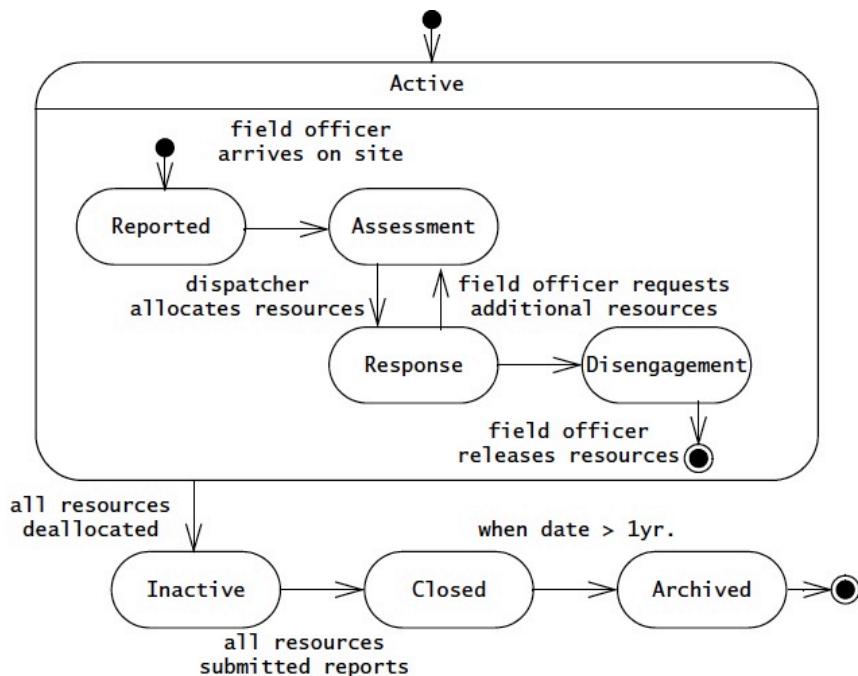
*Dijagram stanja pomaže projektanu da otkrije nove slučajevе korišćenja i da doradi postojeće*

**Sekvencijalni dijagrami** distribuiraju ponašanje sistema po objektima i utvrđuju operacije koje objekti treba da realizuju. Sekvencijalni dijagrami predstavljaju ponašanje sistema iz perspektive jednog slučaja korišćenja.

**Dijagram stanja** jednog objekta predstavlja ponašanje iz perspektive samo jednog objekta. Ovo omogućava da projektant sistema pripremi više formalnog opisa ponašanja objekta, i da utvrdi i nedostatak nekih slučajeva korišćenja. Fokusiranjem na pojedinačna stanja objekta, projektanti mogu da identifikuju i novo ponašanje.

Nije potrebno razviti dijagram stanja za svaki objekat sistema. Samo objekti koji imaju duži životni ciklus i čije ponašanje se menja stanjem u kome se nalaze se uzimaju u obzir za razvoj dijagrama stanja. To su najčešće Control objekti, ređe Entity objekti, a skoro nikada Boundary objekti.

Na slici 4 prikazan je primer jednog dijagrama stanja. Njegovom analizom, projektant može da primeti da li ima definisane sve potrebne slučajevе korišćenja koji prate objekat u svim njegovim stanjima, od otvaranja, pa do zatvaranja. Projektant može dodati neke detalje na neke akcije korisnika, koje menjaju stanje objekta.



Slika 2.4 Dijagram stanja objekta Incident.[1.2]

## PRIMER: KLASE STANICA ZA PRAĆENJE VREMENSKIH PRILIKA

*Kada se utvrde osnovni objekti i njihova glavna svojstva (atributi i operacije), vrši se dalje, detaljnije projektovanje ovih objekata*

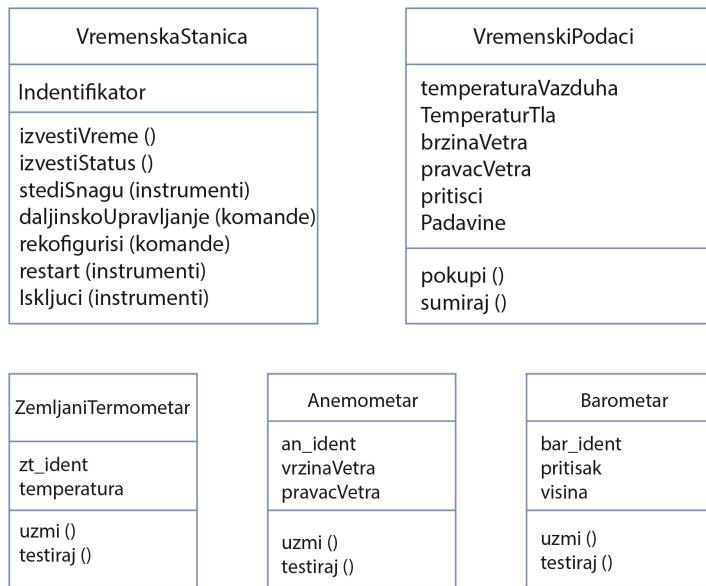
Na slici 1 prikazano je pet klasa utvrđenih za slučaj stanice za praćenje vremenskih prilika, a koje su utvrđene primenom navedenih tehnika. To su Termometar na zemlji, Anemometar (za merenje brzine vetra), Barometar (za merenje pritiska), VremenskaStanica i VremenskiPodaci.

Objekat VremenskaStanica obezbeđuje interfejs stanice sa okruženjem. Ovaj objekt sadrži sve operacije koje pominje scenario koji opisuje njegov rad.

Objekat VremenskiPodaci je odgovoran za obradu vremenskog izveštaja. On šalje informacionom sistemu sumirane podatke dobijene od instrumenata koje sadrži stanica.

Objekti Termometar, Anemometar i Barometar predstavljaju klase koje su direktno povezane sa instrumentima koje predstavljaju. Sadrže neophodne operacije za kontrolu ovih instrumenata. Ovi objekti autonomno prikupljaju u određenoj frekvenciji i lokalno ih smeštaju u memoriju objekata. Na poseban zahtev, ovi podaci se šalju vremenskoj stanici.

Kada se utvrde osnovni objekti i njihova glavna svojstva (atributi i operacije), vrši se dalje, detaljnije projektovanje ovih objekata. Analiziraju se zajednička svojstva klasi, da bi se definisala eventualna hijerarhija klasi. Na primer, u slučaju stanice za



Slika 2.5 Klase objekata stanice za praćenje vremenskih prilika [1.2]

prikupljanje vremenskih prilika, može se definisati superklasa za sve instrumente, npr. nazvana **Instrument** koja sadrži zajednička svojstva, kao što su identifikator, i operacije za čitanje i testiranje podataka. Ovde se mogu dodati i novi atributi u super klasi (**Instrument**), kao što je frekvencija prikupljanja podataka.

## DEFINISANJE KLASA I ATRIBUTA (VIDEO)

*Georgia Tech predavanje - Definisanje klasa i atributa*

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

## DODAVANJE ATRIBUTA (VIDEO)

*Georgia Tech predavanje - Dodavanje atributa*

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

## DODAVANJE OPERACIJA (VIDEO)

*Georgia Tech predavanje - Dodavanje operacija*

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

## DODAVANJE VEZA IZMEĐU KLASA (1) (VIDEO)

*Georgia Tech predavanje - Dodavanje veza između klasa (1)*

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

## DODAVANJE VEZA IZMEĐU KLASA (2) (VIDEO)

*Georgia Tech predavanje - Dodavanje veza između klasa (2)*

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

## DORADA DIJAGRAMA KLASA (VIDEO)

*Georgia Tech predavanje - Dorada dijagrama klasa*

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

## FINALIZACIJA DIJAGRAMA KLASA (VIDEO)

*Georgia Tech predavanje - Finalizacija dijagrama klasa*

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

## KRAJ PROJEKTOVANJA INFORMACIONOG SISTEMA BIBLIOTEKE (VIDEO)

*Georgia Tech predavanje - Kraj projektovanja informacionog sistema biblioteke*

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

## OO SYSTEMS ANALYSIS AND DESIGN - OO DESIGN (VIDEO)

*Trajanje: 45:27 minuta (opciono, ako student želida pogleda)*

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

## ▼ Poglavlje 3

# Specifikacija interfejsa

## ŠTA JE INTERFEJS?

*Interfejs nekog objekta definiše poruke na koje objekat reaguje. Svaka poruka se definiše tzv. potpisom tj. nazivom operacije koju objekat sadrži i njegove argumente (podatke).*

Interfejs nekog objekta (tj. klase objekata) definiše poruke na koje objekat reaguje. Svaka poruka se definiše tzv. *potpisom* (**signature**), tj. nazivom operacije (metoda) koju objekat sadrži i argumenata (podataka) koji su neophodni da bi ta operacija mogla uspešno da se izvrši.

Interfejs ne određuje kako će objekt, tj. klasa koja predstavlja taj objekt, primeniti i izvršiti tu operaciju. Zato se projektovanje interfejsa i projektovanje klase mogu raditi paralelno (istovremeno). Vi možete kasnije menjati način kako se operacija izvršava, a da pri tom ništa ne menjate u interfejsu (ako se signature poruka ne menjaju). To je jedan od prednosti objektno-orientisanih sistema, jer promene koje se rade unutar jedne klase, ne moraju da izazivaju nikakve promene u komunikaciji sa drugim klasama, ako se signature poruka koje razmenjuju (interakcije), ne menjaju.

Projektovanje interfejsa se svodi na definisanje signatura i semantike servisa koje obezbeđuje neki objekt ili grupa objekata (klasa). U UML-u, interfejs se predstavlja isto kao i klasa, samo što nema deo koji je namenjen atributima. Semantika interfejsa se može definisati primenom OCL jezika (**Object Constraint Language**).

Interfejs sadrži listu servisa, ili operacija, koje objekt (ili klasa) objekt koji je povezan sa interfejsom nudi spoljnjem okruženju. Te operacije, menjaju vrednosti podataka u objektu (set metodi) ili čitaju i prosleđuju njihove vrednosti (get metodi).

Jedan objekt može imati i više interfejsa. To se primenjuje najčešće kada se želi da pruže različiti pogledi na objekt, ili da se ograniči pristup određenoj grupi korisnika, samo određenom skupu servisa. U Javi se interfejs posebno definiše, tj. nezavisno od objekta koji realizuje operacije (metode) navedene u interfejsu.

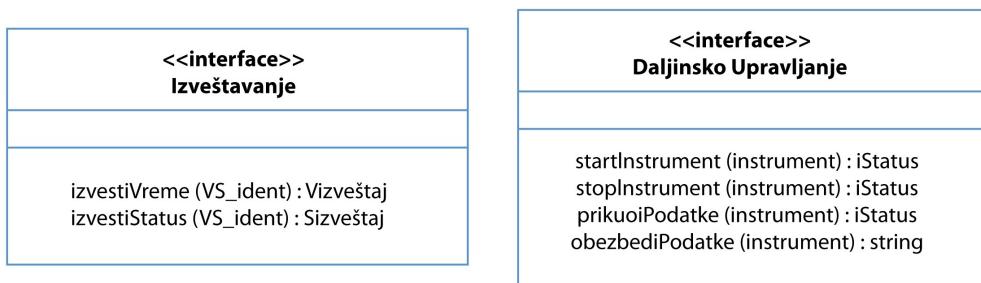
S druge strane, jedan interfejs može da obezbedi pristup i većem broju objekata.

## PRIMER: INTERFEJSI VREMENSKE STANICE

*Dva interfejsa koji su definisani za primer stanice za prikupljanje podataka o vremenskim prilikama. Oni definišu četiri operacije koje realizuje jedan meto*

Slika 1 prikazuje dva interfejsa koji su definisani za primer stanice za prikupljanje podataka o vremenskim prilikama. Na levoj strani je interfejs koji je namenjen izveštavanju, jer sadrži spisak operacija koje se nalaze u objektu

VremenskaStanica i koje pripremaju podatke za izveštaje koji se traže. Drugi interfejs, s desne strane, je namenjen za podršku upravljanja iz daljine, jer obezbeđuje četiri operacije upravljanja. Međutim, iako interfejs navodi četiri operacije, one se sve realizuju u objektu od strane samo jednog metoda objekta **VremenskaStanica**. Njen metod **daljinskaKontrola** izvršava sve četiri operacije



Slika 3.1 Interfejsi objekta VremenskaStanica [1.2]

## UML STRUCTURAL DIAGRAMS: COMPONENT DIAGRAM - GEORGIA TECH - SOFTWARE DEVELOPMENT PROCESS (VIDEO)

*Trajanje: 3,16 minuta*

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

## ▼ Poglavlje 4

# Šabloni projektovanja i njihove vrste

## ŠTA JE ŠABLON PROJEKTOVANJA?

*Šabloni i jezici za šablove su načini opisivanja najbolje prakse, dobrih projektnih rešenja, i prikupljanje iskustva na način koji omogućava drugima*

**Šablon** ([pattern](#)) je opis problema i suština njegovog rešenja koje se može više puta koristiti u različitim slučajevima. Šabloni su način da se ponovo upotrebi znanje i iskustvo drugih projektanata. One obezbeđuju ponovnu upotrebljivost projektnih rešenja koja su se pokazala dobrim u praksi.

Šablon ne daje detaljnu specifikaciju, već više opis akumuliranog iskustva i znanja, u primeni oprobanog rešenja nekog zajedničkog problema. Hillside Group daje sledeću definiciju: „*Šabloni i jezici za šablove su načini opisivanja najbolje prakse, dobrih projektnih rešenja, i prikupljanje iskustva na način koji omogućava drugima da ponovo upotrebe to iskustvo.*“

Obično se neko projektno rešenje softverskog sistema objašnjava navođenjem šabloni koja je korišćenja za njen razvoj. Šabloni za projektovanje se najčešće koriste pri projektovanju objektno-orientisanih sistema. Zato, mnogi šabloni primenjuju objektne karakteristike, kao što je nasleđivanje svojstava i poliformizam. Međutim praksa primenjivanja učaurivanja (engl. [encapsulation](#)) se može primeniti i kod drugih sistema (koji nisu objektno orientisani). Jedan šablon najčešće treba da sadrži bar sledeća četiri elementa:

1. **Naziv šabloni** koji odražava upotrebljivost Šabloni.
2. **Opis problemskog područja** u kome se šablon može primeniti.
3. **Opis rešenja** za delove sistema, njihovih veza i odgovornosti. To nije opis rešenja, već uzorak za njegovu izradu, koji se kreira u svakoj korektnoj situaciji. To je obično u formi dijagrama klase.
4. **Iskaz u posledicama**, tj. o rezultatima i učinjenim kompromisima pri primeni šablonia Na taj način korisnici šablonu (projektanti sistema) mogu da se lakše odluče da li da koriste ponuđenu mustru.

Korisno je istaći zašto je uzorak koristan i opisati situacije u kojima se šablon može uspešno primenjivati. Opis rešenja obuhvata strukturu šablonu, učesnike, kolaboracije i implementacijuš

# OSNOVNE VRSTE ŠABLONA PROJEKTOVANJA

*Šabloni se mogu klasifikovati na sledeće osnovne vrste: šabloni kreiranja, šabloni strukture, šabloni ponašanja i J2EE šabloni*

Pri projektovanju objektno-orientisanih softverskih sistema treba koristiti poznate (objavljene) šabloni za projektovanje softverskih sistema, jer oni su rezultat uspešne prakse u projektovanju ovakvih sistema. Pri tome se koriste dva principa u vidu preporuka:

1. Programirajte objekat (primerak klase) a ne implementaciju
2. Dajte prednost kompoziciji umesto nasleđivanju

Šabloni projektovanja su parcijalna rešenja opštih problema. Šablon projektovanja čini mali broj klasa koji pute delegiranja ili nasleđivanja, obezbeđuju robusno i prilagodljivo rešenje. Te klase se mogu prilagoditi i dopuniti pri projektovanju specifičnog sistema koji se razvija. Šabloni projektovanja daju standardnu terminologiju i specifični su za određeni scenario.

Na primer: Singleton šablon. On pojednostavljuje upotrebi jednog objekta. Svi inženjeri koji prave jedan objekat saopštavaju jedni drugima da koriste Singleton šablon

Šabloni projektovanja (engl. **design patterns**) obezbeđuju najbolja rešenja određenih problema iz prakse razvoja softvera. Njihovom primenom, neiskusni softver inženjer uči kako da projektuje softver na lak i brzi način.

Šabloni se mogu klasifikovati na sledeće osnovne vrste:

1. Šabloni kreiranja
2. Šabloni strukture
3. Šabloni ponašanja
4. J2EE šabloni

**Šabloni kreiranja** daju način kreiranja objekata sa skrivenom logikom kreiranja, tj. bez korišćenja operatora **new**. Na taj način program ima veću fleksibilnost u kreiranju objekata u određenom slučaju korišćenja.

**Šabloni strukture** se bavi kompozicijom klasa i objekata. Koncept nasleđivanja se koristi da bi se definisali interfejsi i da bi se definisao način postavljanja objekata radi dobijanja nove funkcionalnosti

**Šabloni ponašanja** se prvenstveno bave komunikacijom između objekata

**J2EEE šabloni** se prvenstveno bave prezentacionim slojem sistema (npr. GUI). Ovi šabloni su utvrđeni od strane Sun Java Center.

## PRIMENA ŠABLONA PROJEKTOVANJA

*Za uspešnu primeni projektnih šablon potrebno je odgovarajuće iskustvo u njihovom korišćenju. Potrebno je da prepozname situacije u kojima se može p*

Pri projektovanju softverskog sistema, uzmite uvek u obzir mogućnost korišćenja odgovarajućeg šablonu, jer to ubrzava projektovanje, a obezbeđuje i pouzdanost sistema, jer se koristi već provereno projektno rešenje. Postoje knjige sa prikazima različitih šablonu (mustri) za projektovanje softverskih sistema, i njih treba izabrati i kreativno primeniti.

Primenom šablonu vrši se ponovna upotreba već razrađenih konceptualnih rešenja. U slučaju ponovne upotrebe izvršnih komponenti, javlja se problem ograničenja koja se javljaju usled odluka vezanih za detaljno projektovanje, a koje su doneli implementatori (programeri) tih izvršnih komponenata. Ta ograničenja mogu biti vezana za izabrane algoritme u tim komponentama, ali i za tipove interfejsa komponenata. U slučaju kada su ova vrlo konkretna rešenja u sukobu sa zahtevima postavljenim pri projektovanju novog sistema, ponovna upotreba takvih komponenti postaje nemoguća ili unosi veliku neefikasnost rada sistema. Zbog toga, upotreba šablonu (mustri) ne znači upotrebu gotovih, izvršnih komponenata, već samo ponovnu upotrebu ideja, koncepata, iskustva rešavanja sličnog problema, što otvara mogućnost reorganizacije sistema tako da se može upotrebiti odgovarajuća šablon.

Ako se radi na tom, u opštem nivou, mogu se naći odgovarajuće šabloni u knjizi projektnih šablonu i uspešno primeniti. Međutim, ako je vaš problem vrlo specifičan, onda je verovatno da ne možete naći odgovarajuću šablon koju bi mogli da primenite u vašem projektovanju rešenja.

Za uspešnu primeni projektnih šablon potrebno je odgovarajuće iskustvo u njihovom korišćenju. Potrebno je da prepozname situacije u kojima se može primeniti neka šablon. Neiskusni projektanti imaće teškoća da izaberu odgovarajući šablonu iz knjige raspoloživih šablonu

## SOFTWARE DESIGN PATTERNS - OVERVIEW (VIDEO)

*Trajanje: 4,15 minuta*

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

## UVOD O ŠABLONE PROJEKTOVANJA (VIDEO)

*Georgia Tech predavanje - Uvod u šablove projektovanja*

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

## ISTORIJA ŠABLONA PROJEKTOVANJA (VIDEO)

*Georgia Tech predavanje - Istorija šablona projektovanja*

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

## KATALOG ŠABLONA (VIDEO)

*Georgia Tech predavanje - Katalog šablona*

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

## VIDEO PREDAVANJE ZA OBJEKAT "ŠABLONI PROJEKTOVANJA I NJIHOVE VRSTE"

*Trajanje video snimka: 21min 19sek*

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

## ✓ Poglavlje 5

# Šablon Singleton

## KLASA SINGLETON

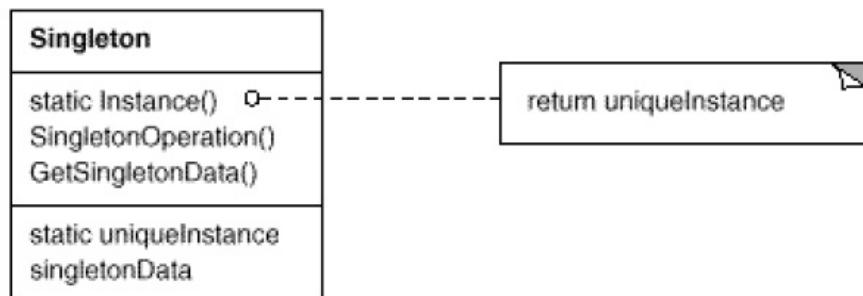
*Šablon Singleton se primenjuje u slučajevima klasa koje imaju samo jedan primerak (instancu), tj. objekat.*

### Opis problema:

- Klasa treba da ima samo jednu instancu, objekat.

### Rešenje:

- klasa vodi računa o svoj jedinoj instanci.
- Obezbeđuje pristup objektu sa neke poznate tačke pristupa.
- Jedina instanca se proširuje podklasama, a klijenti mogu da koriste proširenu instancu bez promene svog koda.



Slika 5.1 S truktura šablona Simgletone [2.4]

### Učesnici:

**Singleton:** Definiše statički metod objekta koji omogućuje klijentima da pristupe svom jedinstvenom objektu (instanci). Klasa **Singleton** je odgovorna za kreiranje svoje jedine instance, tj. objekta .

### Kolaboracije:

- Klijenti pristupaju **Singleton** objektu **isključivo korišćenjem statičkog metoda koji obezbeđuje klasa Singleton**

# ŠABLON SINGLETON: POSLEDICE I IMPLEMENTACIJA

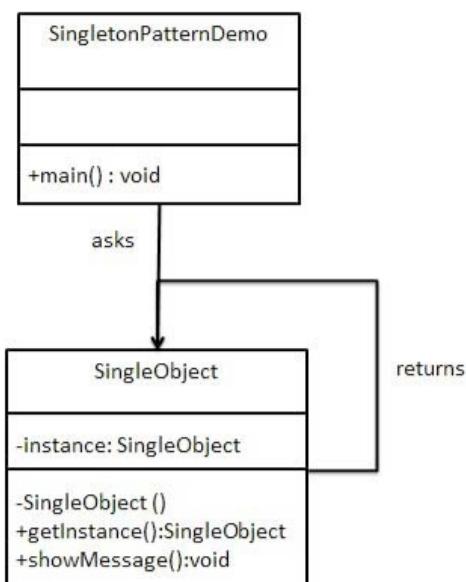
*Klasa Singleton kontroliše kako i kada klijenti pristupaju njenom objektu.*

## Posledice:

1. *Kontrolisan pristup jedinom objektu.* Klasa Singleton kontroliše kako i kada klijenti pristupaju njenom objektu.
2. *Smanjen prostor imena.* Šablon Singleton predstavlja poboljšanje globalnih promenljivih. Onemogućava zagađenje prostora imena sa globalnim promenljivama koje memorišu svoje instance, tj. objekte.
3. *Dozvoljava dopunjavanje metoda.* Klasa Singleton može da ima podklasu te je jednostavno da se konfiguriše aplikacija sa jednim objektom te podklase i to u vreme izvršenja.
4. *Dozvoljava promenljiv broj objekata.* Šablon vam dozvoljava da kreirate i više od jednog objekta klase Singlton. Treba samo promeniti metod koji omogućava pristup Singleton objektu
5. *Veća fleksibilnost u odnosu na metod klase.* Ista se funkcionalnost može da postigne primenom statičkog metoda klase Singleton. Međutim, tada je otežano korišćenje više od jednog objekta klase.

## Implementacija:

1. Kreira se klasa **SingleObject** sa svojim privatnim konstruktorom i koja ima statičku sopstvenu instancu (objekat)
2. Klasa **SingleObject** obezbeđuje statički metod za dobijanje statičke instance (objekta) za vezu sa spoljnjim svetom (klijentima).
3. Naš primer: **SingletonPatternDemo** koristi klasu SingletonObject za dobijanje objekta SingleObject.



Slika 5.2 Implementacija šablona Singletone [2.4]

## ŠABLON SINGLETON: POSTUPAK PRIMENE

*Primena šablona Singletone obuhvata tri koraka: kreiranje klase Singletone, kreiranje objekta klase Singletone i proveru dobijenih rezultata.*

### Korak 1: Kreiranje klase Singleton

```
public class SingleObject {  
  
    //create an object of SingleObject  
    private static SingleObject instance = new SingleObject();  
  
    //make the constructor private so that this class cannot be  
    //instantiated  
    private SingleObject(){}
  
  
    //Get the only object available  
    public static SingleObject getInstance(){  
        return instance;  
    }
  
    public void showMessage(){  
        System.out.println("Hello World!");  
    }
}
```

### Korak 2: Dobijanje objekta klase Singleton

```
public class SingleObject {  
  
    //create an object of SingleObject  
    private static SingleObject instance = new SingleObject();  
  
    //make the constructor private so that this class cannot be  
    //instantiated  
    private SingleObject(){}
  
  
    //Get the only object available  
    public static SingleObject getInstance(){  
        return instance;  
    }
  
    public void showMessage(){  
        System.out.println("Hello World!");  
    }
}
```

}

### Korak 3: Provera rezultata

Hello World!

## SINGLETON DESIGN PATTERN TUTORIAL (VIDEO)

*Trajanje: 18,39 minuta*

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

## ✓ Poglavlje 6

# Šablon Builder

## ŠABLON KREIRANJA BUILDER

*Odvojiti izradu složenih objekata od njihovog predstavljanja, tako da isti postupak izrade može da se primeni za kreiranje različitih predstavljanja objekata*

### Svrha

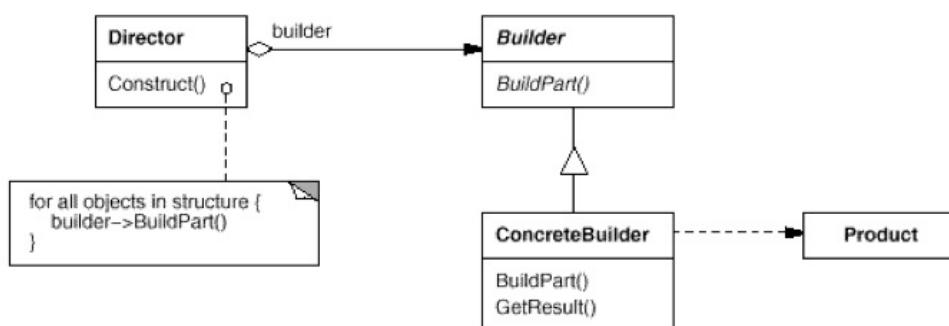
- Odvojiti izradu složenih objekata od njihovog predstavljanja, tako da isti postupak izrade može da se primeni za kreiranje različitih predstavljanja objekata.

### Primena:

Šablon projektovanja Builder se koristi kada:

- je algoritam kreiranja nekog složenog objekta treba da bude nezavisan od delova koji čine objekat i od načina njegovog sastavljanja;
- proces izrade mora da dozvoli različita predstavljanja objekta koji je u izradi.

### Struktura šablona:



Slika 6.1 Struktura šablona kretanja Builder [2.4]

### Učesnici:

1. **Builder:** Specificira apstraktni interfejs za kreiranje delova objekta Product
2. **ConcreteBuilder:** Izrađuje i sastavlja delove proizvoda primenom Builder interfejsa. Definiše i pamti predstavljanja koje je kreirao. Obezbeđuje interfejs za nalaženje proizvoda
3. **Director:** zrađuje neki objekat upotrebom Builder interfejsa

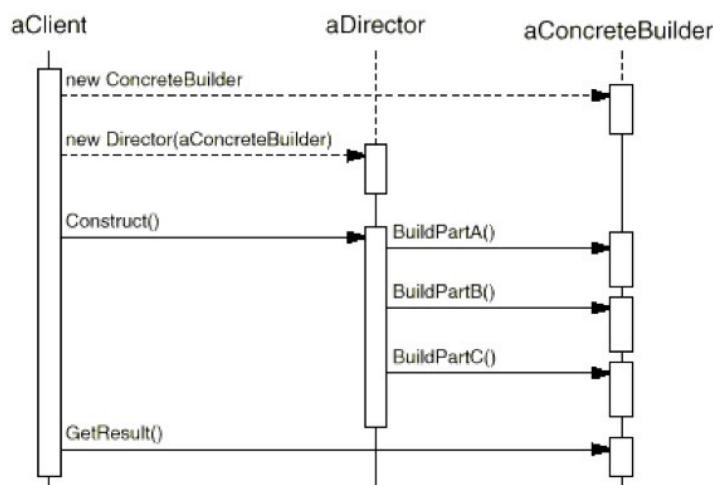
**4. Product:** Predstavlja složeni objekat u izradi. ConcreteBuilder izrađuje unutrašnje predstavljanje proizvoda i definiše proces svog sastavljanja. Uključuje klase koje definišu sastavne delove, uključujući i interfejse za sastavljanje delova u konačan rezultat.

## ŠABLON BUILDER: KOLABORACIJA I POSLEDICE

*Šablon Builder omogućava menjanje unutrašnje predstave objekta, izolovanje izrade i predstavljanja objekata i bolju kontrolu procesa njihove izrade.*

### Kolaboracije:

1. **Client** kreira objekat **Direktor** i konfiguriše ga sa poželjnim **Builder** objektom.
2. **Direktor** obaveštava interfejs **Builder** uvek kada treba da se izradi deo proizvoda.
3. **Builder** obrađuje zahteven od objekta **Director** i dodaje delove proizvodu.
4. **Client** vrši pregled proizvoda preko **Builder-a**



Slika 6.2 Kolaboracija po šablonu Builder [2.4]

### Posledice:

1. Dozvoljava vam da menjate unutrašnje predstavljanje objekta. **Builder** obezbeđuje direktoru apstraktan interfejs za izradu prozvoda. Skriva način sastavljanja proizvoda. Kako je proizvod izrađen preko apstraktnog interfejsa, vi treba samo da promenite unutrašnje predstavljanje objekta da bi definisali novu vrstu graditelja, tj. **Builder**
2. Izoluje program namenjen izradi i predstavljanju. Poboljšana je modularizacija učaurenjem načina izrade i predstavljanja složenih objekata. Klijenti ne moraju da znaju klasu koja definiše unutrašnju strukturu proizvoda
3. Omogućava vam bolju kontrolu procesa izrade. **Builder** izrađuje proizvod korak po korak, pod kontrolom direktora. Kada je proizvod gotov, direktor vrši pregled proizvoda dobijen od graditelja. **Builder** interfejs odražava proces izrade proizvoda

vše nego drugi šabloni kreiranja. To vam daje bolju kontrolu procesa izrade, te i unutrašnje struktura

## ŠABLON BUILDER: IMPLEMENTACIJA

*Implementacioni primer je restoran brze hrane u kome se nude dve vrste burgera i hladno piće.*

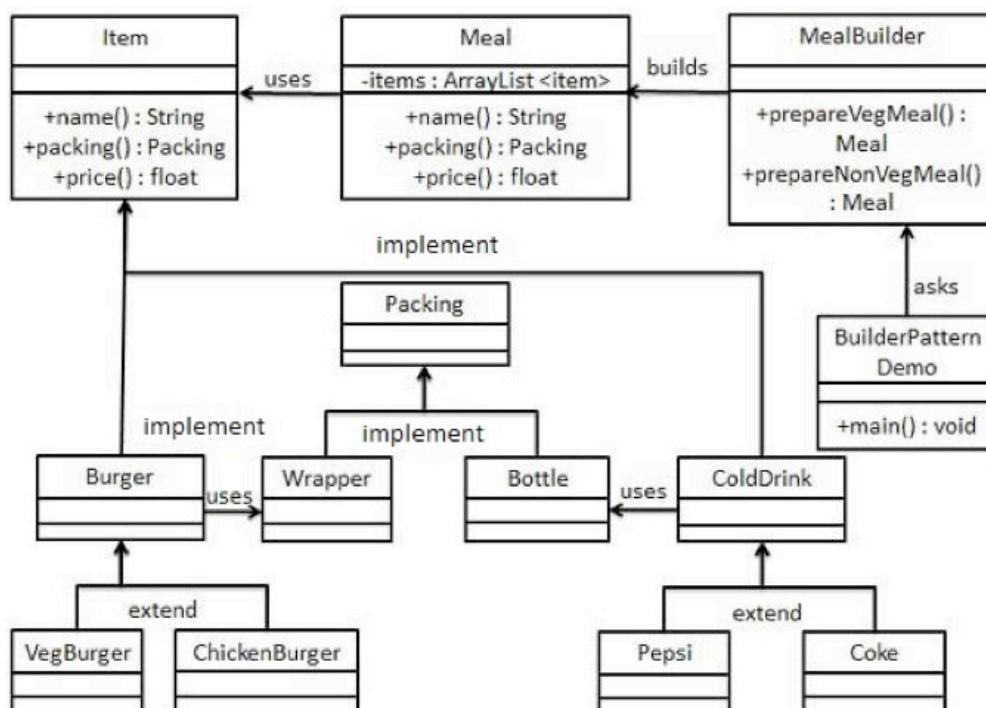
### Primer:

Restoran brze hrane u kome su tipični proizvodi burger i hladno piće. **Burger** može biti **VegBurger** ili **ChikenBurger**, a pakuje se sa kutijom. Hladno piće može biti koka kola ili pepsi kola i pakuje se sa flašom.

### Rešenje:

Kreira se

- **Item** interfejs koji predstavlja hrani i konkretne klase koje ga primenjuju
- **Packing** interfejs koji predstavlja pakovanje i klase koje ga primenjuju
- **Meal** klasa koja ima **ArrayList** listu sa **Item** i **MealBuilder** za izradu različitih tipova **Meal** objekata kombinovanjem **Item** objekata.
- **BuilderPatternDemo** klasa upotrebljava **MealBuilder** za izradu **Meal** objekta.



Slika 6.3 Primer korišćenja šablonu Builder [2.4]

## ŠABLON BUILDER: POSTUPAK PRIMENE (1.-3. KORAK)

*Kreiranje interfejsa, kreiranje konkretnih klasa koje primenjuju interfejsa Packing i kreiranje apstraktnih klasa koje primenjuju Item interfejs*

**Korak 1:** Kreiranje interfejsa

```
public interface Item {  
    public String name();  
    public Packing packing();  
    public float price();  
}  
public interface Packing {  
    public String pack();  
}
```

**Korak 2:** Kreiranje konkretnih klasa koje primenjuju Package interfejs:

```
public class Wrapper implements Packing {  
  
    @Override  
    public String pack() {  
        return "Wrapper";  
    }  
}  
public class Bottle implements Packing {  
  
    @Override  
    public String pack() {  
        return "Bottle";  
    }  
}
```

**Korak 3:** Kreiranje apstraktnih klasa koje primenjuju Item interfejs sa osnovnim funkcionalnostima.

```
public abstract class Burger implements Item {  
  
    @Override  
    public Packing packing() {
```

```
        return new Wrapper();
    }

    @Override
    public abstract float price();
}

public abstract class ColdDrink implements Item {

    @Override
    public Packing packing() {
        return new Bottle();
    }

    @Override
    public abstract float price();
}
```

## ŠABLON BUILDER: POSTUPAK PRIMENE (4 I 5. KORAK)

*Kreiranje klasa koje proširuju klase Burger i ColdDrink i kreiranje klase Meal sa objektom Item.*

**Korak 4:** Kreiranje konkretnih klasa koje proširuju klase **Burger** i **ColdDrink**.

```
public class VegBurger extends Burger {

    @Override
    public float price() {
        return 25.0f;
    }

    @Override
    public String name() {
        return "Veg Burger";
    }
}

public class ChickenBurger extends Burger {

    @Override
    public float price() {
        return 50.5f;
    }

    @Override
```

```
public String name() {
    return "Chicken Burger";
}
}

public class Coke extends ColdDrink {

    @Override
    public float price() {
        return 30.0f;
    }

    @Override
    public String name() {
        return "Coke";
    }
}
```

**Korak 5:** Kreiranje klase **Meal** koja sadrži objekte **Item** koji su gore definisani.

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

public class Meal {
    private List<Item> items = new ArrayList<Item>();

    public void addItem(Item item){
        items.add(item);
    }

    public float getCost(){
        float cost = 0.0f;
        for (Item item : items) {
            cost += item.price();
        }
        return cost;
    }

    public void showItems(){
        for (Item item : items) {
            System.out.print("Item : "+item.name());
            System.out.print(", Packing : "+item.packing().pack());
            System.out.println(", Price : "+item.price());
        }
    }
}
```

## ŠABLON BUILDER: POSTUPAK PRIMENE (6. I 7. KORAK)

### Kreiranje klase MainBuilder i BuilderPatternDemo

**Korak 6:** Kreiranje klase MainBuilder koja kreira Meal objekte.

```
public class MealBuilder {  
  
    public Meal prepareVegMeal (){  
        Meal meal = new Meal();  
        meal.addItem(new VegBurger());  
        meal.addItem(new Coke());  
        return meal;  
    }  
  
    public Meal prepareNonVegMeal (){  
        Meal meal = new Meal();  
        meal.addItem(new ChickenBurger());  
        meal.addItem(new Pepsi());  
        return meal;  
    }  
}
```

**Korak 7:** BuiderPatternDemo uses MealBuider to demonstrate builder pattern

```
public class BuilderPatternDemo {  
    public static void main(String[] args) {  
        MealBuilder mealBuilder = new MealBuilder();  
  
        Meal vegMeal = mealBuilder.prepareVegMeal();  
        System.out.println("Veg Meal");  
        vegMeal.showItems();  
        System.out.println("Total Cost: " +vegMeal.getCost());  
  
        Meal nonVegMeal = mealBuilder.prepareNonVegMeal();  
        System.out.println("\n\nNon-Veg Meal");  
        nonVegMeal.showItems();  
        System.out.println("Total Cost: " +nonVegMeal.getCost());  
    }  
}
```

## ŠABLON BUILDER: POSTUPAK PRIMENE (8. KORAK)

### *Provera rezultata*

**Korak 8:** Provera rezultata

Veg Meal

```
Item : Veg Burger, Packing : Wrapper, Price : 25.0
Item : Coke, Packing : Bottle, Price : 30.0
Total Cost: 55.0
```

Non-Veg Meal

```
Item : Chicken Burger, Packing : Wrapper, Price : 50.5
Item : Pepsi, Packing : Bottle, Price : 35.0
Total Cost: 85.5
```

## BUILDER DESIGN PATTERN (VIDEO)

*Trajanje: 13,04 minuta*

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

## VIDEO PREDAVANJE ZA OBJEKAT "ŠABLON BUILDER"

*Trajanje video snimka: 47min 30sek*

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

## ▼ Poglavlje 7

# Šablon Abstract Factory (Apstraktna fabrika)

## ŠABLON KREIRANJA ABSTRACT FACTORY

*Kod šablona Abstract Factory (Apstraktna fabrika), interfejs je odgovoran za kreiranje fabrike povezanih objekata, bez posebne specifikacije njihovih klasa*

### Namena:

Šablon Apstraktna fabrika ( [Abstract Factory Pattern](#)) radi sa super fabrikom koja kreira druge fabrike. To je fabrika za proizvodnju objekata, Spada u kategoriju šablona kreiranja. Daje najbolji način za kreiranje nekog objekta

### Interfejs:

Kod šablona Apstraktna fabrika, interfejs je odgovoran za kreiranje fabrike povezanih objekata, bez posebne specifikacije njihovih klasa. Svaka generisana fabrika daje objekte na način definisan šablonom Apstraktna fabrika.

### Opis problema:

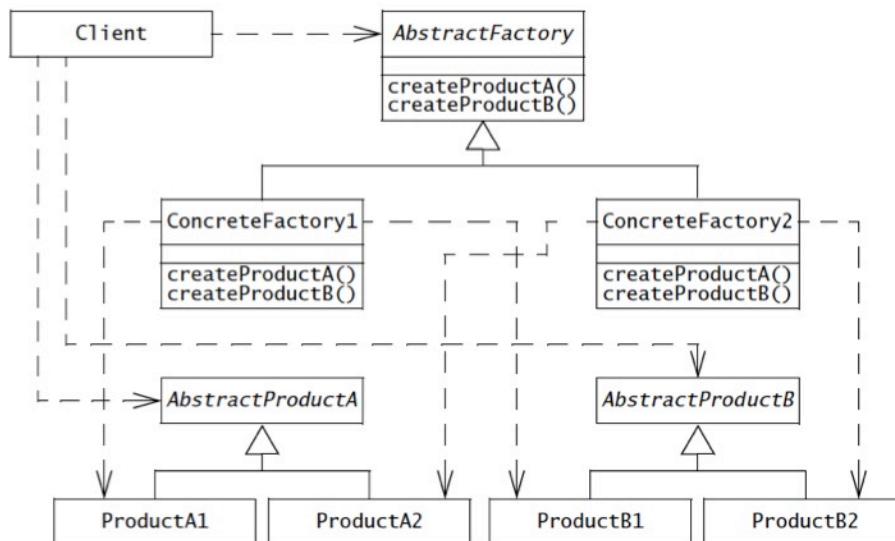
Štititi klijenta od različitih platformi koje obezbeđuju različite implementacije istog skupa koncepta.

### Rešenje:

Platforma predstavlja skup klasa **AbstractProducts**. Svaka od njih predstavlja neki koncept (npr. dugme) koji podržavaju sve platforme. Jedna klasa **AbstractFactory** objavljuje operacije za kreiranje pojedinačnih proizvoda.. Specifična platforma se realizuje sa klasom **ConcreteFactory** i skup klasa **ConcreteProducts** (po jedan za svaki **AbstractProduct**)

**ConcreteFactory** zavisi samo od povezanih **ConcreteProducts** **Client** zavisi samo od **AbstractProducts** i **AbstractFactory** klasa, tako da se može lako primeniti na različitim platformama.

### Struktura:



Slika 7.1 Šablon kreiranja Apstraktna fabrika [2.4]

## ŠABLON ABSTRACT FACTORY: POSLEDICE I IMPLEMENTACIJA

*AbstractFactory klasa štiti klijenta od različitih platformi koje obezbeđuju različite implementacije istog skupa koncepta.*

### Posledice:

**Client** je zaštićen od konkretnih klasa proizvoda. Moguća je zamena srodnih objekata za vreme izvršenja. Dodavanje novih proizvoda je teško, jer se moraju kreirati nove realizacije svake fabrike.

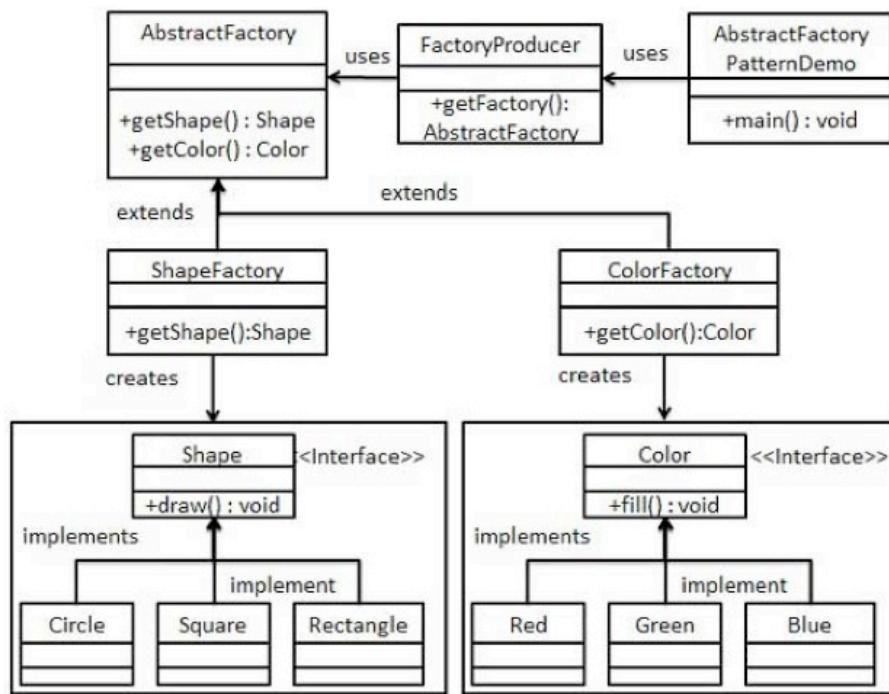
### Implementacija:

1. Kreirati **Shape** i **Color** interfejse i konkretne klase primenom njihovih interfejsa
2. Stvorimo klasu za apstraktну fabriku, tj. klasu **AbstractFactory**
3. Klase **ShapeFactory** i **ColorFactory** kao podklase klase **AbstractFactory**
4. Kreira se klasa za generisanje klasa: klasa **FactoryProducer**

### Primer:

Upotrebljava klasu **FactoryProducer** da bi se dobio objekat klase **AbstractFactory**. Prenosi se informacija o obliku koji želi da se kreira (CIRCLE, RECTANGLE i SQUARE) u klasi.

**AbstractFactory** da bi se dobio tip potrebnog objekta. Prenosi se i informacija o boji (GREEN, RED, BLUE) klasu **AbstractFactory** da bi se dobila boja potrebnog objekta



Slika 7.2 Primer upotrebe šablonu Apstraktna fabrija [2.4]

## ŠABLON ABSTRACT FACTORY: POSTUPAK PRIMENE (1. - 3. KORAKA)

*Primer upotrebljava klasu FactoryProducer da bi se dobio objekat klase AbstractFactory. Prenosi se informacija o obliku koji se želi da kreira (CIRC*

### Korak 1: Kreiranje interfejsa za Shapes

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

### Korak 2: Kreiranje konkretnih klasa koje primenjuju iterfejs

```
public class Rectangle implements Shape {

    @Override
    public void draw() {
        System.out.println("Inside Rectangle::draw() method.");
    }
}

public class Square implements Shape {

    @Override
    public void draw() {

```

```
        System.out.println("Inside Square::draw() method.");
    }
}
```

```
public class Circle implements Shape {

    @Override
    public void draw() {
        System.out.println("Inside Circle::draw() method.");
    }
}
```

### Korak 3: Kreiranje interfejsa Color

```
public class Circle implements Shape {

    @Override
    public void draw() {
        System.out.println("Inside Circle::draw() method.");
    }
}
```

## ŠABLON ABSTRACT FACTORY: POSTUPAK PRIMENE (4. I 5. KORAK)

*Kreiranje konkretnih klasa koje primenjuju isti interfejs i kreiranje klase AbstractFactory radi dobijanja fabrika za Object i Color objekte*

### Korak 4: Kreiranje konkretnih klasa koje primenjuju isti interfejs

```
public class Red implements Color {

    @Override
    public void fill() {
        System.out.println("Inside Red::fill() method.");
    }
}

public class Green implements Color {

    @Override
    public void fill() {
        System.out.println("Inside Green::fill() method.");
    }
}

public class Blue implements Color {
```

```
@Override  
public void fill() {  
    System.out.println("Inside Blue::fill() method.");  
}  
}
```

**Korak 5:** Kreiranje klase **AbstractFactory** radi dobijanja fabrika za **Object** i **Color** objekte

```
public abstract class AbstractFactory {  
    abstract Color getColor(String color);  
    abstract Shape getShape(String shape) ;  
}
```

## ŠABLON ABSTRACT FACTORY: POSTUPAK PRIMENE (6. KORAK)

*Kreiranje podklasa Factory klase AbstractFactory radu generisanja objekata*

**Korak 6:** Kreiranje podklasa Factory klase AbstractFactory radu generisanja objekata

```
public class ShapeFactory extends AbstractFactory {  
  
    @Override  
    public Shape getShape(String shapeType){  
        if(shapeType == null){  
            return null;  
        }  
        if(shapeType.equalsIgnoreCase("CIRCLE")){  
            return new Circle();  
        } else if(shapeType.equalsIgnoreCase("RECTANGLE")){  
            return new Rectangle();  
        } else if(shapeType.equalsIgnoreCase("SQUARE")){  
            return new Square();  
        }  
        return null;  
    }  
  
    @Override  
    Color getColor(String color) {  
        return null;  
    }  
}
```

```
public class ColorFactory extends AbstractFactory {  
  
    @Override  
    public Shape getShape(String shapeType){  
        return null;  
    }  
  
    @Override  
    Color getColor(String color) {  
        if(color == null){  
            return null;  
        }  
        if(color.equalsIgnoreCase("RED")){  
            return new Red();  
        } else if(color.equalsIgnoreCase("GREEN")){  
            return new Green();  
        } else if(color.equalsIgnoreCase("BLUE")){  
            return new Blue();  
        }  
        return null;  
    }  
}
```

## ŠABLON ABSTRACT FACTORY: POSTUPAK PRIMENE (7. I 8. KORAK)

*Kreiranje klase FactoryProducer za dobijanje fabrika za prenos informacija i upotreba klase FactoryProducer*

**Korak 7:** Kreiranje klase FactoryProducer za dobijanje fabrika za prenos informacija

```
public class FactoryProducer {  
    public static AbstractFactory getFactory(String choice){  
        if(choice.equalsIgnoreCase("SHAPE")){  
            return new ShapeFactory();  
        } else if(choice.equalsIgnoreCase("COLOR")){  
            return new ColorFactory();  
        }  
        return null;  
    }  
}
```

**Korak 8:** Upotreba klase FactoryProducer

Upotreba **FactoryProducer** da be se dobole fabrike konkretnih klasa prenosom informacija, kao što je tip.

```
public class AbstractFactoryPatternDemo {  
    public static void main(String[] args) {  
  
        //get shape factory  
        AbstractFactory shapeFactory =  
FactoryProducer.getFactory("SHAPE");  
  
        //get an object of Shape Circle  
        Shape shape1 = shapeFactory.getShape("CIRCLE");  
  
        //call draw method of Shape Circle  
        shape1.draw();  
  
        //get an object of Shape Rectangle  
        Shape shape2 = shapeFactory.getShape("RECTANGLE");  
  
        //call draw method of Shape Rectangle  
        shape2.draw();  
  
        //get an object of Shape Square  
        Shape shape3 = shapeFactory.getShape("SQUARE");  
  
        //call draw method of Shape Square  
        shape3.draw();  
  
        //get color factory  
        AbstractFactory colorFactory =  
            FactoryProducer.getFactory("COLOR");  
  
        //get an object of Color Red  
        Color color1 = colorFactory.getColor("RED");  
  
        //call fill method of Red  
        color1.fill();  
  
        //get an object of Color Green  
        Color color2 = colorFactory.getColor("Green");  
  
        //call fill method of Green  
        color2.fill();  
  
        //get an object of Color Blue  
        Color color3 = colorFactory.getColor("BLUE");  
  
        //call fill method of Color Blue  
        color3.fill();  
    }  
}
```

## ŠABLON ABSTRACT FACTORY: POSTUPAK PRIMENE (9. KORAK)

### *Provera rezultata*

Korak 9: Provera rezultata

```
Inside Circle::draw() method.  
Inside Rectangle::draw() method.  
Inside Square::draw() method.  
Inside Red::fill() method.  
Inside Green::fill() method.  
Inside Blue::fill() method.
```

Slika 7.3 Dobijen rezultat

## ABSTRACT FACTORY DESIGN PATTERN (VIDEO)

*Trajanje: 13,19 minuta*

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

## FACTORY METHOD PATTERN (VIDEO)

*Georgia Tech predavanje - Factory Method Pattern*

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

## PRIMER ZA FACTORY METHOD PATTERN (VIDEO)

*Georgia Tech predavanje - Primer za Factory Method Pattern*

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

## ▼ Poglavlje 8

# Pokazna vežba

## UVOD U ŠABLONE PROJEKTOVANJA

### *Cilj i opis vežbe*

U ovim vežbama studenti će se upoznati sa šablonima projektovanja (Design Patterns). Prvo će biti prikazano koji su to principi Objektno-Orijentisanog programiranja i kako prepoznati kod koji nije Objektno-Orijentisan. Biće prikazana neka od tehnika za odabir odgovarajućeg šablona projektovanja u rešavanju konkretnog problema. Biće opisano kako i kada treba primenjivati šabljone projektovanja i kako samostalno razvijati i unapređivati šabljone projektovanja.

Šabloni projektovanja (Design Patterns) predstavljaju dobru programersku praksu. Dobrim inženjerom se smatraju ljudi koji znaju da upotrebe šabljone projektovanja u adekvatnim situacijama. Šabloni projektovanja nam omogućavaju izbegavanje grešaka u dizajnu već primenu najbolje prakse koja se već pokazala kao kvalitetna u produkciji. Ovo znanje može mnogo da utiče na kvalitet koda i na uspešnost realizacije projekta. U većem timu poznavanje šabljona projektovanja omogućava članovima tima bolju komunikaciju i jasniji kod.

U ovim vežbama biće demonstrirana izrada i primena šabljona projektovanja: Singleton, Apstraktna fabrika (Abstract Factory) i Builder.

Da bi u potpunosti razumeli sadržaj vežbi neophodno je obnoviti Objektno Orijentisane pristupe i principe kao što su:

- Nasleđivanje
- Enkapsulacija
- Polimorfizam

## PREPOZNAVANJE LOŠEG OBJEKTNKO-ORIJENTISANOG KODA

*Postoji nekoliko načina kako se može prepoznati kod koji nije Objektno-Orijentisan. Neki od narednih primera objašnjavaju načine prepoznavanja*

Kao što smo na prethodnom primeru videli nije poželjno primenjivati Design Patterne nad ne objektno orijentisanim kodom. Prema tome treba da vidimo kad je neki kod loš. Evo nekoliko osnovnih situacija koje ukazuju na loš ne objektno orijentisani kod.

Korišćenje globalnih promenljivih. Ova situacija teško da može da se desi u Javi s obzirom

da java ne poznaje mehanizam globalnih promenljivih. Naravno programeri se nekad snađu i naprave neke varijante u kojima se promenljiva ponaša kao globalna (ova situacija je uvek praćena sa još nekoliko ne objektno orijentisanih situacija).

Globalne metode. Kao i u slučaju promenljivih nije preporučljivo imati globalne metode, iako u javi možemo da napravimo static metode koje se ponašaju kao globalne to treba izbegavati. Postoji jako mali broj situacija kada je poželjno ovakvo ponašanje.

Korišćenje public promenljivih. Nikad ni u jednoj situaciji ne treba imati public promenljive. Takve promenljive treba enkapsulirati.

Dugačke metode. Jedno od najjednostavnijih načina da se ustanovi da kod nije dobar je da se pogledaju pojedinačne metode u kodu. Ako neka metoda ne može da stane na jedan programerski ekran to znači da je i kod u celosti loš. Dobar kod podrazumeva da svaka metoda može u celosti da se vidi prilikom programiranja. Ako metoda ne može da se vidi mora da se izvrše odgovarajući refactoring-zi i da se kod dovede u stanje da svaka metoda može da se vidi u celosti bez skrolovanja. Ovo je veoma bitan zahtev koji mnogi programi ne ispunjavaju. Ovaj zahtev je ujedno i osnov za mnoge druge napredne tehnike programiranja pa i primenu Desing Patterna.

Korišćenje dugačkih if-else-if struktura. Ako koristimo bar dve if-else-if naredbe onda skoro sigurno imamo metodu koja ne može da stane u jedan programerski ekran. Tako da je ova situacija usko vezana sa problemom dugačkih metoda. Ovakav kod je ne poželjan. Najjednostavniji način da da se dugačka if-else-if struktura razbijje je da se pretvori u neki oblik polimorfizma, najčešće overload.

Gora, zato što se teže refaktoriše u čiste objektno orijentisane strukture. Postoji nekoliko razloga zašto je ova naredba ne poželjna. Prvi razlog, parametar switch naredbe može da bude samo int ili char nikako objekat što umanjuje primenu objekata i objektno orijentisanih principa. Drugo, switch naredbe skoro uvek prouzrokuju da ne može metoda da u celosti stane na ekran. Treće, u okviru switch naredbe ima previše ključnih reči (break, default, case) što čini kod manje čitljivim. Prilikom programiranja treba u potpunosti izbaciti ovu naredbu iz programerske prakse.

## ODABIR ŠABLONA PROJEKTOVANJA

*Nekoliko koraka prilikom odabira šablona projektovanja (Design Patterna)*

Prilikom projektovanj projekta ili prilikom rešavanje nekog aktuelnog problema u okviru projekta često treba da uzmemu u obzir ovde izložene Design Patterne. Iskusniji projektanti odnosno programeri koji već dugo koriste patterne obično prilikom pogleda na problem znaju koji Design Pattern bi trebalo upotrebiti i kako. Veći probem je kod manje iskusnih koji se tek susreću sa Design Patternima. Manje iskusnim programerima može da se čini da su neki patterni jako slični i da nema neke velike razlike, ili da nemogu da sagledaju problem u celosti. Najčešće programeri savladavaju jedan po jedan design pattern ali i u tim situacijama moraju da imaju nekakvu liniju vodilju kako da odaberu pravi pattern za svoj trenutni problem. Evo par koraka koji mogu da pomognu prilikom odabira pravog Design patterna.

Prvo se odlučimo za grupu Desing Patterna. Builder, Structural ili Behavioral. Nakon toga razmatramo Design Patterne iz te grupe i odbacimo one koji očigledno nemaju veze sa trenutnim problemom. Nakon toga analiziramo moguću upotrebu svakog od Design Patterna

preostalog u toj grupi, na zadati problem. Prilikom analize treba razmišljati, ne samo o tome da li dati Design Pattern rešava dati problem već i o tome kako bi se taj Design Pattern uklopio u postojeći kod. Nakon odabira pravog design patterna proveriti još jednom da li postoji neki pattern koji se adresira na slični problem i izvršiti skraćenu analizu da li možda može da se on koristi umesto postojećeg. Prilikom toga uzimamo u obzir sve Design Patterne a ne samo Design Patterne iz odabrane gurpe. Razlog ovom koraku je potencijalna mogućnost da smo pravobitno pogrešno odabrali grupu Design Patterna. U ovom koraku nam od velike pomoći može biti šema Design Patterna prikazana na slici.

Kada smo se konačno odlučili za pravi design Pattern treba da implementiramo dati Design Pattern, nakon toga treba napisati JUnit testove koji nam definišu ponašanje Patterna u konkretnoj situaciji. I na kraju proveriti da li sve to dobro funkcioniše sa postojećim kodom.

## ✓ 8.1 Vežba za šablon Singleton

### POKAZNI PRIMER 1 - SINGLETON ŠABLON

*Uvod u Singleton šablon projektovanja i njegova najčešća implementacija u kodu*

**Vreme trajanja: 5 minuta**

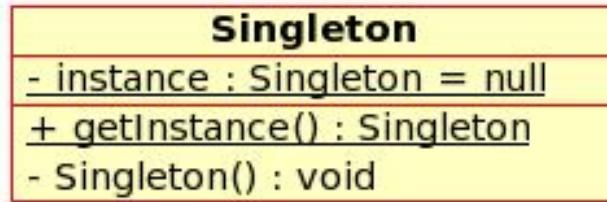
Verovatno najpoznatiji šablon projektovanja. Na veliku žalost ovaj Design Pattern se često pogrešno koristi odnosno neki programeri pomoću njega kreiraju neku vrstu globalnih promenljivih ili metoda.

U okviru ovog šablon projekovanja primjenjen je samo jedan od principa objektno orientisanog programiranja a to je Enkapsulacija. Enkapsulacija je malo čudno primenjena nad konstruktorom tako da je kreiran konstruktor kome može da se pristupi samo iz same klase. Naravno da ovo ne bi moglo da funkcioniše da nema static modifikatora. Pristup enkapsuliranom konstruktoru je moguće samo iz static metoda. U ovom slučaju to se radi pomoću metode **getInstance()**. Da bi ovaj šablon mogao da funkcioniše neophodna je statička instanca same klase.

Singleton klasa je karakteristična po tome da je moguće imati samo jednu instancu ove klase na nivou celokupne aplikacije. Zahvaljujući ovoj činjenici možemo da kreiramo klasu i da je pozivamo odakle god nam treba bez potrebe da prosleđujemo njenu instancu svuda. Singleton klase bi trebalo da budu samo klase koje su jedinstvene na nivou aplikacije i potrebne na više mesta. Na primer neke utiliti klase.

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

```
}
```



Slika 8.1.1 UML prikaz klase [Izvor: Autor]

## POKAZNI PRIMER 2 - FILELOGGER KLASNI MODEL

*Primer Singleton šablon na primeru izrade klase za logovanje (FileLogger) u log.txt fajl*

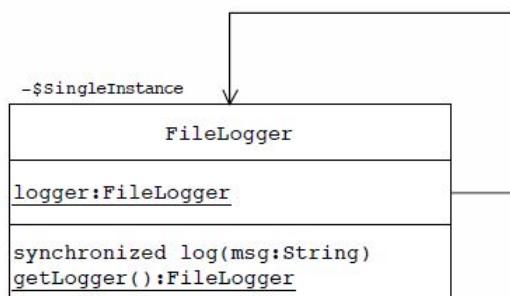
**Vreme trajanja: 10 minuta**

Kao što je ranije napomenuto imati Singleton je korisno kada je potrebno imati jednu fizičku instancu onoga što objekat predstavlja. Ovo je tačno u slučaju FileLogger zato što postoji jedan fizički log fajl. U aplikaciji kada više različitih objekata pokušava da loguje poruke u fajl, potencijalno može biti više instanci FileLogger klase prilikom upotrebe svakog od tih objekata. Ovo može dovesti do različitih pitanja konkurentnog pristupa istog fajla iz različitih objekata. Jedan od rešenja je da se poseduje jedna globalna instanca klase FileLogger u okviru aplikacije i tako jednoj instanci mogu pristupiti svi objekti. Ovo ne rešava problem u potpunosti.

- Ne sprečava klijenta da kreira novu instancu klase FileLogger
- Ne sprečava da više niti u okviru istog klijenta izvršava log metodu.

Da bi se ovo sprečilo primeniće se Monitor koncept i deklarisati metodu log(String param) kao sinhronizovanu (synchronize).

Ovo ne sprečava da više niti klijenta izvršava log metodu ali sprečava klijenta da kreira više instanci klase FileLogger. Korišćenjem sinhronizovane metode rešavamo konkurentan pristup, a primenom Singleton šablon kreiranje više instanci klase.



Slika 8.1.2 UML prikaz Singleton klase FileLogger [Izvor: Autor]

## POKAZNI PRIMER 2- FILELOGGER KLASA IMPLEMENTACIJA

*Implementacija koda klase FileLogger koja se odnosi na UML klasu i njena primena u okviru LoggerTest klase*

Neophodno je napraviti konstruktor klase FileLogger private, i time smo omogućili pristup svim metodama unutar klase pristup tom konstruktoru.

Moguće je (i poželjno) napraviti public interfejs klase za metodu getInstance kako bi se omogućilo klijentu da se pristupi metodi. (U ovom primeru je urađena demonstracija bez interfejsa)

Unutar metode getInstance klijentu je dozvoljen pristup instanci klase FileLogger. Ova public metoda mora da bude static da bi korisnik mogao da je poziva bezinstanciranja klase. Unutar ove metode kreira se i vraća instanca FileLogger klase pristupanjem privatnog konstruktora. Ovo se izvršava samo jedanput prilikom prvog pozivanja ove metode. Svaki naredni poziv getInstance metode vraća istu FileLogger instancu klasa koja je kreirana prvi put i nova instanca klase se ne kreira ponovo.

```
public class LogerTest {  
  
    public LogerTest() {  
        FileLogger log = FileLogger.getFileLogger();  
        log.log("Testiranje log");  
    }  
  
    public static void main(String[] args) {  
        new LogerTest();  
    }  
}
```

```
import java.io.File;  
import java.io.FileNotFoundException;  
import java.io.PrintWriter;  
  
public class FileLogger {  
  
    private static FileLogger logger;  
  
    //sprečavanje klijenta da koristi konstruktor  
    private FileLogger() {}  
  
    public static FileLogger getFileLogger() {  
        if (logger == null) {  
            logger = new FileLogger();  
        }  
        return logger;  
    }  
}
```

```
public synchronized void log(String msg) throws FileNotFoundException {  
    File file = new File("log.txt");  
    if (file.exists()) {  
        try (PrintWriter output = new PrintWriter(file)) {  
            output.print(msg);  
            output.close();  
        }  
    }  
}
```

## ✓ 8.2 Pokazni primer 3 - vežba za šablon Builder

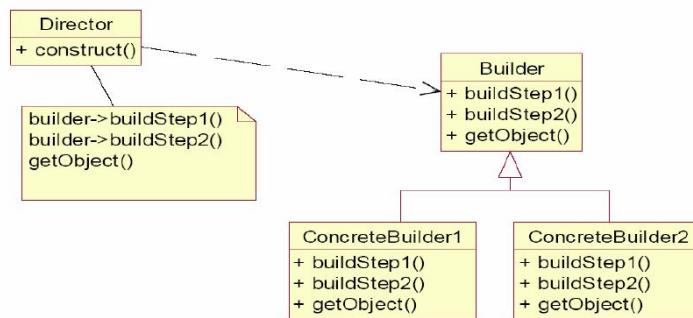
### UPOTREBA ŠABLONA BUILDER (GRADITEĽJ)

*Kada objekat nije moguće napraviti iz jednog puta jer nema dovoljno informacija, onda je dobro koristiti builder šablon projektovanja. Vreme trajanja: 20 minuta*

Na slici 1 je prikazan UML dijagram Design Pattern-a Builder.

U nekim situacijama objekat ne možemo da napravimo u jednom cugu već moramo da ga gradimo korak po korak kako saznajemo nove informacije. U takvim situacijama od velike pomoći može da nam bude Design Pattern Builder. Iako builder Desing Pattern može da se kreira i upotrebom jedne klase u opštem slučaju možemo da imamo više konkretnih buildera za istu stvar pa tada treba da imamo i apstraktnu nad klasu koju instanciraju konkretnе builder klase. Kao što ćemo videti u narednom primeru apstraktnu nad klasu može zameniti Interface.

```
public interface BuildInterface {  
    public void addIme(String ime);  
    public void add(Calendar datum);  
    public void add(OsnovnaSkola skola);  
    public void add(SrednjaSkola skola);  
    public void add(Fakultet faks);  
    public void add(Kurs kurs);  
    public void add(Sertifikat setrifikat);  
    public void add(boolean pusi);  
    public void add(int godine);  
    public CV dajMiCV(); }
```



Slika 8.2.1 UML prikaz šablona Builder

## KREIRANJE BUILDER ŠABLONA

### *Izrada interfejsa/Apstraktne klase i izvedenih klasa na primeru CV*

Želimo da napravimo Builder klasu koja će nam pomoći u pravljenju CV-a. Osnovna ideja je da sam CV ima veliki broj potencijalnih elemenata i da gradimo taj objekat tako što dodajemo elemente po potrebi. Nezgodno bi bilo da moramo ceo CV da kreiramo iz jednog konstruktora jer bi moralo sve u napred da se zna i ne bi bili u mogućnosti da ga naknadno dopunjujemo. Rešenje za ovaj problem se krije u Design Patternu Builder.

Umesto apstraktne klase koristićemo interface. Apstraktna klasa u Design Patternu Builder služi za definisanje metoda koje konkretna Builder klasa mora da ima. Za ovu potrebu je mnogo praktičnije koristiti Interface.

Kao što je prikazano ovaj interface definiše metode koje su nam neophodne za kreiranje CV-a.

### Obična informacija

```

public class ObicnaInformacija {
    private String informacija;
    public ObicnaInformacija(String informacija){ this.informacija = informacija; }
    public String getInformacija(){ return informacija; }
    public void setInformacija(String informacija){ this.informacija = informacija; }
    @Override public String toString()
    { return informacija; }
}

```

Za potrebe kreiranja CV-a napravićemo nekoliko klasa koje će nam kasnije pomoći u izgradnji objekta pomoću Design Patterna Builder.

Kao osnova za ove dodatne klase napravićemo klasu koja ima osnovne informacije. Ovu klasu ćemo kasnije da nasledimo u svakoj klasi koja to koristi.

Ova obična infomracija ima samo jedan podatak a to je string informacija.

### Izvadene klase

```

public class OsnovnaSkola extends ObicnaInformacija {
    public OsnovnaSkola(String skola){ super(skola);}
}

```

```
public class SrednjaSkola extends ObicnaInformacija {  
    public SrednjaSkola(String skola){ super(skola); }  
}  
public class Fakultet extends ObicnaInformacija {  
    Fakultet(String faks){  
        super(faks); } }
```

## BUILDER KLASA

### Izrada Builder klase i kreiranje klase CV

Konkretnе klase које ће нам помоћи у раду Builder klase наследјују класу ObicnaInformacija. Кao конкретне класе направићемо конкретне класе OsnovnaSkola, SrednjaSkola, Fakultet itd.... Ове класе ће нам помоћи да помоћу њих kreiramo одређена поглавља у CV-u, koji kreira Builder. Na ovom mestu neћemo do kraja implementirati ове класе (пошто су мање bitne za rad Builder Design Patterna) ali uz predavanje можете da skinete i primer где je sve do kraja implementirano.

Da bi направили builder klasu за наше потребе неophodno je da implementiramo traženi interface BuilderInterface. Iako deluje мало nepotrebno правити interface koji implementira само једна клаша ово је добра programersка пракса. Ako se pojavi потреба да се CV (у овом примеру) kreira на други начин у односу на онaj који smo implementirali можемо да направимо још један builder sa traženom implementacijom koji наследjuje isti interface uz zamenu konkretnog objekta.

Zahvaljujući tome свака измена је дaleко једноставнија. Sve više искусни програмери започињу писање клаша тако што прво definišu interface па тек онда kreiraju клашу koja implemetnira taj interface.

Kao што се вidi искористили smo постојање različitih klasa да uvedemo overload методу add(). Zahvaljujući činjenici da svaki element dodajemo kao posebnu клашу можемо да направимо jednu методу add() која vrši dodавање у CV а она ће радити različit posao u zavisnosti od конкретне клаше коју јој пошалјемо.

```
public class Builder implements BuildInterface {  
    private CV moj = new CV();  
    public Builder() { }  
    public void addIme(String ime)  
    { moj.dodaj("Ime i prezime: \t" + ime); }  
    public void add(Calendar datum) {  
        SimpleDateFormat dateFormat = new SimpleDateFormat("dd.MM.yyyy");  
        moj.dodaj("Rođen: \t" + dateFormat.format(datum.getTime()));  
    }  
    public void add(OsnovnaSkola skola)  
    { moj.dodaj("Završio osnovnu školu: \t" + skola); }  
    public void add(SrednjaSkola skola)  
    { moj.dodaj("Završio srednju školu: \t" + skola); }  
    public void add(Fakultet faks)  
    { moj.dodaj("Završio fakultete: \t" + faks); }  
    public void add(Kurs kurs)
```

```
{ moj.dodaj("Pohađao kurs:\t" + kurs); }
public void add(Sertifikat sertifikat)
{ moj.dodaj("Ima Sertifikat:\t" + sertifikat); }
public void add(boolean pusi)
{ if(pusi) moj.dodaj("Pušač"); }
public void add(int godine)
{ moj.dodaj("Godina radnog staža:\t" + godine); }
public CV dajMiCV(){
return moj;
}
```

Klasa CV u sebi sadrži kompletan CV koji je kreirala klasa Builder. Ova klasa samo sadrži podatak, string tekst CV-a i metodu dodaj koju koristi Builder da bi gradio CV.

Klasa CV koja sadrži CV je oslobođena posla kreiranja CV-a. Da nismo primenili Builder Design Pattern ova klasa bi verovatno bila prevelika i što je još važnije imala bi više zaduženja. Upotreboru Design Patterna Builder kod je jednostavan i elegantan i svaka klasa ima svoje jedno jednostavno zaduženje.

```
public class CV {
private StringBuilder cv = new StringBuilder();
public CV() { }
public void dodaj(String sledeće) {
cv.append("\n");
cv.append(sledeće);
}
@Override public String toString()
{ return cv.toString(); }
}
```

## UPOTREBA BUILDER KLASE

### *Upotreba kreiranog šablonu projektovanja Builder na primeru CV*

Cela moć ovog Desing Patterna krije se u upotrebi klase Builder. Na prethodnom primeru je prikazana upotreba Buildera prilikom kreiranja CV-a.

Građenjem našeg CV-a upotrebom klase Builder. Klasi Builder prosleđujemo klase koje definišu različite delove CV-a, kao što su informacije o osnovnoj školi, srednjoj školi, fakultetu i godinama radnog iskustva. Primetićemo da smo za sve ove operacije koristili istu metodu, metodu **add()**. Kreirali smo klasu koja je zadužena za odgovarajući tip informacija a nakon toga smo tu informaciju poslali klasi Builder. Upotreba klase Builder je na ovaj način izuzetno jednostavna jer uvek se poziva ista metoda.

Na kraju građenja našeg CV-a pozivamo metodu **dajMiCV** koja nam vraća konkretan izgrađen CV. Ako je potrebno možemo naknadno dodati još informacija u CV upotrebom Buildera i ponovo pomoći metode dajMiCV uzeto novu verziju CV-a.

```
Builder pravljeneCva = new Builder();
pravljeneCva.addIme("Aleksandar");
```

```
Calendar rodjenje = Calendar.getInstance();
rodjenje.set(Calendar.YEAR, 1999);
rodjenje.set(Calendar.MONTH, 7);
rodjenje.set(Calendar.DAY_OF_MONTH, 14);
pravljenjeCva.add(rodjenje);
pravljenjeCva.add(new OsnovnaSkola("Isidora Sekulić"));
pravljenjeCva.add(new SrednjaSkola("XIV Beogradska Gimnazija"));
pravljenjeCva.add(new Fakultet("FIT - Beograd"));
pravljenjeCva.add(13);
CV prikaz = pravljenjeCva.dajMiCV();
System.out.println(prikaz);
```

## OO PRINCIPI PRIMENJENI NA BUILDER

*Neki od najvažnijih Objektno-orientisanih principa primenjenih na klasu Builder*

Interface ili Apstraktna klasa

- BuilderInterface (interface)
- Builder

### NadKlase

1. ObicnaInformacija

2. OsnovnaSkola

3. SrednjaSkola

4. Fakultet

5. Kurs

6. Sertifikat

U Design Patternu Builder su primenjeni sva tri osnovna principa Objektno Orientisanog programiranja.

Klasa Builder je implementirala interface BuilderInterface i time smo iskoristili princip implementacije interface-a. Pomoćne klase u izgradnji CV-a imaju svoju malu hijerarhiju. Imamo nad klasu ObicnaInformacija koju nasleđuju sve ostale klase. I time smo kreirali jednu malu i hijerarhiju pomoćnih klasa.

### Enkapsulacija

Klasa CV se nalazi u okviru klase Builder se ne vidi se prilikom kreiranja CV-a. Tokom procesa kreiranja CV-a mi koristimo samo klasu Builder dok je klasa CV enkapsulirana u okviru klase Builder i pristupa joj se samo pomoću metode dajMiCV()

StringBuilder u okviru CV klase se takođe ne vidi odnosno pristupa mu se preko metoda dodaj i toString. Svi ovi podaci su nevidljivi za onog ko koristi ove klase, korisnik klasa može samo da pristupi metodama preko kojih je predviđen pristup tim podacima i time se ostvaruje enkapsulacija podataka. Podatak informacija se nalazi u nad klasi ObičnaInformacija i pristupa mu se posredno iz konstruktora podklase i metode toString podklase.

### Polimorfizam

Ključni mehanizam u ovom primeru je polimorfizam, odnosno Overload. Klasa Builder ima 8 metoda koje se zovu add, sve metode vraćaju isto void ali sve metode primaju različite tipove

podataka. Ovo obezbeđuje kako jednostavnu upotrebu Overload metoda add(), to znači da sve što želimo da dodamo dodajemo preko metode add().

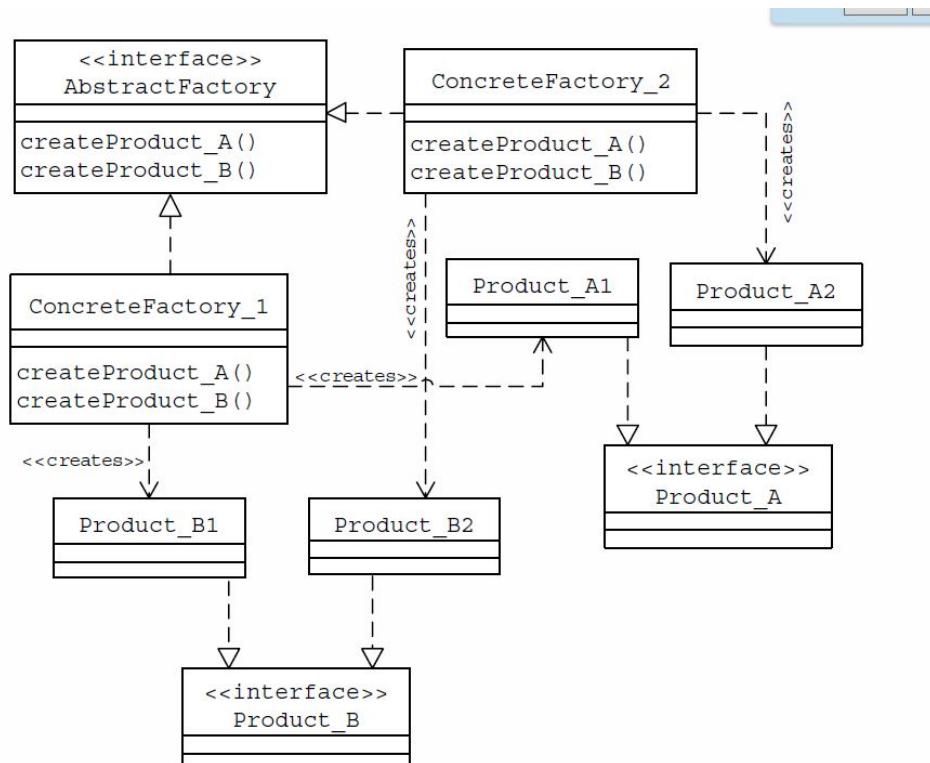
Na kraju da zaključimo kako je veoma efikasan Design Pattern Builder koji može dosta da nam pomogne u stvari samo ima adekvatnu primenu osnovnih principa OO. Ovo se naravno odnosi i na sve ostale Design Patterne koje ćemo učiti i one koje nećemo obradivati u okviru predmeta CS112 Objektno Orientisano Modelovanje.

## ▼ 8.3 Pokazni primer 4 - vežba za Apstraktnu fabriku

### ABSTRACTFACTORY ŠABLON

*Definisanje šablonu Abstract Factory, tj. abstraktna fabrika. Vreme trajanja: 20 minuta*

Na slici je prikazan UML dijagram apstraktne fabrike. Kao što se može videti Apstraktna fabrika ima dva dela. Prvi deo odnosi se na hijerarhiju proizvoda, a drugi na Fabrike tih proizvoda. Svaka fabrika i svaki proizvod imaju svoju apstraktnu nad klasu. U delu programa koji koriste klase svi proizvodi i fabrike se vode pod apstraktnim proizvodom i apstraktnom fabrikom, a konkretnе instance proizvoda za konkretnu fabriku se kreiraju ispod tih apstraktnih



Slika 8.3.1 UML dijagram AbstractFactory šablonina projektovanja [Izvor: Autor]

## DEFINISANJE APSTRAKTNE FABRIKE

*Definisanje apstraktne fabrike, apstraktnih i konkretnih proizvoda i njihova značaja i upotreba*

Kreiranje apstraktnih proizvoda:

```
abstract class Vrata { }
abstract class Prozor { }
```

U ovom slučaju kreiraju se dva apstraktna proizvoda vrata i prozori. Proizvodi se nikada samostalno ne prave već se dobijaju kao usluga rada fabrike, odnosno fabrika isporučuje proizvode. Fabrika će uvek isporučivati prozor ili vrata, a ispod će se nalaziti neki od konkretnih proizvoda.

Kreiranje konkretnih proizvoda. Cela konstrukcija postoji zbog kreiranja konkretnih proizvoda. U ovom primeru imamo Vrata i Prozore. Odnosno na kući možemo da imamo ili Aluminijumska vrata i prozore ili Drvena vrata i prozore. Mi ne želimo da u isto vreme imamo i jedno i drugo i zato smo i odabrali design pattern AbstractFactory. Primer konkretnih proizvoda:

```
class DrvenaVrata extends Vrata { }
class DrveniProzor extends Prozor { }
class AluminijumVrata extends Vrata { }
class AluminijumProzor extends Prozor { }
```

Svaki od ovih konkretnih proizvoda mora da nasledi odgovarajući abstraktni proizvod. Treba imati na umu da svaka metoda koja nam treba u nekom proizvodu mora da bude definisana u abstraktnoj nad klasi, inače joj nikad nećemo moći pristupiti.

Prvo kreiramo abstraktnu fabriku. Uloga fabrike je da generiše konkretne proizvode. Svaki od tih proizvoda se neće samostalno kreirati već će se tražiti usluga klase fabrika koja će da isporučuje konkretnu instancu proizvoda. Ova abstraktna fabrika ima ulogu da definiše sve proizvode koje mogu konkretne fabrike da naprave. S obzirom da svaki proizvod ima svoj odgovarajući abstraktni proizvod u abstraktnoj fabrici ćemo koristiti samo te abstraktne proizvode. U primeru datom u kodu prikazujemo abstraktnu fabriku koja je u stanju da definiše proizvodnju dva proizvoda Vrata i Prozora.

```
abstract class AbstractFactory
{
    // Methods
    abstract public Vrata createVrata();
    abstract public Prozor createProzor();
}
```

## KORIŠĆENJE APSTRAKTNE FABRIKE

*Izrada konkretnih fabrika Aluminuma i drveta i primena šablonu apstraktne fabrike*

Kada imamo apstraktnu fabriku možemo da pređemo na pravljenje konkretnе fabrike. U primeru prikazanom u kodu pravimo konkretnu fabriku Vrata i Prozora i nazivamo je DrvenaFabrika. Ova drvena fabrika je zadužena za kreiranje konkretnog prozora odnosno vrata. Svaka od metoda createVrata() i createProzor() vrše implementaciju metoda definisanih u apstraktnoj klasi AbstractFactory. Rezultat rada je apstraktna klasa Prozor odnosno apstraktna klasa Vrata. Naravno s obzirom da su ovo apstraktne klase naša metoda mora da kreira instancu stvarnih klasa, tako da naše metode prave instance klasa konkretnih proizvoda DrvenaVrata i DrveniProzor. Na ovaj način fabrika sakriva konkretnе implementacije jer kad se instanca primi od strane fabrike ovi proizvodi se vode kao Vrata i Prozor a ne kao DrvenaVrata i DrveniProzor.

```
class DrvenaFabrika extends AbstractFactory {  
    @Override public Vrata createVrata(){  
        return new DrvenaVrata();  
    }  
    @Override public Prozor createProzor(){  
        return new DrveniProzor();  
    }  
}
```

Na isti način pravimo i aluminijum fabriku

```
class AluminijumFabrika extends AbstractFactory {  
    @Override public Vrata createVrata(){  
        return new AluminijumVrata();  
    }  
    @Override  
    public Prozor createProzor(){  
        return new AluminijumProzor();  
    }  
}
```

Kompletна priprema koju smo do sad napravili ima samo jedan cilj sakrivanje konkretnog proizvoda od korisnika. Upotreba ovog šablona je sledeća: Kreiramo instancu apstraktne klase AbstractFactory tako što ispod podmetnemo konkretnu fabriku u primeru DrvenaFabrika. Nakon toga možemo u celom programu da kreiramo vrata i prozore tako što ćemo pozivati metode createVrata i createProzor. Sve to radimo pod instancom koja predstavlja apstraktnu fabriku. Šta se događa kad želimo sa drvenih proizvoda da se prebacimo na aluminijumske. U tom slučaju treba samo na jednom mestu da umesto DrvenaFabrika instanciramo AluminijumFabrika i u celoj aplikaciji dobijena vrata i prozori će biti sad aluminijumski.

```
AbstractFactory af = new DrvenaFabrika();  
Vrata v1 = af.createVrata();  
Prozor p1 = af.createProzor();
```

## PRINCIPI PRIMENJENI NA APSTRAKTNU FABRIKU

*Neki od Objektno-orientisanih principa je primenjeno na šablon Apstraktne fabrike, a to su hijerarhija, enkapsulacija i polimorfizam*

### Hijerarhija

## 1. AbstractFactori (abstraktna klasa)

- DrvenaFabrika
- AluminijumFabrika

## 2. Vrata (abstraktna klasa)

- DrvenaVrata , • AluminijumVrata

## Prozor (abstraktna klasa)

- DrveniProzor • AluminijumProzor

Ceo koncept abstraktne fabrike bi bio nemoguć da nema nekoliko osnovnih objektno orientisanih principa. Kao najvažniji za ovaj Design Pattern svakako možemo da izdvojimo nasleđivanje. Korišćenjem nasleđivanja napravili smo abstrakte klase proizvoda i abstrakte klase fabrike. Nasleđivanjem ovih abstraktnih klasa smo kreirali konkretne klasе. Ovde je primenjuje još jedna važna osobina nasleđivanja a to je da instance dece mogu da se smeste pod instance roditeljske nad klase čak i kad je nad klasa abstraktna.

## Enkapsulacija

- Nije pokazana u primeru
- U stvarnoj primeni svi podaci u klasama moraju da budu enkapsulirani

Druga dva principa objektno orientisanog programiranja takođe su primenjeni u ovom Design Patternu. Enkapsulacija nije nigde u ovom primeru direktno prikazana tako ali u stvarnoj implementaciji svi podaci u konkretnim klasama moraju da budu private odnosno enkapsulirani. Međutim ideja enkapsulacije je u ovom Design Pattern-u jako prisutna jer mi sakrivamo (enkapsuliramo) konkretnu implementaciju.

## Polimorfizam

- Sve metode koje nam trebaju moraju da se definišu u apstraktnim klasama.
- Konkretne klase moraju da implementiraju metode iz apstraktnih klasa

Polimorfizam je u šablonu projektovanja Abstract Factori prisutan i on obezbeđuje da iste metode createVrata i createProzor mogu da se implementiraju na različite načine u različitim podklasama. Ovo je tipičan primer Override metoda. Još jedna činjenica vezana za polimorfizam i abstract factory, s obzirom na samu implementaciju ovog šablonu projektovanja nameće se činjenica da ne možemo da koristimo metodu ako ne koristimo override mehanizam. Odnosno u okviru proizvoda koji dobijemo možemo da koristimo samo one metode koje su definisane u apstraktnoj nadklasi i implementirane u konkretnim klasama.

**Napomena:** Konkretne klase ne smeju da imaju ni jednu javnu metodu koja nije definisana u apstraktnim klasama. Metodama može da se pristupi samo kroz override mehanizam.

## UPOTREBA PRIMENE APSTRAKNTE FABRIKE

### *Nekoliko načina gde je dobro/poželjno koristiti Apstraktnu fabriku*

Design Pattern apstraktna fabrika ima širok spektar primene. Evo nekoliko situacija u kojim je poželjno koristiti Design Pattern Abstract Factory.

- Pristup raznim Bazama podataka – Nije redak slučaj da se zahteva da ista aplikacija može da radi sa nekoliko različitih baza podataka. Naravno svaka baza može da ima svoje specifičnosti

primene. U toj situaciji koristi se Design Pattern Abstract Factory. U tom slučaju konkretni proizvodi su klase koje rade sa bazom dok fabrika određuje koji skup klasa će se koristiti. Tako da možemo da imamo na primer Oracle fabriku koja rukovodi kreiranjem klasa koje pristupaju Oracle-voj bazi i Sybase fabriku koja kreira obekte koji umeju da rade sa Sybase bazom podataka. Kad želimo da u okviru aplikacije umesto jedne baze koristimo drugu dovoljno je da to samo na jednom mestu zamenimo i biće zamenjeno u celoj aplikaciji. Naravno zahvaljujući mehanizmu apstraktne fabrike.

- GUI komponente - želimo da promenimo izgled komponentama GUI-a ako je ceo gui napravljen po principima apstraktne fabrike dovoljno je da zamenimo fabriku pa će i sve komponente da se izmene. Apstraktna fabrika je najjednostavniji način da se ugradi takva mogućnost u aplikaciju.
- Matematički poračuni - postoje situacije kad se koristi neki sistem pa bi u slučaju da se koristi taj sistem sve klase i njihovi metodi trebali da imaju jedan način rada, a u drugom slučaju drugi način. To može da se ostvari pomoću Apstraktne Fabrike.
- Proizvodi koji moraju da se slažu - na primeru koji je prikazan u predhodnom primeru dali smo baš ovakav primer.

## OO SYSTEMS ANALYSIS AND DESIGN - USE CASE REALIZATIONS (VIDEO)

*Trajanje: 35:03 minuta*

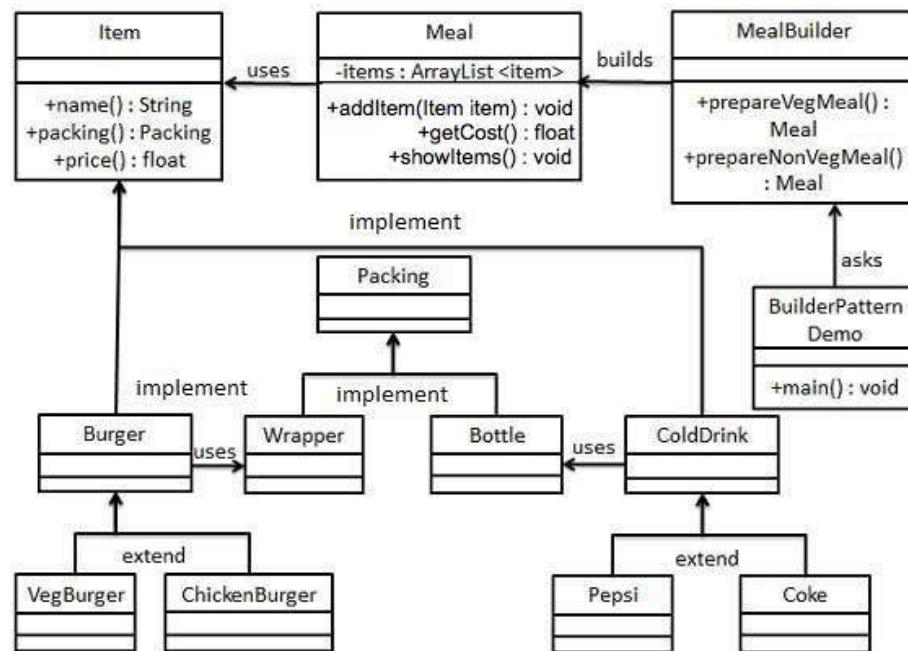
**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

## ❖ 8.4 Pokazni primer 5 - sistem restorana

### PRIMENA ŠABLONA BUILDER NA SISTEM RESTORANA

*Naredna sekcija definiše primenu Singleton šablon na sistem restorana. Trajanje 15 minuta*

Na osnovu dijagrama klase koji je definisan na slici 1 možemo zaključiti da je za sistem restorana najpogodnije primeniti **Builder šablon**. Slika 2 definiše klasni dijagram nakon primene Builder šablon. Na slici 2 možemo videti interfejs **MealBuilder** koji sadrži metodu **prepareMeal** i odnosi se na izgradnju samog objekta obroka (**Meal**) koji se dalje sastoji od mnogo **Item**-a.



Slika 8.4.1 Primer primene Builder Patterna [Izvor: Autor]

Slika 8.4.2 RMOS sistem sa primenom Builder šablonu [Izvor: Autor]

## ✓ Poglavlje 9

### Individualna vežba

#### METODA PROJEKTOVANJA: ZADACI 1 I 2 ZA INDIVIDUALNI RAD

*Zadaci koji se odnose na deo lekcije koja se bavi metodama projektovanja softvera, bez primene šablon projekovanja.*

**1. Zadatak** (10 minuta): Za studiju slučaja - Sistem za prikupljanje sa stanica za merenje meteoreoloških podataka, definisane su mnoge klase. Na slici 1 prikazano je samo jedan deo tih klasa. Utvrdite druge oobjekte tog sistema, analizom slučajeva korišćenja i njihovih scenarija. Napravite hijerarhijsku strukturu klasa tih objekata, uz korišćenje i svojstva nasleđivanja.

VremenskaStanica	VremenskiPodaci
Indentifikator	
izvestiVreme ()	temperaturaVazduha
izvestiStatus ()	TemperaturTla
stediSnagu (instrumenti)	brzinaVetra
daljinskoUpravljanje (komande)	pravacVetra
rekofigurisi (komande)	pritisci
restart (instrumenti)	Padavine
Iskljuci (instrumenti)	pokupi ()
	sumiraj ()
ZemljaniTermometar	Anemometar
zt_ident	an_ident
temperatura	vrzinaVetra
uzmi ()	pravacVetra
testiraj ()	uzmi ()
	testiraj ()
Barometar	
	bar_ident
	pritisak
	visina
	uzmi ()
	testiraj ()

Slika 9.1 Deo utvrđenih klasa saistema za merenje meteroloških podataka [Izvor: Autor]

**2. Zadatak** (10 minuta): Razvijte projektno rešenje stanice za prikupljanje meteoreoloških podataka, koje će pokazati interakciju između podsistema i instrumenata koji prikupljaju meteorološke podatke. Upotrebite sekvenčnalni dijagram da prikažete tu interakciju.

## METODA PROJEKTOVANJA: ZADACI 3-5 ZA INDIVIDUALNI RAD

*Ovi zadaci očekuju da student uradi projektovanje traženog softverskog sistema.*

**3. Zadatak** (10 minuta): Utvrđite moguće objekte sledećih sistema, i za njih razvijte objektno-orientisano projektno rešenje. Možete sami usvojiti predpostavke potrebne za ovo projektovanje:

- a) Sistem za upravljanje vremenom i dnevnicima zaposlenih u jednoj firmi. Sistem ima za cilj da omogućava zakazivanje sastanaka grupe zaposlenih. Kada je potrebno organizovati sastanak određene grupe zaposlenih, sistem analizira njihove elektronske rokovnike, i nalazi vremenski slot u kome su svi oni slobodni i zakazuje sastanak, obaveštavajući sve članove grupe. Ukoliko ne može da nađe slobodan vremenski slot, sistem pregovara sa korisnicima tako da promenama u svojim rokovnicima, omoguće zajednički vremenski slot za sastanak.
- b) Stanica za prodaju goriva za automobile projektovana je za automatski rad, bez radnika za sipanje goriva. Vozač treba da ubaci svoju kreditnu karticu u čitač kartica, koji je povezan sa pumpom. Kartica se verifikuje komunikacijom sa kompjuterom izdavaoca kartice, određuje se gornja granica goriva koje se može uzeti. Vozač onda može da sipa gorivo. Kada završi sipanje i vrati crevo na svoje mesto, sistem zadužuje kreditnu karticu za vrednost sisanog goriva. Kreditna kartica se vraća vozaču, od strane čitača. Ako kreditna kartica nije ispravna ili važeća, pumpa je vraća vozaču pre nego što bi počelo punjenje (nema punjenja goriva).

**4. Zadatak** (10 minuta): Uradite sekvenčnalni dijagram za zadatak 3.a.

**5. Zadatak** (5 minuta): Uradite dijagram stanja sa zadatkom 3.a

## PRIMENA ŠABLONA BUILDER: ZADACI OD 1 DO 4 ZA INDIVIDUALNI RAD

*Zadaci za samostalni rad studenta.*

**Zadatak 1** (15 minuta)

Nacrtati klasni dijagram sistema koji vodi evidenciju o predispitnim obavezama studenta za neki predmet (testovi, zalaganje, projekat, domaći zadaci, kolokvijum) i na dijagramu primeniti **šablon Builder** kod kreiranja izveštaja o ukupnim predispitnim obavezama za predmet po studentu.

**Zadatak 2** (10 minuta)

Na dijagramu iz zadatka 1 dodati interfejs za **šablon Builder** i interfejs za predispitnu obavezu, kao i nadklasu za predispitnu obavezu koja treba da ima naziv obaveze, broj poena i datum kada je izvršena. Svaka predispitna obaveza treba da ima konstantu koja predstavlja max broj poena po obavezi.

**Zadatak 3** (10 minuta)

Na dijagramu iz prethodnog zadatka **kreirati konkretnе klase predispitnih obaveza**. Klasa Zadatak, klasa Projekat (koji ima dodatne podatke, poeni za funkcionalnost, poeni za dokumentaciju, poeni za testove, pri čemu ukupan broj poena ne sme da pređe max broj poena za obavezu). Demonstrirati izmenjen rad Buidler šablona projektovanja sa svim informacijama šta je student uradio i koliko ukupnih poena ima.

#### Zadatak 4 (10 minuta)

Iz klasnog dijagrama kreiranog u prethodnim zadacima generisati Java klase automatski korišćenjem PowerDesigner-a.

## PRIMENA ŠABLONA ABSTRACT FACTORY, SINGLTON I BUILDER: ZADACI OD 5 DO 7 ZA INDIVIDUALNI RAD

*Zadaci su definisani za samostalni rad studenta.*

#### Zadatak 5 (15 minuta)

Implementirati **Abstract Factotory** (Apstraktnu fabriku) koja menja dugmiće na ekranu promenom fabrike. Napraviti fabriku velikih dugmića i malih dugmića. I promenom fabrike iz menija menjaju se i dugmići.

#### Zadatak 6 (15 minuta)

Na primeru rada sa bazom podataka koja treba da obezbedi osnovne CRUD operacije proizvoda koji karakterišu naziv, cena i opis, primeniti **Singleton šablon**. Kreirati klasni model i generisati kod u Javi.

#### Zadatak 7 (15 minuta)

Osmisliti i kreirati klasni model centralizovanog sistema za upravljanje radom banke. Klasni model uskladiti da ima primer korišćenja **Singleton i Builder šablonu**.

## ▼ Zaključak

### REZIME LEKCIJE

#### *Poruke izloženog gradiva*

1. Projektovanje softvera i njegova implementacija su dve spregnute aktivnosti. Nivo detalja koje projektno rešenje softvera treba da sadrži zavisi od tipa softvera koji se razvija i od toga da li se koristi razvoj vođen planom ili se koristi agilni pristup u projektovanju.
2. Proces projektovanje objektno-orientisanog softverskog sistema uključuje aktivnosti projektovanja arhitekture sistema, utvrđivanja osnovnih objekata sistema, opisivanje projektnog rešenja upotrebom različitih UML objektnih modela. I dokumentovanjem interfejsa komponenata.
3. Broj i vrste UML modela koji se radi u toku projektovanja može da bude različit. Koriste se statički modeli (modeli klasa, modeli generalizacije, modeli asocijacije) i dinamički modeli (seqvencijalni modeli, modeli stanja).
4. Interfejsi komponenata moraju se precizno definisati tako da ih drugi objekti mogu da koriste. U tu svrhu se koristi UML stereotip za interfejse.
5. Pri razvoju softvera, uvek gledajte da u što većoj meri koristite postojeći softver, bilo u vidu komponenti, u vidu servisa, ili u vidu celih sistema.
6. Upravljanje konfiguracijom je proces upravljanja promenama softverskog sistema koji je u razvoju. Vrlo je bitno da razvojni tim međusobno sarađuje u toku razvoja softvera.
7. Najčešće se softver razvija na razvojnoj platformi, a realizuje na izvršnoj platformi kod kupca. Na razvojnoj platformi se koristi sistem za razvoj softvera (IDE), a razvijeni izvršni kod softvera se onda prebacuje na ciljnu mašinu, tj. računar na kome će raditi u toku njegove eksploracije (tzv. izvršna platforma).
8. Razvoj softvera sa otvorenim (izvornim) kodom je razvoj softvera čiji je izvorni kod javno dostupan. To znači da mnogi mogu da predlože promene i poboljšanja softvera.
9. Upoznavanje sa svrhom šabloni projektovanja i njihovom kategorizacijom.
10. Ovladavanje primenom šablon kreiranja:
  - 10.1. Singleton
  - 10.2. Builder
  - 10.3. Abstract Factory

### LITERATURA

#### *Preporučena literatura*

##### **1. Obevezna literature:**

1. Predavanja, vežbanja i dodatni materijali objavljeni na sistemu za e-učenje, 2014
2. Ian Sommerville, Software Engineering, Tenth Edition, Pearson Education Inc., 2016.

## 2. Dopunska literatura:

1. B. Bruegge, A. Dutoit, Object-Oriented Software Engineering – Using OML, Patterns, and Java, Thirth Edition, Prentice Hall, 2010
2. O'Reilly, Head First Design Patterns
3. Partha Kuchana, Software Architecture Design patterns in Java
4. Design Patterns - Elements of Reusable Object-Oriented Software, Eric Gamma, Richard Helm, Ralph Johnson, Jogns Viisides, 19
5. Design Patterns in Java Tutorial, tutorialspoint.com

**3. Veb lokacije :** Na ovim veb lokacijama možete naći opise mnogih šablon projekovanja sa primerima, te se prepućuje da ih proučite

- <http://www.netobjectives.com/resources/books/design-patterns-explained>
- <https://www.oodesign.com/>



## SE201 - UVOD U SOFTVERSKO INŽENJERSTVO

Projektni šablone struktura

Lekcija 09

PRIRUČNIK ZA STUDENTE

# SE201 - UVOD U SOFTVERSKO INŽENJERSTVO

## Lekcija 09

### ***PROJEKTNI ŠABLONE STRUKTURA***

- ▼ Projektni šablone struktura
- ▼ Poglavlje 1: Šablon Adapter
- ▼ Poglavlje 2: Šablon Bridge
- ▼ Poglavlje 3: Šablon Decorator
- ▼ Poglavlje 4: Šablon Façade
- ▼ Poglavlje 5: Šablon Composite
- ▼ Poglavlje 6: Pokazna vežba – Šabloni strukture
- ▼ Poglavlje 7: Individualna vežba - Zadaci za samostalni rad
- ▼ Zaključak

Copyright © 2017 – UNIVERZITET METROPOLITAN, Beograd. Sva prava zadržana. Bez prethodne pismene dozvole od strane Univerziteta METROPOLITAN zabranjena je reprodukcija, transfer, distribucija ili memorisanje nekog dela ili čitavih sadržaja ovog dokumenta., kopiranjem, snimanjem, elektronskim putem, skeniranjem ili na bilo koji drugi način.

Copyright © 2017 BELGRADE METROPOLITAN UNIVERSITY. All rights reserved. No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise, without the prior written permission of Belgrade Metropolitan University.

## ❖ Uvod

# UVOD

### *Cilj lekcije*

Cilj ove lekcije da vas upozna sa osnovama projektovanja objektno-orientisanog softvera primenom UML i da ukaže na važne aspekte implementacije dobijenog projektnog rešenja . Ova lekcija vam omogućava da:

- razumete najvažnije aktivnosti nekog uopštenog objektno-orientisanog procesa projektovanja;
- razumete različite modele koji se koriste za dokumentovanje nekog objektno-orientisanog projektnog rešenja;
- znate ideju projektnih šablonai kako se one koriste za višestruku upotrebu znanja i iskustva u projektovanju;

Da bi projektovanje softvera bilo brže, uspešnije i kvitetni, ono se najčešće oslanja na prethodna iskustva. Ta iskustva se stalno prikupljaju i predstavljaju u vidu tzv. šabloni za projektovanje (engl. design patterns).U ovom poglavlju biće izloženi sledeći šabloni šabloni strukture:

- Adapter,
- Bridge,
- Decorator,
- Facade i
- Composite,

Kako svaki šablon ima naziv koji je originalno dat na engleskom jeziku, i kao takav opštepoznat u svetu, to se ovde ne daje prevod ovih originalnih naziva, te se u tekstu koriste originalni nazivi na engleskom. To isto važi i za sve klase i interfejse koji se pominju pri objašnjavanju šabloni za projektovanje.

NAPOMENA:

Zbog obimnosti s jedne strane, i zbog održavana jedinstvenosti nastavne jedinice, s druge strane, ova vrlo obimna lekcija se predaje tri nedelje, te na tradicionalnoj nastavi ima 3 x (3+1+3) časova, tj. ukupno 9 časova predavanja, 3 časa pokaznih vežbi i 9 časova individualnih vežbi.. **Ova predavanja i vežbe se izvode 7., 8. i 9. nedelji u semestru.**

## UVODNI VIDEO

*Trajanje video snimka: 1min 36sek*

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

# ✓ Poglavlje 1

## Šablon Adapter

### ŠABLON STRUKTURE ADAPTER

*Šablon Adapter konvertuje interfejs klase u drugi interfejs u skladu sa očekivanjem klijenta*

#### Svrha šablonu:

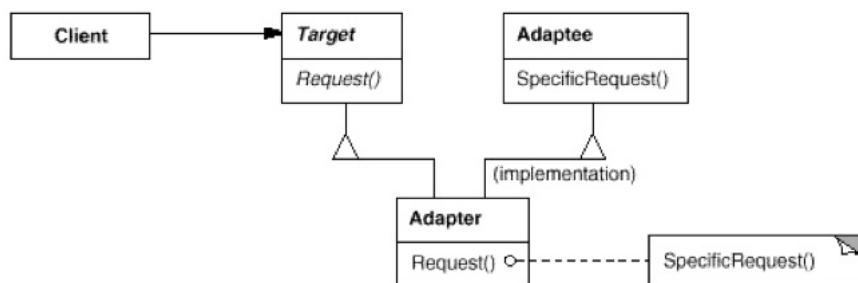
Konverte interfejs klase u drugi interfejs u skladu sa očekivanjem klijenta. Adapter omogućava da klase rade zajedno, što ne bi mogli da rade zbog nekompatibilnih interfejsa. Povezuje dva nekompatibilna interfejsa

#### Primenljivost:

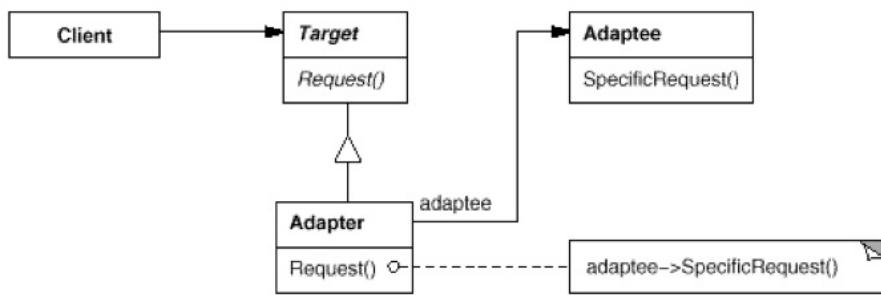
Primena Adapter šablonu kada želita da:

- upotrebite postojeću klasu a njen interfejs ne odgovara vašoj potrebi
- kreirate višestruko upotrebljivu klasu koja sarađuje sa klasom koja nema kompatibilan interfejs
- upotrebite nekoliko postojećih podklasa, a nije praktično da prilagodite njihove interfejsne praviljenjem podklase za svaku od njih. Adapter objekat prilagođava interfejs svoje klase-roditelja (adapter objekta)

#### Struktura:



Slika 1.1 Struktura šablonu Adapter klase [2.4]



Slika 1.2 Struktura šablona Adapter objekta [2.4]

## ŠABLON ADAPTER: PRIMER PRIMENE

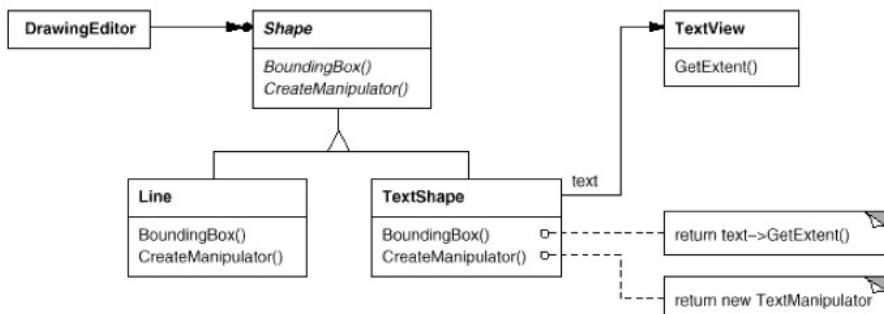
*Često Adapter obezbeđuje funkcionalnost koju adaptirana klasa nema (pre primene Adaptera).*

### Primer upotrebe:

Često Adapter obezbeđuje funkcionalnost koju adaptirana klasa nema. Primer: **createManipulator()** - pomera objekte, pa sada i tekstove.

**Manipulator** je apstraktna klasa za objekte koji znaju kako da animiraju **Shape** kao odgovor na ulaz korisnika (npr. pomeranje oblika na novu lokaciju). Koristiti se podklasa **Manipulator** klase za različite oblike, npr. **TextManipulator** za **TextShape** objekte

Sa **TextManipulator** objektom, **TextShape** dodaje funkcionalnost koju **TextView** nema a koju **Shape** zahteva.



Slika 1.3 Primer primene šablona Adapter [2.4]

### Učesnici:

1. Ciljna klasa – **Target** (pr. Shape); Definiše interfejs za specifični domen koji klijent koristi
2. Klijent – **Client** (pr. DrawingEditor): Sarađuje sa objektima koji zadovoljavaju Target interfejs
3. Klasa koja se prilagođava – **Adaptee** (pr. TextView): Prilagođava interfejs Adaptee Target interfejsu

### Kolboracija:

1. Klijenti pozivaju operacije **Adapter** objekta
2. **Adapter** reaguje i poziva operacije pobjekta **Ataptee** koje ispunjavaju zahtev

## ŠABLON ADAPTER: POSLEDICE

*Prilagođava Adaptee klasu Target klasi koristeći posebnu Adapter klasu. Dopušta da sa Adapter prekrije posnašanje Adaptee klase jer je Adapter podklasa klase Adaptee*

### Posledice:

#### Adapter klase:

- Prilagođava Adaptee klasi Target koristeći posebnu Adapter klasu.
- Dopušta da Adapter prekrije posnašanje Adaptee klase jer je Adapter podklasa klase Adaptee.
- Uvodi samo jedan objekat, i dodatni pokazivač je neophodan da bi se došlo do Adaptee objekta.

#### Adapter objekta:

- Dopušta da Adapter radi sa više objekata klase Adaptee. Adapter može takođe da doda funkcionalnost za sve Adaptee objekte
- Otežava prekrivanje posnašanje klase Adaptee. Zahteva se korišćenje podklasa klase Adaptee nad kojom Adapter deluje (a ne samu klasu Adaptee).

### Stvari na koje treba obratiti pažnju:

1. *Koju dozu prilagidavanja može da obezbedi Adapter? To zavisi od sličnosti Target interfejsa sa interfejsom Adaptee.*
2. *Ubačeni adapteri.* Prilagodavanje interfejsa omogućava rad naših klasa sa postojećim sistemima koje očekuju interfejse iste klase. Ovo su klase sa ugrađenim prilagodljivim interfejsima

## ŠABLON ADAPTER: IMPLEMENTACIJA

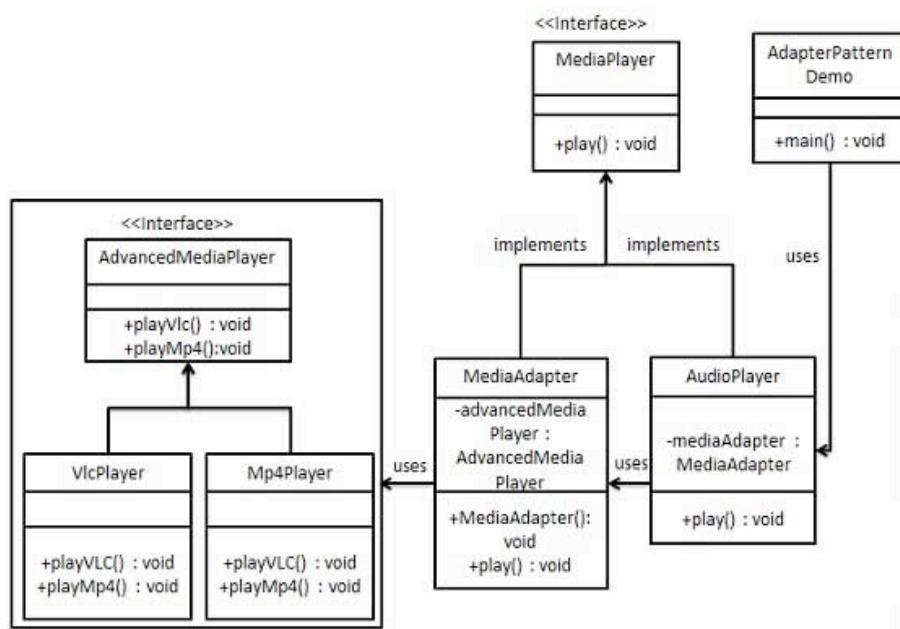
*Cilj je napraviti da AudioPayer koristi i druge formate. Kreira se adapter klasa MediaAdapter koja primenjuje MediaPlayer interfejs i koristi objekte A*

### Primer:

- Klasa **AudioPayer** primenjuje interfejs **MediaPlayer**
- **AudioPayer** koristi mp3 format audio fajlova
- Klasa **AdvancedMediaPlayer** koristi **AdvancedMediaPlayer** interfejs.

- Cilj: napraviti da **AudioPayer** koristi i druge formate
- Kreira se adapter klasa **MediaAdapter** koja primenjuje **MediaPlayer** interfejs i koristi objekte **AdvancedMediaPlayer** klase da bi koristila različite formate.
- **AudioPayer** koristi adapter klasu **MediaAdapter** prebacujući joj audio tip bez znanja stvarne klase koja može da koristi dati format.
- **AdapterPaternDemo** daje demo klasu koja koristi **MediaPlayer** klasu koja obezbeđuje korišenje različitih formata

Sledeća slika prikazuje dijagram klasa ovog primera:



Slika 1.4 Dijagram klasa primea upotrebe šablona Adapter [2.4]

## ŠABLON ADAPTER: POSTUPAK PRIMENE (1. I 2. KORAK)

*Kreiranje interfejsa za klase **MediaPlayer** i **AdvancedMediaPlayer** i reiranje konkretnih klasa koje primenjuju **AdvancedMediaPlayer** interfejs*

**Korak 1:** Kreiranje interfejsa za klase **MediaPlayer** i **AdvancedMediaPlayer**

```

public interface MediaPlayer {
    public void play(String audioType, String fileName);
}

public interface AdvancedMediaPlayer {
    public void playVlc(String fileName);
    public void playMp4(String fileName);
}
  
```

**Korak 2:** Kreiranje konkretnih klasa koje primenjuju **AdvancedMediaPlayer** interfejs

```
public class VlcPlayer implements AdvancedMediaPlayer{
    @Override
    public void playVlc(String fileName) {
        System.out.println("Playing vlc file. Name: " + fileName);
    }

    @Override
    public void playMp4(String fileName) {
        //do nothing
    }
}

public class Mp4Player implements AdvancedMediaPlayer{

    @Override
    public void playVlc(String fileName) {
        //do nothing
    }

    @Override
    public void playMp4(String fileName) {
        System.out.println("Playing mp4 file. Name: " + fileName);
    }
}
```

## ŠABLON ADAPTER: POSTUPAK PRIMENE (3. I 4. KORAK)

*Kreiranje Adapter klase koja primenjuje MediaPlayer interfejs i kreiranje konkretne klase koja primenjuje MediaPayer interfejs*

**Korak 3:** Kreiranje Adapter klase koja primenjuje MediaPlayer interfejs

```
ublic class MediaAdapter implements MediaPlayer {

    AdvancedMediaPlayer advancedMusicPlayer;

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

    public void play(String fileName) {
        advancedMusicPlayer.play(fileName);
    }
}
```

```
    advancedMusicPlayer = new Mp4Player();
}
}

@Override
public void play(String audioType, String fileName) {
    if(audioType.equalsIgnoreCase("vlc")){
        advancedMusicPlayer.playVlc(fileName);
    }else if(audioType.equalsIgnoreCase("mp4")){
        advancedMusicPlayer.playMp4(fileName);
    }
}
```

**Korak 4:** Kreiranje konkretne klase koja primenjuje **MediaPlayer** interfejs

```
public class AudioPlayer implements MediaPlayer {
    MediaAdapter mediaAdapter;

    @Override
    public void play(String audioType, String fileName) {

        //inbuilt support to play mp3 music files
        if(audioType.equalsIgnoreCase("mp3")){
            System.out.println("Playing mp3 file. Name: " + fileName);
        }
        //mediaAdapter is providing support to play other file formats
        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");
        }
    }
}
```

## ŠABLON ADAPTER: POSTUPAK PRIMENE (5. I 6. KORAK)

*Upotreba AudioPayer za rad sa različitim tipovima audio formata i prikaz rezultat*

**Korak 5:** Upotreba AudioPayer za rad sa različitim tipovima audio formata

```
public class AdapterPatternDemo {  
    public static void main(String[] args) {  
        AudioPlayer audioPlayer = new AudioPlayer();  
  
        audioPlayer.play("mp3", "beyond the horizon.mp3");  
        audioPlayer.play("mp4", "alone.mp4");  
        audioPlayer.play("vlc", "far far away.vlc");  
        audioPlayer.play("avi", "mind me.avi");  
    }  
}
```

**Korak 6:** Prikaz rezultata

```
Playing mp3 file. Name: beyond the horizon.mp3  
Playing mp4 file. Name: alone.mp4  
Playing vlc file. Name: far far away.vlc  
Invalid media. avi format not supported
```

Slika 1.5 Prikaz dobijenog rezultata

## ADAPTER DESIGN PATTERN (VIDEO)

*Trajanje:12, 31 minuta*

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

## VIDEO PREDAVANJE ZA OBJEKAT "ŠABLON ADAPTER"

*Trajanje video snimka: 18min 48sek*

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

## ▼ Poglavlje 2

# Šablon Bridge

## ŠABLON STRUKTURE BRIDGE

*Svrha šablonu: Odvajanje apstrakcije od svoje implementacije tako da se obe klase mogu nezavisno menjati*

### **Primena:**

Bridge se koristi kada treba da odvojimo apstrakciju od njene implementacije da se mogu nezavisno menjati. Odvaja klasu implementacije od apstraktne klase obezbeđujući most između njih, tj. strukturu za njihovo povezivanje. Šablon koristi interfejs koji ima funkciju mosta koji omogućava funkcionalnost konkretnе klase nezavisno od interfejsa klase za implementaciju. Obe vrste klase mogu se strukturno menjati bez međusobnog uticaja.

### **Svrha šablonu:**

Odvajanje apstrakcije o svoje implementacije tako da se obe klase mogu nezavisno menjati

### **Motivacija /razlog nastanka šablonu):**

Kada apstrakcija ima nekoliko mogućih implementacija, onda se primenjuje nasleđivanje.. Apstraktna klasa definiše interfejs za apstrakciju, a konkretnе podklase primenjuju ga na različite načine. Međutim, ovaj pristup ne obezbeđuje dovoljnu fleksibilnost. Nasleđivanje vezuje za stalno implementaciju za apstrakciju, te ju je teško menjati, proširiti i ponovo upotrebiti nezavisno od apstrakcije i implementacije

### **Primenljivost:**

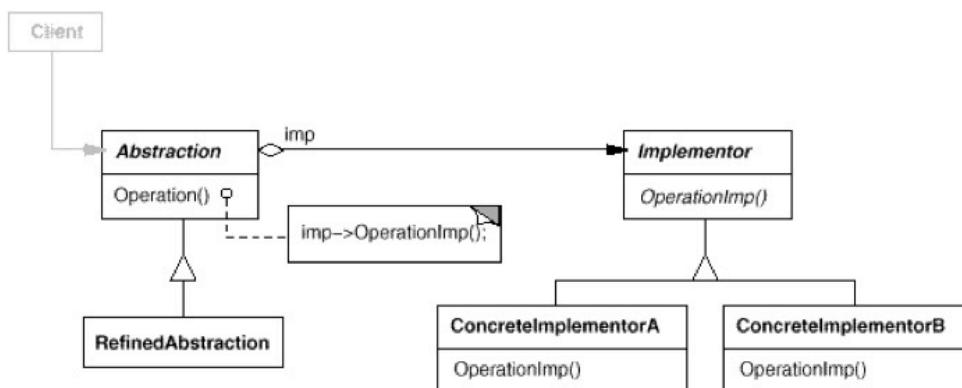
Upotrebite Bridge šablon kada:

- Želite da izbegnete stalnu vezu između apstraktne klase i njene implementacije
- Obe apstrakcije i njihove implementacije se šire sa podklasama. Bridge šablon kombinuje različite apstrakcije i implementacije i nezavisno ih proširuje
- Promene u implementaciji neke apstrakcije ne bi trebalo da ima nikakav uticaj na klijente, tj. klijenti ne moraju da vrše svoju kompilaciju
- Kada treba da podelite objekt na dva dela.
- Kada želite da delite implementaciju sa više objekata, a ti treba da bude skriveno od klijenata.

# ŠABLON BRIDGE: OPIS

*Abstraction objekat prosleđuje zahtev klijenta svom Implementator objektu*

## Struktura:



Slika 2.1 Struktura šablonu Bridge [2.4]

## Učesnici:

- Abstraction:** Definiše interfejs apstrakcije. Održava referencu ka tipu objekta implementatora
- RefinedAbstraction:** Proširuje interfejs klase Abstraction
- Implementor:** Definiše interfejs za implementacione klase
- ConcreteImplementor:** Primjenjuje Implementor interfejs i definiše njegovu konkretnu implementaciju

## Kolaboracija:

- Abstraction prosleđuje zahtev klijenta svom Implementator objektu

## Posledice:

- Odvajanje interfejsa i implementacije:* Implementacije nije permanentno vezana za interfejs. Implementacija se može raditi u vreme izvršenja. Nezavisna je promena Abstraction i Implementor klasa. Podsticanje primene slojevite arhitekture
- Poboljšano proširenje* Nezavisnost proširenja Abstraction i Implementor klasa
- Skrivanje implementacionih detalja od klijenata* Izolacija klijenta od detalja implementacije

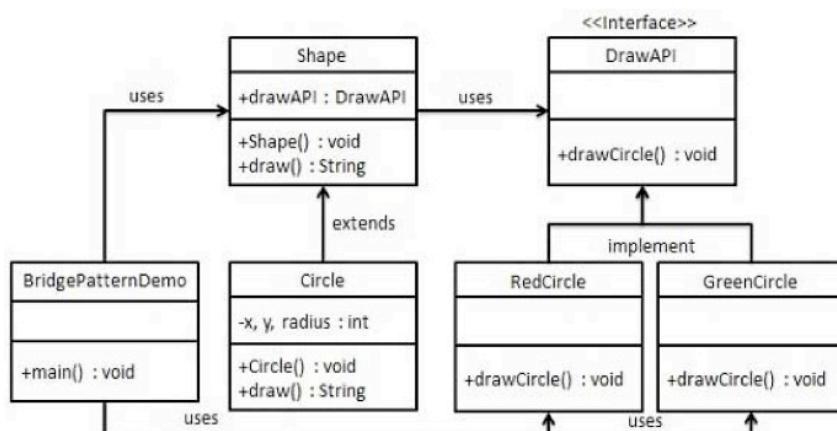
## ŠABLON BRIDGE: PRIMER I 1. KORAK PRIMENE

*Krug (Circle object) se crta u različitim bojama upotrebom istog metoda apstraktne klase (Shape), ali koristeći različite mostove ka klasama implementacije*

### Primer:

1. Krug se crta u različitim bojama upotrebom istog metoda apstraktne klase, ali koristeći različite mostove sa klasama implementacije
2. Interfejs **DrawAPI** je most povezivanja konkretne klase i implementacione klase.
3. Klase **RedCicrcle** i **GreenCircle** primenjuju **DrawAPI** interfejs.
4. **Shape** je apstraktna klasa koja koristi objekat **DrawAPI**
5. **BridgePatternDemo** je demo klasa koja upotrebljava Shape klasu za crtanje krugova različitih boja.

Sledeća slika prikazuje dojagram klasa ovog primera:



Slika 2.2 Primer implementacije šablonu Bridge [2.4]

### Korak 1: Kreiranje Bridge interfejsa implementatora

```
public interface DrawAPI {
    public void drawCircle(int radius, int x, int y);
}
```

## ŠABLON BRIDGE: POSTUPAK PRIMENE (2. I 3. KORAK)

*Kreiranje konkretnе implementacione klase koja primenjuje DrawAPI interfejs i kreiranje apstraktne klase Shape koja upotrebljava DrawAPI interf*

**Korak 2:** Kreiranje konkretnе implementacione klase koja primenjuje **DrawAPI** interfejs

```
public class RedCircle implements DrawAPI {  
    @Override  
    public void drawCircle(int radius, int x, int y) {  
        System.out.println("Drawing Circle[ color: red, radius: "  
            + radius +", x: " +x+", "+ y +"]");  
    }  
}  
  
public class GreenCircle implements DrawAPI {  
    @Override  
    public void drawCircle(int radius, int x, int y) {  
        System.out.println("Drawing Circle[ color: green, radius: "  
            + radius +", x: " +x+", "+ y +"]");  
    }  
}
```

**Korak 3:** Kreiranje apstraktne klase **Shape** koja upotrebljava **DrawAPI** interfejs

```
public abstract class Shape {  
    protected DrawAPI drawAPI;  
    protected Shape(DrawAPI drawAPI){  
        this.drawAPI = drawAPI;  
    }  
    public abstract void draw();  
}
```

## ŠABLON BRIDGE: POSTUPAK PRIMENE (4. I 5. KORAK)

*Kreiranje konkretnе klase koja primenjuje interfejs Shape i upotreba klase Shape i DrawAPI za crtanje obojenih krugova*

**Korak 4:** Kreiranje konkretnе klase koja primenjuje interfejs **Shape**

```
public class Circle extends Shape {  
    private int x, y, radius;  
  
    public Circle(int x, int y, int radius, DrawAPI drawAPI) {  
        super(drawAPI);  
        this.x = x;  
        this.y = y;  
        this.radius = radius;  
    }  
  
    public void draw() {  
        drawAPI.drawCircle(radius,x,y);  
    }  
}
```

**Korak 5:** Upotreba klase **Shape** i **DrawAPI** za crtanje obojenih krugova

```
public class BridgePatternDemo {  
    public static void main(String[] args) {  
        Shape redCircle = new Circle(100,100, 10, new RedCircle());  
        Shape greenCircle = new Circle(100,100, 10, new GreenCircle());  
  
        redCircle.draw();  
        greenCircle.draw();  
    }  
}
```



**Korak 6:** Prikaz rezultata

```
Drawing Circle[ color: red, radius: 10, x: 100, 100]
Drawing Circle[ color: green, radius: 10, x: 100,
100]
```

Slika 2.3 Dobijen rezultat

## BRIDGE DESIGN PATTERN (VIDEO)

*Trajanje: 14,51 minuta*

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

## VIDEO PREDAVANJE ZA OBJEKAT "ŠABLON BRIDGE I ŠABLON DECORATOR"

*Trajanje video snimka: 31min 24sek*

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

## ✓ Poglavlje 3

# Šablon Decorator

## ŠABLON STRUKTURE: DECORATOR

*Decorator šablon dozvoljava dodavanje nove funkcionalnosti postojećeg objekta bez menjanja njegove strukture*

### Primena:

Decorator šablon dozvoljava dodavanje nove funkcionalnosti postojećeg objekta bez menjanja njegove strukture. Ovaj šablon kreira Decorator klasu omotač originalne klase i obezbeđuje dodatnu funkcionalnost bez promena potpisa metoda klase

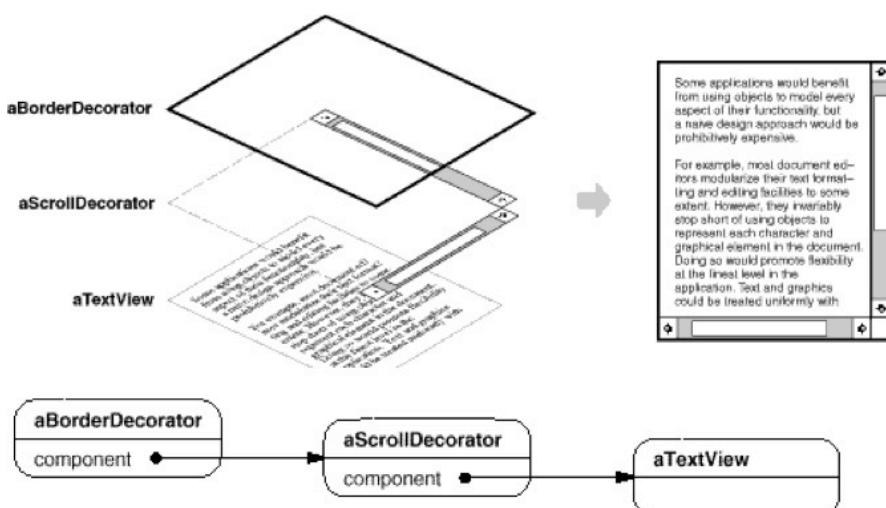
### Svrha šablon-a:

Donosi dinamičku dodatnu odgovornost nekom objektu.. Decorator objekti obezbeđuju fleksibilnu alternativu korišćenju podklasa za proširenje funkcionalnosti.

### Motivacija (razlog nastanka šablon-a):

Nekada želimo da dodamo funkcionalnost samo nekim objektima, a ne celoj klasi (npr. GUI). Jedan način za ovo je primena nasleđivanja. To je nefleksibilno i statički, jer svi objekti jedne klase to dobijaju. Drugi fleksibilniji način je da se objekat omota drugom frugim (Decorator) koja dodaje potrebnu funkcionalnost .

### Primer primene:

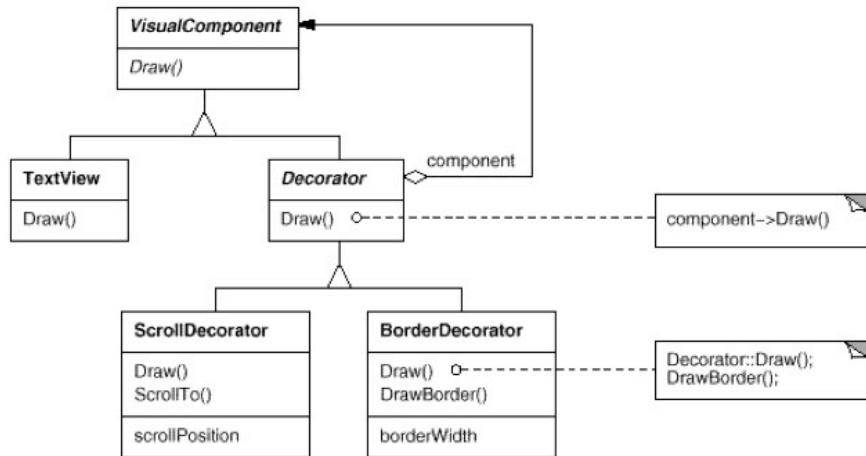


Slika 3.1 Primer u kome se može primeniti adapter Decorator [2.4]

## ŠABLON DECORATOR: OPIS

*Šablon Decorator se upotrebljava za dodavanje odgovornosti individualnim objektima dinamički i transparentno, bez uticaja na sam objekat.*

Na slici 2 dat je dijagram klasa razmatranog primera:

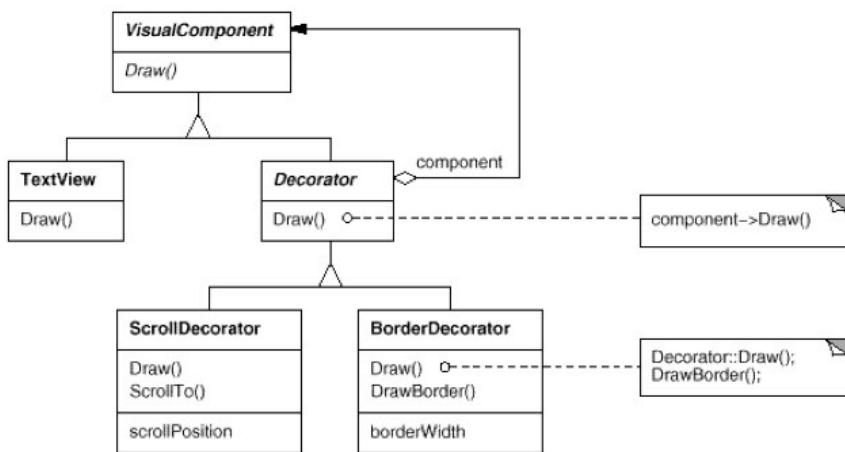


Slika 3.2 Dijagram klasa primera [2.4]

**Primenljivost:** Upotrebite Decorator

- da bi dodali odgovornosti individualnim objektima dinamički i transparentno, bez uticaja na sam objekat,
- za odgovornosti koje se mogu kasnije ukloniti,
- Kada nije praktično primeniti proširenje podklasama.

**Struktura:**



Slika 3.3 Dijagram strukture pri primeni šablona Decorator [2.4]

**Učesnici:**

1. **Component:** Definiše interfejs za objekte kojim se dinamički dodaje dodatna funkcionalnost
2. **ConcreteComponent:** Definiše objekat sa dodatnom odgovornošću
3. **Decorator:** Održava referencu na Component objekat i definiše interfejs koji je usaglašen sa interfejsom Component objekta
4. **ConcreteDecorator:** Dodaje odgovornosti komponenti

## ŠABLON DECORATOR: KOLABORACIJE I POSLEDICE

*Šablon Decortor obezbeđuje veću fleksibilnost nego staticko nasleđivanje, ne zavisi od identiteta objekta i radi sa malo malih objekata koji se razlikuju sa*

### Kolaboracije:

1. Decorator prosleđuje zahteve Component objektima.
2. Opciono, može da izvrši dodatne operacije pre i posle slanja zahteva.

### Posledice:

Decorator šablon ima više prednosti:

1. Veća fleksibilnost nego staticko nasleđivanje. Mogu se dodavati i uklanjati u vreme izvršenja. Svojstva se mogu i dva puta dodavati. Inkrementalno dodavanje odgovornosti primenom Decorator objekata u fazi izvršenja (trošak kada je potrebno)
2. Decorator i njegova komponenta nisu identični, te ne zavise od identiteta objekta.
3. Mnogo malih objekata. Razlikuju se samo sa malim inkrementalnim dodacima.

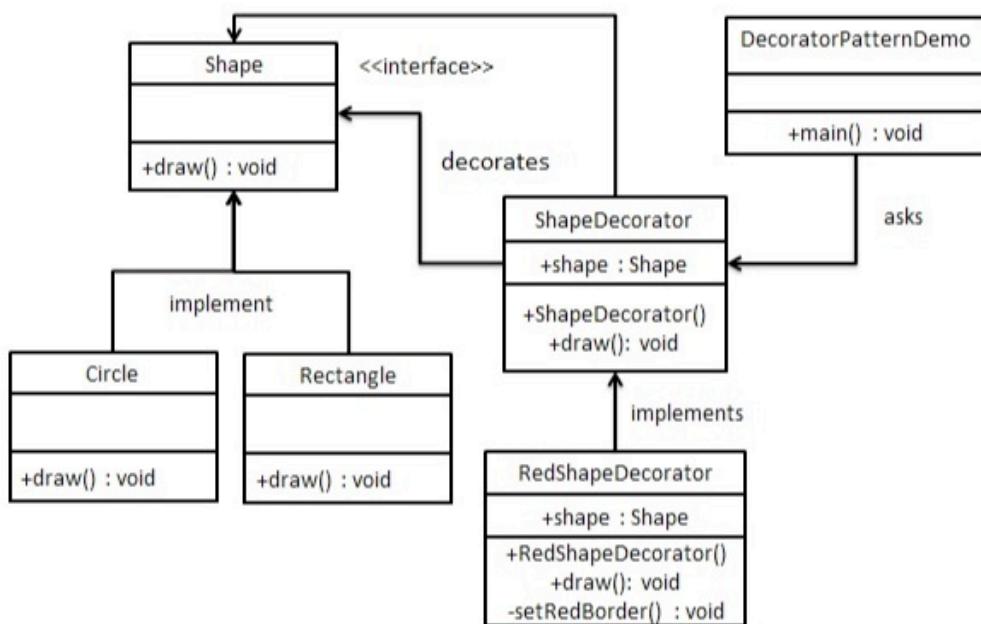
## ŠABLON DECORATOR: PRIMER

*Kreirati apstraktni Decorator klasu ShapeDecorator koja primenjuje Shape interfejs koji primenjuje krug (Circle) ili pravougaonik (Rectangle) klase, a ko*

### Primer za implementaciju:

- Kreirati **Shape** objekat i konkretne klase koje primenjuju Shape interfejs.
- Kreirati apstraktni **Decorator** klasu **ShapeDecorator** koja primenjuje **Shape** interfejs i ima **Shape** objekta kao promenljivu instance.
- **RedShapeDecorator** je konkretna klasa koja primenjuje ShapeDecorator
- **DecoratorPatternDemo** je demo klasa koja koristi **RedShapeDecorator** za dekoraciju Shape objekta.

Dijagram klasa u primeru implementacije: je dat na slici 4 .



Slika 3.4 Dijagram klasa primera implementacije šablonu Decorator [2.4]

## ŠABLON DECORATOR: POSTUPAK PRIMENE (1., 2. I 3. KORAK)

*Kreiranje interfejsa, kreiranje konkretnih klasa za isti interfejs i kreiranje abstractne Decorator klase koja primenjuje Shape interfejs*

### Korak 1: Kreanje interfejsa

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

### Korak 2: Kreiranje konkretnih klasa za isti interfejs

```
public class Rectangle implements Shape {

    @Override
    public void draw() {
        System.out.println("Shape: Rectangle");
    }
}
```

```
}
```

```
public class Circle implements Shape {
```

```
    @Override
```

```
    public void draw() {
```

```
        System.out.println("Shape: Circle");
```

```
    }
```

```
}
```

**Korak 3:** Kreiranje abstractne Decorator klase koja primenjuje Shape interfejs

```
public abstract class ShapeDecorator implements Shape {
```

```
    protected Shape decoratedShape;
```

```
    public ShapeDecorator(Shape decoratedShape){
```

```
        this.decoratedShape = decoratedShape;
```

```
    }
```

```
    public void draw(){
```

```
        decoratedShape.draw();
```

```
    }
```

```
}
```

## ŠABLON DECORATOR: POSTUPAK PRIMENE (4. I 5. KORAK)

*Kreiranje konkretne Decorator klase koja proširuje ShapeDecorator klasu i upotreba RedShapeDecorator radi dekorisanja Shape objekata*

**Korak 4:** Kreiranje konkretne Decorator klase koja proširuje ShapeDecorator klasu

```
public class RedShapeDecorator extends ShapeDecorator {  
  
    public RedShapeDecorator(Shape decoratedShape) {  
        super(decoratedShape);  
    }  
  
    @Override  
    public void draw() {  
        decoratedShape.draw();  
        setRedBorder(decoratedShape);  
    }  
  
    private void setRedBorder(Shape decoratedShape){  
        System.out.println("Border Color: Red");  
    }  
}
```

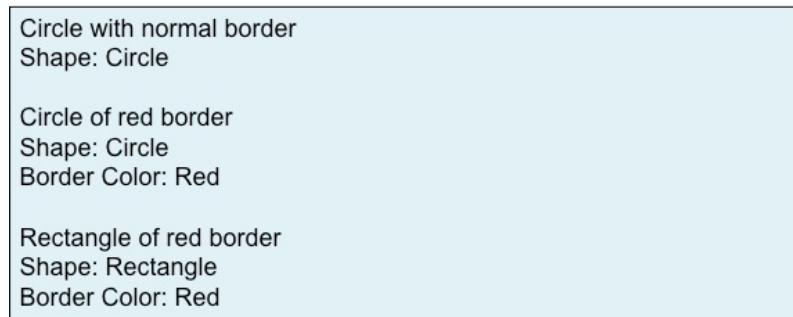
#### Korak 5: Upotreba RedShapeDecorator radi dekorisanja Shape objekata

```
public class AdapterPatternDemo {  
    public static void main(String[] args) {  
        AudioPlayer audioPlayer = new AudioPlayer();  
  
        audioPlayer.play("mp3", "beyond the horizon.mp3");  
        audioPlayer.play("mp4", "alone.mp4");  
        audioPlayer.play("vlc", "far far away.vlc");  
        audioPlayer.play("avi", "mind me.avi");  
    }  
}
```

## ŠABLON DECORATOR: POSTUPAK PRIMENE ( 6. KORAK)

### *Prikaz rezultata*

**Korak 6:** Prikaz rezultata



Slika 3.5 Prikaz dobijenog rezultata

## DECORATOR DESIGN PATTERN (VIDEO)

*Trajanje: 12,56*

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

## ✓ Poglavlje 4

# Šablon Façade

## ŠABLON STRUKTURE FAÇADE

*Šablon Façade sakriva složenost sistema i obezbeđuje interfejs klijentu za njegovu komunikaciju sa sistemom*

### Primena:

Šablon Façade sakriva složenost sistema i obezbeđuje interfejs klijentu za njegovu komunikaciju sa sistemom.. Ovaj šablon koristi jednu klasu koja obezbeđuje uprošćene metode koje koristi klijent, a delegira pozive metodima postojećih klasa sistema.

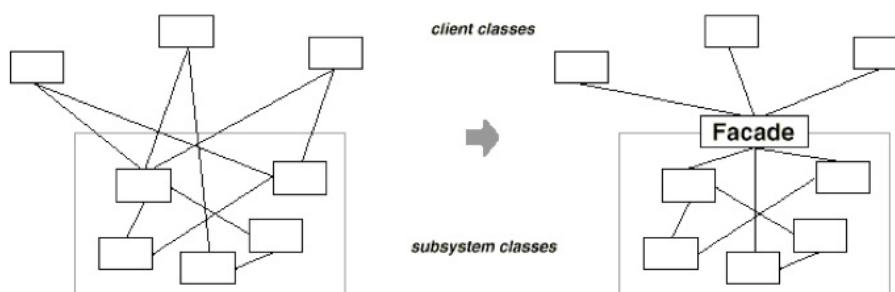
### Svrha šablon-a:

Obezbeđenje ujedinjenog interfejsa ka skupu interfejsa podistema. **Façade** definiše interfejs višeg nivoa koji omogućava lakše korišćenje podistema.

### Motivacija (razlog nastanka šablon-a):

Struktiranjem sistem u podisteme smanjuje kompleksnost sistema. Zajednički cilj projektovanja je da se minimizira komunikacija i zavisnost među podistemima. To se postiže sa **Façade** objektom koji obezbeđuje jedinstve, uprošćen interfejs ostalim delovima podistema.

Na slici 1 prikazan je slučaj kada se koristi šablon **Façade**



Slika 4.1 Skica slučaja kada se koristi šablon Facade [2.4]

### Primenljivost:

Upotrebljavajte šablon **Facade** kada:

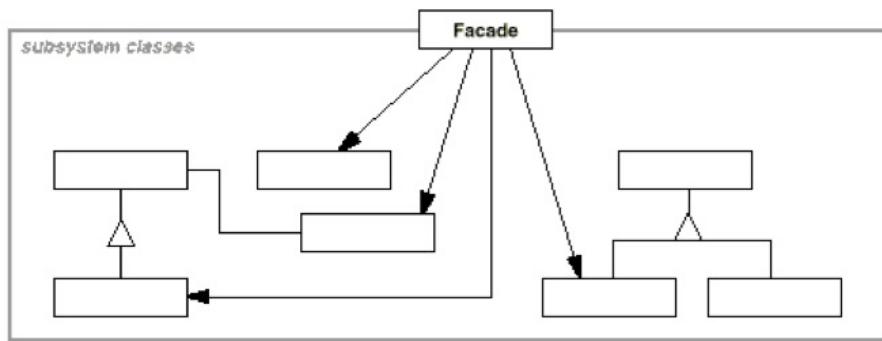
- želite da obezbedite jednostavan interfejs za složeni podistem
- postoje mnoge zavisnosti između klijenata i implementacionih klasa neke abstrakcije. Façade odvaja podistem od klijenata.

- želite da postavite podsisteme u slojeve. Façade definiše ulaznu tačku

## ŠABLON FAÇADE: OPIS

*Klijenti komuniciraju sa podsistomom šaljući zahteve Façade objektu, koji ih prosleđuje odgovarajućim objektim podistema*

### Struktura:



Slika 4.2 Struktura za primenu šablonu Façade [2.4]

### Učesnici:

- Facade:** Zna koja klase podsistema su odgovorne za neki zahtev. Delegira zahteve klijenta odgovarajućim objektima podsistema
- Klase podsistema:** Ostvaruju funkcionalnost podsistema. Realizuju posao dodeljenim Facade objektu. Nemaju znanje o Facade objektu, imaju samo referencu ka njemu

### Kolaboracije:

- Klijenti komuniciraju sa podistemom šaljući zahteve Façade objektu, koji ih prosleđuje odgovarajućim objektim podsistema
- Klijenti koji upotrebljavaju Façade, nemaju direktni pristup objektima njegovih podsistemima

### Posledice:

- Facade izoluje klijente od podsistema, te time smanjuje broj objekata sa kojim klijent treba da radi, te time pojednostavljuje njegovo korišćenje.
- Obezbeđuje slabo povezivanje između podsistema i klijenata. Objekti sistema imaju jake međusobne veze. Slaba povezanost omogućava promenu objekata podsistema bez uticaja na klijente.
- Ne sprečava aplikacije da upotrebljavaju klase podsistema ako to žele.

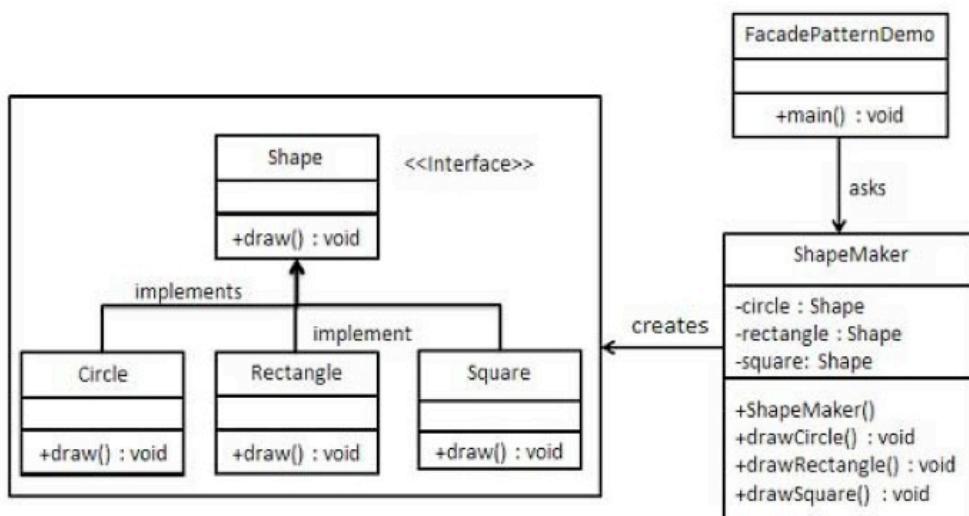
## ŠABLON FAÇADE: PRIMER

*Façade klasa je ShapeMaker klasa koja upotrebljava konkretnе klase (Circle, Rectangle i Square) preko Shape interfejsa.*

### Primer:

- Kreirati **Shape** interfejs i konkretne klase koje primenjuju **Shape** interfejs.
- **Facade** klasa **ShapeMaker** se posebno definiše
- **ShapeMaker** klasa upotrebljava konkretne klase kojima delegira pozive korisnika.
- **FacadePatternDemo** je demo klasa koja upotrebljava ShapeMaker klasu.

### Dijagram klase primera:



Slika 4.3 Dijagram klase primera implementacije adaptera Façade [2.4]

## ŠABLON FAÇADE: POSTUPAK PRIMENE (1., 2. I 3. KORAK)

*Kreiranje interfejsa, kreiranje konkretnih klasa koje primenjuju isti interfejs i Kreiranje Façade klase.*

### Korak 1: Kreiranje interfejsa

```

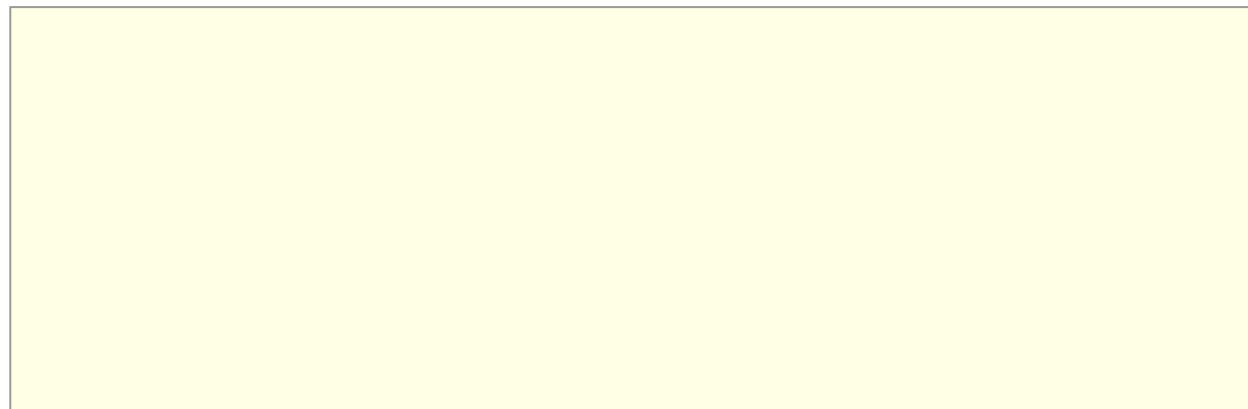
public interface Shape {
    void draw();
}
  
```

### Korak 2: Kreiranje konkretnih klasa koje primenjuju isti interfejs

```
public class Rectangle implements Shape {  
  
    @Override  
    public void draw() {  
        System.out.println("Rectangle::draw()");  
    }  
}  
  
public class Square implements Shape {  
  
    @Override  
    public void draw() {  
        System.out.println("Square::draw()");  
    }  
}  
public class Circle implements Shape {  
  
    @Override  
    public void draw() {  
        System.out.println("Circle::draw()");  
    }  
}
```

### Korak 3: Kreiranje Façade klase

```
public class ShapeMaker {  
    private Shape circle;  
    private Shape rectangle;  
    private Shape square;  
  
    public ShapeMaker() {  
        circle = new Circle();  
        rectangle = new Rectangle();  
        square = new Square();  
    }  
  
    public void drawCircle(){  
        circle.draw();  
    }  
    public void drawRectangle(){  
        rectangle.draw();  
    }  
    public void drawSquare(){  
        square.draw();  
    }  
}
```



## ŠABLON FAÇADE: POSTUPAK PRIMENE (4. I 5. KORAK)

*Upotreba Façade objekta za crtanje različitih oblika i prikaz rezultata*

**Kodak 4:** Upotreba Facade objekta za crtanje različitih oblika

```
public class FacadePatternDemo {  
    public static void main(String[] args) {  
        ShapeMaker shapeMaker = new ShapeMaker();  
  
        shapeMaker.drawCircle();  
        shapeMaker.drawRectangle();  
        shapeMaker.drawSquare();  
    }  
}
```

**Korak 5:** Prikaz rezultata

```
Circle::draw()  
Rectangle::draw()  
Square::draw()
```

Slika 4.4 Dobijen rezultat [2.4]

## FACADE DESIGN PATTERN (VIDEO)

*Trajanje:11,31 minute*

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

## ✓ Poglavlje 5

### Šablon Composite

#### ŠABLON STRUKTURE COMPOSITE

*Composite šablon opisuje kako da se koristi rekurzija u postavljanju objekata, tako da klijenti ne moraju da se bave tom razlikom.*

##### Primena:

Composite šablon se koristi kada treba tretirati grupu objekata na jednostavni način kao jedan objekat. Composite šablon postavlja objekte u strukture stabla radi predstavljanja dela, kao i cele hijerarhije. Šablon kreira klasu koja sadrži grupu sopstvenih objekata. Klasa omogućava promenu grupe istih objekata.

##### Svrha šablon-a:

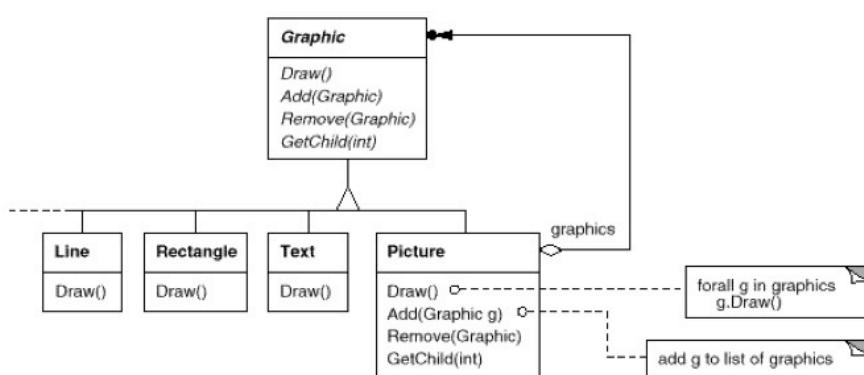
Composite objekti u strukturi stabla predstavljaju delove i celu hijerarhiju objekata strukture. Composite objekat omogućava klijentima da rade sa pojedinačnim objektima i skupovima objekata na jedinstven način.

##### Motivacija (razlog nastamka šablon-a):

Grafičke aplikacije kao što su editori za crtanje i sistemi za crtanje šema, omogućavaju korisnicima da izgrađuju složene dijagrame korišćenjem jednostavnih komponenti. Korisnik mogrubiše komponente gradeći veće komponente, koji postaju delovi još većih komponenti.

Primer: Primena primitiva Text i Lines i dr.

Problem: Program koji koristi ove različite klase mora da tretira primitive i kontejner objekte različito, iako ih korisnik tretira najčešće na isti način.



Slika 5.1 Primer za primenu Composite šablon-a [2.4]

Composite šablon opisuje kako da se koristi rekurzija u postavljanju objekata, tako da klijenti ne moraju da se bave tom razlikom.

Ključno rešenje: korišćenje apstraktne klase (**Graphic**) koja predstavlja i primitive i njihove kontejnere

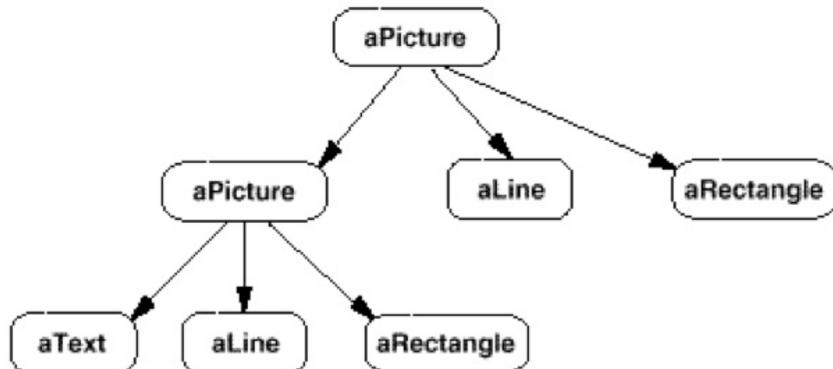
## ŠABLON COMPOSITE: OPIS

*Upotrebite Composite šablon kada želite da predstavite hijerarhiju objekata i omogućite klijentima da ignorišu razliku između složenih objekata i pojedinačnih objekata*

### Primenljivost:

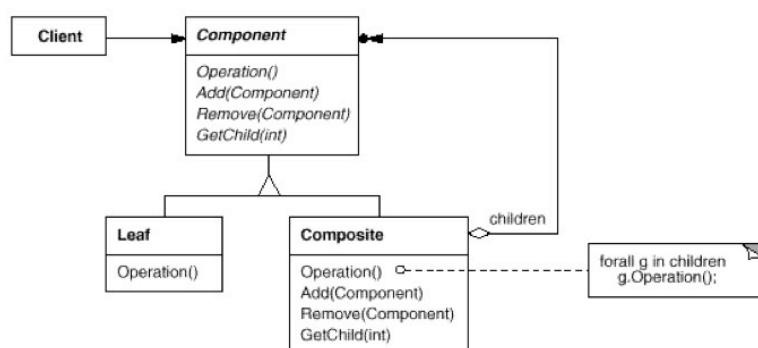
Upotrebite Composite šablon kada želite da:

- predstavite hijerarhiju objekata
- omogućite klijentima da ignorišu razliku među složenih objekata i pojedinačnih objekata. Klijenti će tretirati sve objekte u hijerarhiji na jednak način.

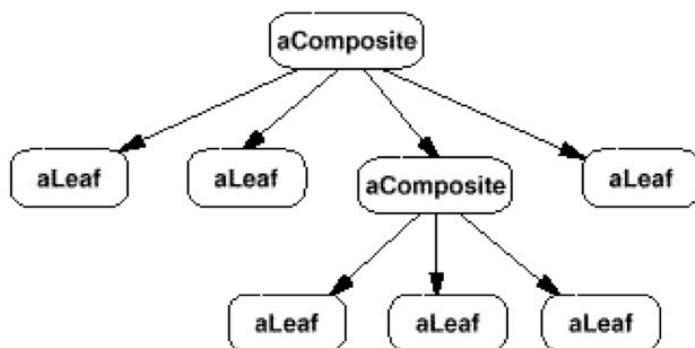


Slika 5.2 Slika kao struktura grafičkih elemenata [2.4]

### Struktura:



Slika 5.3 Struktura šablona Composite [2.4]



Slika 5.4 Tipična Composite struktura objekta [2.4]

## ŠABLON COMPOSITE: UČESNICI I KOLABORACIJE

*Klijenti upotrebljavaju interfejs Component klase za interakciju sa objektima u složenoj strukturi. Klasa Composite definiše ponašanje komponenata...*

### Učesnici:

1. **Client:** Manipuliše sa objektima u strukturi preko Component interfejsa
2. **Component (npr. Graphic):** Deklarira interfejs za objekte u strukturi. Primjenjuje podrazumevajuće ponašanje za interfejs koji je zajednički za sve klase. Deklarira interfejs za pristup i rad sa podkomponentama ("deca"). (Opciono) definiše interfejs za pristup "roditeljima" komponente (nadkomponente) u strukturi rekurzije, i primjenjuje ako je to odgovarajuće
3. **Leaf (npr. Rectangle, Line, Text, itd.):** Predstavlja list-objekte u strukturi. List-objekat nema "decu". Definiše ponašanje primitivnih objekata u strukturi
4. **Composite (npr. Picture):** Definiše ponašanje komponenata sa "decom". Skladišti "decu" komponente. Primjenjuje operacije rada sa "decom" u **Component** interfejsu

### Kolaboracije:

1. Klijenti upotrebljavaju interfejs Component klase za interakciju sa objektima u složenoj strukturi
2. Ako je primalac Leaf, onda se zahtev direkto odraduje.
3. Ako je primalac Composite, onda se najčešće prosleđuje podkomponenta, a ("deci"), uz moguće izvođenje nekih prethodnih operacija pre ili posle prosleđivanja

## ŠABLON COMPOSITE: POSLEDICE

*Šablon Composite definiše hijerarhiju klasa koju čine primitivni i složeni oblici. Primitivni objekti mogu biti delovi složenijih objekata, koji rekurzivno,*

### Posledice:

Šablon Composite:

- **Definiše hijerarhiju klasa** koju čine primitivni oblici i složeni oblici. Primitivni objekti mogu biti delovi složenijih objekata, koji rekurzivno, mogu biti delovi drugih složenih objekata, itd. Program klijenta mora da bude spreman da pored primitivnog objekta, da preuzme i složen objekat;
- **Pojednostavljuje klijenta.** Klijenti na isti način tretiraju složene strukture i pojedinačne (primitivne) objekte. Njihov program je jednostavniji, jer ne moraju da proveravaju da li rade sa primitivnim ili složenim objektom.;
- **Olakšava dodavanja svih vrsta komponenata.** Novodefinisan **Leaf** ili **Composite** podklase automatski rade sa postojećim strukturama i programom klijenta. Klijent ne moraju da se menjaju za nove **Component** klase.;
- **Čini vaše projektno rešenje uopštenije.** Nedostatak olakšanog dodavanja novih komponenata je otežano ograničavanje dodavanja složenih komponenata. Ponekad želite da vaš složeni objekat ima samo određene komponente. Sa **Composite**, treba da koristite provere u vreme izvršenja programa.

### Implementacija:

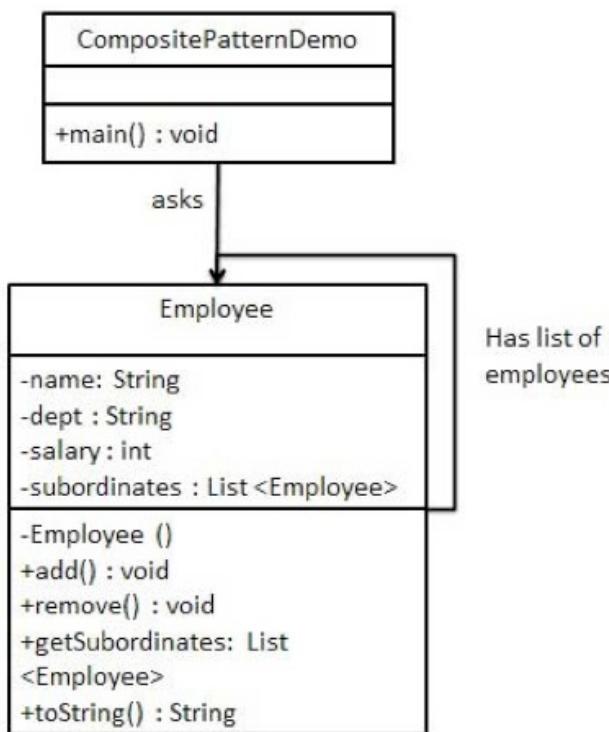
Pri primeni Composite šablona treba obratiti pažnju na mnoga pitanja

1. Eksplisitne reference na "roditelja". Ovo olakšava rad sa hijerarhijskom strukturom objekata, kao i njihovo brisanje.
2. Deljenje komponenata. Korisno je ostvarite deljenje komponenata, ali je teže ostvarljivo kada one imaju samo jednog "roditelja".
3. Maksimizacija **Component** interfejsa. Treba da definiše što veći broj operacija klasa strukture.
4. Deklarisanje operacije upravljanja "decom". Da li ih deklarisati u **Component** ili u **Composite** klasi?
5. Da li **Component** primenjuje listu **Component** objekata? Samo sa malim brojem "dece".
6. Redosled "dece".
7. Keširanje radi poboljšanja performansi.
8. Ko može da briše komponente?
9. Koja je najbolja struktura podataka za smeštaj komponenata?

## ŠABLON COMPOSITE: PRIMER I 1. KORAK REŠAVANJA

Klasa **Employee** je **Composite** klasa, **CompositePatternDemo** klasa upotrebljava **Employee** klasu radi dodavanja hijerarhijskog nivoa organizacionih jedinica

**Primer:** Klasa **Employee** je **Composite** klasa, **CompositePatternDemo** klasa upotrebljava **Employee** klasu radi dodavanja hijerarhijskog nivoa organizacionih jedinica i za štampanje zaposlenih.



Slika 5.5 Klasa CompositePatternDemo [2.4]

**Korak 1:** Kreiranje **Employee** klase sa listom **Employee** objekata

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

public class Employee {
    private String name;
    private String dept;
    private int salary;
    private List<Employee> subordinates;

    // constructor
    public Employee(String name, String dept, int sal) {
        this.name = name;
    }

    // ...
}
```

```
        this.dept = dept;
        this.salary = sal;
        subordinates = new ArrayList<Employee>();
    }

    public void add(Employee e) {
        subordinates.add(e);
    }

    public void remove(Employee e) {
        subordinates.remove(e);
    }

    public List<Employee> getSubordinates(){
        return subordinates;
    }

    public String toString(){
        return ("Employee :[ Name :" + name
            + ", dept :" + dept + ", salary :"
            + salary+ " ]");
    }
}
```

## ŠABLON COMPOSITE: POSTUPAK PRIMENE (2. I 3. KORAK)

*Upotreba Employee klase za kreiranje i štampanje hijerarhije i prikaz rezultata*

**Korak 2:** Upotreba Employee klase za kreiranje i štampanje hijerarhije

```
public class CompositePatternDemo {
    public static void main(String[] args) {
        Employee CEO = new Employee("John", "CEO", 30000);
```

```
Employee headSales = new Employee("Robert","Head Sales", 20000);

Employee headMarketing = new Employee("Michel","Head Marketing", 20000);

Employee clerk1 = new Employee("Laura","Marketing", 10000);
Employee clerk2 = new Employee("Bob","Marketing", 10000);

Employee salesExecutive1 = new Employee("Richard","Sales", 10000);
Employee salesExecutive2 = new Employee("Rob","Sales", 10000);

CEO.add(headSales);
CEO.add(headMarketing);

headSales.add(salesExecutive1);
headSales.add(salesExecutive2);

headMarketing.add(clerk1);
headMarketing.add(clerk2);

//print all employees of the organization
System.out.println(CEO);
for (Employee headEmployee : CEO.getSubordinates()) {
    System.out.println(headEmployee);
    for (Employee employee : headEmployee.getSubordinates()) {
        System.out.println(employee);
    }
}
}
```

**Korak 3:** Prikaz rezultata

```
Employee :[ Name : John, dept : CEO, salary :30000 ]  
Employee :[ Name : Robert, dept : Head Sales, salary :20000 ]  
Employee :[ Name : Richard, dept : Sales, salary :10000 ]  
Employee :[ Name : Rob, dept : Sales, salary :10000 ]  
Employee :[ Name : Michel, dept : Head Marketing, salary :20000 ]  
Employee :[ Name : Laura, dept : Marketing, salary :10000 ]  
Employee :[ Name : Bob, dept : Marketing, salary :10000 ]
```

Slika 5.6 Dobijeni rezultat na monitoru računara [2.4]

## COMPOSITE DESIGN PATTERN (VIDEO)

*Trajanje: 16,46 minuta*

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

## VIDEO PREDAVANJE ZA OBJEKAT "ŠABLON COMPOSITE I ŠABLON FAÇADE"

*Trajanje video snimka: 26min 5sek*

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

## ✓ Poglavlje 6

# Pokazna vežba – Šabloni strukture

## ŠABLONI PROJEKTOVANJA

### *Opis i cilj lekcije*

Na ovom času ćemo videti kako se implementiraju design patterni. Videćemo primere implementacije Design Pattern-a Bridge, Dekorater. Osim što je demonstrirana implementacija Design Patterna pokazano je i kako se dati Pattern koristi. Za svaki Design Pattern je dato na kojim principima OO programiranja se zasnivaju i kako se ti OO principi implementiraju u okviru Design Patterna.

Svaki veći projekat podrazumeva upotrebu šablona projektovanja. Postoji veliki broj šabloni. Problem nastaje što u realnim uslovima nije lako ustanoviti koji šablon izabrati za primenom, a poseban problem je što nam uglavnom treba kombinacija poznatih šabloni ili poznate šablone treba prilagoditi, malo promeniti, da bi bili rešenje neke situacije. Da bi umeli da prilagodimo postojeće šablone situaciji, moramo da poznajemo mehanizme funkcionisanja šabloni, a to se najbolje vidi na primerima.

### ✓ 6.1 Pokazni primer 1 - vežba za šablon Bridge

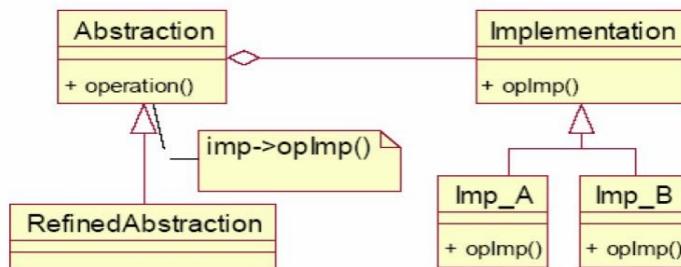
## SVRHA ŠABLONA BRIDGE

*Svrha i upotreba Bridge šabloni projektovanja. Vreme trajanja: 20minuta*

Na slici 1 prikazan je UML dijagram Bridge šabloni projektovanja. Osnovna namena ovog šabloni je da razdvoji implementaciju od apstrakcije. Da bi realizovali ovaj koncept potrebno je da imamo nadklasu (apstraktnu klasu ili interface) u kojima su definisane sve moguće metode koje implementacija može da ima.

Ovaj šablon se koristi kad možemo da imamo više implementacija za isti tip podatka. Prikazaćemo jedan primer Bridge šabloni projektovanja.

Posebna pogodnost ovog šabloni je laka izmena i održavanje. Ako dođe do promene implementacije ne moramo da menjamo apstrakciju i obrnuto.



Slika 6.1.1 UML prikaz Bridge šablona projektovanja [Izvor: Autor]

## IZRADA BRIDGE ŠABLONA PROJEKTOVANJA

*Izrada Apstraktne klase korisnik i dve različite implementacije iste klase*

Definišemo apstraktnu implementaciju tako što definišemo apstraktну klasu **Korisnik** koja u sebi ima metode **uplata**, **isplata** i **mesecniObracun** koje će služiti apstrakciji za pristup implementaciji. Ova Apstraktna klasa ima jedan konkretni podatak stanje, a to je trenutno stanje korisnika i taj podatak mora da postoji u svim implementacijama zato smo ga i deklarisali u nadklasi.

```

public abstract class Korisnik {
    protected double stanje;
    public Korisnik() { }
    public double getStanje() {
        return stanje;
    }
    public abstract void mesecniObracun();
    public abstract void uplata(double value);
    public abstract void isplata(double value);
}

```

Pravimo prvu konkretnu implementaciju u okviru Bridge šablona projektovanja. Konkretna implementacija će biti klasa **Stedisa**. Ova klasa je zadužena za jednu varijantu implementacije Korisnik klase. Posebnu pažnju treba obratiti na metod **mesecniObracun** koji poziva druge private metode i na taj način ostvaruje specifičnu implementaciju koja je vezana za klasu Stedisa.

```

public class Stedisa extends Korisnik {
    public Stedisa() { }
    public void mesecniObracun() {
        dodajKamatu();
        odbiBancineUsluge();
        uzimimDaNePrimeti();
    }
    private void dodajKamatu() {
        stanje *= 1.1;
    }
    private void odbiBancineUsluge()
}

```

```
{ stanje -= 150; }
private void uzmimuDanePrimeti()
{ stanje *= 0.999; }
public void uplata(double value)
{ stanje += value; }
public void isplata(double value)
{ stanje -= value; }
}
```

Druga konkretna implementacija klase **Korisnik** je klasa **Zaposleni**. Ova klasa kreira drugu varijantu klase Korisnik to je slučaj kad je korisnik ujedno i zaposleni. Metoda mesecniObracun drugačija je od predhodne varijante odnosno ima drukčiju implementaciju.

```
public class Zaposleni extends Korisnik {
public Zaposleni() { }
public void mesecniObracun()
{
dodajKamatu();
upaltiPlatu();
odbiBancineUsluge();
}
private void dodajKamatu()
{ stanje *= 1.1; }
private void upaltiPlatu()
{ stanje += 70000; }
private void odibiBancineUsluge()
{ stanje -= 150; }
public void uplata(double value)
{ stanje += value; }
public void isplata(double value)
{ stanje -= value; }
}
```

## BRIDGE APSTRAKCIJA

### *Kreiranje klase za komunikaciju sa klijentom banke*

Prelazimo na deo apstrakcije u okviru Design Patterna Bridge. Ovaj deo Design Patterna je neophodan da bi ostvario klasu za komunikaciju sa svima kome je potreban Klijent Banke, bez obzira da li je on štediša ili zaposleni banke. Odnosno informacija šta je Klijent Banke treba da ostane nepoznata onome ko koristi ovu klasu. Svaka izmena u unutrašnjem ponašanju Klijenta, na primer način izračunavanja mesečnog obračuna, treba da ostane nepoznata onome ko koristi ovu klasu čak i sam deo apstrakcije treba da ne zna ovo. Klasa Klijent ima sve javne metode kao i klase koje su zadužene za implementaciju s tom razlikom što klasa Klijent ne vrši nikakvo izračunavanje sa tim podacima.

```
public class Klijent
{
```

```
private Korisnik kor;
public Klijent(Korisnik kor)
{ this.kor = kor; }
public void mesecniObracun()
{ kor.mesecniObracun(); }
public double stanje()
{ return kor.getStanje(); }
public void isplata(double suma)
{ kor.isplata(suma); }
public void uplata(double suma)
{ kor.uplata(suma); }
```

## UPOTREBA BRIDGE ŠABLONA

*Upotreba Bridge šablona projektovanja na kreiranom primeru.*

Pre nego što vidimo kako se koristi Bridge Design Pattern, napravićemo jednu utility metodu. Metoda će biti zadužena za prikaz svih klijenata i primaće apstraktnu klasu Klijent tako da možemo da pošaljemo bilo koju konkretnu klasu koja nasleđuje Klijent apstraktnu klasu. Sama metoda radi po principu varijabilnog broja argumenata što znači da metodi možemo da pošaljemo jednog klijenta, više njih razdvojenih zarezom ili niz klijenata. U okviru same metode implementiramo kao da je poslat niz klijenata. Ako je bio poslat jedan onda ćemo praktično u promenljivoj ili imati samo jedan element niza, a ako ih prosledimo sa zarezima sama java će to pretvoriti u niz koji možemo na ovaj način da obradujemo. Da bi ulepšali prikaz formatirali smo prikaz koristeći DecimalFormat javinu klasu i napravili sopstvenu masku za formatiranje.

```
private void prikazSvihKlijenata(Klijent ... kli)
{
    DecimalFormat decFormat = new DecimalFormat("#,##0.00");
    System.out.println("\nStanje Klijenata");
    for(Klijent k : kli)
        System.out.println("\nStanje Klijenta: " + decFormat.format(k.stanje()));
}
```

Da bi demonstrirali mogućnosti ovog Design Patterna prvo ćemo napraviti niz Klijenata koji u sebi imaju konkretne korisnike. Ti korisnici mogu biti tipa Stedisa ili tipa Zaposleni. Kasnije u radu nećemo se više obraćati tipu korisnika već samo klijentu.

```
Klijent[] klijenti = new Klijent[6];
klijenti[0] = new Klijent(new Zaposleni());
klijenti[1] = new Klijent(new Stedisa());
klijenti[2] = new Klijent(new Stedisa());
klijenti[3] = new Klijent(new Stedisa());
klijenti[4] = new Klijent(new Stedisa());
klijenti[5] = new Klijent(new Zaposleni());
Random slu = new Random();
```

Pristupamo metodama mesecniObracun, uplata i isplata tako što pristupamo Klasi Klijent. U slučaju da moramo po zahtevu da izmenimo unutrašnje ponašanje mesečnog obračuna nad jednim tipom korisnika onda ta ispravka neće uticati na ostale klase u programu. Ništa ne mora da se menja osim konkretne metode koja se menja.

```
for(int i = 0; i < 20; i++) {  
    klijenti[slu.nextInt(6)].mesecniObracun();  
    klijenti[slu.nextInt(6)].uplata(  
        (double)slu.nextInt(20000));  
    klijenti[slu.nextInt(6)].isplata(  
        (double)slu.nextInt(20000));  
    prikazSvihKlijenata(klijenti);  
}
```

## OBJEKTNI PRINCIPI PRIMENJENI NA BRIDGE

### *Opis nasleđivanja, enkapsulacije i polimorfizma kao objektno-orientisanih principa u Bridge šablonu projektovanja*

U okviru Bridge-a koriste se sva tri principa OOP. Nasleđivanje je primenjeno u delu implementacije gde je napravljena apstraktna nad klasa pod nazivom Korisnik koju nasleđuju konkretne klase Stedisa i Zaposleni. U okviru samog šablonu projektovanja ova hijerarhija predstavlja implementacioni deo Bridge Design Patterna.

U realnim situacijama ovu apstraktну klasu možemo zameniti interface-om. Ovde je odlučeno da se koristi apstraktna klasa zbog stanja koje je zajedničko za sve podklase pa iz tog razloga je zgodno da se dati podatak digne u nadklasu. Ako ne postoje zajednički podaci onda je preporučljivije koristiti interface. U konkretnoj primeni može da nam se dogodi da je deo ovog šablonu zadužen za apstrakciju potreбno razdeliti na više klase. U toj situaciji preporučljivo je da i Apstraktioni deo ima svoju hijerarhiju.

U nekoj situaciji može da se implementira ovaj pattern i bez hijerarhije ali to je redak slučaj upotrebe Bridge šablonu projektovanja.

Osnovna ideja šablonu projektovanja Bridge je enkapsulacija odnosno sakrivanje. Brigde ostvaruje sakrivanje konkretne implementacije preko definicije odnosno apstrakcije. Osim ovog načelnog principa i u samom kodu ovog šablonu projektovanja primenjena je enkapsulacija na nekoliko mesta. Prvo korisnik u klasi Klijent je sakriven, drugo stanje u klasi korisnik je takođe enkapsulirano i važi samo za svu decu klase Korisnik. Ovaj podatak je nevidljiv van hijerarhije.

Polimorfizam je takođe zastupljen u ovoj implementaciji Design Patterna Bridge. Imamo apstraktne metode uplata(), isplata() i mesecniObracun() koje su implementirane tek u svojoj deci, u klasama Zaposleni i Stedisa. Ovo je naravno samo jedna varijanta implementacije ovog Design Patterna i svakako postoji još mogućnosti kako ovo uraditi.

U dodatnom materijalu za ove vežbe dat je kompletan kod ovog primera pa je poželjno da se proba kako bi se videli rezultati rada ovog koda. Takođe može da se proba unos raznih logičkih izmena i primetiće se da će izmene veoma logično biti samo na jednom mestu. Tako

da Bridge Design Pattern pomaže da kod bude u skladu sa principima Objektno Orijentisanog programiranja.

Razmislite i pokušajte da smislite bolju implementaciju ovih metoda.

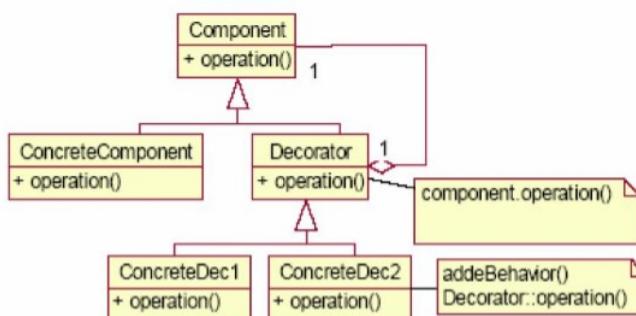
## ✓ 6.2 Pokazni primer 2 - vežba za šablon Decorator

### SVRHA ŠABLONA DECORATER

*Svrha i upotreba šabloni Decorater. Vreme trajanja: 15 minuta*

Na slici 1 je prikazan UML dijagram šablon projekovanja Dekorater. Osnovna ideja Dekoratera je da imamo komponentu koja radi svoj posao i posebnu klasu Dekorater koja će da obezbeđuje dodatno ponašanje na osnovnu komponentu. Ovaj šablon projekovanja je primenjen na mnogim mestima u okviru samog Java programskog jezika odnosno SUN-ovog API-a. Na primer kad kreiramo Jtable komponentu mi možemo da je smestimo u klasu JScrollPane pa ćemo običnoj tabeli dodati mogućnost da ima skrolovanje. Ako želimo možemo da iskoristimo i klasu TableRowSorter koja je također Dekorater i koja obezbeđuje tabeli mogućnost sortiranja redova.

Osnovna implementacija ovog šablon projekovanja je preko apstraktne definicije komponente odnosno Dekoratera. Konkretna komponenta nasleđuje apstraktну komponentu, a konkretni Dekorateri nasleđuju apstraktni Dekorater. Na dijagramu se vidi da mora da se ispuni uslov da metode koje koristi konkretna komponenta moraju da budu definisane u apstraktnoj komponenti i da posle sve klase koje je nasleđuju imaju implementirane te metode. U slučaju Dekoratera te metode moraju da budu preusmerene na konkretnu komponentu, odnosno dekorater ako treba da napravi izmene u primljenim podacima i posle ih prosledi konkretnoj komponenti. Konkretna komponenta nije ni svesna da postoji dekorater.



Slika 6.2.1 UML dijagram šabloni dekorater [Izvor: Autor]

### DEKORATER NAPOMENA

*Napomena i saveti prilikom upotrebe dekorater šabloni projekovanja*

Iz prethodne analize rada Decorator Design Patterna može da se zaključi da je on idealan kad treba postojećoj komponenti dodati novu funkcionalnost. Umesto da se unose razni if-else mehanizmi u postojeću klasu i/ili da se povećavaju zaduženja iste klase mi možemo da implementiramo Design Pattern Decorater i na taj način da održimo visoku čitljivost koda jedinstvenost klasa-objekata i da povećamo funkcionalnost. Jednom rečju da održimo visok nivo Objektno Orientisanosti koda. Druga velika prednost Dekoratera u odnosu na potencijalno drugačija rešenja je da klasa kojoj se dodaje funkcionalnost tog nije ni svesna tako da se povećava transparentnost ovog rešenja.

U svakom slučaju Design Pattern Dekorater obezbeđuje:

- Lako dodavanje novih funkcionalnosti
- Lako održavanje koda
- Velika čitljivost koda bez obzira na uvođenje velikog broja varijanti i funkcionalnosti
- Veća fleksibilnost u korišćenju koda

## IZRADA KOMPONENTI ŠABLONA DECORATER

### *Izrada apstrakcije komponente odnosno KlijentInterface i klijent konkretnu komponentu*

U našem primeru ćemo umesto apstraktne klase komponente da koristimo interface. Korišćenje interface-a umesto apstraktne klase se koristi kad god nemamo nekih podataka koje bi apstraktna klasa trebalo da sadrži. Pravimo Interface KlijentInterface koji definiše metode koje će morati naša konkretna komponenta da implementira. Ove iste metode će morati i dekorater klasa da implementira.

```
public interface KlijentInterface
{
    int getBrojRacuna();
    double getDug();
    double getMesecnaKamata();
    String getNaziv();
    double getStanje();
    void isplata(double isplata);
    void mesecniObracun();
    void setBrojRacuna(int brojRačuna);
    void setDug(double dug);
    void setMesecnaKamata(double mesecnaKamata);
    void setNaziv(String naziv);
    void uplata(double uplata);
}
```

```
public class Klient implements KlijentInterface{
    private String naziv;
    private int brojRacuna;
    private double stanje;
    private double dug;
```

```
private double mesecnaKamata;
private boolean kredit;
public Klijent() { }
public Klijent(String naziv, int brojRacuna, double stanje, double dug, double
mesecnaKamata)
{
this.naziv = naziv;
this.brojRacuna = brojRacuna;
this.dug = dug;
this.stanje = stanje;
this.mesecnaKamata = mesecnaKamata;
}
public Klijent(Klijent k)
{
this.naziv = k.naziv;
this.brojRacuna = k.brojRacuna;
this.dug = k.dug;
this.stanje = k.stanje;
this.mesecnaKamata = k.mesecnaKamata;
this.kredit = k.kredit;
}
public String getNaziv()
{ return naziv; }
public void setNaziv(String naziv)
{ this.naziv = naziv; }
public int getBrojRacuna()
{ return brojRacuna; }
public void setBrojRacuna(int brojRacuna)
{ this.brojRacuna = brojRacuna; }
public double getStanje()
{ return stanje; }
public double getDug()
{ return dug; }
public void setDug(double dug)
{ this.dug = dug; }
public double getMesecnaKamata()
{ return mesecnaKamata; }
public void setMesecnaKamata(double mesecnaKamata)
{ this.mesecnaKamata = mesecnaKamata; }
public void mesecniObracun() {
if(dug>0)
{
stanje -= mesecnaKamata;
dug -= mesecnaKamata;
} else
{ stanje *= 1.02; }
}
public void uplata(double uplata)
{ stanje += uplata; }
public void isplata(double isplata)
{ stanje -= isplata; }
}
```

## APSTRAKTNI ŠABLON DECORATER

*Izrada apstraktnog klijent dekoratera koji postavlja podrazumevana ponašanja svih metoda koje su kreirane u interfejsu.*

Klasa koja implementira dekorater se zove KlijentDekorater i naravno mora da implementira intreface KlijentInterface. Ova klasa je apstraktna klasa za konkretnе dekoratere koje ћemo kasnije napraviti. Uloga ove klase je da kreira default ponašanje svih metoda koje su definisane u interface-u KlijentInterface. Implementacija ovih metoda podrazumeva kreiranje veze između dekoratera i konkretnе komponente. Konkretna komponenta se prima u konstruktoru ove apstraktne klase. Ova klasa sadrži u sebi instancu konkretnе komponente. Mi vodimo instancu konkretnе komponente preko zadatog interface-a. Koristimo Interface umestu konkretnе komponente jer na taj način možmo da implementiramo veći broj dekoratera jer svaki od njih implementira dati interface kao i konkretna klasa. U ovoj varijanti konkretni dekorater ne zna da li je primio konkretnu komponentu ili drugi dekorater. Nakon kreiranja ovakve apstraktne klase biće nam lakše da implementiramo konkretni dekorater jer će klasa koja predstavlja konkretni dekorater morati da override-uje samo one metode kojima želi da promeni ponašanje.

```
public abstract class KlijentDekorater implements KlijentInterface {  
    protected KlijentInterface klijent;  
    KlijentDekorater(KlijentInterface k)  
    { klijent = k; }  
    public int getBrojRacuna()  
    { return klijent.getBrojRacuna(); }  
    public double getDug()  
    { return klijent.getDug(); }  
    public double getMesecnaKamata()  
    { return klijent.getMesecnaKamata(); }  
    public String getNaziv()  
    { return klijent.getNaziv(); }  
    public double getStanje()  
    { return klijent.getStanje(); }  
    public void setBrojRacuna(int brojRacuna)  
    { klijent.setBrojRacuna(brojRacuna); }  
    public void setDug(double dug)  
    { klijent.setDug(dug); }  
    public void setMesecnaKamata(double mesecnaKamata)  
    { klijent.setMesecnaKamata(mesecnaKamata); }  
    public void setNaziv(String naziv)  
    { klijent.setNaziv(naziv); }  
    public void uplata(double uplata)  
    { klijent.uplata(ulata); }  
    public void isplata(double isplata)  
    { klijent.isplata(isplata); }  
    public void mesecniObracun()  
    { klijent.mesecniObracun(); }
```

## PRVI KONKRETNI DECORATER

*Izrada konkretnog dekoratera koji predstavlja usluge uplate/isplate i implementira apstraktni dekorater*

Konkretni dekorater ćemo implementirati u klasi UslugeUplataIsplata. Ova klasa mora da nasledi apstraktну klasu KlijentDekorater koja u sebi sadrži implementirane sve metode od zadatog interface-a KlijentInterface. Konstruktor ove klase mora da primi interface KlijentInterface koji će predstavljati instancu konkretne komponente koju dekorišemo ili drugi dekorater. Ovaj dekorater će da izmeni metode uplata() i isplata() i dodaće proviziju banke. Na ovaj način nismo narušili osnovnu klasu i ona radi bez uzimanja provizije a posao uzimanja provizije smo izveli preko ovog dekoratera. Ako dođe do izmene u politici banke i provizija se promeni osnovna komponenta se neće menjati menjamo samo dekorater.

```
public class UslugeUplataIsplata extends KlijentDekorater
{
    public UslugeUplataIsplata(KlijentInterface klijent)
    { super(klijent); }
    @Override
    public void uplata(double uplata)
    {
        klijent.uplata(uplata);
        klijent.isplata(uplata*0.02); //2% Uzima banka
    }
    @Override public void isplata(double isplata) {
        if(isplata*1.02 > klijent.getStanje())
        {
            klijent.isplata(isplata * 1.02);
        }
        else{System.out.println("Nema dovoljno para na računu" +
        "isplata nije obavljena");
    }
}
```

## DRUGI KONKRETNI ŠABLON DECORATER

*Izrada drugog konkretnog šablon-a Decorater koji proverava stanje. On takođe predstavlja implementaciju apstraktnog dekoratera*

Kreiramo još jedan dekorater. Osnovna komponenta nema proveru stanja odnosno ne proverava da li isplata može da se izvede odnosno da li ima dovoljno para na računu za takvu transakciju. Pravimo dekorater klasu koji se zove ProveraStanja koja će da proverava da li ima dovoljno para na računu za traženu transakciju. Uvođenjem ovog i prethodnog dekoratera. Mi smo podelili posao na nekoliko klasa. Osnovna komponenta odrađuje bazični posao klijenta dok ova dva dekoratera svaki radi deo posla vezanog za promet na klijentovom računu. Na ovaj način svaka klasa ima svoje jednostavno kratko zaduženje. Svaka klasa je jasna i lako čitljiva.

```
public class ProveraStanja extends KlijentDekorater {  
    public ProveraStanja(KlijentInterface klijent)  
    { super(klijent); }  
    @Override public void isplata(double isplata) {  
        if(getStanje() < isplata)  
            System.out.println("Nema dovoljno para na računu" +  
                "isplata nije obavljena");  
        else  
            klijent.isplata(isplata); }  
    @Override  
    public void mesecniObracun()  
    {  
        if(klijent.getStanje() > getDug())  
            klijent.mesecniObracun();  
        else  
            System.out.println("Klijent nema dovoljno para na računu");  
    }  
}
```

## PRIMENA ŠABLONA DECORATER

### *Primena šabloni Decorater na kreiranom primeru*

Ovo je jedna od implementacija kreiranog Dekorater šabloni projektovanja. Kao što je ranije napomenjeno u zavisnosti od svrhe kreiranja Dekoratera šabloni projektovanja zavisi i njegova primena.

```
Klijent[] klijenti = new Klijent[6];  
klijenti[0] = new Klijent("Pera", 0001, 10000, 0, 100);  
klijenti[1] = new Klijent("Dragana", 0002, 10000, 100000, 1000);  
klijenti[2] = new Klijent("Miroslava", 0003, 100000, 0, 1000);  
klijenti[3] = new Klijent("Miloš", 0004, 10000, 20000, 1000);  
klijenti[4] = new Klijent("Ivana", 0005, 100000, 0, 1000);  
klijenti[5] = new Klijent("Jovana", 0006, 100000, 20000, 1000);  
KlijentInterface[] klijentReal= new KlijentInterface[klijenti.length];  
for(int i = 0; i<klijentReal.length; i++)  
{  
    klijentReal[i] = new ProveraStanja  
(new UslugeUplataIsplata(klijenti[i]));  
}  
Random slu = new Random();  
for(int i = 0; i < 20; i++)  
{  
    klijentReal[slu.nextInt(6)].mesecniObracun();  
    klijentReal[slu.nextInt(6)].uplata((double)slu.nextInt(20000));  
    klijentReal[slu.nextInt(6)].isplata((double)slu.nextInt(20000));  
    prikazSvihKlijenata(klijentReal);  
}
```

## OBJEKTNI PRINCIPI PRIMENJENI NA ŠABLON DECORATER

### *Opis nasleđivanja, enkapsulacije i polimorfizma kao objektno-orientisanih principa primenjenih u Dekorater šablonu projektovanja*

U Design Pattern-u dekorater objektno orijentisani princip nasleđivanja igra veliku ulogu. Kao što smo videli početak cele konstrukcije je apstraktna klasa koja definiše komponentu i dekorater. U našem slučaju to je bio interface KlijentInterface. Ovaj interface su implementirale klase Klijent koja predstavlja konkretnu komponentu i klasa KlijentDekorater koja je apstraktna klasa za sve dekoratere. Ovde dolazimo i do drugog dela hijerarhije gde apstraktna klasa KlijentDekorater predstavlja koren klasu svih dekoratera. Ovu klasu nasleđuju klase ProveraStanja i UslugeUplatalsplata.

Kao što smo videli osnova ovog Design Patterna je upravo hijerarhija odnosno definisano ponašanje koje imaju sve klase u okviru Decorater Design Patterna.

Ni jedna klasa u celom Design Patternu ne sme da ima ni jednu metodu van osnovnog definisanog interface-a (KlijentInterface).

Da bi svaka klasa u okviru Design Pattern mogla da funkcioniše kao proizvod za sebe bilo je neophodno sve podatke adekvatno enkapsulirati. Tako da smo sva polja klijentske klase enkapsulirali i onemogućili direktni pristup tim poljima već samo kroz definisane pristupne metode. U apstraktnoj klasi KlijentDekorater smo kreirali instancu tipa KlijentInterface (što u radu može biti konkretna komponenta odnosno instanca klase Klijent ili neki od njegovih dekoratera). Instanca tipa KlijentInterface je portected tipa tako da mogu da joj pristupaju pod klase odnosno konkretni dekorateri. U našem slučaju to su klase UslugeUplatalsplata i ProveraStanja.

U Design Patternu Decorater se koristi polimorfizam i to njegov mehanizam Override. Sve metode koje postoje u ovom Desing Pattern-u su definisane u Interface-u KlijentInterface sve klase samo vrše Override ovih metoda. S obzirom da su klase Klijent i KlijentDekorater u istom hijerarhijskom nivou klasa KlijentDekorater u sebi sadrži instancu klase Klijent i sve pozive Overrde-ovanih metoda preusmerava na instancu klase Klijent. U delu pod hijerarhije dekoratera imamo još očigledniji Override gde konkretni dekorateri vrše Override samo onih funkcija kojima menjaju funkcionalnost.

### ✓ 6.3 Pokazni primer 3 - vežba za šablon Composite

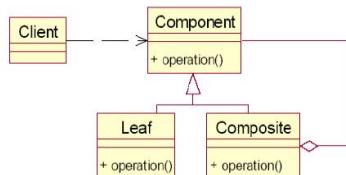
## COMPOSITE ŠABLON PROJEKTOVANJA

*Šta predstavlja Composite šablon projektovanja i način primene.  
Vreme trajanja: 10 minuta*

Na slici 1 prikazan je UML dijagram Design Patterna Composite. Ovaj Desing Pattern se koristi u situacijama kad imamo strukturu u strukturi. Na primer želimo da napravimo program u kome ćemo imati mapirane sve elemente neke konstrukcije sa svojim podacima, recimo veš mašina. Problem može nastati ako ne razmišljamo o ovom problemu već krenemo sa implementacijom klase koje će predstavljati pojedinačne elemente te mašine. Završićemo sa gomilom klasa koje se drugačije zovu (motor, osovina, bubanj), pri čemu će sve klase imati otprilike iste podatke naziv, datum ugradnje, id, isl... Naravno najveći problem će tek tad doći kako da se sve te klase povežu pa da se zna koji deo pripada kom poddelu. Rešenje ovog problema se nalazi u Design Patternu Composite za koji će biti potrebno samo par klasa (umesto svega onog što smo napravili). Osnovna arhitektura Design Patterna Composite je u apstraktnoj klasi koja definiše osnovno ponašanje i koja sadrži u sebi kolekciju istog tipa. Instance koje se nalaze u kolekciji mogu biti tipa Composite ili Leaf. Leaf predstavlja list odnosno poslednju komponentu odnosa komponentu koja u sebi ne sadrži konkretnu komponentu. Ovo sve je moguće zahvaljujući mehanizmu građenja hijerarhije kao i ostalim mehanizmima Objektno Orientisanog programiranja.

Design Pattern Composite se koristi kad nam treba veliki broj elemenata iste ili slične strukture. Također ima veliku upotrebnu vrednost kod kontejnerskih elemenata. Na primer Dokument u sebi sadrži sekcije, sekcije sadrže pasuse, pasusi sadrže rečenice, rečenice sadrže slova i brojeve.

Ovo je tipičan primer gde se koristi Design Pattern Composite. Još jedna upotrena ovog Design Pattern-a je efikasna simulacija strukture drveta jer struktura koju gradi ovaj Design Pattern najviše podseća na drvo gde elementi Composite predstavljaju grane a elementi Leaf predstavljaju listove odnosno konačne elemente.



Slika 6.3.1 UML prikaz šablonu Composite [Izvor: Autor]

## COMPOSITE INTERFACE

*Opis interfejsa/apstraktne klase koja opisuje rad Composite šablonu projektovanja*

Da bi kreirali primer Composite Design Patterna prvo moramo da definišemo apstraktnu klasu koja definiše elemente ovog Patterna odnosno ako usvojimo termin Drvo onda elemente budućeg drveta. S obzirom da svaka grana drveta mora da ima kolekciju svojih pod grana to ćemo realizovati pomoću kolekcije LinkedList. Ovo sve će ovde biti urađeno preko interface-a. Dakle, definišemo interface Element koji će nam definisati metode koje mora da ima svaka konkretna implementacija ovog interface-a. U ovom slučaju to su metode koje se odnose na delove veš mašine.

```

public interface Element {
    public String getNazivElementa();
  
```

```
public void setNazivElementa(String naziv);
public Calendar getGodinaUgradnje();
public void setGodinaUgradnje(Calendar godinaUgradnje);
public boolean isGarancija();
public boolean isOsnovni();
public void add(Element el);
public void remove(Element el);
public LinkedList getAllPodElementi();
public Element getElement(int index);
}
```

## COMPOSITE – KONKRETNA IMPLEMENTACIJA

*Implementacija konkretnе klase koja Composite sa UML dijagrama šablonu Composite*

Kasu smo nazvali MasinskiDeo i ona implementira interface Element. Samim tim što je ova klasa implementirala interface Element implementirali smo sve metode ovog interface-a. Treba obratiti pažnju na LinkedList Elemenata. Ovo je predstavlja listu svih pod elemenata ove klase. Kao element ove kolekcije može da bude i instanca same ove klase, jer i ona implementira interface Element.

```
public class MasinskiDeo implements Element
{
    private String naziv;
    private Calendar godinaUgradnje;
    private LinkedList<Element> elementi = new LinkedList();
    public MasinskiDeo() { }
    public MasinskiDeo(String naziv, int godina)
    {
        this.naziv = naziv;
        Calendar datum = Calendar.getInstance();
        datum.set(Calendar.YEAR, godina);
        datum.set(Calendar.MONTH, 1);
        datum.set(Calendar.DAY_OF_MONTH, 1);
        godinaUgradnje = datum;
    }
    public String getNazivElementa() { return naziv; }
    public Calendar getGodinaUgradnje()
    { return godinaUgradnje; }
    public boolean isGarancija()
    {
        Calendar danas = Calendar.getInstance();
        Calendar proba = (Calendar) godinaUgradnje.clone();
        proba.roll(Calendar.YEAR, true);
        return danas.before(proba);
    }
    public boolean isOsnovni()
    { return elementi.isEmpty(); }
    public void add(Element el)
```

```

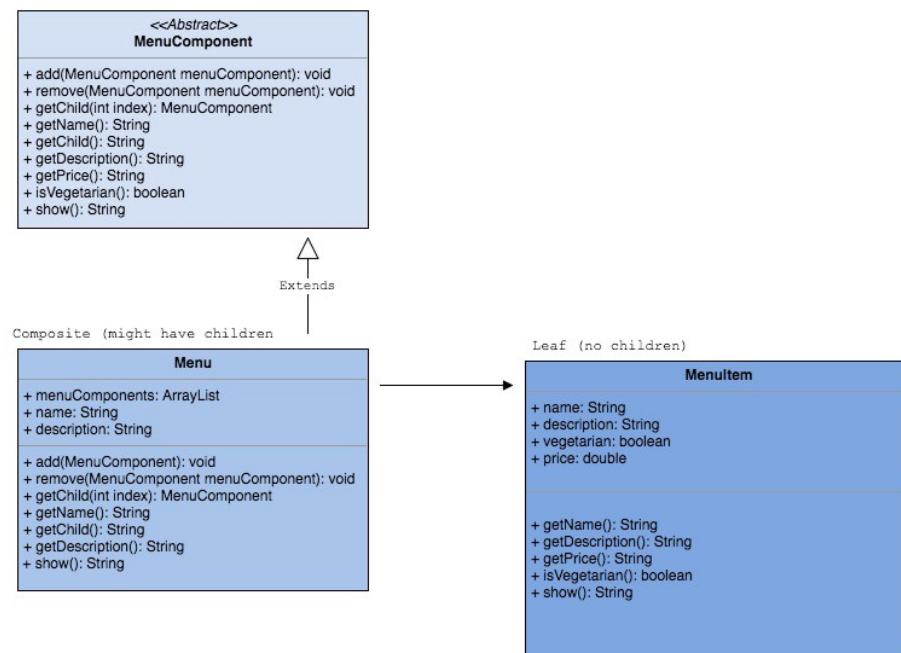
{ elementi.add(el); }
public void remove(Element el)
{ elementi.remove(el); }
public LinkedList getAllPodElementi()
{ return elementi; }
public Element getElement(int index)
{ return elementi.get(index); }
public void setNazivElementa(String naziv)
{ this.naziv = naziv; }
public void setGodinaUgradnje(Calendar godinaUgradnje)
{ this.godinaUgradnje = godinaUgradnje; }
@Override public String toString() {
String text = "\nElement: " +naziv + "\tGodina ugradnje: " +
godinaUgradnje.get(Calendar.YEAR);
for(Element el : elementi)
{ text += "\t"+el; }
return text;
}
}

```

## ▼ 6.4 Pokazni primer 4 - sistem restorana

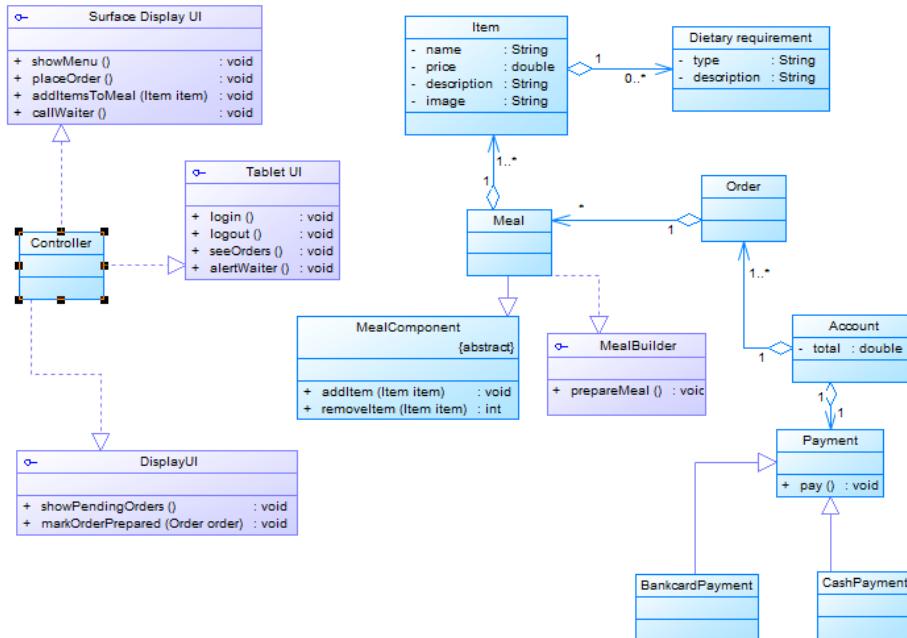
### PRIMENA COMPOSITE ŠABLONA NA SISTEM RESTORANA

*Ova sekcija daje primenu Composite šablonu na konkretnom sistemu restorana. Trajanje 5 minuta*



Slika 6.4.1 Primer primene Composite šablona projektovanja [Izvor: Autor]

Na osnovu slike 1 gde je data konkretna primena Composite šablona, izmenjena je struktura klasnog dijagrama na slici 2 tako da klasa Meal nasleđuje klasu MealComponent. Definisano je da Meal sada nasleđuje metode addItem i removeItem i predstavlja kompozitni objekat sačinjen od Item-a koji dalje nema objekte dece.



Slika 6.4.2 Klasni dijagram RMOS sistema nakon primene Composite šablona [Izvor: Autor]

## ✓ Poglavlje 7

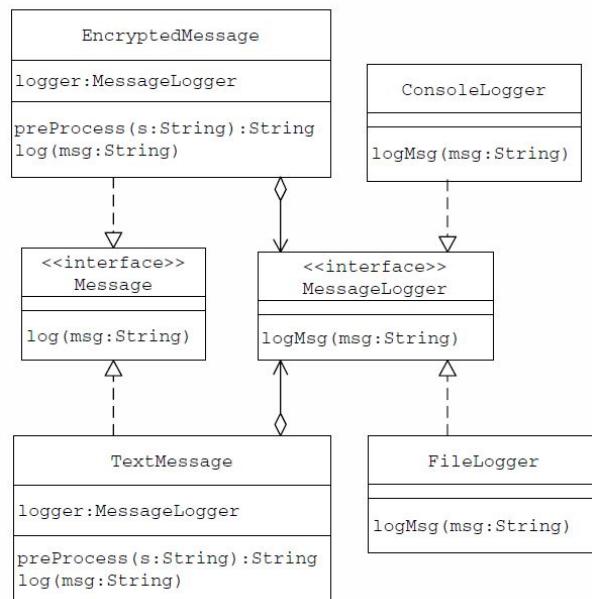
# Individualna vežba - Zadaci za samostalni rad

## ZADACI ZA INDIVIDUALNI RAD - ZADACI OD 1 DO 2

*Potrebno je primeniti znanja stečena na ovim vežbama, izrada i implementacija Bridge (Most) i Decorator (Dekorater) šablon projekata.*

### Zadatak 1 (25 minuta)

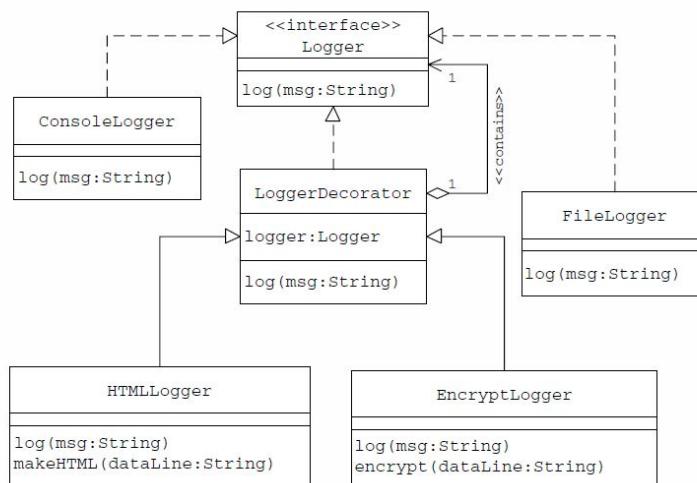
Kreirati aplikaciju koja primenjuje Most (**Bridge**) šablon projekata kao na slici 1 .



Slika 7.1 UML prikaz implementacije šablonu Bridge [Izvor: Autor]

### Zadatak 2 (25 minuta)

Kreirati aplikaciju koja primenjuje Dekorater (**Decorator**) šablon projekata kao na slici 2 .



Slika 7.2 UML prikaz implementacije šablona Decorator [Izvor: Autor]

## ZADACI ZA INDIVIDUALNI RAD - ZADACI OD 3 DO 5

*Slučaj kada Bridge šablon ne može da se primeni, Composite i Decorator u različitim sistemima.*

### **Zadatak 3** (20 minuta)

Bridge šablon se dosta koristi kod sistema koji se dele na podsisteme. Međutim, ta primena Bridge šablona nije uvek moguća. Navedite jedan primer podsistema u kome Bridge šablon ne može da se koristi i detaljno opišite razloge za nemogućnost ovakve primene.

### **Zadatak 4** (20 minuta)

Na primeru formiranja slike dodavanjem različitih oblika i simbola, nacrtati klasni dijagram koji će demonstrirati rad Composite obrasca. Generisati kod u Java programskom jeziku.

### **Zadatak 5** (30 minuta)

Na primeru odobravanja kredita i analize svih pojedinačnih dokumenata koji se prikupljaju i dodeljuju kreditnoj dokumentaciji, primeniti Composite i Decorator obrazac projektovanja. Generisati kod u Java programskom jeziku.

## ✓ Zaključak

### REZIME LEKCIJE

#### *Pouke izloženog gradiva*

1. Projektovanje softvera i njegova implementacija su dve spregnute aktivnosti. Nivo detalja koje projektno rešenje softvera treba da sadrži zavisi od tipa softvera koji se razvija i od toga da li se koristi razvoj vođen planom ili se koristi agilni pristup u projektovanju.
2. Proces projektovanje objektno-orientisanog softverskog sistema uključuje aktivnosti projektovanja arhitekture sistema, utvrđivanja osnovnih objekata sistema, opisivanje projektnog rešenja upotrebom različitih UML objektnih modela. I dokumentovanjem interfejsa komponenata.
3. Broj i vrste UML modela koji se radi u toku projektovanja može da bude različit. Koriste se statički modeli (modeli klasa, modeli generalizacije, modeli asocijacije) i dinamički modeli (seqvencijalni modeli, modeli stanja).
4. Interfejsi komponenata moraju se precizno definisati tako da ih drugi objekti mogu da koriste. U tu svrhu se koristi UML stereotip za interfejse.
5. Pri razvoju softvera, uvek gledajte da u što većoj meri koristite postojeći softver, bilo u vidu komponenti, u vidu servisa, ili u vidu celih sistema.
6. Upravljanje konfiguracijom je proces upravljanja promenama softverskog sistema koji je u razvoju. Vrlo je bitno da razvojni tim međusobno sarađuje u toku razvoja softvera.
7. Najčešće se softver razvija na razvojnoj platformi, a realizuje na izvršnoj platformi kod kupca. Na razvojnoj platformi se koristi sistem za razvoj softvera (IDE), a razvijeni izvršni kod softvera se onda prebacuje na ciljnu mašinu, tj. računar na kome će raditi u toku njegove eksploracije (tzv. izvršna platforma).
8. Razvoj softvera sa otvorenim (izvornim) kodom je razvoj softvera čiji je izvorni kod javno dostupan. To znači da mnogi mogu da predlože promene i poboljšanja softvera.

### LITERATURA

#### *Preporučena literatura*

##### **1. Obevezna literatura:**

1. Predavanja, vežbanja i dodatni materijali objavljeni na sistemu za e-učenje, 2014
2. Ian Sommerville, Software Engineering, Tenth Edition, Pearson Education Inc., 2016.

##### **2. Dopunska literatura:**

1. B. Bruegge, A. Dutoit, Object-Oriented Software Engineering – Using OML, Patterns, and Java, Thirth Edition, Prentice Hall, 2010
2. O'Reilly, Head First Design Patterns

3. Partha Kuchana, Software Architecture Design patterns in Java
4. Design Patterns - Elements of Reusable Object-Oriented Software, Eric Gamma, Richard Helm, Ralph Johnson, Jogns Viisides, 19
5. Design Patterns in Java Tutorial, [tutorialspoint.com](http://tutorialspoint.com)

**3. Veb lokacije :** Na ovim veb lokacijama možete naći opise mnogih šablona projektovanja sa primerima, te se prepućuje da ih proučite

1. <http://www.netobjectives.com/resources/books/design-patterns-explained>
2. <https://www.odesign.com/>



## SE201 - UVOD U SOFTVERSKO INŽENJERSTVO

Šabloni projektovanja ponašanja  
sistema

Lekcija 10

PRIRUČNIK ZA STUDENTE

# SE201 - UVOD U SOFTVERSKO INŽENJERSTVO

## Lekcija 10

### **ŠABLONI PROJEKTOVANJA PONAŠANJA SISTEMA**

- ▼ Šabloni projektovanja ponašanja sistema
- ▼ Poglavlje 1: Šablon Mediator
- ▼ Poglavlje 2: Šablon Memento
- ▼ Poglavlje 3: Šablon State
- ▼ Poglavlje 4: Šablon Strategy
- ▼ Poglavlje 5: Šablon Observer
- ▼ Poglavlje 6: Šablon Visitor
- ▼ Poglavlje 7: Pokazna vežba
- ▼ Poglavlje 8: Individualna vežba
- ▼ Zaključak

Copyright © 2017 – UNIVERZITET METROPOLITAN, Beograd. Sva prava zadržana. Bez prethodne pismene dozvole od strane Univerziteta METROPOLITAN zabranjena je reprodukcija, transfer, distribucija ili memorisanje nekog dela ili čitavih sadržaja ovog dokumenta., kopiranjem, snimanjem, elektronskim putem, skeniranjem ili na bilo koji drugi način.

Copyright © 2017 BELGRADE METROPOLITAN UNIVERSITY. All rights reserved. No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise, without the prior written permission of Belgrade Metropolitan University.

## ✓ Uvod

# UVOD

### *Cilj lekcije*

Cilj ove lekcije da vas nauči kako da koristite šablone projektovanja ponašanja softverskog sistema i to sledećih šablonu:

- Mediator,
- Memento,
- State,
- Strategy,
- Observer i
- Visitor

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

## ✓ Poglavlje 1

# Šablon Mediator

## ŠABLON PONAŠANJA MEDIATOR

*Šablon Mediator omogućava definisanje objekta koji učauruje (skriva) kako niz objekata međusobno komuniciraju. Mediator podržava labavo povezivanje*

### **Svrha šablon-a:**

Definisanje objekta koji učauruje (skriva) kako niz objekata međusobno komuniciraju. Mediator podržava labavo povezivanje objekata na taj način što odlaže eksplisitno povezivanje.

### **Motivacija (razlog kreiranja ovog šablon-a):**

OO projektovanje podstiče distribuciju ponašanja objekata. Ovo dovodi do mnogih veza između objekata. Ako objekat ima mnogo veza sa drugim objekatima, onda se smanjuje mogućnost njegove upotrebljivosti u drugim OO sistemima (jer može da traži te veze).

Kao primer, navešćemo meni korisničkog interfejsa (slika 1). Isti meni u različitim sistemima zahteva druge klase koje ga implementiraju. To se radi za podklasama i može da bude komplikovano.

Problem se pojednostavljuje sa Mediator objektom. Mediator je odgovoran za kontrolu i koordinaciju interakcija grupe objekata. Mediator je posrednik koji objekte povezuje u grupu. Objekti znaju samo za Mediator, a ne za druge objekte. Ovo smanjuje broj veza.

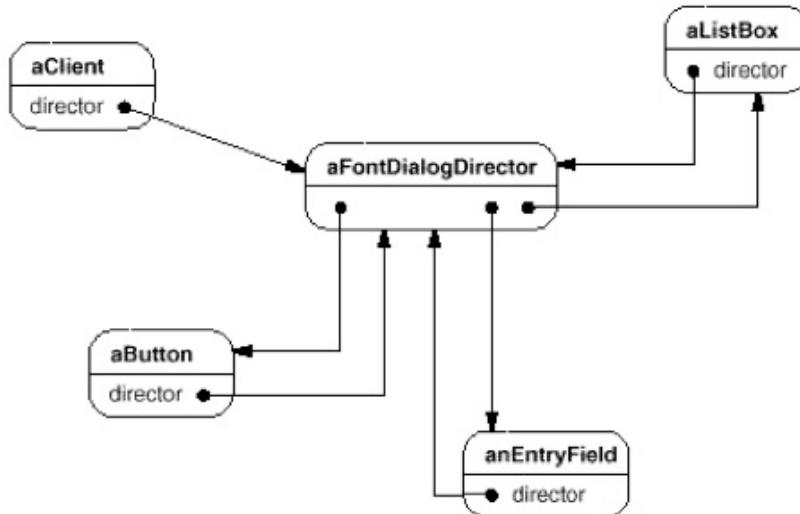


Slika 1.1 Primer korisničkog menija [4]

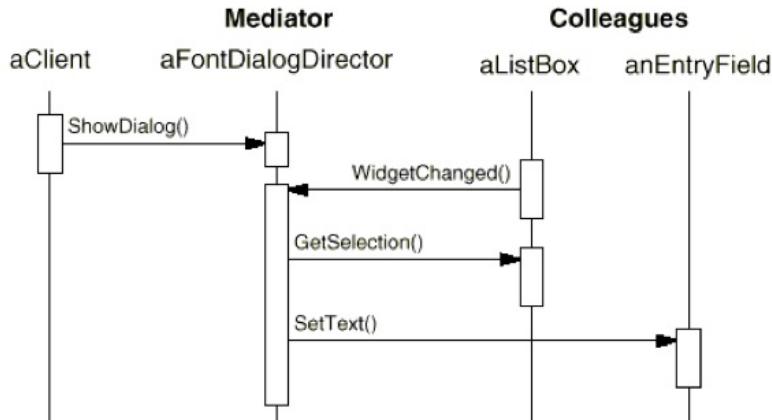
## ŠABLON MEDIATOR: PRIMER UPOTREBE

*Mediator je odgovoran za kontrolu i koordinaciju interakcija grupe objekata. Mediator je posrednik koji objekte povezuje u grupu, a oni znaju samo za M*

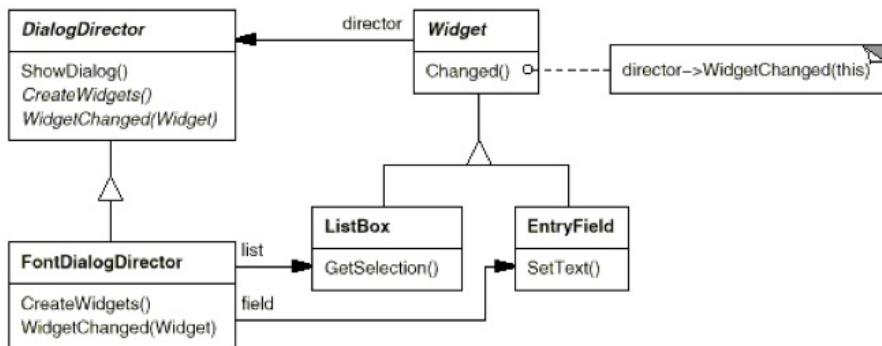
Slika 2 prikazuje objekat **aFontDialogDirector** koji je ustvari **Mediator** objekat. Slika 3 prikazuje dijagram interakcije, a slika 4 dijagram klasa za ovaj primer.



Slika 1.2 Mediator objekat: aFontDialogDirector [4]



Slika 1.3 Dijagram interakcije [4]



Slika 1.4 Dijagram klasa [4]

## ŠABLON MEDIATOR: PRIMENA, STRUKTURA, UČESNICI I KOLABORACIJA

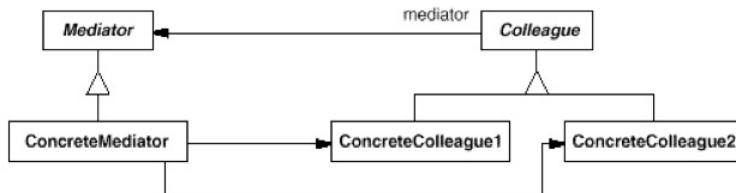
*Primenite Mediator šablon kada skup objekata međusobno komunicira na jasno definisan, ali složen način, a veze nisu strukturirane i teško su razumlji*

### Primena:

Primenite Mediator šablon kada:

- skup objekata međusobno komunicira na jasno definisan ali složen način. Veze nisu strukturisane i teško su razumljive;
- ponovna upotreba objekta je otežana jer komunicira sa drugim objektima;
- ponašanje koje se distribuira među nekoliko klasa trebalo bi da bude prilagođljivo bez upotrebe mnogo podklaasa.

### Struktura:



Slika 1.5 Struktura klasa pri primeni Mediator šablona [4]

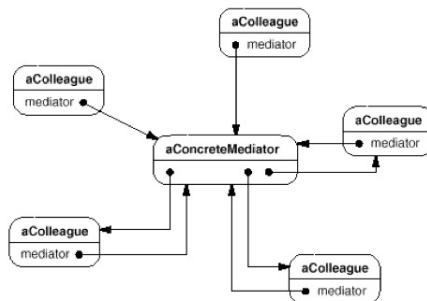
Slika 6 prikazuje jedan tipičan primer strukture objekata u kojoj se primenjuje šablon Mediator.

#### Učesnici:

- **Mediator** (DialogDirector): Definiše interfejs komunikacije sa Colleagues objektima
- **ConcreteMediator** (FontDialogDirector); Primjenjuje kooperativno ponašanje koordiniranjem Colleagues objekata
- **Colleagues** klase (IsitBox, EntryField)

#### Kolaboracija:

1. **Colleagues** šalju i primaju zahteve od **Mediator** objekta.
2. **Mediator** objekat primjenjuje kooperativno ponašanje preusmeravanje primljenih zahteva između odgovarajućih Colleagues objekata.



Slika 1.6 Primer tipične strukture objekata po Mediator šablonu [4]

## ŠABLON MEDIATOR: POSLEDICE I IMPLEMENTACIJA

*Šablon ograničava korišćenje podklasa, razdvaja Colleagues objekte, pojednostavljuje protokole objekata, olakšava sagledavanje interakcije objeka*

#### Posledice:

Mediator šablon obezeđuje sledeće povoljnosti, ali i nedostatke:

1. Ograničava korišćenje podklasa. **Mediator** lokalizuje ponašanje, koje bi inače bilo distribuirano mnogim objektima. Promena ponašanja se vrši korišćenjem podklase

samo **Mediator** klase, a **Colleagues** klase se mogu ponovo upotrebljavati bez njihove promene.

2. *Razdvajanje Colleagues objekata:* Mediator promoviše slabe veze među **Colleagues** objekata. Objekti **Colleagues** i **Mediator** se mogu nezavisno ponovo upotrebljavati

3. *Pojednostavljuje protokole objekta:* Mediator zamenjuje interakcije tipa "više sa više" sa tipom "jedan sa više" između **Mediatora** i **Colleagues** objekata. Vezu tipa "jedan prema više" je lakše za razumevanje, održavanje

4. *Apstraktan je način rada objekata.* Posebnim odvajanjem interakcije među objektima i pojedinačnog ponašanja objekata, olakšava sagledavanje interakcije objekata u sistemu.

5. *Centralizacija kontrole.* Kako **Mediator** preuzima protokole interakcije objekata, on može da postane vrlo složen, i tada postaje težak za održavanje.

### Implementacija:

Relevantna su sledeća pitanja implementacije:

1. *Izostavljanje apstraktne klase Mediator.* Kada se koristi samo jedan **Mediator** objekat, nema potrebe da se definiše apstraktna **Mediator** klasa. Ako se koristi više **Mediator** objekata, onda **Mediator** klasa obezbeđuje apstraktno povezivanje **Colleagues** objekata te oni onda rade sa više **Mediator** klase.

2. *Komunikacija Colleagues-Mediator.* **Colleagues** počinju komunikaciju sa **Mediator** objektom kada se pojavi određeni događaj. Zato, implementacija **Mediatora** se može izvršiti korišćenje **Observer** šablona. **Colleagues** klase imaju ulogu **Subjects** klase, šaljući obaveštenja **Mediator** objektu uvek kada promene stanje. Tada **Mediator** proširuje efekte ove promene na druge **Colleagues** objekte.

## ŠABLON MEDIATOR: PRIMER I 1. KORAK REŠAVANJA

*Mediator klasa preuzima svu komunikaciju između objekata (kao posrednik), te zbog labavog povezivanja objekata, omogućava jednostavnije održavanje programskog koda*

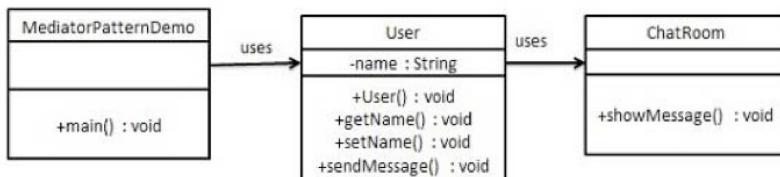
### Primer:

Mediator šablon se koristi da se smanji složenost komunikacije između više objekata ili klasa.

**Mediator** klasa preuzima svu komunikaciju između objekata (kao posrednik), te zbog labavog povezivanja objekata, olakšava održavanje programskog koda.

Slika 7 prikazuje implementacioni primer:

- **ChatRoom** klasi više **User** objekata šalje poruke
- **ChatRoom** prikazuje sve poruke svim **User** objektima.



Slika 1.7 Implementacioni primer [4]

### Korak 1: Kreiranje Mediator klase

```
import java.util.Date;

public class ChatRoom {
    public static void showMessage(User user, String message){
        System.out.println(new Date().toString()
            + " [" + user.getName() +"] : " + message);
    }
}
```

## ŠABLON MEDIATOR: POSTUPAK PRIMENE (2. I 3. KORAK)

*Kreiranje User klase i upotreba User objekta radi pokazivanja njihovih komunikacija*

### Korak 2: Kreiranje User klase

```
public class User {
    private String name;

    public String getName() {
        return name;
    }
}
```

```
public void setName(String name) {  
    this.name = name;  
}  
  
public User(String name){  
    this.name = name;  
}  
  
public void sendMessage(String message){  
    ChatRoom.showMessage(this,message);  
}  
}
```

**Korak 3:** Upotreba User objekta radi pokazivanja njihovih komunikacija

```
public class MediatorPatternDemo {  
    public static void main(String[] args) {  
        User robert = new User("Robert");  
        User john = new User("John");  
  
        robert.sendMessage("Hi! John!");  
        john.sendMessage("Hello! Robert!");  
    }  
}
```

**Korak 4:** Verifikacija rezultata

```
Thu Jan 31 16:05:46 IST 2013 [Robert] : Hi! John!  
Thu Jan 31 16:05:46 IST 2013 [John] : Hello! Robert!
```

Slika 1.8 Prikaz rezultata

## MEDIATOR DESIGN PATTERN (VIDEO)

*Trajanje: 18,30 minuta*

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

## VIDEO PREDAVANJE ZA OBJEKAT "ŠABLON MEDIATOR I ŠABLON MEMENTO"

*Trajanje video snimka: 30min 29sek*

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

## ✓ Poglavlje 2

# Šablon Memento

## ŠABLON PONAŠANJA MEMENTO

*Koristite Memento šablon kada je potrebno da se memoriše stanje objekta da bi se kasnije povratilo to stanje, dobili implementacioni detalji i time prekinu*

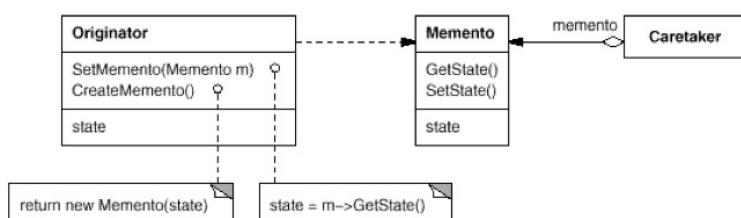
### Svrha šablon-a:

Bez remećenja učaurenja, prihvata i eksternalizuje unutrašnje stanje nekog objekta, tako da može kasnije da ga obnovi.

### Primenljivost:

Koristite Memento šablon kada je: potrebno da se memoriše stanje objekta da bi se kasnije povratilo to stanje, i potreban direktni interfejs radi dobijanja stanja implementacionih detalja i time prekinuti učaurenje objekta.

### Struktura:



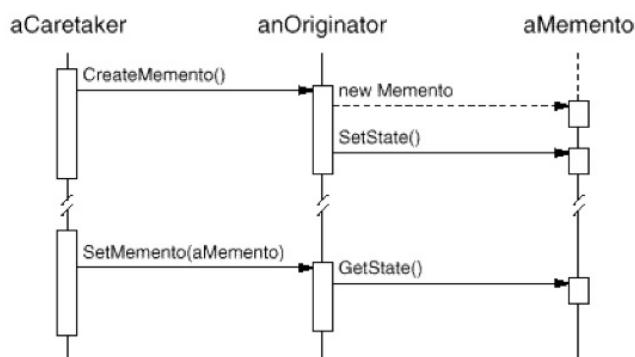
Slika 2.1 Struktura klasa pri primeni šablon-a Memento [4]

**Caretaker** zahteva Memento od **Originator** objekta, zadržava ga celo vreme, i vraća ga **Originator** objektu

### Učesnici:

1. **Memento (npr. SolverState):** Memoriše unutrašnje stanje **Originator** objekta. Štiti pristup od drugih objekata (sem od **Originator** objekta)
2. **Originator (npr. ConstraintSolver):** Kreira Memento objekat sa njegovim unutrašnjim stanjem Upotrebljava Memento da povrati svoje prethodno unutrašnje stanje
3. **Caretaker (UNDO mehanizam):** Štiti sigurnost **Memento** objekta

### Kolaboracije:



Slika 2.2 Struktura klasa pri primeni šablona Memento [4]

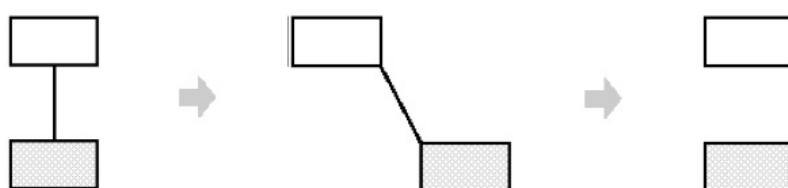
## ŠABLON MEMENTO: MOTIVACIJA

*Memento je objekat koji memoriše unutrašnja stanja Originator objekta. UNDO operacija preuzima to stanje od Memento objekta.*

### Motivacija (razlog nastanka šablona):

U praksi, ponekad je potrebno vratiti neki objekat u njegovo prethodno stanje. Međutim objekti skrivaju (učauruju) svoje stanje, te ono nije vidljivo spolja drugim objektima. Slika 1 prikazuje jedan takav primer, tj. grafički editor.

Grafički editor podržava veze među objektima. Korisnik povezuje dva pravougaonika linijom, i ta veza ostaje i ako se jedan pravougaonik pomeri (slika 1). Ovaj zadatak se rešava korišćenjem sistema sa ograničenjem (constraint-solving system)



Slika 2.3 Pomeranje pravougaonika u grafičkom editoru [autor]

**ConstraintSolver** objekat zapisuje veze kada se naprave i generiše matematičke jednačine koje ih opisuju. Rešava te jednačine uvek kada korisnik izvrši promenu u dijagramu. Na osnovu dobijenog rezultata, **ConstraintSolver** upotrebljava rezultate proračuna da bi dobio i prikazao grafičke elemente sa odgovarajućim vezama.

Rešenje sa memorisanjem rastojanja pomeranja nije dobro, jer ne garantuje da će se svi objekti pojaviti na starim lokacijama prilikom UNDO operacije. Problem se rešava sa Memento šablonom. **Memento** je objekat koji memoriše unutrašnja stanja nekog drugog objekta (**Originator**). UNDO operacija preuzima od **Memento** objekta prethodno stanje tog **Originator** objekta. **Originator** inicijalizira **Memento** objekat sa svojim trenutnim stanjem. Samo **Originator** može da stvori i da koristi informaciju o svom stanju, drugi objekti to ne vide. Konkretno, ovaj postupak se u slučaju primera grafičkog editora, sprovodi na sledeći način:

1. Editor zahteva **Memento** objekat od **ConstraintSolver** objekta, pri operaciji pomeranja grafičkog objekta.
2. **ConstraintSolver** kreira i vraća **Memento** objekat, tj. **SolverState** objekat, koji sadrži strukturu podataka koja opisuje unutrašnje stanje promenljivih i jednačina **ConstraintSolver** objekta
3. Pri UNDO operaciji, **ConstraintSolver** menja svoje unutrašnje stanje jer mu editor vraća **SolverState** objekat
4. Na osnovu informacije iz **SolverState** objekta, **ConstraintSolver** menja svoju unutrašnju strukturu i vraća jednačine i njihove promenljive u prethodno stanje

## ŠABLON MEMENTO: POSLEDICE I IMPLEMENTACIJA

*Šablon pojednostavljuje Originator objekat, ali upotreba Memento objekta može da zahteva veliku memoriju, a postoje i skriveni troškovi*

### Posledice:

Očuvanje granica učaurivanja: Sem Originator objekta, niko drugi ne može da dobije njegovo stanje od Memento objekta.

1. *Pojednostavljenje Originator objekta:* Ako klijenti upravljaju svojim stanjem, onda to pojednostavljuje Originator objekat i klijenti ne moraju da obaveštavaju Originator kada su obavili operaciju obnavljanja stanja.
2. *Upotreba Memento objekata može biti skupa.* Ako Originator mora da kopira i memoriše veliku količinu podataka u Memento objektu, ili ako klijenti često kreiraju i vraćaju Memento objekte Originator objektu
3. *Definisanje uskih i širokih interfejsa:* U nekim jezicima je teško obezbediti da samo Originator može da pristupi stanju Momento objekta
4. *Skriveni troškovi rada sa Memento objektima.* **CareTaker** je odgovoran da uništi **Memento** objekat o kome se brine. Međutim, on nema informaciju koliko je informacija u Memento objektu.

### Implementacija:

Treba imati u vidu dve stvari:

1. *Podrška programskom jeziku.* Memento objekti imaju dva interfejsa. Širok – za rad sa Originator objektom, i uski – za rad sa ostalim objektima.
2. *Memorisati inkrementalne promene.* Kada se Memento objekti kreiraju, i vraćaju svom Originator objektu, predvidiljivim redosledom, ona Memento može da memoriše samo inkrementalne promene unutrašnjeg stanja Originator objekta.

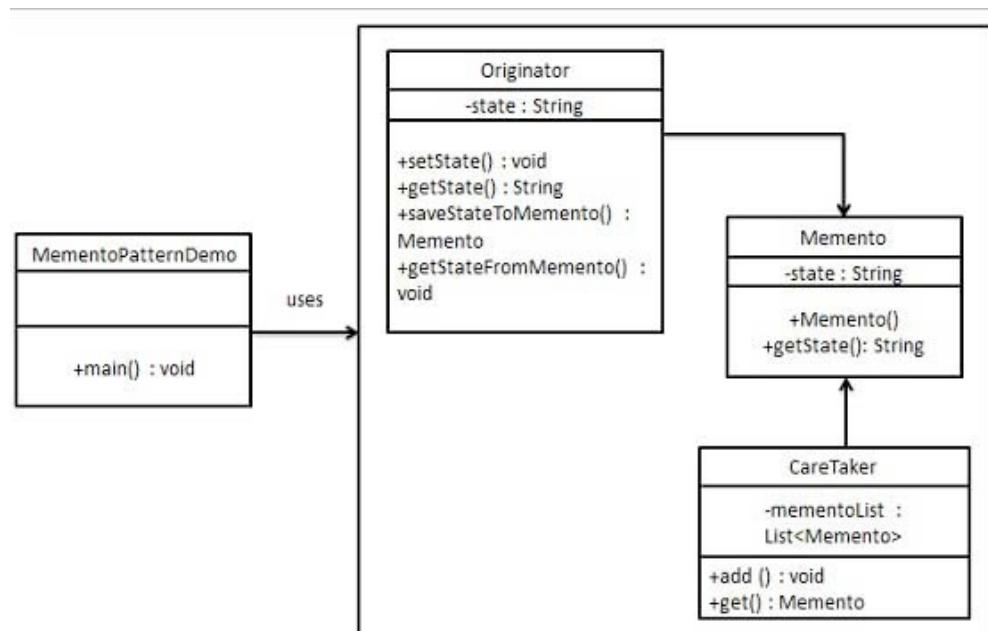
## ŠABLON MEMENTO: STRUKTURA

*Memento šablon koristi tri klase: Memento, CareTaker i MementoPatternDemo klasu.*

Memento šablon koristi tri klase.

1. **Memento** klasa sadrži unutrašnja stanja objekta Originator koji je potrebno kasnije obnoviti.
2. **CareTaker** objekat koji je odgovoran za povratak stanja objekta na osnovu stanja dobijenog od **Memento** objekta.
3. **MementoPatternDemo** klasa upotrebljava **CareTaker** i **Originator** objekte da obnovi stanja objekata.

#### Struktura šablon:



Slika 2.4 Struktura šablon [4]

## ŠABLON MEMENTO: POSTUPAK PRIMENE (1. I 2. KORAK)

*MementoPatternDemo klasa upotrebljava CareTaker i Originator objekte da obnovi stanja objekata, koja su privremeno memorisana u Memento objektu.*

#### Korak 1: Kreiranje **Memento** klase

```

public class Memento {
    private String state;

    public Memento(String state){
        this.state = state;
    }

    public String getState(){
        return state;
    }
}

```

```
}
```

## Korak 2: Kreiranje **Originator** klase

```
public class Originator {  
    private String state;  
    public void setState(String state){  
        this.state = state;  
    }  
    public String getState(){  
        return state;  
    }  
    public Memento saveStateToMemento(){  
        return new Memento(state);  
    }  
    public void getStateFromMemento(Memento Memento){  
        state = Memento.getState();  
    }  
}
```

## ŠABLON MEMENTO: POSTUPAK PRIMENE (3. I 4. KORAK)

*Kreiranje CareTaker klase i upotreba CareTaker i Originator objekata*

**Kreiranje 3:** Kreiranje **CareTaker** klase

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

public class CareTaker {
    private List<Memento> mementoList = new ArrayList<Memento>();

    public void add(Memento state){
        mementoList.add(state);
    }

    public Memento get(int index){
        return mementoList.get(index);
    }
}
```

**Korak 4:** Upotreba **CareTaker** i **Originator** objekata

```
public class MementoPatternDemo {
    public static void main(String[] args) {
        Originator originator = new Originator();
        CareTaker careTaker = new CareTaker();
        originator.setState("State #1");
        originator.setState("State #2");
        careTaker.add(originator.saveStateToMemento());
        originator.setState("State #3");
        careTaker.add(originator.saveStateToMemento());
        originator.setState("State #4");

        System.out.println("Current State: " + originator.getState());
        originator.getStateFromMemento(careTaker.get(0));
        System.out.println("First saved State: " + originator.getState());
    }
}
```

```
    originator.getStateFromMemento(careTaker.get(1));
    System.out.println("Second saved State: " + originator.getState());
}
}
```

## ŠABLON MEMENTO: POSTUPAK PRIMENE (5. I 6. KORAK)

### *Verifikacija i prikaz rezultata*

**Korak 5:** Verifikacija rezultata

Donja slika priznaje dobijene rezultata;

```
Playing mp3 file. Name: beyond the horizon.mp3
Playing mp4 file. Name: alone.mp4
Playing vlc file. Name: far far away.vlc
Invalid media. avi format not supported
```

Slika-5 Dobijen rezultat [4]

## MEMENTO DESIGN PATTERN (VIDEO)

*Trajanje: 20,29 minuta*

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

## ✓ Poglavlje 3

# Šablon State

## ŠABLON PONAŠANJA STATE

*Šablon State dozvoljava objektu da menja svoje ponašanje kada se promeni njegovo unutrašnje stanje.*

### Svrha šablonu:

Dozvoljava objektu da menja svoje ponašanje kada se promeni njegovo unutrašnje stanje.  
Objekat inicira promenu svoje klase.

### Motivacija (razlog kreiranja ovog šablonu):

Prepostavimo klasu **TCPConnection** koja predstavlja neku mrežnu vezu.

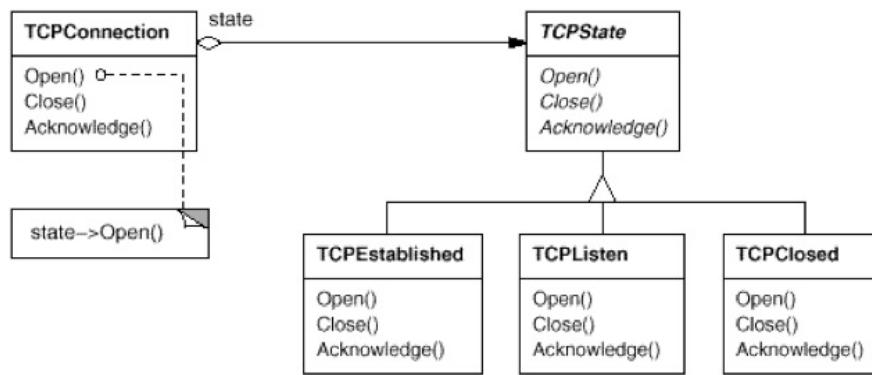
**TCPConnection** objekat može biti u jednom od stanja:

1. Established,
2. Listening,
3. Closed

**TCPConnection** različito odgovara na zahteve iz drugih objekata, zavisno od sopstvenog stanja

Ključna ideja: Koristiti apstraktnu klasu **TPVState** sa interfejsom koji je zajednički za sve klase koje predstavljaju različita operativna stanja. Specifično ponašanje **TCPState** klase se definiše odgovarajućom podklasom.

- Klasa **TCPConnection** održava stanje objekta (podklase klase **TCPState**) koji predstavlja trenutno stanje objekta **TCPConnection**.
- Klasa **TCPConnection** delegira ovom objektu sve zahteve koji su specifični od njenog stanja
- **TCPConnection** upotrebljava objekat svoje podklase klase **TCPState** radi izvršenja operacija koje su specifične za trenutno stanje mreže.
- Kod promene stanja, **TCPConnection** menja stanje objekta koji koristi.



Slika 3.1 Primer sa TCPConnection [4]

## ŠABLON STATE: PRIMENLJIVOST

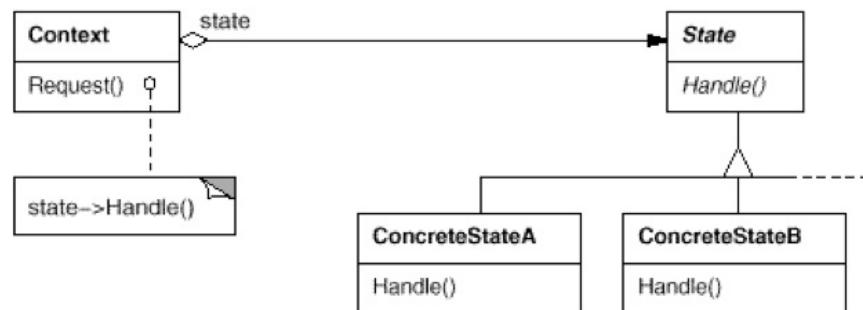
*Šablon State se koristi kada ponašanje objekta zavisi od sopstvenog stanja i mora da promeni svoje ponašanje u vreme izvršenja, a operacije...*

### Primenljivost:

Šablon State se koristi u sledećim slučajevima:

- Ponašanje objekta zavisi od sopstvenog stanja i mora da promeni svoje ponašanje u vreme izvršenja
- Operacije imaju velike, uslovne isklaze koji zavise od stanja objekta.

### Struktura:



Slika 3.2 Struktura klasa pri primeni šablon State [4]

### Učesnici:

**Context (npr., TCPConnection):** Definiše interfejs od interesa klijenata. Održava objekat **ConcreteState** podklase koja definiše trenutno stanje.

**State (npr. TCPState):** Definiše interfejs koji skriva ponašanje povezano sa posebnim stanjem **Context** objekta.

**ConcreteState** podklase: Svaka podklasa primenjuje ponašanje zavisno od stanja objekta **Context**.

**Kolaboracija:**

1. Kolaboracije delegiraju zahtev zavistan od stanja objekta **ConcreteStateObject**.
2. **Context** može da se prebaci kao argument objektu **State** koji obrađuje zahtev. Ovo dozvoljava **State** objektu pristup objektu **Context**.
3. **Context** je primarni interfejs za klijente koji mogu da konfigurišu **Context** sa **State** objektima. Kada je **Context** konfigurisan, njegovi klijenti ne treba da direktno komuniciraju sa **State** objektima. **Context** ili **ConcreteState** podklasa može da odluči koje stanje treba da sledi, a na osnovu određenih okolnosti.

## ŠABLON STATE: POSLEDICA PRIMENE I PRIMER

*Šablon omogućava lokalizaciju ponašanje uslovljeno stanjem i raspodeljuje ponašanja na različita stanja, čini eksplittnim promenu stanja i zajedničko ko*

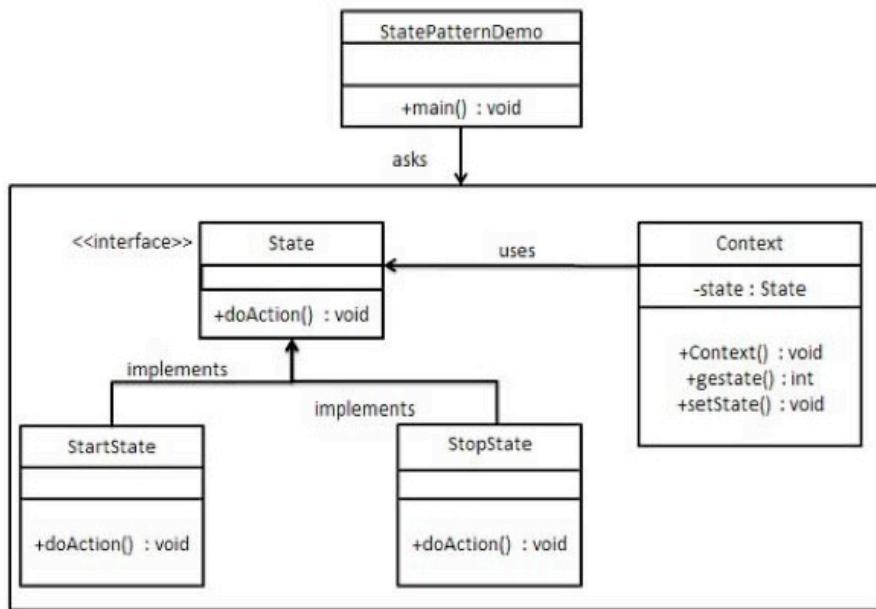
**Posledice:**

1. Lokalizuje ponašanje uslovljeno stanjem i raspodeljuje ponašanja za različita stanja. State šablon stavlja celo ponašanje koje je povezano sa određenim stanjem u jedan objekat. Na taj način lako je dodavati i menjati stanja i tranzicije korišćenjem podklasa.
2. Čini eksplicitnim tranzicije stanja.
3. **State** objekti se mogu deliti.

**Implementaciona pitanja:**

1. Ko definiše stanje tranzicija? Najotvoreniji pristup je da se to definiše u podklasi klase State.
2. Tabelarno definisati alternative. Za svako stanje, tabela definiše vezu ulaza i stanja tranzicija.
3. Kreiranje i uništavanje **State** objekata. Da li kreirati **State** objekte samo kada su potrebi, ili ih stalno imati?
4. Upotreba dinamičkog nasleđivanja, ako jezik to podržava.

**Primer za implementaciju:**



Slika 3.3 Struktura primera za impleentaciju [4]

## ŠABLON STATE: POSTUPAK PRIMERA (1., 2. I 3. KORAK)

*Kreiranje interfejsa, kreiranje konkretnih klasa koje pimenuju isti interfejs*

### Korak 1: Kreiranje interfejsa

```

public interface State {
    public void doAction(Context context);
}
    
```

### Korak 2: Kreiranje konkretnih klasa koje pimenuju isti interfejs

```

public class StartState implements State {

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

    public String toString(){
        return "Start State";
    }
}

public class StopState implements State {

    public void doAction(Context context) {
        System.out.println("Player is in stop state");
    }
}
    
```

```
        context.setState(this);
    }

    public String toString(){
        return "Stop State";
    }
}
```

**Korak 3:** Kreiranje **Context** klase

```
public class Context {
    private State state;

    public Context(){
        state = null;
    }

    public void setState(State state){
        this.state = state;
    }

    public State getState(){
        return state;
    }
}
```

## ŠABLON STATE: POSTUPAK PRIMERA (4. I 5. KORAK)

*Upotreba Context objekta da bi se videla promena ponašanja kada se State promeni i verifikacija rezultata*

**Korak 4:** Upotreba **Context** objekta da bi se videla promena ponašanja kada se **State** promeni.

```
public class StatePatternDemo {
    public static void main(String[] args) {
        Context context = new Context();
```

```
StartState startState = new StartState();
startState.doAction(context);

System.out.println(context.getState().toString());

StopState stopState = new StopState();
stopState.doAction(context);

System.out.println(context.getState().toString());
}

}
```

#### Korak 5: Verifikacija rezultata

Player is in start state

Start State

Player is in stop state

Stop State

## STATE DESIGN PATTERN (VIDEO)

*Trajanje: 20,50 minuta*

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

## VIDEO PREDAVANJE ZA OBJEKAT "ŠABLON STATE I ŠABLON STRATEGY"

*Trajanje video snimka: 25min 55sek*

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

## ✓ Poglavlje 4

# Šablon Strategy

## ŠABLON PONAŠNJA STRATEGY

*Šablon omogućava definisanje familije algoritama, učaurenje svakog od njih, i njihovu razmenljivost, a algoritmi se mogu nezavisno menjati od strane klijenta*

### Svrha šablon-a:

- Definisanje familije algoritama, učaurenje svakog od njih, i njihovu razmenljivost
- Strategy omogućava da se algoritam nezavisno menja od klijenata koji ga koriste.

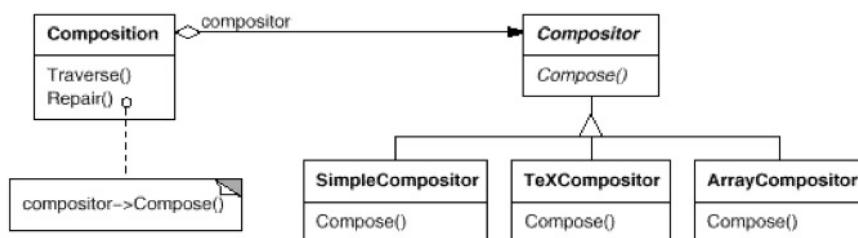
### Motivacija (razlog nastanka šablon-a):

Mnogi algoritmi pretvaraju tok nekog teksta u tekst sa linijama. Stavljanje ovih algoritma u klase koje ih koriste nije dobro:

- Klijenti koji žele linije dobijaju složeniji kod.
- Različiti algoritmi su odgovarajući za različita vremena.
- Teško je dodati novi algoritam i promeniti postojeći algoritam kada je korišćenje linija sa tekstovima integralni deo klijenta.

Ovi problemi se izbegavaju definisanjem klase koje učauruju različite algoritme za stavljanje teksta u linije, tj. u formatiran tekst.

Na slici 1 je prikazan tipičan slučaj.



Slika 4.1 Tipičan slučaj za primenu šablon-a Strategy [4]

**Klasa Composition** je odgovorna za održavanje i menjanje linija prikazanog teksta i prikazivača teksta.. Startegija deljenja teksta na linije se stavlja u podklase apstraktne klase **Composition**. Na slici su prikazane tri strategije rešavanja.

Kada **Composition** reformiše tekst, prebacuje odgovornost na klasu **Compositor**. Klijent bira objekat podklase **Compositor**

## ŠABLON STRATEGY: OPIS

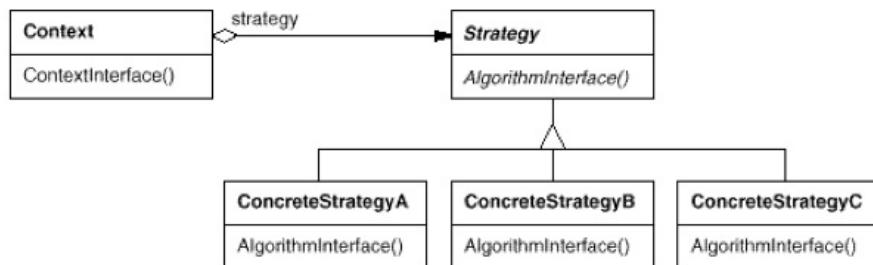
*Upotrebite Strategy šablon kada se povezane klase razlikuju samo u ponašanju, kada imate potrebu za različitim varijantama algoritma*

### Primenljivost:

Upotrebite Strategy šablon kada:

- se povezane klase razlikuju samo u ponašanju.
- imate potrebu za različitim varijantama algoritma.
- algoritam upotrebljava podatke koje klijent ne treba da zna.
- klasa definiše puno različitih ponašanja i to se onda u operacijama nalazi u iskazima sa više uslova. Svaki uslov zamenite odgovarajućom klasom Startegy šablona.

### Struktura:



Slika 4.2 Dijagram klasa primeri šablonu Strategy [4]

### Učesnici:

1. **Strategy (npr. Compositor)**: Deklariše zajednički interfejs za sve podržane algoritme. Context koristi ovaj interfejs da poziva željeni algoritam.
2. **ConcreteStrategy (npr. SimpleComposition, TexComposition, ArrayComposition)**: Implementira algoritam upotrebom Strategy interfejsa.
3. **Context (npr. Composition)**: Konfigurisan sa **ConcreteStrategy** objektom. Održava referencu na **Startegy** objekat. Definiše interfejs za pristup objekta **Strategy** svojim podacima.

### Kolaboracija:

1. **Strategy i Context** su u vezi da bi primenili izabran algoritam. **Context** objekat može da prenese sve zahtevane podatke u algoritam u **Startegy** objekat, a kada se poziva određen algoritam. Alternativno, **Context** može da se sam prebaci kao argument operacije **Strategy** klase.
2. **Context** prosleđuje zahteve svojih klijenata objektu **Strategy**, odn. **ConcreteStrategy**.

## ŠABLON STRATEGY: POSLEDICE

*Šablon omogućava primenu familije povezanih algoritama, alternativno korišćenje podklasa i zamenu uslovnih iskaza.*

### Posledice:

Startegy šablon omogućava sledeće dobre i loše posledice svoje primene:

1. Familije povezanih algoritama. Njih definiše hijerarhija podklasa **Strategy** klase.
  2. Alternativa korišćenju podklasa. Sa podklasom **Context** klase može se direktno uneti novo ponašanje. Ali, to ne može raditi dinamički, tj. u vreme izvršenje programa, već pre kompilacije. Stavljanjem algoritma u posebne **Strategy** klase omogućava vam da menjate algoritme nezavisno od njihovog **Context** objekta, što olakšava izbor, razumevanje i proširenje.
  3. Strategije zamenjuju uslovne iskaze. Stavljanjem ponašanja u poseban objekt **Strategy** klase, eliminiše ove uslovne iskaze.
- 
1. Izbor implementacije. Strategije omogućavaju različite implementacije istog ponašanja.
  2. Klijenti moraju da budu svesni različitim strategija. Ali, klijent mora da prethodno razume te strategije. Upotrebite **Strategy** šablon kada je to nevažno za klijente
  3. Komunikacioni trošak između **Strategy** u **Context** objekta. Ne koristite šablon ako je efikasnost bitna, te je čvršće povezivanje brže.
  4. Povećanje broja objekata. **Strategije** povećavaju broj objekata u nekoj aplikaciji.

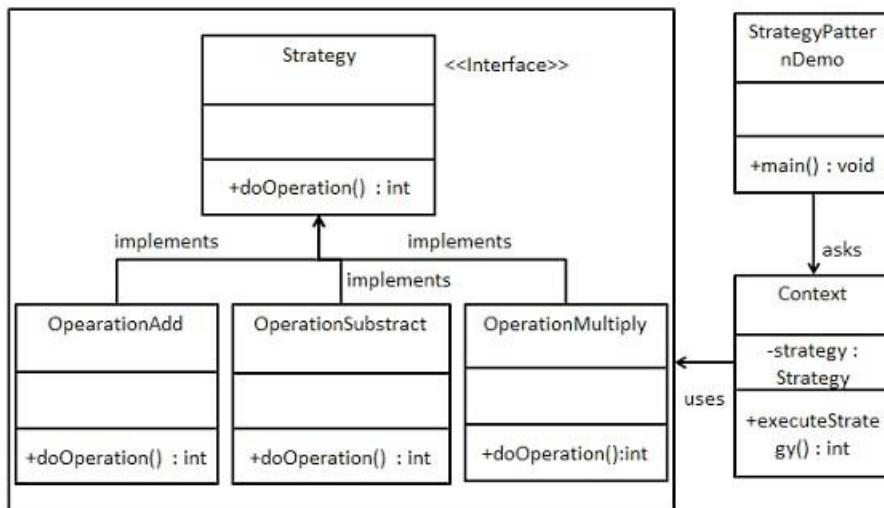
## ŠABLON STRATEGY: PRIMER I 1. KORAK REŠAVANJA

*Definisanje interfejsa **Strategy** i **Context** objekta koji daju objektu **ConcreteStrategy** efikasnost pristupa podacima iz **Context** objekta i obrnuto,*

### Implementaciona pitanja:

1. Definisanja interfejsa **Strategy** i **Context** objekata. Ovi interfejsi daju objektu **ConcreteStrategy** efikasnost pristupa podacima iz **Context** objekta, i obrnuto.
2. Strategije, predstavljene kao izabrani parametri.
3. Opciono stvaranje **Strategy** objekata

### Struktura primera implementacije:



Slika 4.3 Dijagram klasa u primeru implementacije [4]

#### Korak 1: Kreiranje interfejsa

```

public interface Strategy {
    public int doOperation(int num1, int num2);
}

```

## ŠABLON STRATEGY: POSTUPAK PRIMENE (2. I 3. KORAK)

*Kreiranje konkretnih klasa koje primenjuju isti interfejs i kreiranje Context klase*

#### Korak 2: Kreiranje konkretnih klasa koje primenjuju isti interfejs

```

public class OperationAdd implements Strategy{
    @Override
    public int doOperation(int num1, int num2) {
        return num1 + num2;
    }
}

```

```
    }
}

public class OperationSubtract implements Strategy{
    @Override
    public int doOperation(int num1, int num2) {
        return num1 - num2;
    }
}

public class OperationMultiply implements Strategy{
    @Override
    public int doOperation(int num1, int num2) {
        return num1 * num2;
    }
}
```

### Korak 3: Kreiranje **Context** klase

```
public class Context {
    private Strategy strategy;

    public Context(Strategy strategy){
        this.strategy = strategy;
    }

    public int executeStrategy(int num1, int num2){
        return strategy.doOperation(num1, num2);
    }
}
```

## ŠABLON STRATEGY: POSTUPAK PRIMENE (4. I 5. KORAK)

*Upotreba **Context** objekta radi prikaza ponašanja i verifikacija rezultata*

### Korak 4: Upotreba **Context** objekta radi prikaza ponašanja

```
public class StrategyPatternDemo {
    public static void main(String[] args) {
        Context context = new Context(new OperationAdd());
        System.out.println("10 + 5 = " + context.executeStrategy(10, 5));

        context = new Context(new OperationSubtract());
        System.out.println("10 - 5 = " + context.executeStrategy(10, 5));

        context = new Context(new OperationMultiply());
        System.out.println("10 * 5 = " + context.executeStrategy(10, 5));
    }
}
```

}

**Korak 5:** Verifikacija rezultata

$$\begin{aligned}10 + 5 &= 15 \\10 - 5 &= 5 \\10 * 5 &= 50\end{aligned}$$

Slika 4.4 Prikaz rezultata na monitoru računara [4]

## STRATEGY DESIGN PATTERN (VIDEO)

*Trajanje: 11,31 minuta*

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

## STRATEGY PATTERN (VIDEO)

*Georgia Tech predavanje - Startegz Pattern*

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

## PRIMER ZA STRATEGY PATTERN (VIDEO)

*Georgia Tech predavanje - Primer za Strategy Pattern*

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

## ✓ Poglavlje 5

### Šablon Observer

## ŠABLON PONAŠANJA OBESERVER

*Šablon omogućava definisanje "jedan-na-mnoge" zavisnost između objekata, tako da kada se jednom objektu promeni stanje, svi objekti zavisni od njega*

#### **Svrha šablon-a:**

Definisanje "jedan-na-mnoge" zavisnost između objekata, tako da kada se jedan objekt promeni stanje, svi objekti zavisni od njega se automatski obaveštavaju i menjaju.

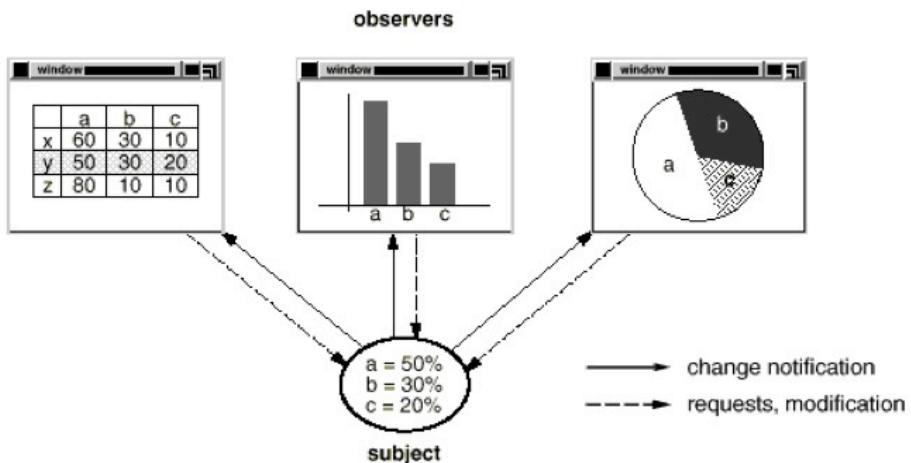
#### **Motivacija (razlog nastanka šablon-a):**

Pri deljenju sistema na kolekciju kooperišućih klasa potrebno je obezbititi održavanje konzistentnost povezanih objekata. Izbegava se čvrsto povezivanje objekata jer smanjuje višestruka upotrebljivost

Primer: Prezentacioni deo i aplikacioni podaci grafičkog korisničkog interfejsa. Ostvaruje se nezavisnost prezentacionog dela od dela sa podacima

Observer šablon opisuje kako da se postave ove relacije. Ključni objekti su **Subject** i **Observer**. Observer objekti se obaveštavaju uvek kada dođe do promene stanja **Subject** objekta. Zbog toga se koristi "publish-subscription" ("prijava publikovanja") interakcija. **Subject** objekat je izdavač obaveštenja. Ne zna ko su Observer objekti. Bilo koji broj **Observer** objekta mogu da se upišu u primaoca obaveštenja

Slika 1 prikazuje jedan tipičan slučaj kada je korisno primeniti šablon Observer. Isti podaci se prikazuju na tri različita načina. Ako klijent promeni neki od podataka u jednom od prikaza, neophodno je da sistem tu promenu prikaže i na duga dva prikaza.



Slika 5.1 Slučaj kada se primenjuje šablon Observer [4]

## ŠABLON OBSERVER: OPIS

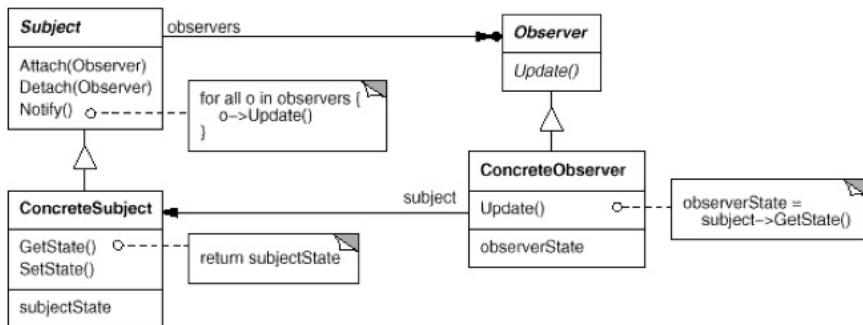
*Primenjuite šablon kada apstrakcija ima dva aspekta, i kada jedan zavisi od drugoga i kada promena jednog objekta zahteva promenu drugih objekata*

### Primenljivost:

Primenjujte ova šablon:

- Kada abstrakcija ima dva aspekta, i kada jedan zavisi od drugoga
- Kada promena jednog objekta zahteva promenu drugih objekata, a vi ne znate koliko je takvih objekata
- Kada bi trebalo da jedan objekat da obavesti druge objekte bez potreba da zna ko su ti objekti ne želite da ovi objekti budu čvrsto povezani

### Struktura:



Slika 5.2 Struktura klasa kada se primenjuje pablon Observer [4]

### Učesnici:

1. **Subject:** Zna svoje **Observer** objekte. Njihov broj nije ograničen.

2. Obezbeđuje interfejs za dodavanje ili uklanjanje **Observer** objekti
3. **Observer:** Definiše interfejs za objekte koji treba da se obaveste kada se menja Subject objekat.
4. **ConcreteSubject:** Memoriše stanje **ConcreteSubject** objekata i šalje obaveštenje svojim **Observer** objekatima kada dođe do promene njegovog stanja
5. **ConcreteObserver:** Održava referencu na **ConcreteSubject** objekta, memoriše stanje koje treba da ostane konzistentno stanje **Subject** objekta i Implementira **Observer** interfejs radi održavanje svog konsistentnog stanja sa stanjem **Subject** objekta

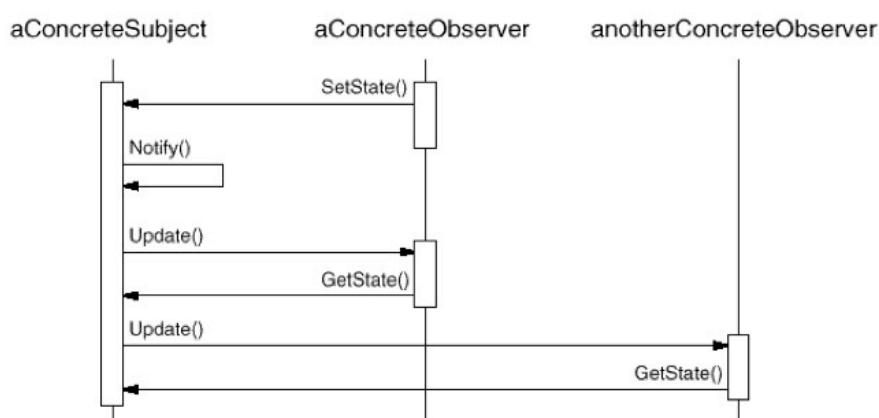
## ŠABLON OBSERVER: KOLABORACIJA I POSLEDICE

*Observer šablon omogućava vam da nezavisno menjate Subject objekte i Observer objekte. Možete dodati Observer objekte bez promene Subject objekta*

### Kolaboracije:

1. **ConcreteSubject** obaveštava svoje **ConcreteObserver** objekte uvek kada dođe do promene njegovog stanja
2. Kada dobije informaciju o promeni **ConcreteSubject** objekta, **ConcreteObserver** objekat može ispitivati **ConcreteSubject** za informaciju. **ConcreteObserver** koristi ovu informaciju da obnovi stanje sa stanjem **ConcreteSubject** objekta.

### Dijagram kolaboracije:



Slika 5.3 Dijagram interakcije koji opisuje kolaboraciju angažovanih objekata [4]

### Posledice:

Observer šablon omogućava vam da nezavisno menjate **Subject** objekte i **Observer** objekte. Možete dodati **Observer** objekte bez promene **Subject** objekata ili drugih **Observer** objekata

Ostale prednosti:

1. Apstraktno povezivanje između **Subject** i **Observer** objekata. Svi **Subject** objekti imaju svoju listu **Observer** objekata. Svaki ima interfejs apstraktne **Observer** klase. **Subject** ne zna konkretnu klasu **Observer** interfejsa. Ovakvo povezivanje je apstraktno i minimalno.
2. *Podrška emitovanja komunikacije.* Obaveštenje ne sadrži adresu primaoca. Automatski se emituje svim zainteresovanim objektima koji su prijavljeni za prijem poruka.
3. *Neočekivane promene.* Otežana je kontrola automatskog širenja promena povezanih objekata.

## ŠABLON OBSERVER: IMPLEMENTACIJA

*Postoji niz pitanja na koja morate da obratite pažnju pri implementaciji ovog šablonu.*

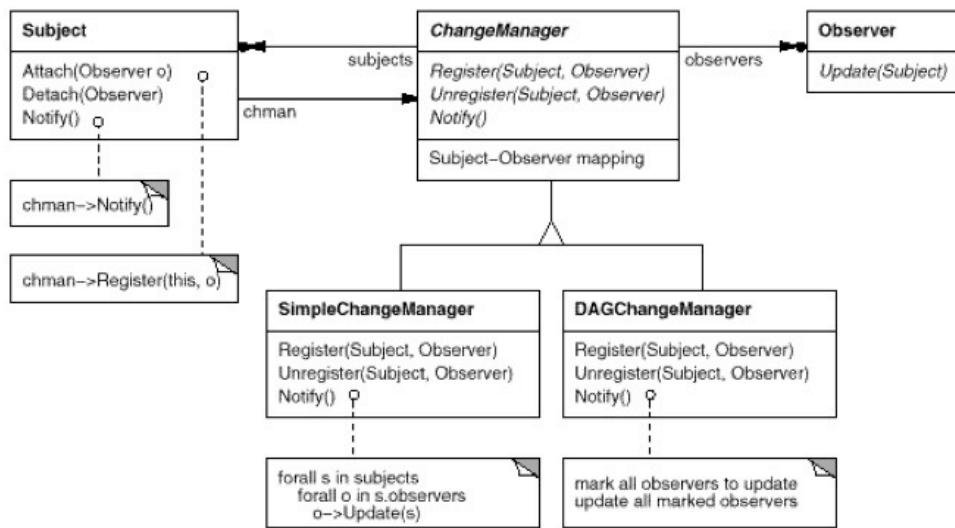
### Problemi implementacije:

Pri implementaciji (primeni) šablonu Observer, morate obratiti pažnju na sledeća pitanja, za koje morate da nađete zadovoljavajuće odgovore, tj. rešenja.:

1. Mapiranje **Subject** objekata u njihove **Observer** objekte.
2. Praćenje više od jednog objekta.
3. Ko pokreće promene?
4. Apsolutne reference uopštenih Subjekt objekata
5. Obezbeđenje da se stanje **Subject** objekta samostalno usklađuje i pre obaveštavanja.
6. Izbegavanje promena protokola specifičnih za **Observer** objekte.
7. Eksplicitna specifikacija modifikacije interesa.
8. Učaurenje složene semantike promena.
9. Kombinovavanje **Subject** i **Observer** klase.

### Struktura primera implementacije:

Na slici 4 prikazan je dijagram klasa jednog primera u kome se primenjuje šablon Observer, radi ilustracije.

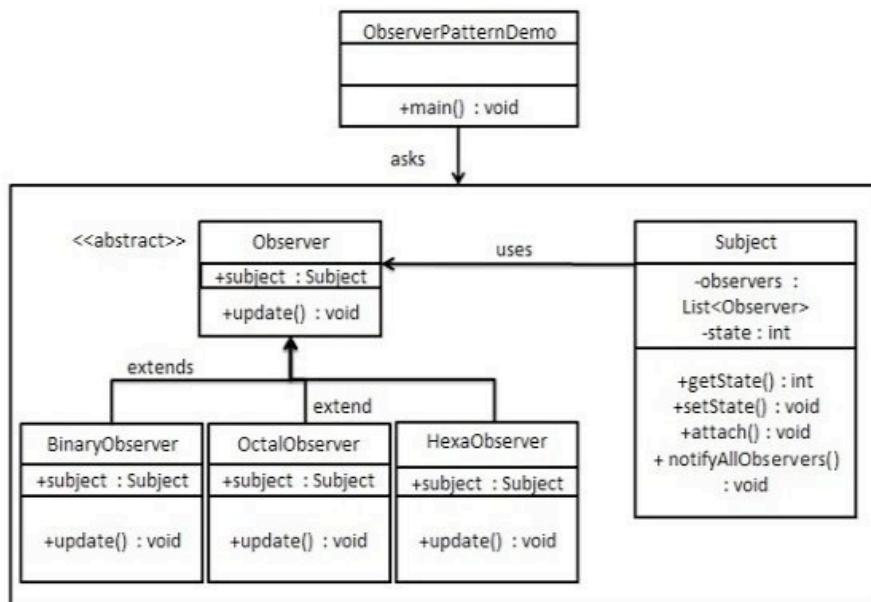


Slika 5.4 Struktura primera implementacije [4]

## ŠABLON OBSERVER: PRIMER I 1. KORAK REŠAVANJA

*Kreiranje Subject klase za dati problem*

**Struktura:**



Slika 5.5 Struktura klasa pri primeni šablonu Observer [4]

**Korak 1:** Kreiranje **Subject** klase

```

import java.util.ArrayList;
import java.util.List;
public class Subject {
    private List<Observer> observers
  
```

```
= new ArrayList<Observer>();  
private int state;  
  
public int getState() {  
    return state;  
}  
public void setState(int state) {  
    this.state = state;  
    notifyAllObservers();  
}  
public void attach(Observer observer){  
    observers.add(observer);  
}  
public void notifyAllObservers(){  
    for (Observer observer : observers) {  
        observer.update();  
    }  
}  
}
```

## ŠABLON OBSERVER: 2. I 3. KORAK

*Kreiranje Observer klase i kreiranje konkretne Observer klase*

**Korak 2:** Kreiranje **Observer** klase

```
public abstract class Observer {  
    protected Subject subject;  
    public abstract void update();  
}
```

**Korak 3:** Kreiranje konkretne **Observer** klase

```
public class BinaryObserver extends Observer{

    public BinaryObserver(Subject subject){
        this.subject = subject;
        this.subject.attach(this);
    }

    @Override
    public void update() {
        System.out.println("Binary String: "
            + Integer.toBinaryString(subject.getState()) );
    }
}

public class OctalObserver extends Observer{

    public OctalObserver(Subject subject){
        this.subject = subject;
        this.subject.attach(this);
    }

    @Override
    public void update() {
        System.out.println("Octal String: "
            + Integer.toOctalString(subject.getState()) );
    }
}

public class OctalObserver extends Observer{

    public OctalObserver(Subject subject){
        this.subject = subject;
        this.subject.attach(this);
    }

    @Override
    public void update() {
        System.out.println("Octal String: "
            + Integer.toOctalString(subject.getState()) );
    }
}
```

## ŠABLON OBSERVER: 4. I 5. KORAK

*Upotreba Subject i konkretni Object objekata i verifikacija rezultata*

**Korak 4:** Upotreba **Subject** i konkretni **Object** objekata

```
public class ObserverPatternDemo {  
    public static void main(String[] args) {  
        Subject subject = new Subject();  
  
        new HexaObserver(subject);  
        new OctalObserver(subject);  
        new BinaryObserver(subject);  
  
        System.out.println("First state change: 15");  
        subject.setState(15);  
        System.out.println("Second state change: 10");  
        subject.setState(10);  
    }  
}
```

**Korak 5:** Verifikacija rezultata

```
First state change: 15  
Hex String: F  
Octal String: 17  
Binary String: 1111  
Second state change: 10  
Hex String: A  
Octal String: 12  
Binary String: 1010
```

Slika 5.6 Prikaz dobijenih rezultata na monitoru računara

## OBSERVER DESIGN PATTERN (VIDEO)

*Trajanje: 22,26 minuta*

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

## VIDEO PREDAVANJE ZA OBJEKAT "ŠABLON OBSERVER I ŠABLON VISITOR"

*Trajanje video snimka: 31min 19sek*

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

# ✓ Poglavlje 6

## Šablon Visitor

### ŠABLON PONAŠANJA VISITOR

*Visitor vam omogućava da definišete novu operaciju bez promene klase elemenata na koje se izvršava*

#### Svrha šablon-a:

Predstavljanje operacije koja se izvršava na elemente strukture objekata.. Visitor vam omogućava da definišete novu operaciju bez promene klase elemenata na koje se izvršava

#### Motivacija (razlog nastanka šablon-a):

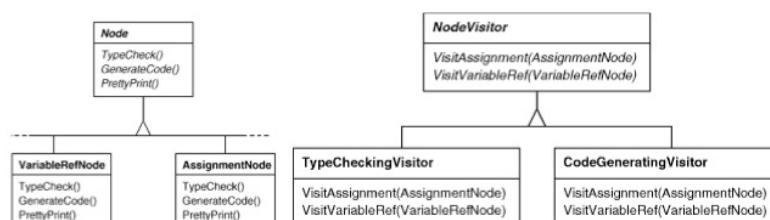
Prepostavimo kompjajler koji predstavlja programe kao apstraktna sintaksna stabla. Treba da ima više operacija za:

- proveru definisanih tipova
- optimizaciju programskega koda
- analizu toka
- analizu dodeljivanja vrednosti promenljivih

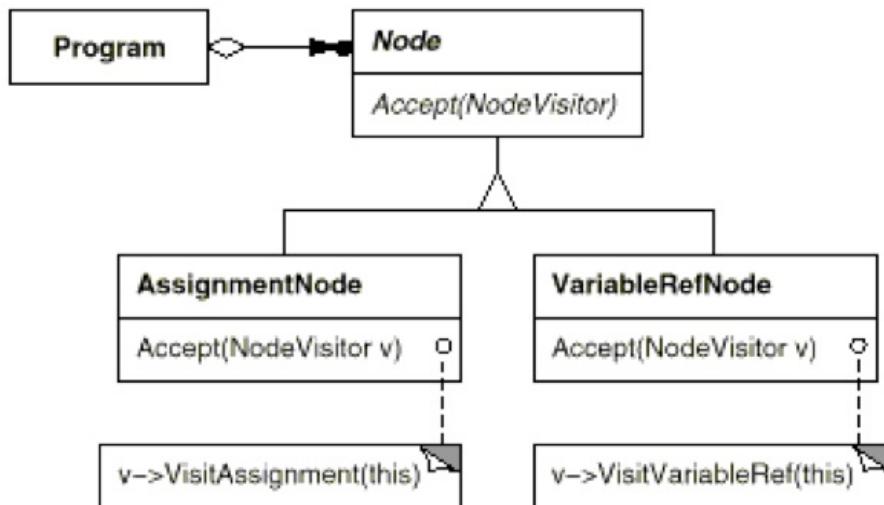
Za svaku operaciju neophodna je posebna klasa u čvoru (Slika 1). Ovakav sistem sa hijerarhijom čvorova je tažak za razumevanje, održavanje i menjanje. Zato se pravi NodeVisitor deklariše operaciju za svaki klasu čvora. Visitor šablon učauroju operacije za svaki fazu kompilacije u Visitor klase za tu klasu. Sa Visitor šablonom, definisite dve hijerarhije klase za:

- elemente koji deluju na hijerarhiju čvorova
- Visitor elemente koji definišu operacije na elemente NodeVisitor hijerarhije.

Možete dodati novu operaciju definisanjem nove podklase u hijerarhiji Visitor klasa.



Slika 6.1 Deo stabla sa čvorovima [4]



Slika 6.2 Primena šablonu Visitor [4]

## ŠABLON VISITOR: PRIMENLJIVOST I STRUKTURA

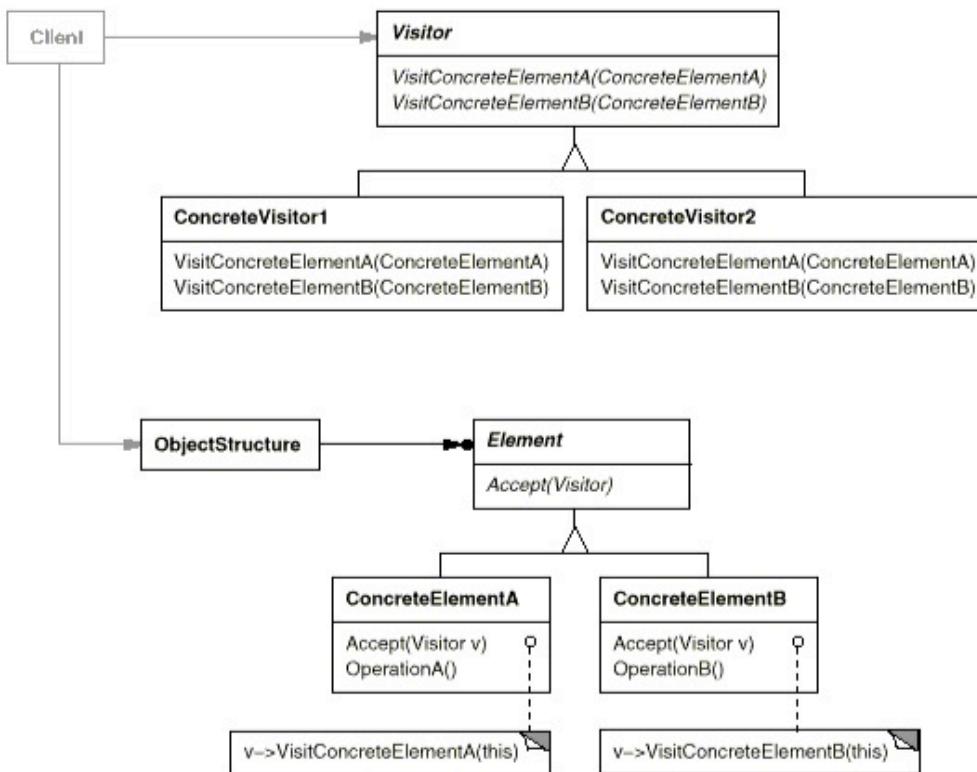
*Koristite Visitor šablon kada struktura objekata sadrži mnogo klase objekata sa mnogo različitih interfejsa, kada postoji mnogo nepovezanih operacija*

### Primenljivost:

Koristite Visitor šablon kada:

- struktura objekata sadrži mnogo klase objekata sa mnogo različitih interfejsa,
- postoji mnogo različitih i nepovezanih operacija nad objektima strukture, a želite da izbegnete "zagadjenje" njihovih klasa sa ovim operacijama; Kada istu strukturu objekata koristi više operacija, iskoristite Visitor šablon da bi stavili ove operacije samo kod onih klasa koje ih koriste.
- klase koje definišu strukturu objekata se retko menjaju, ali vi često želite da definisete nove operacije nad ovom strukturu.

### Struktura:



Slika 6.3 Struktura klasa pri primeni šablona Visitor

## ŠABLON VISITOR: UČESNICI I KOLABORACIJA

*Klijent kreira ConcreteVisitor objekat koji prelazi po strukturi objekata, posećujući svaki element sa Visitor operacijom, radi utvrđivanja stanja Elementa*

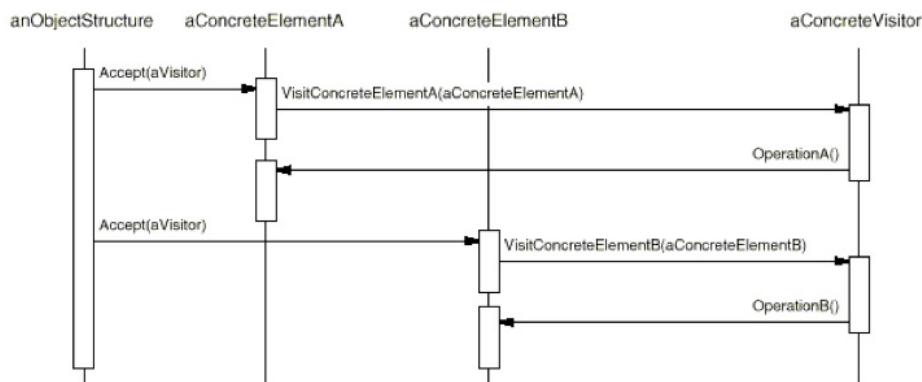
### Učesnici:

- Visitor (npr. NodeVisitor)**: Deklarše neku `Visit` operaciju za svaku klasu **ConcreteElement** u strukturi objekata. Potpis i ime operacije identificuje klasu koja šalje zahtev **Visitor** objektu. Tada **Visitor** može da direktno pristupi elementu preko njegovog posebnog interfejsa.
- ConcreteVisitor (npr. TypeCheckingVisitor)**: Primjenjuje svaku operaciju koja je deklarisana od strane **Visitor** klase. Svaka operacija primjenjuje deo algoritma definisanog za odgovarajuću klasu objekta strukture. **ConcreteVisitor** memorise lokalno stanje. Ono akumulira rezultate za vreme prelaženja objekata strukture.
- Element (npr. Node)**: Definiše `Accept` operaciju koja koristi **Visitor** objekat kao argument
- ConcreteElement (npr. AssignmentNode, VariableRefNode)**: Primjenjuje `Accept` operaciju koja uzima **Visitor** objekat kao argument
- ObjectStructures (npr. Program)**: Može da nabroji svoje elemente. Može da obezbedi interfejs kojim **Visitor** pristupa njegovim elementima. Može biti mešavina ili kolekcija

### Kolaboracija:

1. Klijent koji koristi **Visitor** šablon mora da kreira **ConcreteVisitor** objekat koji prelazi po strukturi objekata, posećujući svaki element sa visitor operacijom.
2. Kada poseti element, on poziva **Visitor** operaciju koja odgovara toj klasi. Element je argument te operacije i dozvoljava joj da pristupe njegovom stanju, ako je potrebno.

### Dijagram interakcije:



Slika 6.4 Dijagram interakcije objekata u kolaboraciji [4]

## ŠABLON VISITOR: POSLEDICE

*Svaka struktura objekata koristi pridodatu Visitor klasu. Ona deklariše **VisitConcreteElement** operaciju za svaku klasu **ConcreteElement** koja definiše*

### Posledice:

1. *Visitor lako dodaje nove operacije* Sa novim Visitor objektom, lako dodajete novu operaciju.
2. *Visitor prikuplja potrebne operacije i odvaja ih od nepotrebnih.* To uprošćuje i klase koje definišu elemente, a i algoritme koje definiše Visitor objekti.
3. *Dodavanje novih ConcreteElement klasa je teško.* Visitor šablon otežava dodavanje novih podklasa klasi Element.
4. *Pristup hijerarhiji klasa.* Alternativno, može se koristiti i Iterator objekt, ali on se koristi za pristup elementima strukture koji su različitog tipa.
5. *Akumulacija stanja.* Visitor objekti mogu da akumuliraju stanja elemenata koje su posetili, prenoseći jedan dodatni argument u operacijama koje sprovode, ili da za ovo koriste neku globalnu promenljivu.
6. *Razbijanje učarenja.* Šablon zahteva korišćenje nekih javnih operacija, što ruši učarenje

### Implementacija:

Svaka struktura objekata koristi pridodatu **Visitor** klasu. Ona deklariše **VisitConcreteElement** operaciju za svaku klasu **ConcreteElement** koja definiše strukturu

objekata. Svaka **Visit** operacija deklariše argument koji čini poseban **ConcreteElement**, što joj dozvoljava direktni pristup interfejsu klasi **ConcreteElement**. Klase **ConcreteElement** prekriva svaku **Visit** operaciju da bi primenio operaciju posetioca.

#### Pitanja za implementaciju:

1. *Duplo obaveštenje.* Operacija koja se mora izvršiti zavisi od tipa Visitor objekta i tipa Element objekta koju posećuje.
2. *Ko je odgovoran za prelazak po strukturi objekata?* Visitor mora da poseti svaki objekat strukture. Kako on tamo dolazi? Ta odgovornost se može dodeliti:
  3. strukturi objekata
  4. Visitor objektu
  5. posebnom iteratoru objekata

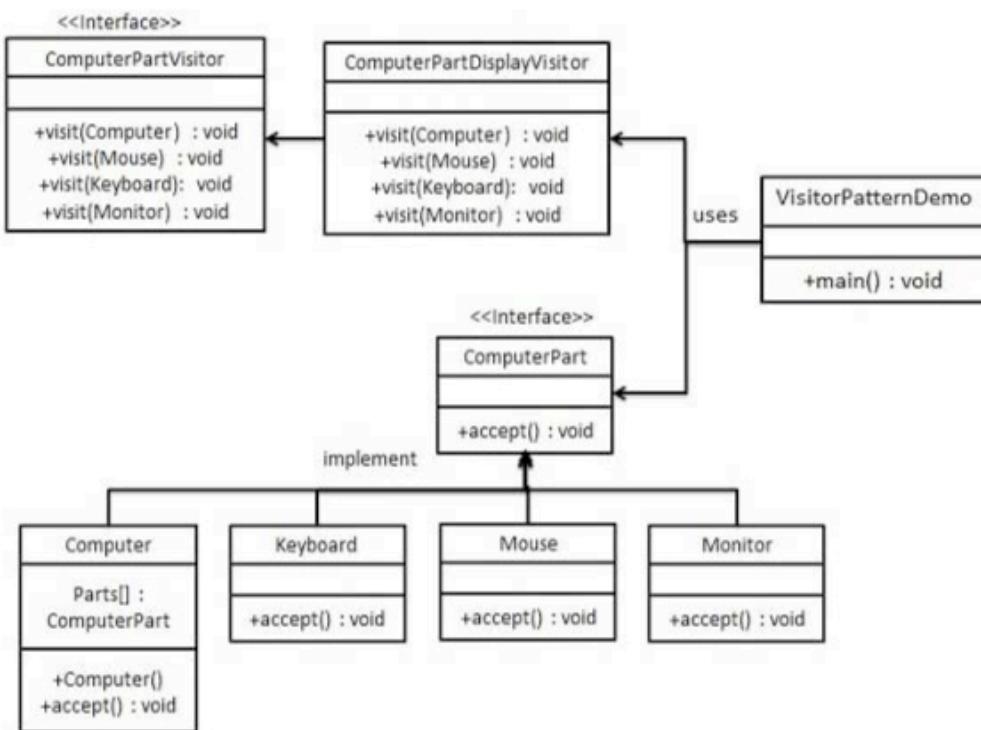
## ŠABLON VISITOR: PRIMER

*Kreirati ComputerPart interfejs koji definiše accept() operaciju, i konkretne klase Keyboard, Mouse, Monitor i Computer primenjuju ComputerPart interface*

#### Primer za implementaciju:

- Kreirati **ComputerPart** interfejs koji definiše **accept()** operaciju.
- Konkretne klase **Keyboard**, **Mouse**, **Monitor** i **Computer** primenjuju **ComputerPart** interfejs. Drugi interfejs, **ComputerPartVisitor** definiše operacije Visitor klase.
- **Computer** upotrebljava konkretni **Visitor** objekat za sprovođenje odgovarajuće akcije
- **VisitorPatternDemo** je demo klasa koja primenjuje Visitor šablon.

#### Dijagram klasa primera za implementaciju:



Slika 6.5 Dijagra klasa primjera implementacije šablona Visitor

## ŠABLON VISITOR: POSTUPAK PRIMENE (1. I 2. KORAKA)

*Definisanje interfejsa elementa i kreiranje konkretnih klasa*

**Korak 1:** Definisanje interfejsa elementa

```

public interface class ComputerPart {
    public void accept(ComputerPartVisitor computerPartVisitor);
}
  
```

**Korak 2:** Kreiranje konkretnih klasa

```

public class Keyboard implements ComputerPart {

    @Override
    public void accept(ComputerPartVisitor computerPartVisitor) {
        computerPartVisitor.visit(this);
    }
}

public class Monitor implements ComputerPart {

    @Override
    public void accept(ComputerPartVisitor computerPartVisitor) {
        computerPartVisitor.visit(this);
    }
}
  
```

```
}

public class Mouse implements ComputerPart {

    @Override
    public void accept(ComputerPartVisitor computerPartVisitor) {
        computerPartVisitor.visit(this);
    }
}

public class Computer implements ComputerPart {

    ComputerPart[] parts;

    public Computer(){
        parts = new ComputerPart[] {new Mouse(), new Keyboard(), new Monitor()};
    }

    @Override
    public void accept(ComputerPartVisitor computerPartVisitor) {
        for (int i = 0; i < parts.length; i++) {
            parts[i].accept(computerPartVisitor);
        }
        computerPartVisitor.visit(this);
    }
}
```

## ŠABLON VISITOR: POSTUPAK PRIMENE (3. I 4. KORAKA)

*Definisanje interfejsa koji predstavlja Visitor klasu i kreiranje konkretne visitor klase*

**Korak 3:** Definisanje interfejsa koji predstavlja Visitor klasu

```
public interface ComputerPartVisitor {  
    public void visit(Computer computer);  
    public void visit(Mouse mouse);  
    public void visit(Keyboard keyboard);  
    public void visit(Monitor monitor);  
}
```

**Korak 4:** Kreiranje konkretne visitor klase

```
public class ComputerPartDisplayVisitor implements ComputerPartVisitor {  
    @Override  
    public void visit(Computer computer) {  
        System.out.println("Displaying Computer.");  
    }  
    @Override  
    public void visit(Mouse mouse) {  
        System.out.println("Displaying Mouse.");  
    }  
    @Override  
    public void visit(Keyboard keyboard) {  
        System.out.println("Displaying Keyboard.");  
    }  
    @Override  
    public void visit(Monitor monitor) {  
        System.out.println("Displaying Monitor.");  
    }  
}
```

## ŠABLON VISITOR: POSTUPAK PRIMENE (5. I 6. KORAKA)

*Upotreba ComputerPartDisplayVisitor za prikaz delova Computer objekta i verifikacija rezultata*

**Korak 5:** Upotreba **ComputerPartDisplayVisitor** za prikaz delova **Computer** objekta

```
public class VisitorPatternDemo {  
    public static void main(String[] args) {  
  
        ComputerPart computer = new Computer();  
        computer.accept(new ComputerPartDisplayVisitor());  
    }  
}
```

#### Korak 6: Verifikacija rezultata

Displaying Mouse.  
Displaying Keyboard.  
Displaying Monitor.  
Displaying Computer.

Slika 6.6 Prikaz rezultata [4]

## VISITOR DESIGN PATTERN (VIDEO)

*Trajanje: 13,30 minuta*

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

## ✓ Poglavlje 7

### Pokazna vežba

#### PROJEKTOVANJE PONAŠANJA SISTEMA

*U nastavku su pokazni primeri koji primenjuju različite obrasce projektovanja prilikom modelovanja ponašanja definisanog sistema.*

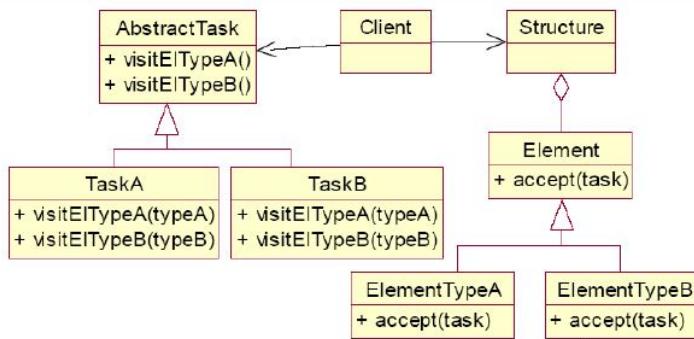
#### ✓ 7.1 Pokazni primer 1 - Vežba za šablon Visitor

#### VISITOR ŠABLON PONAŠANJA

*Način korišćenja i upotreba šablona Visitor. Trajanje: 20 minuta*

Na slici je prikazan UML dijagram Design patterna Visitor. Ovaj Design Pattern pomaže prilikom rasterećivanja klase dodatnim zaduženjima. Ako nam treba da dodamo extra metode na postojeću klase to može da se uradi i upotrebom ovog Design Patterna. Kao što se vidi iz samog dijagrama ovde će biti zastupljeni svi osnovni principi objektno orijentisanog programiranja. Element ima svoje konkretnе implementacije i AbstractTask ima svoje konkretnе implementacije TaskA i TaskB. Dakle građenje hierarhije. Takođe se primećuje da se metoda accept iz klase Element Override-uje u podklasama kao što se i metode visitElTypeA() i visitElTypeB() iz apstraktne klase AbstractTask Override-ju u klasama TaskA i TaskB. Iz samog Dijagrama se vidi da nam Vizitor obezbeđuje mehanizam zahvaljujući kom možemo da pokrenemo metod koji će se odnositi na drugu klasu. Ovo je pogotovu korisno kad treba da kombinujemo ponašanje jedne grupe klasa sa drugom grupom. Bez ovog Design Patterna morali bi da kreiramo onoliko klasa koliko postoji kombinacija. Ovako je dovoljno da kreiramo samo klase koje se kombinuju.

Design Pattern Visitor ima nekoliko karakterističnih osobina koje omogućavaju dodavanje funkcionalnosti na više različitih klasa. Tom prilikom možemo da koristimo mehanizam posećivanja klasa radi uzimanja podataka-informacija. Ovaj Design Pattern nam olakšava posao dodavanja novih funkcija i jedino na šta mora da se vodi računa je pravilna enkapsulacija.



Slika 7.1.1 UML prikaz šabloni Visitor [Izvor: Autor]

## VISITOR INTERFEJSI

### *Kreiranje interfejsa Visitor (Task) & Interface Pica (element)*

Kao primer upotrebe Visitor Design Patterna napravićemo program koji će da simulira rad tri picerije i tri različita načina dostave pice. Na ovom primeru ćemo videti kako Design Pattern Visitor može da izvede kombinaciju klasa radi postizanja potrebnog efekta. Počinjemo od interface-a Visitor i interface Pizza. Ova dva interface-a će definisati ponašanje klasa koje će predstavljati konkretnе picerije (interface Pizza) i konkretnе načine naručivanja pice (interface Visitor). Svaka picerija mora da ima implementiranu metodu order() koja će da daje željeni način naručivanja a svaki posetioč (klase koje implementiraju interface Visitor) metodu visit koja kao parametar prima interface Pizza odnosno tokom rada će joj biti prosleđena konkretna Piceri-a.

```

interface Visitor
{
    public void visit(Pizza p);
}
interface Pizza
{
    public String order();
}

```

## VISITOR IMPLEMENTACIJA

### *Konkretna implementacija klase PizzaPlanet, PizzaHut, Maestro, ByPickup datog primera*

#### **PizzaPlanet (element koncretan)**

Prvo ćemo kreirati klasu PizzaPlanet tako što ćemo implementirati interface Pizza. Klasa PizzaPlanet ima jedan podatak to je naziv picerije i jednu metodu koja je diktirana od strane interface-a Pizza. Metoda se zove order() i vraća naziv picerije od koje je naručena pica.

```
class PizzaPlanet implements Pizza
{
final String name = "PizzaPlanet";
public String order()
{
return name;
}
```

### PizzaHut (element konkretan)

Kreiramo sledeću piceriju u obliku klase PizzaHut tako što ćemo implementirati interface Pizza. Klasa PizzaHut ima jedan podatak to je naziv picerije i jednu metodu koja je diktirana od strane interface-a Pizza. Metoda se zove order() i vraća naziv picerije od koje je naručena pica.

```
class PizzaHut implements Pizza
{
final String name = "PizzaHut";
public String order()
{
return name;
}
```

### Maestro (konkretan element)

Kreiramo sledeću piceriju u obliku klase Maestro tako što ćemo implementirati interface Pizza. Klasa Maestro ima jedan podatak to je naziv picerije i jednu metodu koja je diktirana od strane interface-a Pizza. Metoda se zove order() i vraća naziv picerije od koje je naručena pica.

```
class Maestro implements Pizza
{
final String name = "Maestro";
public String order()
{
return name;
}
```

### ByPickup (visitor)

Kreramo Visitore. Počnjemo od prvog načina naručivanja pice a to je uzimanjem pice za poneti koji je predstavljen klasom ByPickup. Ova klasa kao i svi ostali posetnici mora da implementira interface

**Visitor.** Od podataka ova klasa ima ime i naziv metoda naručivanja pice. Ime je promenljiva koja predstavlja piceriju i koju dobijamo pozivajući metodu order picerije koju smo dobili kroz metod visit. Metod visit prima interface Pizza. Ovaj metod je tražen od strane interface Visitor.

```
class ByPickup implements Visitor {
private String name;
```

```
private final String method = "By pick up";
public void visit(Pizza p) {
name = p.order();
}
public String toString() {
return name + " " + method;
}
}
```

## VISITOR IMPLEMENTACIJA NASTAVAK

### *Implementacija klasa ByEatin, ByDelivery, Dinner*

#### **ByEatin (visitor)**

Kreramo sledeći Visitor. Idemo sa sledećim načinom naručivanja pice a to je jedenjem pice u piceriji koji je predstavljen klasom ByEatin. Ova klasa kao i svi ostali posetnici mora da implementira interface Visitor. Od podataka ova klasa ima ime i naziv metoda naručivanja pice. Ime je promenljiva koja predstavlja piceriju i koju dobijamo pozivajući metodu order picerije koju smo dobili kroz metod visit. Metod visit prima interface Pizza. Ovaj metod je tražen od strane interface Visitor.

```
class ByEatin implements Visitor {
private String name;
private final String method = "By eat in";
public void visit(Pizza p) {
name = p.order();
}
public String toString() {
return name + " " + method;
}
}
```

#### **ByDelivery (visitor)**

Kreramo sledeći Visitor. Kreiramo poslednji način naručivanja pice a to je donošenjem pice na kućnu adresu koji je predstavljen klasom ByDelivery. Ova klasa kao i svi ostali posetnici mora da implementira interface Visitor. Od podataka ova klasa ima ime i naziv metoda naručivanja pice. Ime je promenljiva koja predstavlja piceriju i koju dobijamo pozivajući metodu order picerije koju smo dobili kroz metod visit. Metod visit prima interface Pizza. Ovaj metod je tražen od strane interface Visitor.

```
class ByDelivery implements Visitor {
private String name;
private final String method = "By delivery";
public void visit(Pizza p) {
name = p.order();
}
public String toString() {
return name + " " + method;
}
}
```

```
}
```

### Dinner (client)

Spajamo sve ovo u strukturu. Klasa Dinner nam predstavlja davalac informacija. Ova klasa ima dve metode jedna getDinner nam vraća slučajno odabranu piceriju. Druga metoda howto nam vraća slučajan odabir načina naručivanja pice. Kao što se vidi iz koda ova klasa nam je pomoćna klasa koja će nam služiti za testiranje rada visitora.

```
class Dinner {  
    public Pizza getDinner() {  
        switch ((int) (Math.random() * 3))  
        {  
            case 0: return new PizzaPlanet();  
            case 1: return new PizzaHut();  
            case 2: return new Maestro();  
            default: return null;  
        }  
    }  
    public Visitor howto() {  
        switch ((int) (Math.random() * 3))  
        {  
            case 0: return new ByPickup();  
            case 1: return new ByEatin();  
            case 2: return new ByDelivery();  
            default: return null;  
        }  
    }  
}
```

## KORIŠĆENJE VISITOR ŠABLONA

### *Način upotrebe kreiranog primera šablona Visitor*

Testiramo rad Design Patterna Visitor. Za potrebe testiranja pravimo listu pizcerija i unutra smještamo sve picerije koje smo kreirali. PizzaPlanet, PizzaHut i Maestro. Da bi posle imali lakši prolazak kroz tu listu kreiramo iterator od te liste. Kreiramo instancu tipa Dinner. Ova klasa nam obezbeđuje slučajan odabir picerije pozivom metode getDinner.

```
class Test {  
    public static void main(String[] args)  
    {  
        List pizzaList = new ArrayList();  
        pizzaList.add(new PizzaPlanet());  
        pizzaList.add(new PizzaHut());  
        pizzaList.add(new Maestro());  
        Iterator it = pizzaList.iterator();  
        System.out.println("How many pizza restaurants in this area ? ");  
        while (it.hasNext()) {
```

```
System.out.println((Pizza) it.next().order());  
}  
Dinner d = new Dinner();  
Pizza pza = d.getDinner();  
Visitor v = d.howto();  
v.visit(pza);  
System.out.println("\nWhich store for dinner?");  
System.out.println(v);  
}  
}
```

## OO PRINCIPI ŠABLONA VISITOR

*Principi objektno-orientisanog programiranja koji su primenjeni na šablon Visitor*

U datom primeru imamo izgrađene dve hijerarhije. Prva je hijerarhija picerija. Interface Pizza implementiraju konkretnе picerije PizzaPlanet, PizzaHut i Maestro. Druga hijerarhija u korenu ima interface Visitor i to je hierarhija naših posetioč klasa koje definišu, način naručivanja pica. Klase koje implementiraju interface Visitor su ByPickup, ByEatin i ByDelivery.

Postoji jedan potencijalni problem koji se odnosi na Design Pattern Visitor. Programer može da dođe u iskušenje da dozvoli posetiocu da pristupi podacima koji bi trebali da budu enkapsulirani. Razlog za ovo je jednostavnost kodiranja i veća moć ovog Design Patterna u slučaju da posetioc može da pristupi svim elementima klase. Naravno ovo ne bi smelo da se dozvoli odnosno svi podaci moraju da ostanu strogo enkapsulirani pa makar na uštrbu mogućnosti exploatacije ovog Design Patterna.

Polimorfizam u okviru ovog Design Patterna se svodi na implementaciju metoda iz interface-a u konkretnim klasama i to u obe hierarhije.

## ▼ 7.2 Pokazni primer 2 - Vežba za šablon Observer

## OBSERVER ŠABLON PROJEKTOVANJA

*Uvodne napomene, način implementacije i korišćenja šablonu „osluškivač“ (Observer)*

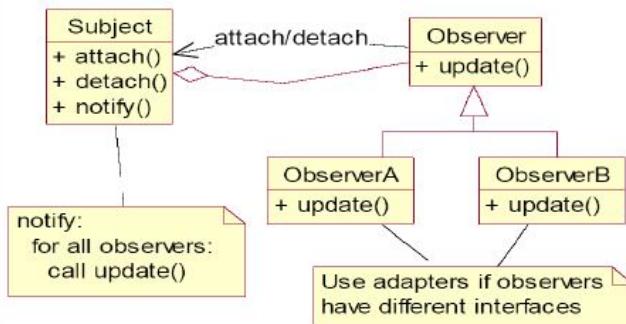
### Trajanje: 20 minuta

Na slici 1 je prikazan UML dijagram Design Patterna. Često problematična situacija je kad imamo objekat koji u sebi ima podatke koje treba da koriste drugi objekti, a objekat izmeni neki od tih podataka što treba da povuče ostale objekte da obnove svoje stanje. Na primer imamo klasu sa podacima koji se prikazuju u tabeli i na dijagramu. Kad objekat promeni

svoje podatke tabela i dijagram treba da promene svoj sadržaj. Situacija može da bude dodatno zakomplikovana situacijom kad postoje još neki objekti zainteresovani za promenu tog objekta na primer naknadno dodat dijagram u obliku pite. Ovaj problem se uspešno rešava Design Patternom Observer koji obezbeđuje da svaki objekat bude pravovremeno i adekvatno obavešten o promeni a da sam objekat ne mora ni da zna koji konkretno objekti ga nadgledaju. U ovom Design Patternu učestvuju dve strane objekat koji treba da se nadgleda i klase koje su zainteresovane za izmene u objektu. Najjednostavniji i ujedno najefikasniji način da se implementira Design Pattern Observer je da obe strane implementiraju odgovarajuće interface. U nastavku ćemo prikazati kod interface-a za obe strane.

Najveći problem u komunikaciji objekata koji pripadaju različitim klasama je njihova prevelika uvezanost.

Oba objekta učesnika u komunikaciji u toj običnoj varijanti treba da imaju instancu onog drugog objekta. Na taj način su te klase postale suviše uvezane odnosno ne možemo da jednu od njih prebacimo u drugi projekat jer će zahtevati prisustvo i one druge klase. Na ovaj način gubimo osnovnu osobinu objekata da je svaki objekat proizvod za sebe koji može višestruko da se koristi. Ovaj problem može uspešno da se reši upotrebom Design Patterna Observer. Posebna prednost Observer Design Pattera je upravo činjenica da objekat koji je zainteresovan za stanje drugog objekta mora da zna instancu tog objekat (što uvek odgovara logici programa), dok objekat koji se posmatra ne zna koji konkretno objekti ga nadgledaju. Nadgledani objekat ima listu zainteresovnih objekata i svaki objekat koji je zainteresovan prijavi se na lisu. Svi objekti na listi će biti obavešteni kad dođe do promene objekta. Ako neki objekat više ne želi da nadgleda dati objekat uvek može da se odjavи sa liste zainteresovanih.



Slika 7.2.1 UML prikaz šablonu Observer [Izvor: Autor]

## OBSERVER PRIMER

*Primer kreiranja aplikacije izveštaja korišćenjem šablonu Observer (osluškivač)*

Karakteristike aplikacije koje će obuhvatati su sledeće:

- Korisnik može da izabere odeljenje za koje želi da vidi prikaz izveštaja
- Nakon odabira odeljenja, dva tipa izveštaja se prikazuju mesečni (Monthly report - lista svih transakcija za trenutni mesec za odabranu odeljenje) i YTD sales chart - grafikon koji prikazuje prihode u svakom mesecu tokom godine.
- Nakon promene odeljenja menjaju se i prikazi dva izveštaja.

Na osnovu gore datog opisa aplikacije možemo zaključiti da su dva objekta koja predstavljaju izveštaje zavisni od objekta koji predstavlja odabir odeljenja. U ovom primeru odlično je primeniti šablon Observer jer treba da osluškuje promenu odeljenja i na osnovu toga obavesti dva različita prikaza da promene svoje podatke u odnosu na odabrano odeljenje.

Na osnovu toga možemo definisati klase i kompletan model koji će biti ispraćen ovim zadatkom.

```

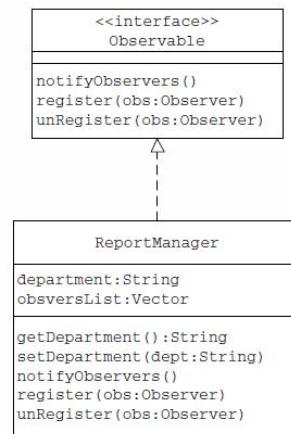
public interface Observable {
    public void notifyObservers();
    public void register(Observer obs);
    public void unRegister(Observer obs);
}

```

Da bi se kreirao Observer šablon neophodno je kreirati interfejs koji je prikazan u listingu iznad. Pored interfejsa u tabeli su prikazane klase koje čine observer šablon.

Class	Role	Functionality
ReportManager	Subject	Displays the necessary UI for the user to select a department. Maintains the user selected department in an instance variable.
MonthlyReport	Observer	Displays the monthly report for the selected department.
YTDChart	Observer	Displays the YTD sales chart for the selected department.

Slika 7.2.2 Tabela-1 prikaz klasa sa opisom [Izvor: Autor]



Slika 7.2.3 Prikaz implementacije interfejsa Observer [Izvor: Autor]

## IMPLEMENTACIJA OBSERVER INTERFEJSA

### *Objašnjenje implementacione klase ReportManager interfejsa Observer*

Klasa ReportManager obezbeđuje implementacije metoda koje se nalaze u interfejsu Observable. Oba zavisna izveštaja koriste ove metode da bi se registrovale kao observer. Klasa ReportManager čuva sve obesverere koji su prijavljeni u observerList vektoru. Kada se

odabere odeljenje koje se prikazuje ono predstavlja stanje objekta ReportManager i nalazi se u promenljivoj department. Kada se setuje nova vrednost u promenljivu department, onda se poziva notifyObservers metoda. Deo notifyObserver metode unutar ReportManager klase poziva refreshData(Observable) metodu svakog od registrovanih observer-a.

Pored implementacija metoda Observable interfejsa klasa ReportManager prikazuje i grafički deo koji je dat na slici 4. Ovaj interfejs omogućava korisniku da odabere odeljenje (department).

Nepohodno je i da definišemo interfejs Observer koji implementiraju obe klase koje čine izveštaje MonthlyReport i YTDCChart.

```
public interface Observer {  
    public void refreshData(Observable subject);  
}
```

ReportManager koristi ovaj interfejs kako bi obavestio svoje Observer-e odnosno posmatrače koji su prijavljeni.

```
public class ReportManager extends JFrame  
    implements Observable {  
  
    public static final String newline = "\n";  
    public static final String SET_OK = "OK";  
    public static final String EXIT = "Exit";  
  
    private JPanel pSearchCriteria;  
    private JComboBox cmbDepartmentList;  
    private JButton btnOK, btnExit;  
    private Vector observersList;  
    private String department;  
  
    public ReportManager() throws Exception {  
        super("Observer Pattern - Example");  
  
        observersList = new Vector();  
  
        // Create controls  
        cmbDepartmentList = new JComboBox();  
        btnOK = new JButton(ReportManager.SET_OK);  
        btnOK.setMnemonic(KeyEvent.VK_S);  
        btnExit = new JButton(ReportManager.EXIT);  
        btnExit.setMnemonic(KeyEvent.VK_X);  
  
        pSearchCriteria = new JPanel();  
  
        //Create Labels  
        JLabel lblDepartmentList  
            = new JLabel("Select a Department:");  
  
        ButtonHandler vf = new ButtonHandler(this);
```

```
btnOK.addActionListener(vf);
btnExit.addActionListener(vf);

JPanel buttonPanel = new JPanel();

//-----
GridBagLayout gridbag = new GridBagLayout();
buttonPanel.setLayout(gridbag);
GridBagConstraints gbc = new GridBagConstraints();
buttonPanel.add(lblDepartmentList);
buttonPanel.add(cmbDepartmentList);
buttonPanel.add(btnOK);
buttonPanel.add(btnExit);

gbc.insets.top = 5;
gbc.insets.bottom = 5;
gbc.insets.left = 5;
gbc.insets.right = 5;

gbc.gridx = 0;
gbc.gridy = 0;
gridbag.setConstraints(lblDepartmentList, gbc);
gbc.anchor = GridBagConstraints.WEST;
gbc.gridx = 1;
gbc.gridy = 0;
gridbag.setConstraints(cmbDepartmentList, gbc);

gbc.anchor = GridBagConstraints.EAST;
gbc.insets.left = 2;
gbc.insets.right = 2;
gbc.insets.top = 40;
gbc.gridx = 0;
gbc.gridy = 6;
gridbag.setConstraints(btnOK, gbc);
gbc.anchor = GridBagConstraints.WEST;
gbc.gridx = 1;
gbc.gridy = 6;
gridbag.setConstraints(btnExit, gbc);

Container contentPane = getContentPane();
contentPane.add(buttonPanel, BorderLayout.CENTER);
try {
    UIManager.setLookAndFeel(new WindowsLookAndFeel());
    SwingUtilities.updateComponentTreeUI(
        ReportManager.this);
} catch (Exception ex) {
    System.out.println(ex);
}

initialize();
setSize(250, 200);
setVisible(true);
}
```

```
private void initialize() throws Exception {
    // fill some test data here into the listbox.
    cmbDepartmentList.addItem("HardWare");
    cmbDepartmentList.addItem("Electronics");
    cmbDepartmentList.addItem("Furniture");
}

public void register(Observer obs) {
    // Add to the list of Observers
    observersList.addElement(obs);
}

public void unRegister(Observer obs) {
    // remove from the list of Observers
}

public void notifyObservers() {
    // Send notify to all Observers
    for (int i = 0; i < observersList.size(); i++) {
        Observer observer
            = (Observer) observersList.elementAt(i);
        observer.refreshData(this);
    }
}

public String getDepartment() {
    return department;
}

public void setDepartment(String dept) {
    department = dept;
}

class ButtonHandler implements ActionListener {

    ReportManager subject;

    public void actionPerformed(ActionEvent e) {

        if (e.getActionCommand().equals(ReportManager.EXIT)) {
            System.exit(1);
        }
        if (e.getActionCommand().equals(ReportManager.SET_OK)) {
            String dept = (String) cmbDepartmentList.getSelectedItem();
            //change in state
            subject.setDepartment(dept);
            subject.notifyObservers();
        }
    }
}

public ButtonHandler() {
```

```
}

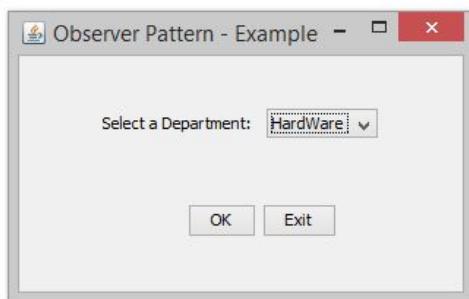
public ButtonHandler(ReportManager manager) {
    subject = manager;
}

}

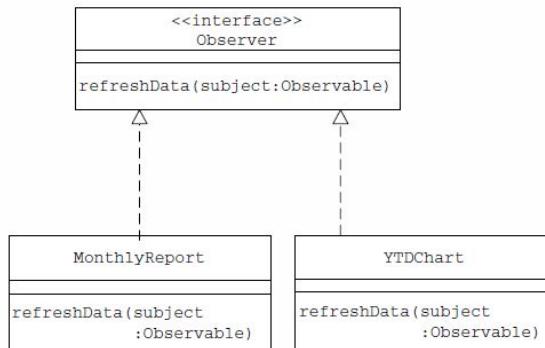
}// end of class
```

## IMPLEMENTACIJA KONKRETNIH KLASA

### Kreiranje klase SupervisorView



Slika 7.2.4 Prikaz prozora ReportManager GUI [Izvor: Autor]



Slika 7.2.5 UML prikaz implementacije interfejsa Observer [Izvor: Autor]

Tipično, klijent će prvo kreirati instance (ReportManager). Kada se kreira Observer objekat (npr: MonthlyReport, YTDChart) kreira, on se prosleđuje Subject instanci kroz konstruktor metodu. Primer koda dat je ispod

```
//Client Code
public class SupervisorView {
    ...
}
```

```
public static void main(String[] args) throws Exception {  
    //Create the Subject  
    ReportManager objSubject = new ReportManager();  
    //Create Observers  
    new MonthlyReport(objSubject);  
    new YTDChart(objSubject);  
}  
}//end of class
```

## IMPLEMENTACIJA KLASA IZVEŠTAJA

*Prikaz implementacije klase MonthlyReport i YTDChart. Kompletna implementacija je prikazana u dodatnim materijalima.*

```
public class MonthlyReport extends JFrame implements Observer {  
  
    public static final String newline = "\n";  
  
    private JPanel panel;  
    private JLabel lblTransactions;  
    private JTextArea taTransactions;  
    private ReportManager objReportManager;  
  
    public MonthlyReport(ReportManager inp_objReportManager)  
        throws Exception {  
        super("Observer Pattern - Example");  
        objReportManager = inp_objReportManager;  
  
        // Create controls  
        panel = new JPanel();  
        taTransactions = new JTextArea(5, 40);  
        taTransactions.setFont(new Font("Serif", Font.PLAIN, 14));  
        taTransactions.setLineWrap(true);  
        taTransactions.setWrapStyleWord(true);  
  
        //Create Labels  
        lblTransactions  
            = new JLabel("Current Month Transactions");  
  
        //For layout purposes, put the buttons in a separate panel  
        JPanel buttonPanel = new JPanel();  
  
        buttonPanel.add(lblTransactions);  
        buttonPanel.add(taTransactions);  
  
        Container contentPane = getContentPane();  
        contentPane.add(buttonPanel, BorderLayout.CENTER);  
        try {  
            UIManager.setLookAndFeel(new WindowsLookAndFeel());  
            SwingUtilities.updateComponentTreeUI(  
                this);  
        } catch (Exception e) {  
            e.printStackTrace();  
        }  
    }  
}
```

```
        MonthlyReport.this);
    } catch (Exception ex) {
        System.out.println(ex);
    }

    setSize(400, 300);
    setVisible(true);
    objReportManager.register(this);

}

public void refreshData(Observable subject) {
    if (subject == objReportManager) {
        //get subject's state
        String department = objReportManager.getDepartment();

        lblTransactions.setText(
            "Current Month Transactions - "
            + department);
        Vector trnList
            = getCurrentMonthTransactions(department);
        String content = "";
        for (int i = 0; i < trnList.size(); i++) {
            content = content
                + trnList.elementAt(i).toString() + "\n";
        }
        taTransactions.setText(content);
    }
}

private Vector getCurrentMonthTransactions(String department)
{
    Vector v = new Vector();
    FileUtil futil = new FileUtil();
    Vector allRows = futil.fileToVector("Transactions.dat");

    //current month
    Calendar cal = Calendar.getInstance();
    cal.setTime(new Date());
    int month = cal.get(Calendar.MONTH) + 1;

    String searchStr = department + "," + month + ",";
    int j = 1;
    for (int i = 0; i < allRows.size(); i++) {
        String str = (String) allRows.elementAt(i);
        if (str.indexOf(searchStr) > -1) {

            StringTokenizer st
                = new StringTokenizer(str, ",");
            st.nextToken(); //bypass the department
            str = "    " + j + ". " + st.nextToken() + "/"
                + st.nextToken() + "~~~"
                + st.nextToken() + "Items" + "~~~"
        }
    }
}
```

```
        + st.nextToken() + " Dollars";
    j++;
    v.addElement(str);
}
}
return v;
}
}// end of class
```

```
import java.io.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.util.*;
import com.sun.java.swing.plaf.windows.*;

public class ReportManager extends JFrame
    implements Observable {

    public static final String newline = "\n";
    public static final String SET_OK = "OK";
    public static final String EXIT = "Exit";

    private JPanel pSearchCriteria;
    private JComboBox cmbDepartmentList;
    private JButton btnOK, btnExit;
    private Vector observersList;
    private String department;

    public ReportManager() throws Exception {
        super("Observer Pattern - Example");

        observersList = new Vector();

        // Create controls
        cmbDepartmentList = new JComboBox();
        btnOK = new JButton(ReportManager.SET_OK);
        btnOK.setMnemonic(KeyEvent.VK_S);
        btnExit = new JButton(ReportManager.EXIT);
        btnExit.setMnemonic(KeyEvent.VK_X);

        pSearchCriteria = new JPanel();

        //Create Labels
        JLabel lblDepartmentList
            = new JLabel("Select a Department:");

        ButtonHandler vf = new ButtonHandler(this);

        btnOK.addActionListener(vf);
        btnExit.addActionListener(vf);
```

```
JPanel buttonPanel = new JPanel();

//-----
GridBagLayout gridbag = new GridBagLayout();
buttonPanel.setLayout(gridbag);
GridBagConstraints gbc = new GridBagConstraints();
buttonPanel.add(lblDepartmentList);
buttonPanel.add(cmbDepartmentList);
buttonPanel.add(btnOK);
buttonPanel.add(btnExit);

gbc.insets.top = 5;
gbc.insets.bottom = 5;
gbc.insets.left = 5;
gbc.insets.right = 5;

gbc.gridx = 0;
gbc.gridy = 0;
gridbag.setConstraints(lblDepartmentList, gbc);
gbc.anchor = GridBagConstraints.WEST;
gbc.gridx = 1;
gbc.gridy = 0;
gridbag.setConstraints(cmbDepartmentList, gbc);

gbc.anchor = GridBagConstraints.EAST;
gbc.insets.left = 2;
gbc.insets.right = 2;
gbc.insets.top = 40;
gbc.gridx = 0;
gbc.gridy = 6;
gridbag.setConstraints(btnOK, gbc);
gbc.anchor = GridBagConstraints.WEST;
gbc.gridx = 1;
gbc.gridy = 6;
gridbag.setConstraints(btnExit, gbc);

Container contentPane = getContentPane();
contentPane.add(buttonPanel, BorderLayout.CENTER);
try {
    UIManager.setLookAndFeel(new WindowsLookAndFeel());
    SwingUtilities.updateComponentTreeUI(
        ReportManager.this);
} catch (Exception ex) {
    System.out.println(ex);
}

initialize();
setSize(250, 200);
setVisible(true);
}

private void initialize() throws Exception {
    // fill some test data here into the listbox.
```

```
        cmbDepartmentList.addItem("HardWare");
        cmbDepartmentList.addItem("Electronics");
        cmbDepartmentList.addItem("Furniture");
    }

    public void register(Observer obs) {
        // Add to the list of Observers
        observersList.addElement(obs);
    }

    public void unRegister(Observer obs) {
        // remove from the list of Observers
    }

    public void notifyObservers() {
        // Send notify to all Observers
        for (int i = 0; i < observersList.size(); i++) {
            Observer observer
                = (Observer) observersList.elementAt(i);
            observer.refreshData(this);
        }
    }

    public String getDepartment() {
        return department;
    }

    public void setDepartment(String dept) {
        department = dept;
    }

    class ButtonHandler implements ActionListener {

        ReportManager subject;

        public void actionPerformed(ActionEvent e) {

            if (e.getActionCommand().equals(ReportManager.EXIT)) {
                System.exit(1);
            }
            if (e.getActionCommand().equals(ReportManager.SET_OK)) {
                String dept = (String) cmbDepartmentList.getSelectedItem();
                //change in state
                subject.setDepartment(dept);
                subject.notifyObservers();
            }
        }
    }

    public ButtonHandler() {
    }

    public ButtonHandler(ReportManager manager) {
```

```

        subject = manager;
    }

}

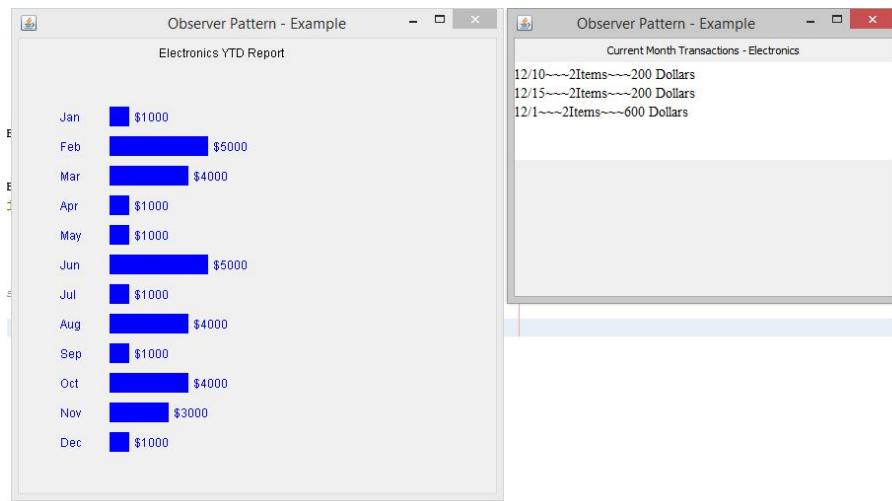
} // end of class

```

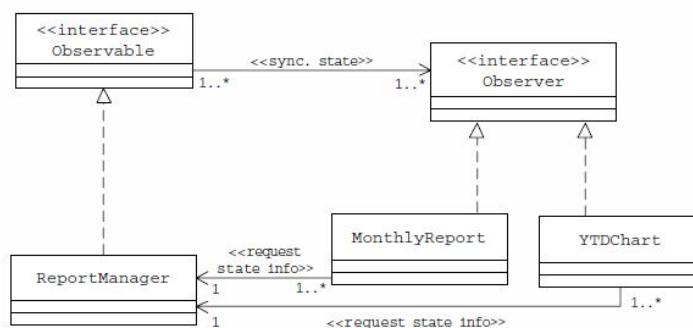
## LOGIČKI TOK

### *Logički tok, prikaz klasnog dijagrama i dijagrama sekvenci*

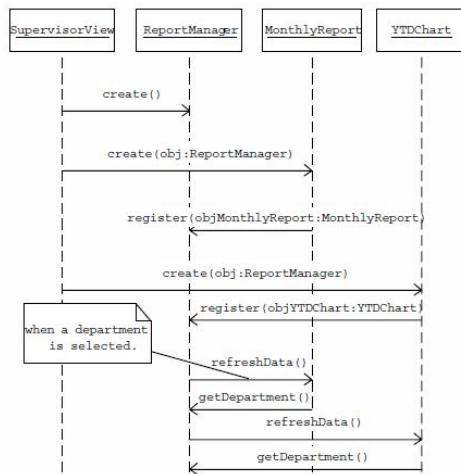
1. Svaki put kada korisnik izabere odeljenje (department) i klikne na OK dugme, ReportManager menja stanje promenljive department.
2. Čim je novo stanje postavljeno, ReportManager poziva refreshData(Observable) metod na oba registrovana osluškivača odnosno u ovom slučaju MonthlyReport i YTDChart objekat.



Slika 7.2.6 GUI prikaz dva izveštaja iz primera [Izvor: Autor]



Slika 7.2.7 Povezanost klasa [Izvor: Autor]



Slika 7.2.8 Tok podataka rada programa [Izvor: Autor]

## IZBOR ŠABLONA (VIDEO)

*Georgia Tech predavanje - Izbor šablonu*

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

## KVIZ: IZBOR ŠABLONA (VIDEO)

*Georgia Tech predavanje - Jviz pitanje o izboru šablonu*

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

## ODGOVOR NA KVIZ PITANJE (VIDEO)

*Georgia Tech predavanje - Odgovor na kviz pitanje o izboru šablonu*

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

## NEGATIVNI ŠABLONI PROJEKTOVANJA

*Šabloni koji pokazuju kako sene sme projektovati*

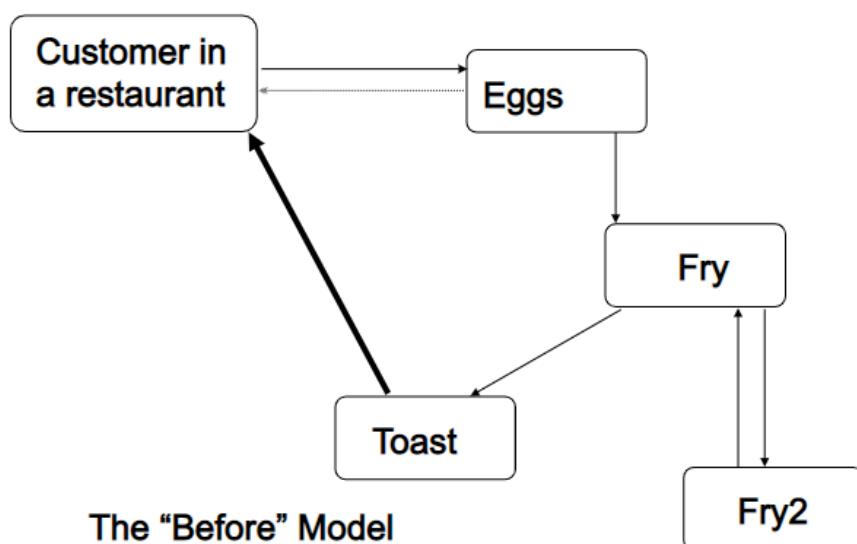
**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

## ✓ 7.3 Pokazni primer 3 - sistem restorana

### PRIMENA MEDIATOR ŠABLONA NA SISTEM RESTORANA

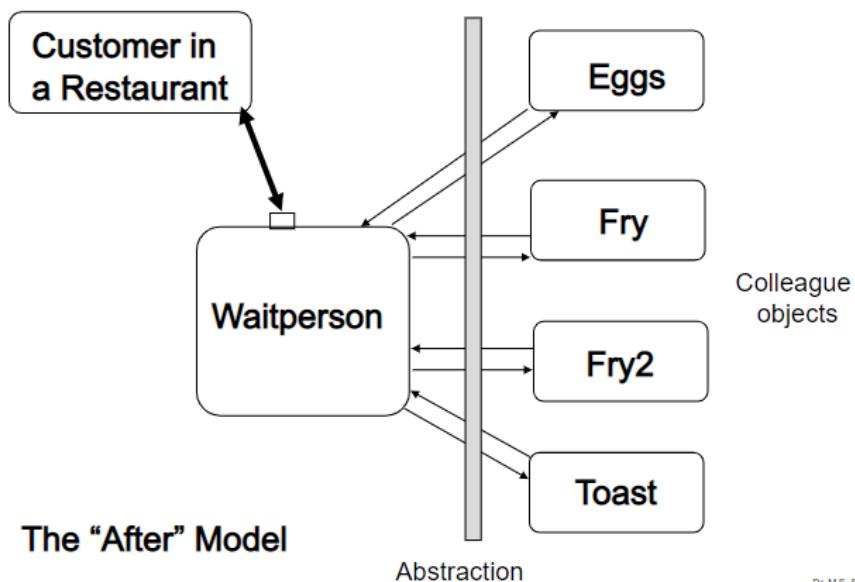
*U ovoj sekciji dat je apstraktni prikaz primene Mediator šablon na RMOS sistem. Trajanje: 5 min.*

Na slici 1 je dat prikaz sistema pre primene Mediator šablon u sistemu restorana. Kako klase u našem sistemu nisu detaljno definisane, primena ovog šablon bi samo dodatno zakomplikovala klasni model i nećemo ga primeniti u finalnoj verziji klasnog modela.



Slika 7.3.1 Primena Mediator šablon na sistem restorana - stanje pre primene [Izvor: Autor]

Primena ovog šablon je značajna jer sada svaka komponenta ima samo poznavanje o mediator komponenti. Fleksibilnost je postignuta tako što svaki individualni objekat može da menja sopstvene metode bez da utiče na ostale objekte



Slika 7.3.2 Primena Mediator šablon na sistem restorana - stanje nakon primene [Izvor: Autor]

## ▼ Poglavlje 8

### Individualna vežba

## ZADACI ZA INDIVIDUALNI RAD - ZADACI OD 1 DO 3

*Zadaci za samostalni rad studenata, primena Mediator, Memento i Strategy šablon projektovanja.*

#### **Zadatak 1: Šablon Mediator** (25 minuta)

Primenite Dialog klasu radnog okvira za GUI aplikacije. Dialog prozor sadrži kolekciju grafičkih i negrafičkih kontrola. Klasa Dialog obezbeđuje mehanizam koji olakšava interakciju između ovih kontrola. Na primer, kada se izabere nova vrednost u ComboBox objektu Pojavljuje se Label sa novom vrednosti. I ComboBox i Lable nisu svesni struktura drugog i sve interakcije su upravljane Dialog objektom. Svaka kontrola nije svesna postojanja drugih kontrola.

#### **Zadatak 2: Šablon Memento** (25 minuta)

Primnите šablon Memento za projektovanje aplikacije Calculator koja ima i operaciju Undo. (referenca: <https://www.oodesign.com/memento-pattern.html>)

#### **Zadatak 3: Šablon Strategy** (25 minuta)

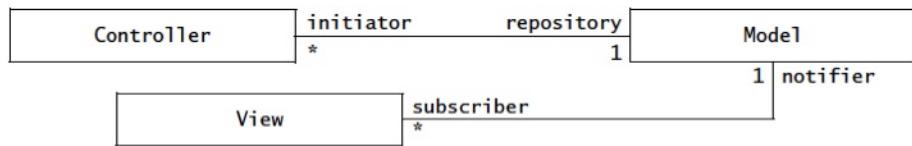
Primenite šablonom Strategy za projektovanje aplikacije koja simulira i analizira interakciju robota ((referenca: <https://www.oodesign.com/memento-pattern.html>)

## ZADACI ZA INDIVIDUALNI RAD - ZADACI OD 4 DO 5

*Zadaci za samostalni rad studenata, primena Observer i Visitor šablon projektovanja.*

#### **Zadatak 4: Observer šablon projektovanja** (25 minuta)

Primenite šablon Observer za realizaciju aktivnosti upisivanja i obaveštavanja u softverskom sistemu koji primenjuje MVC arhitekturu (Model/View/Controller) dat na slici 1. U slučaju primene MVC arhitekture, Controller prihvata ulaze od korisnika i šalje poruku objektu Model, koji je odgovoran za sve podatke na sistemu. Kada se primenom protokola upiši/obavesti (subscribe/notify) izvrši neka promena podataka u objektu Model, ta promena stanja se prenosi objektu View, koji to prikazuje korisniku.



Slika 8.1 MVC arhitektura softvera [Izvor: Autor]

### Zadatak 5: Visitor šablon projektovanja (25 minuta)

Primenite Visitor šablon za slučaj sistema za naručivanje taksi usluga. Korisnik usluge poziva operatera taksi udruženja i naručuje taksi. Kada taksi dođe i preuzme korisnika, taksi preuzima kontrolu vožnje korisnika do željene adrese..

## ▼ Zaključak

### REZIME LEKCIJE

#### *Pouke izloženog gradiva*

U ovoj lekciji smo analizirali primenu šablona projektovanja ponašanja softverskih sistema. Proučili smo sledeće šablonе:

- Mediator,
- Memento,
- State,
- Strategy,
- Observer i
- Visitor

### LITERATURA

#### *Preporučena literatura*

##### **Obevezna literatura:**

1. Onian nastavni materijal za predmet SE201 Uvod u softversko inženjerstvo, 2018/19, Univerzitet Metropolitan
2. Ian Sommerville, Software Engineering, Tenth Edition, Pearson Education Inc., 2016.

##### **Dopunska literatura:**

1. O'Reilly, Head First Design Patterns
2. Partha Kuchana, Software Architecture Design patterns in Java
3. Design Patterns - Elements of Reusable Object-Oriented Software, Eric Gamma, Richard Helm, Ralph Johnson, Jorgens Vissides, 1997
4. Design Patterns in Java Tutorial, tutorialspoint.com

**Veb lokacije :** Na ovim veb lokacijama možete naći opise mnogih šablon projekovanja sa primerima, te se prepućuje da ih proučite.

- <http://www.netobjectives.com/resources/books/design-patterns-explained>
- <https://www.odesign.com/>



## SE201 - UVOD U SOFTVERSKO INŽENJERSTVO

### Evolucija softvera

Lekcija 13

PRIRUČNIK ZA STUDENTE

# SE201 - UVOD U SOFTVERSKO INŽENJERSTVO

## Lekcija 13

### *EVOLUCIJA SOFTVERA*

- ▼ Evolucija softvera
- ▼ Poglavlje 1: Procesi evolucije softvera
- ▼ Poglavlje 2: Dinamika procesa evolucije
- ▼ Poglavlje 3: Održavanje softvera
- ▼ Poglavlje 4: Upravljanje starim softverom
- ▼ Poglavlje 5: Pokazna vežba
- ▼ Poglavlje 6: Individualna vežba
- ▼ Zaključak

Copyright © 2017 – UNIVERZITET METROPOLITAN, Beograd. Sva prava zadržana. Bez prethodne pismene dozvole od strane Univerziteta METROPOLITAN zabranjena je reprodukcija, transfer, distribucija ili memorisanje nekog dela ili čitavih sadržaja ovog dokumenta., kopiranjem, snimanjem, elektronskim putem, skeniranjem ili na bilo koji drugi način.

Copyright © 2017 BELGRADE METROPOLITAN UNIVERSITY. All rights reserved. No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise, without the prior written permission of Belgrade Metropolitan University.

## ✓ Uvod

### UVOD

*Posle puštanja u rad prve verzije softvera, nastaje dugi period njegovog evolutivnog menjanja u toku njegovog životnog veka.*

Cilj ove lekcije da vam objasni zašto je evolucija softvera važan deo softverskog inženjerstva i da vam opiše proces evolucije softvera. Ova lekcija vam omogućava da:

- razumete da je promena neizbežna ako softverski sistemi treba da ostanu korisni i da su razvoj softvera i njegova evolucija integrисани u spiralnom modelu razvoja;
- razumete procese evolucije softvera i uticajne faktore koji deluju na ove procese;
- naučite o različitim tipovima održavanja softvera i faktorima koji utiču na troškove održavanja; i
- razumete kako nasleđeni sistemi se mogu procenjivati radi odlučivanja da li da se odbace, održavaju, poprave ili zamene.

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

## ▼ Poglavlje 1

### Procesi evolucije softvera

#### ŠTA JE EVOLUCIJA SOFTVERA?

*Softversko inženjerstvo je jedan spiralni proces razvoja i evolucije, sa zahtevima, projektovanjem, implementacijom i testiranjem softvera tokom celog njegovog života.*

Isporučen softver se mora da menja iz bar dva razloga.

1. da bi se otklanjale uočeni nedostaci i greške.
2. da bi se prilagođavao novim zahtevima korisnika.

Zbog toga, jednom urađen softver, ceo svoj radni vek najčešće prolazi kroz razne vrste iymena i dopuna, tj. On je u stalnoj evoluciji. Evolucija softvera je stalna promena softvera posle njegovog inicijalnog razvojai isporuke naručiocu, ili tržištu. .

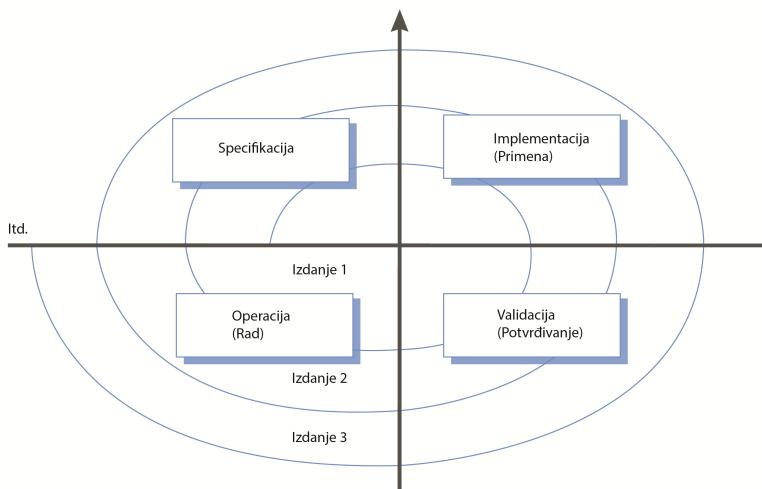
Neke analize pokazuju, da

- oko 85 do 90% troškova vezanih za softver kompanije troše za evoluciju softvera.
- da troškovi evolucije čine dve trećine ukupnih troškova za softver

Kompanija koja je razvila takav softver najčešće predaje odgovornost za dalji razvoj i modifikacije softvera kompaniji koja ga je naručila i koja mora da ima odgovarajući kadar za takav razvoj. Druga varijanta je da kompanija angažuje neku drugu organizaciju da podržava evolutivan razvoj njenog softvera.

Iz navedenih razloga, treba posmatrati softversko inženjerstvo kao jedan spiralni proces sa zahtevima, projektovanjem, implementacijom i testiranjem tokom celog njegovog životnog veka.

Posle verzije 1, ciklusni razvoj se nastavlja, sa navedenim osnovnim aktivnostima, te se dobija verzija 2, pa 3 itd. (slika 1 ).



Slika 1.1 Spiralni model razvoja i evolucija [1.2]

Ovaj model evolucije softvera podrazumeva da je jedna organizacija odgovorna i za njegov inicijalni razvoj, a i za njegovu evoluciju. To je model koji se najčešće koristi kod softverskih paketa, koji su razvijeni za prodaju na tržištu.

## EVOLUCIJA I SERVISIRANJE SOFTVERA

*Evolucija je faza u kojoj se vrše značajne promene u arhitekturi softvera i njegovoj funkcionalnosti. Za vreme servisiranja, vrše se sitnije promene.*

Firma koja je inicijalno razvila softver, može da sklopi ugovor sa drugom firmom koja onda preuzima posao evolucije tog softvera. To može da dovede do prekida u procesu, tj. do diskontinuiteta. Na primer, može da se desi, da dokumentacija o zahtevima i projektnom rešenju ne bude poznata organizaciji – partneru, koja vrši dalju evoluciju softvera. Kompanije, takođe, se mogu spajati ili reorganizovati, što dovodi do uvođenja novog softvera., npr. firme sa kojom se spajaju. Onda, najčešće utvrde da moraju da izvrše određene promene na softveru.

Kada prelazak iz faze razvoje softvera, u fazu evolucije posle isporuke softvera, nije neprimetan, te se ta faza često naziva "**održavanjem softvera**".

Na slici 2 prikazan je alternativan model evolucije softvera tokom njegovog životnog veka. Model razlikuje fazu evolucije od faze servisiranja. **Evolucija** je faza u kojoj se vrše značajne promene u arhitekturi softvera i njegovoj funkcionalnosti. Za vreme **servisiranja**, vrše se sitnije promene, ali neophodne promene.

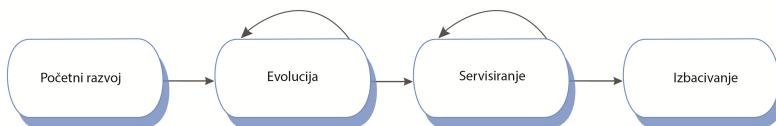
Za vreme evolucije, softver se uspešno koristi, i paralelno postoji tok stalnih promena u zahtevima. Međutim, sa promenama softvera, menja se i njegova struktura, što postepeno dovodi do njegove degradacije, tj. opadanja performansi softvera.

Promene postaju teže i samim tim, i skuplje. To se obično dešava posle nekoliko godina korišćenja softvera, kada se vrše i promene hardverskog i softverskog okruženja u kome

softver radi. U jednom momentu, značajne promene softvera, zbog novih zahteva, postaju sve manje isplative. U tom momentu, sistem prelazi iz faze efolucije u fazu servisiranja.,.

Za vreme faze servisiranja, softver je i dalje koristan, ali se na njemu rade samo male, neophodne izmene. U to vreme, kompanija počinje da razmišlja o zameni softvera novim.

U konačnoj fazi (**phase-out**), softver se i dalje koristi, ali se više ne vrše nikakve izmene na njemu. Korisnici moraju sami da reše probleme na koje najdu.



Slika 1.2 Evolucija i servisiranje [1.2]

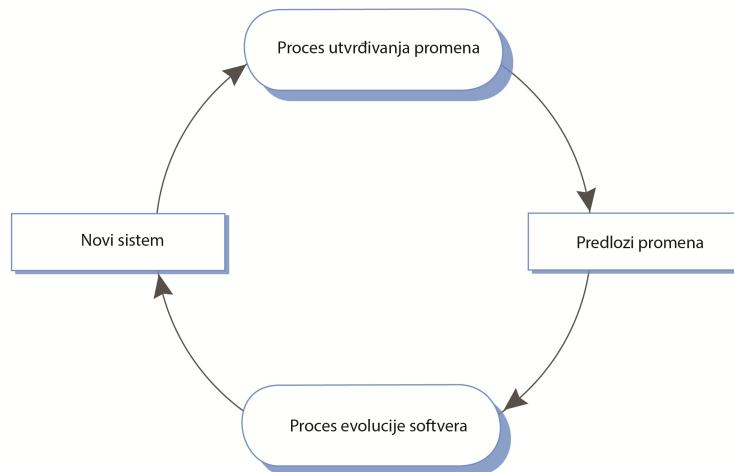
## PROCESI EVOLUCIJE SOFTVERA

*Glavni pokretač evolucije softvera u svim organizacijama su predlozi za promenu sistema*

Procesi evolucije softvera zavise od tipa softvera, od proces razvoja softvera koji se koriste u organizaciji i od veština ljudi koji u tome učestvuju. U nekim organizacijama, evolucija softvera je neformalni proces u kome se zahtevi za promenama formulišu razgovorima između korisnika softvera i inženjera razvoja. U drugim organizacijama, to može biti jedan formalizovan proces, sa strukturisanom dokumentacijom u svakoj fazi procesa.

Glavni pokretač evolucije softvera u svim organizacijama su predlozi za promenu sistema. Oni dolaze iz potrebe da se realizuju još nerealizovani zahtevi sistema, zbog utvrđenih novih zahteva, zbog izveštaja o greškama u sistemu, a i zbog javljanja novih ideja za poboljšanje softvera od strane razvojnog tima. Procesi utvrđivanja zahteva sistema evolucije su cikličnog karaktera i ponavljaju se tokom životnog ciklusa sistema (slika 3)

Zahtevi za promenom softvera bi trebalo da ukazuju na komponentu softvera koja se treba modifikovati. Na taj način se mogu proceniti troškovi realizacije te promene kao i njen efekat.

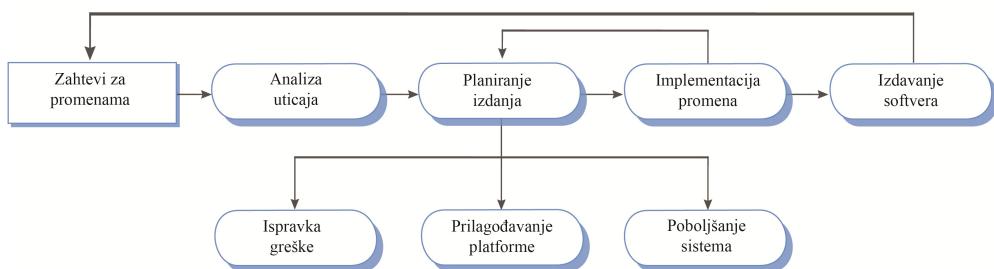


Slika 1.3 Procesi utvrđivanje zahteva za promenom i evolucije [1.2]

## OSNOVNE AKTIVNOSTI PROCESA EVOLUCIJE

*Troškovi i efekat promena se ocenjuju da bi se donela odluka o prihvatanju*

Na slici 4 prikazan je proces evolucije softvera (Arthur, 1988) koji uključuje osnovne aktivnosti procesa. Troškovi i efekat promena se ocenjuju da bi se donela odluka o prihvatanju. Ako je ona pozitivna, onda ona određuje i verziju softvera u kojoj će ta promena biti primenjena. Kada su promene promenjene i potvrđene, izdaje se novo izdanje (verzija) softvera. Ovaj ciklusni proces se stalno ponavlja.



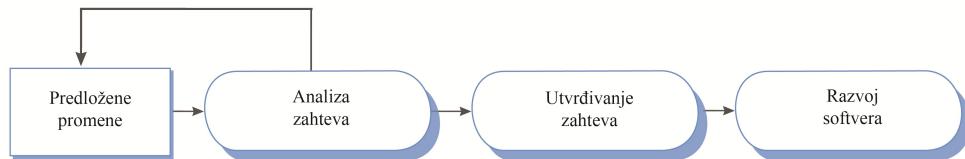
Slika 1.4 Proces evolucije softvera [1]

## IMPLEMENTACIJA PROMENE

*Mora se proceniti da li promena neće izazvati neke nove probleme.*

Proces evolucije softvera i uvođenja promena u softver je jedan iterativni proces razvoja, u kome se nova izdanja softvera projektuju, implementiraju i testiraju. Specifičnost je u prvoj fazi primene promena mora da postoji dobro razumevanje samog softvera, jer u ovom procesu prvo bitni razvojni tim nije više odgovaran za implementaciju promena. Mora se proceniti da li neće promena da izazove neke nove probleme.

U idealnom slučaju, u fazi implementacije promena, trebalo bi izvršiti promenu specifikacije sistema, projektnih rešenja i implementacije softvera, u skladu sa izvršenim promenama (slika 5). Predložene promene se analiziraju i potvrđuju (validacija). Komponente sisteme se ponovo projektuju i implementiraju, a sistem testira.



Slika 1.5 Implementacija promene [1.2]

Zahtri za promenom sistema ponekad se izazvani problemima u sistemu i koji traže hitnu reakciju. Ove urgentne promene su uslovljene sledećim razlozima:

1. Javljanje ozbiljne greške u sistemu koja se mora otkloniti da bi sistem mogao normalno da radi.
2. Došlo je do promenama u radnom okruženju koje imaju neočekivane efekte i sprečavaju normalan rad sistema.
3. Došlo je do promena u načinu poslovanja (novi konkurenti, novi propisi) koji zahtevaju hitne promene u sistemu.

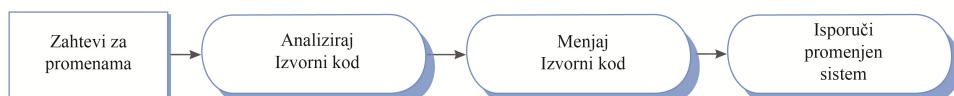
## PROCES HITNE OPRAVKE I PROMENE SISTEMA

*Sistemi koji imaju puno hitnih ispravki u svojoj istoriji (menja se kod, ali ne i projektna dokumentacija) brže stare, jer nove promene postaju teže izvodljive*

Zbog hitnosti, ove promene se ne mogu realizovati normalnom procedurom (slika 5), već se i bez promene zahteva i projektnog rešenja, odmah pristupa promeni koda (slika 6). Opasnost je, naravno, da se time ne poremeti konsistentnost softvera, jer se naknadne dorade dokumentacije obično ne vrše, te stvarni sistem odudara od sistema u dokumentaciji.

Zbog hitnosti, hitne intervencije ne obezbeđuju i najbolji mogući način promene. Zato, sistemi koji imaju puno hitnih ispravki u svojoj istoriji, brže stare, jer nove promene postaju teže izvodljive i teže se održavaju (tzv. „špageti problem“). Normalno bi bilo, da po izvršenoj hitnoj intervenciji na sistema, da se sproveđe pažljiva analiza, i ako je potrebno, promena izvede na drugačiji način, i dokumentuje. Ali, u praksi se to retko dešava, jer te aktivnosti obično nemaju

visok prioritet i to vodi ubrzanom pogoršanju softverske strukture i skraćivanju njegovog životnog veka.



Slika 1.6 Proces hitne opravke i promene sistema. [1.2]

Kod primene agilnih metoda razvoja softvera, zbog inkrementalnog razvoja softvera, prelazak iz procesa razvoja u proces evolucije je neosetan. Primena tehnika automatskog regresionog testiranja je koristan kada se vrše promene u sistemu. Praktično, evolucija samo produžava proces agilnog razvoja softvera.

Mogu se javiti problemi kada razvojni tip predaje dalju nadležnost tima za evoluciju softvera:

1. Problem se javlja kada je razvojni tim primenjivao agilne metoda, a tim za evoluciju primenjuje planski pristup u razvoju softvera. Timu za evoluciju nedostaje potpunija dokumentacija o softveru.
2. Problem se javlja i u suprotnom slučaju, kada je razvojni tim primenjivao planski pristup u razvoju softvera, a tim za evoluciju primenjuje agilne metode. U tom slučaju tim za evoluciju mora da počne ni od čega kada definiše automatske testove te ne dolazi do uprošćavanja koda, kao što se очekuje od agilnog razvoja. U ovakovom slučaju, reinženjering softvera može biti rešenje problema, kako bi se on poboljšao pre nego što se primene agilne metode u fazi evolucije softvera.

## EVOLUTIONARY PROTOTYPING PROCESS - GEORGIA TECH - SOFTWARE DEVELOPMENT PROCESS (VIDEO)

*Trajanje: 2:08 minuta*

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

## VIDEO PREDAVANJE ZA OBJEKAT "PROCESI EVOLUCIJE SOFTVERA"

*Trajanje video snimka: 20min 50sek*

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

## ▼ Poglavlje 2

### Dinamika procesa evolucije

#### LEHMANOVI ZAKONI

*Lehman-ovi zakoni iskazuju izvesne uočene zakonitosti u evoluciji softvera.*

Pojedini istraživači (Lehman i Belady) uočili su izvesne zakonitosti u evoluciji softvera:

1. Održavanje sistema je jedan neizbežan proces. Kako se okruženje sistema menja, javljaju se novi zahtevi i sistem se mora menjati. Kada se novi sistem aktivira, on utiče na okruženje, te se i ono menja, te proces evolucije nastavlja svoj ciklični razvoj.
2. Kako se sistem menja, tako se njegova struktura pogoršava. Preventivno održavanje je jedini način da se ovo izbegne. To znači da je neophodno da se investira vreme i novac za promenu strukture softvera bez promene njegove funkcionalnosti.
3. Veliki sistemi imaju svoju dinamiku koja je postavljena u ranoj fazi procesa razvoja. To određuje dalji trend procesa održavanja softvera i ograničava broj mogućih promena. Zakon je posledica uticaja strukturalnih faktora koji utiču i ograničavaju promene sistema, i organizacionih faktora (brzina donošenja odluka) koji utiču na proces evolucije.
4. Veliki projekti programiranja rade u stanju „zasićenja“. To znači da neka promena u resursima ili ljudima ima neprimetan uticaj na evoluciju sistema u dužem periodu. To znači, kao i u 3. Zakonu, da evolucija programa je dosta nezavisna od odluka menadžmenta. Zakon potvrđuje da su veliki timovi razvoja često neproduktivni zbog složenih komunikacija koje dominiraju nad njihovim radom.
5. *Dodavanje nove funkcionalnosti sistemu neminovno unosi nove greške sistema. Što više novih funkcija unosite, unosite i više grešaka u sistem. To traži ubrzo novo izdanje softvera sa ispravkama ovih grešaka. Znači, ne pravite inkremente sa velikim promenama funkcionalnosti ako niste spremni da ubrzo pravite novo izdanje da bi ispravili unete greške.*

Pored ovih pet zakona, Lehman je kasnije dodao nove. Šesti i sedmi zakon su slični i kažu da korisnici softvera postaju nezadovoljni ako se on ne održava i ako se ne dodaju nove funkcionalnosti. Poslednji, osmi zakon se bavi povratnim procesima.

.

## SEDAM LEHMAN-OVIH ZAKONA

*Lehman-ove zakone treba uzeti u obzir kada planirate proces održavanja softvera.*

Na slici 1 prikazani su svi Lehmanovi zakoni u celosti. Ove zakone treba uzeti u obzir kada planirate proces održavanja

Zakon	Opis
Kontinuirana promena	Program koji se koristi u stvarnom okruženju mora da se menja ili postaje progresivno nekoristan za to okruženje.
Povećanje složenosti	Sa promenama u softveru, njegova struktura postaje složenija. Moraju se odvojiti dodatni resursi za održavanje i uproščavanje strukture.
Velika evolucija programa	Program evolucije i samoregulišući proces. Atributi sistema, kao što su veličina, vreme između izdanja, i broj prijavljenih grešaka je približno nepromenljiv u svakom izdanju.
Organizaciona stabilnost	U toku životnog veka programa, njegova brzina menjanja je približno konstantna i nezavisna od dodeljenih resursa u njegovom razvoju.
Konzervacija familijarnosti	Za vreme životnog veka sistema, ingrementna promena u svakom izdanju je približno ista.
Stalni rast	Funkcionalnost sistema konstantno raste da bi zadržala zadovoljstvo korisnika.
Opadajući kvalitet	Kvalitet sistema opada ukoliko se ne menja u skladu sa promenama u njegovom radnom okruženju.
Povratni sistem	Procesi evolucije višestruke povratne sprege i višestruke agente i treba da ih tretirate kao povratne sisteme da bi ostvarili značajno poboljšanje proizvoda.

Slika 2.1 Tabela-1 Lehman-ovi zakoni [1.2]

## LEHMAN LOW (VIDEO)

*Trajanje: 4,31 minuta*

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

## ▼ Poglavlje 3

### Održavanje softvera

#### SADRŽAJ

*Održavanje softvera praćeno je značajnim troškovima jer se u dugom periodu vrše povremene manje ili veće dorade softvera.*

U ovom poglavlju, izlažu se sledeći aspekti održavanja softvera:

1. Vrste i troškovi održavanja
2. Previđanje troškova održavanja softvera
3. Reinženjering softvera
4. Preventivno održavanje

#### VIDEO PREDAVANJE ZA OBJEKAT "ODRŽAVANJE SOFTVERA"

*Trajanje video snimka: 34min 40sek*

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

#### ▼ 3.1 Vrste i troškovi održavanja

##### VRSTE ODRŽAVANJA

*Održavanje softvera je proces menjanja sistema posle njegove isporuke*

Održavanje softvera je proces menjanja sistema posle njegove isporuke. Promene obuhvataju otklanjanje grešaka u programiranju, ali i veće greške koje otklanjaju greške u projektnom rešenju ili u specifikaciji sistema, ali u zadovoljenju novih zahteva. Implementacija promene postojećeg sistema se primenjuje promenom postojećih komponenti sistema, ali i dodavanjem novih.

Postoji tri tipa održavanja softvera:

1. **Popravka greški:** Otklanjanje greški u kodu (najmanji troškovi popravke), u projektnom rešenju (veći troškovi) jer izaziva promenu nekoliko komponenti, i u zahtevima (najveći troškovi), jer može da zahteva i redizajn sistema.
2. **Prilagođenje okruženju:** Ova promena u softveru je izazvana promenama platforme i okruženja u kome radi sistem (hardver, softver).
3. **Dodavanje funkcionalnosti:** Promene u softveru su nužne zbog zadovoljenja novih organizacijskih ili poslovnih promena. Ove promene sistema su obično znatno ozbiljnije nego u prethodna dva slučaja.

U praksi se koriste različiti nazivi za ove tipove održavanje softvera, a i granice između ovih tipova održavanja nisu uvek jasno određene.

Istraživanja pokazuju da *troškovi održavanja u IT budžetu imaju veći udio nego razvoj novog softvera* (oko 2/3 održavanje, 1/3 novi razvoj). Veći deo troškova održavanja su vezani na zadovoljenju novih zahteva ili prilagođavanju okruženju, nego otklanjanju grešaka u softveru (slika 1 ).



Slika 3.1.1 Distribucija troškova održavanja softvera [1.2]

## TROŠKOVI ODRŽAVANJA SOFTVERA

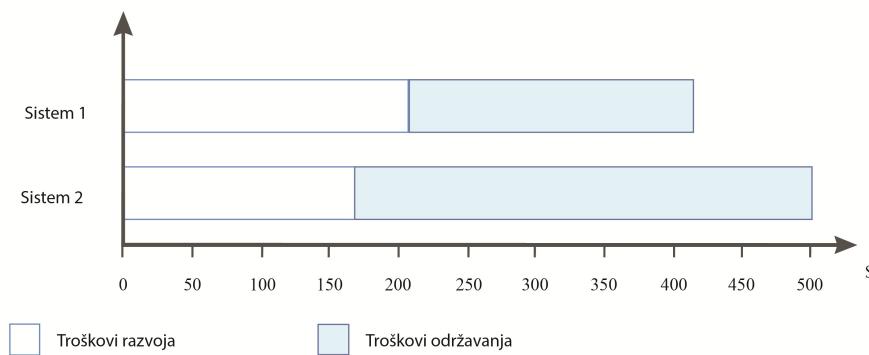
*Ukupni troškovi održavanja softvera u toku njegovog životnog veka mogu da se smanje u odnosu na troškove razvoja ako se o održavanju vodi računa još u fazi razvoja softvera.*

U slučaju tzv. **ugrađenih sistema** koji rade u realnom vremenu, troškovi održavanja su i četiri puta veći nego troškovi razvoja. Razlog za ovo zadovoljenje zahteva pouzdanosti i performansi koje prouzrokuju visok stepen integracije njihovih komponenata, te zbog toga, troškovi održavanja su veći. Doduše, primenom objektno-orientisanih programiranja u razvoju ugrađenih sistema dovodi do smanjivanju troškova održavanja jer olakšavaju promenu njihovih komponenti.

S ciljem da se smanje troškovi održavanja softvera, ulažu se naporci da se razviju metodi razvoja softvera koji će voditi računa o održavanju još u fazi razvoja softvera (npr. primena tehniki dobijanje precizne specifikacije, primena objektno-orientisanog razvoja, upravljanje konfiguracijom softvera).

Slika 2 pokazuje kako ukupni troškovi održavanja softvera u toku njegovog životnog veka mogu da se smanje u odnosu na troškove razvoja ako se o održavanju vodi računa još u fazi razvoja softvera. U slučaju sistema 1 trošak razvoja je bio uvećan za 25.000 dolara, zbog vođenja računa o održavanju softvera. Ovo je dovelo do ušteda od \$100.000 u održavanju ovog sistema u toku njegovog životnog veka. Očigledno, više ulaganja u razvoj softvera dovodi do opadanja troškova održavanja softvera kasnije.

Da bi se smanjili troškovi održavanja softvera koji se razvija **metodama agilnog razvoja**, neophodno je menjati i njegovu strukturu (engl. **refactoring**), kako bi sporije „stario“. Kod **planski vođenom razvoju softvera** retko se ulažu dodatni naporci da se kod učini lakšim za održavanje, jer to povećava troškove razvoja. S druge strane, uštede u održavanju dolaze mnogo kasnije, što nije mnogo atraktivno za mnoge kompanije koje se bore za opstanak na tržištu ili su usmerene ka ostvarivanju što većih profita (zbog pritiska akcionara i bonusa za menadžment).



Slika 3.1.2 Troškovi razvoja i održavanja softvera [1.2]

## RAZLOZI ZA POVEĆANJE TROŠKOVA ODRŽAVANJA

*Skuplje je dodati novu funkcionalnost već postojećem softveru nego je ugraditi pri njegovom razvoju*

Skuplje je dodati novu funkcionalnost već postojećem softveru nego je ugraditi pri njegovom razvoju, iz sledećih razloga:

- Stabilnost tima:** Održavanje obično radi novi tim koji ne razume sistem tako dobro kao tim koji ga je razvio, te im treba više vremena za razumevanje sistema, pre nego što ga menjaju.
- Loša praksa razvoja:** Ugovor o održavanju je obično nezavisан od ugovora o razvoju sistema. Taj posao može biti dat i nekoj kompaniji koji i nije razvila softver. Zato, kompanija koja razvija sistem, nema motiv da pri razvoju softvera vodi računa o njegovom kasnjem održavanju. Čak suprotno, razvojni tim je motivisan da smanji

što više troškove razvoja, ne vodeći računa da to može da dovede do većih troškova održavanja u budućnosti.

**3. Veštine zaposlenih:** Zaposleni koji se bave održavanjem je obično manje iskusan i ne poznaje dobro oblast primene softvera. Pored toga, stari sistemi su često pisani u programskim jezicima koji su prevaziđeni, te ljudi u održavanju nemaju dovoljno iskustva sa tim jezicima. Sve ovo povećava troškove održavanja.

**4. Starost programa i njegova struktura:** S promenama u softveru, postepeno se kvari njegova struktura, te on postaje teži za razumevanje i održavanje. Stari sistemi nisu ni razvijeni primenom modernih tehnika razvoja softvera. Pri njihovom razvoju, njihova struktura nije optimizovana za razumevanje koda, već za njegovu efikasnost. Dokumentacija sistema je ili izgubljena ili je nekonistentna. Stari sistemi ne koriste upravljanje konfiguracijama, te je često otežano nalaženje prave verzije njegovih komponenti.

Prva tri razloga su vezana za problem shvatanja u industriji softvera. Mnoge organizacije i dalje vide održavanje i razvoj softvera kao dve nezavisne aktivnosti. Održavanje često se smatra aktivnošću nižeg ranga, te se ne poklanja pažnja održivosti sistema u toku faze razvoja sistema. Četvrti problem je lakše rešiv, jer se primenom odgovarajućih softverskih tehnika može se uspešno održavati struktura softvera u toku njegovog životnog veka, a i razumljivost sistema.

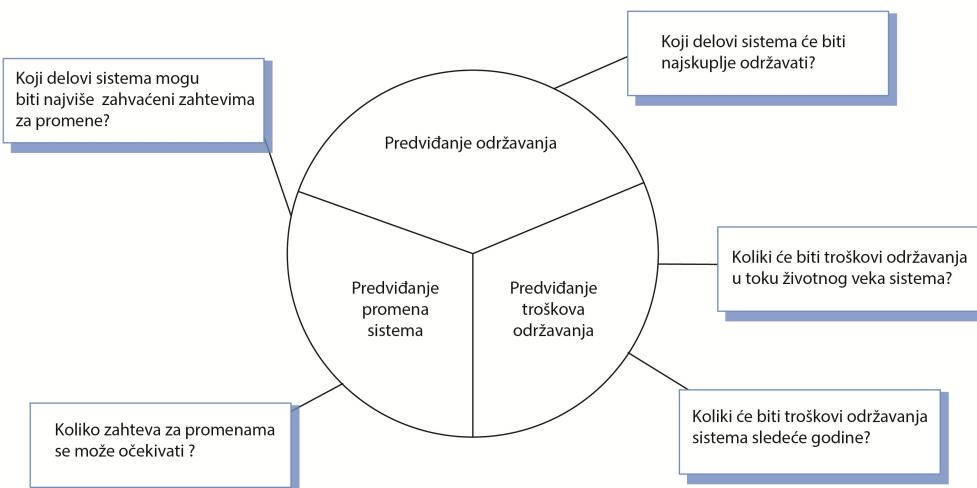
## ▼ 3.2 Predviđanje troškova održavanja softvera

### PREDVIĐANJE TROŠKOVA ODRŽAVANJA

*Poželjno je proceniti ukupne troškove održavanja sistema u datom vremenskom periodu*

Poželjno je predvideti promene sistema, kao i delove sisteme koji će biti najteži za održavanje. Poželjno je proceniti ukupne troškove održavanja sistema u datom vremenskom periodu (slika 1 ).

Da bi se predvideo broj zahteva za promenama nekog sistema, potrebno je dobro razumeti odnos između sistema i spoljnog okruženja. Ukoliko su ti odnosi složeniji, treba očekivati veći broj zahteva za promenama.



Slika 3.2.1 Predviđanje održavanja [1.2]

## MERE ZA SMANJENJE TROŠKOVA ODRŽAVANJA

*Da bi se smanjili troškovi održavanja, preporučljivo je da se složene komponente zamene sa jednostavnijim alternativnim rešenjima*

Kako oceniti složenost odnosa između sistema i okruženja? Uzmite u obzir sledeće:

- Broj i složenost sistemskih interfejsa:** Što ih je više, i što su složeniji, veća je verovatnoća da će se menjati.
- Broj nasledno nestabilnih sistemskih zahteva:** Zahtevi koji odražavaju politiku organizacije i procedure, uvek se češće menjaju nego zahtevi koji se odnose na stabilne karakteristike u nekom području.
- Poslovni procesu u kojim se sistem koristi:** Kako se poslovni procesi razvijaju i menjaju, tako se generišu zahtevi za promenama. Što je veći broj poslovnih procesa koji ih upotrebljavaju, veći je broj zahteva za promenama.

Primenom metrike softvera, meri se njegova složenost. Na taj način se utvrđuju složenije komponente nekog sistema. Što su komponente složenije, to se teže održavaju. Da bi se smanjili troškovi održavanja, preporučljivo je da se složene komponente zamene sa jednostavnijim alternativnim rešenjima.

Kada se neki sistem pusti u rad, obradom nekih podataka, možete oceniti mogućnosti njegovog održavanja. Kakva je metrika procesa potrebna da bi se ocenila sposobnost održavanja sistema? Evo nekih primera:

- Broj zahteva za korektivno održavanje:** Ako je broj izveštaja o greškama veći od broja ispravljenih grešaka programa u toku procesa održavanja, onda je održavanje sistema lošije, tj. opada njegova sposobnost održavanja.
- Prosečno vreme potrebno za izradu analize uticaja promena:** Ovo odražava broj komponenata koje su pod uticajem neke promene. Ako vreme da se ovo utvrdi raste, to ukazuje da je sve veći broj komponenata zahvaćen promenama, te se smanjuje sposobnost održavanja sistema.

**3. Prosečno vreme za realizaciju zahteva za promenom:** Ako vam je potrebno više vremena da realizujete (primenite, implementirate) sistem i njegovu dokumentaciju, ukazuje da sposobnost održavanja sistema opada.

**4. Broj izuzetnih zahteva za promenama:** Ako se povećava broj ovih vanrednih zahteva, onda to ukazuje da se smanjuje sposobnost održavanja sistema.

Ove informacije obezbeđuju da se izvrše predviđanja sposobnosti i održavanja sistema, i da se na osnovu toga, predvide troškovi održavanja. Naravno, intuicija i iskustvo su ovde neophodni za bolje predviđanje. Zato je vrlo važno dobro razumevanje postojećeg koda kao i znanje o razvoju novog koda.

## ▼ 3.3 Reinženjering softvera

### ŠTA JE REINŽENJERING SOFTVERA?

*Reinženjering softvera je promena strukture arhitekture sistema, ponovno pisanje koda primenom modernih programskih jezika, promena struktura i vrednosti podataka sistema.*

Stari programi vremenom, zbog zastarele tehnologije, ili zbog menjanja programa tokom njihove evolucije, postaju spori i neefikasni. Pored toga, mnoge stare sisteme, teško je razumeti i menjati. Oni su optimizirani za performanse, ili za korišćenje memoriskog prostora, a na račun jasnoće koda. Vremenom, početna struktura će se znatno promeni i uruši stalnim promenama softvera.

Da bi ovi stari sistemi postali lakši za održavanje, potrebno je izvršiti reinženjering sistema kako bi se poboljšala struktura sistema i njegova razumljivost. **Reinženjering softvera** je promena strukture arhitekture sistema, ponovno pisanje koda primenom modernih programskih jezika, promena struktura i vrednosti podataka sistema, kao i promena strukture i vrednosti sistemskih podataka:

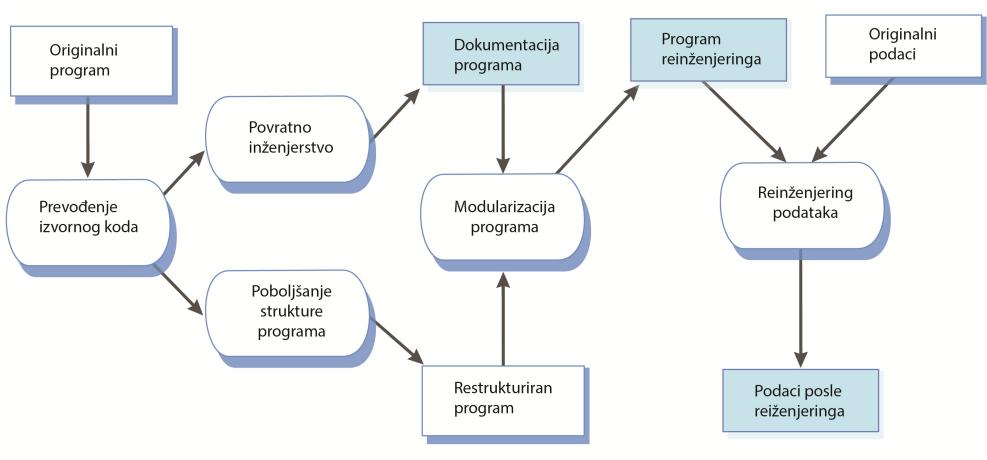
Funkcionalnost softvera se ne menja, i treba izbeći da se vrše velike promene u arhitekturi sistema. Na taj način se mogu ostvarite dve koristi od reinženjering softvera umesto njegove zamene potpuno novim sistemom.

- Smanjiti rizik:** Razvoj i primena potpuno novog sistema, a naročito ako je on kritičan, je uvek rizični potez, jer se mogu javiti greške koje mogu da odlože njegovu primenu. Sa reiženjeringom starog softvera, taj rizik je niži.
- Smanjeni troškovi:** Troškovi reinženjeringa mogu biti višestruko niži nego troškovi razvoja novog sistema.

### PROCES REINŽENJERINGA

*Proces reinženjeringa starog softvera sadrži niz aktivnosti koje poboljšavaju performanse starog softvera.*

Slika 1 prikazuje opšti model procesa reinženjeringa softvera. Na ulazu je stari program a na izlazu je poboljšan i restrukturiran isti program.



Slika 3.3.1 Proces reinženjeringa starog programa [1.2]

Proces reinženjeringa sadrži sledeće aktivnosti:

- 1. Prevođenje izvornog koda:** Uz pomoć odgovarajućeg alata, vrši se konverzija sa nekog starog programskega jezika u noviju verziju istog jezika ili u neki potpuno drugi i savremeniji programski jezik.
- 2. Povratno inženjerstvo:** Analizom izvornog koda radi se programska dokumentacija, tj. opisuje se njegova organizacija i funkcionalnost. Često i ova aktivnost može da se automatizuje.
- 3. Poboljšanje strukture programa:** Analizira se struktura programa i vrši se njena promena da bi se program lakše čitao i razumeo. Ovaj proces se može samo delimično automatizovati.
- 4. Modularizacija programa:** Grupišu se međusobno povezani delovi programa i eliminišu se, redundantnosti u sistemu (ponavljanja koda). U nekim slučajevima može se modifikovati i struktura programa. Na primer, umesto nekoliko, koristi se samo jedna baza podataka. Ovo je ručni proces.
- 5. Reinženjering podataka:** U skladu sa promenama u programu, menjaju se i podaci. To obuhvata i promenu šeme baze podataka i konverziju postojećih baza podataka u nove strukture. Tada se i podaci „čiste“, tj. uklanjuju se pogrešni podaci, dupli slogovi podataka i dr. Postoje alati za podršku reinženjeringa podataka.

## TROŠKOVI I PROBLEMI REINŽENJERINGA

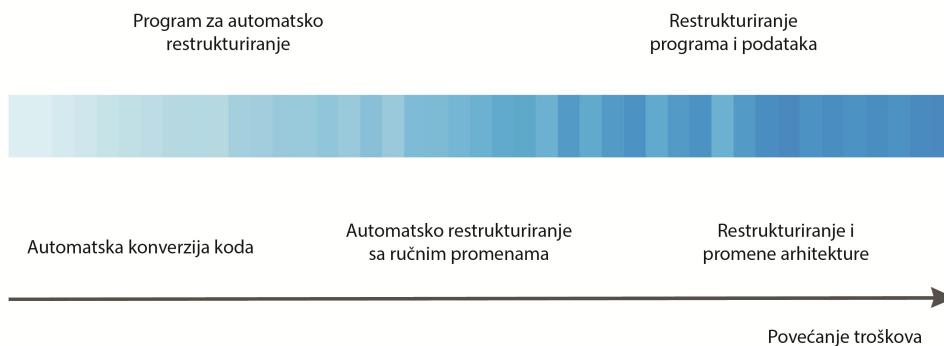
*Program posle reinženjeringa ipak nije tako jednostavan za održavanje kao što je obično potpuno novi sistem koji koristi savremene metode softverskog inženjerstva.*

U praksi, reinženjering programa ne mora da obuhvati sve navedene aktivnosti. Na primer, ne morate da prevodite izvorni program u novu verziju jezika ili u novi jezik. Takođe, nije vam

potrebna programska dokumentacija ako možete da reinženjering sprovedete automatski. Reinženjering podataka se vrši samo ako se promenama u programu menja struktura podataka.

Kako preći sa starog programa na promjenjen program posle reinženjeringu? Za to se često koriste adaptori koji sakrivaju originalne interfejse softverskog sistema i prikazuju nove, koji su bolje strukturisani i koji se mogu koristiti od strane drugih komponenti. Vi ste time upakovali stari softver „u novo odelo“, tj prekrili ste ga novim interfejsima, i time dali druge servisne mogućnosti sistema.

Zavisno od opsega preduzetih aktivnosti reinženjeringu, zavisi i visina troškova koji ga prate (slika 2). Najskuplji deo je promena arhitekture sistema



Slika 3.3.2 Troškovi reinženjeringu u zavisnosti od primenjenih pristupa u reinženjeringu [1.2]

Šta su problemi sa reinženjeringom softvera? Postoje praktične granice poboljšanja starog programa reinženjeringom. Na primer, funkcionalno pisan program ne možete pretvoriti u objektno-orientisan program. Radikalna reorganizacija upravljanja podacima se ne može automatizovano sprovesti. Program posle reinženjeringu ipak nije tako jednostavan za održavanje kao što je obično potpuno novi sistem koji koristi savremene metode softverskog inženjerstva.

## 3.4 Restrukturiranje softvera

### PREVENTIVNO ODRŽAVANJA RESTRUKTURIRANJEM SOFTVERA

*Restrukturiranje je poboljšanje strukture programa, smanjivanje njegove složenosti i olakšavanje njegovog razumevanja.*

Restrukturiranje je proces poboljšanja programa s ciljem da se uspori njegova degradacija (opadanje performansi, povećanje složenosti, otežano snalaženje radi uvođenja promena i dr.) usled promena kojim je vremenom izložen. Restrukturiranje je poboljšanje strukture programa, smanjivanje njegove složenosti i olakšavanje njegovog razumevanja.

Restrukturiranje ne dodaje novu funkcionalnost programu, već se sam program poboljšava. U tom smislu, restrukturiranje je jedan od oblika „preventivnog održavanja“ softvera jer smanjuje probleme koji se javljaju usled promena softvera u budućnosti.

Koja je onda razlika restrukturiranja i reinženjeringa softvera?

1. **Reinženjering** se vrši posle izvesnog vremena održavanja softvera i povećanja troškova održavanja. Upotreboom automatskih alata za obradu i reinženjering starih sistema, vi kreirate novi sistem koji se lakše održava.
2. **Restrukturiranje koda (refactoring code)** je stalni proces poboljšanja za vreme razvoja i evolucije softvera. Sa njim mi izbegavamo degradaciju strukture i koda koji dovode do povećavanja troškova i do drugih teškoća održavanja sistema.

Restrukturiranje je sastavni deo agilnih metoda, kao što je ekstremno programiranje, jer se ovi metodi baziraju na promeni. Zbog čestih (malih) promena, program, teži degradaciji, te programeri često restrukturišu svoje programe kako bi izbegli degradaciju. Zbog primene regresivnog testiranja pri primeni agilnih metoda, smanjuje se rizik unošenja novih grešaka preko restrukturiranja. Sve unete greške bi trebalo da budu detektovane (otkrivene) primenom testova. Međutim, restrukturiranje nije zavisno od drugih aktivnosti primene agilnih metoda, te se može primeniti u bilo kom pristupu razvoja.

## PRIMENA RESTRUKTURIRANJA SOFTVERA

*Poboljšanje softvera se vrši restrukturiranjem programskog koda, ali i projektnog rešenja, što je skuplji i složeniji metod poboljšanja, i primenjuje se kada samo restrukturiranje koda nije dovoljn*

Kada primeniti restrukturiranje? Odgovor: kada se pojavi neka od sledećih situacija:

1. **Dupliranje koda:** Uočavate isti ili vrlo sličan kod na različitim mestima u programu. Taj kod onda zamenjujete jednim metodom kojega onda pozivate na tim mestima.
2. **Dugački metodi:** Vidite metod koji je vrlo dugačak (puno linija koda). Onda ga zamenjujete se više manjih metoda.
3. **Iskazi SWICH:** U programu ima na dosta mesta iskaz „switch“. Kod objektno-orientisanih sistema se oni mogu zameniti upotreboom polimorfizma
4. **Cirkulacija podataka:** Cirkulacija podataka se javlja kada se ista grupa podataka (polja u klasama, parametri u metodima) javljaju na više mesta u programu. To se otklanja njihovim učaurenjem u okviru jednog objekta.
5. **Spekulativna uopštjenost:** Do ovoga dolazi kada programeri uključuju uopštene iskaze u program, jer očekuju da će biti potrebna u budućnosti. To se jednostavno, može ukloniti.

Primenom strukturnih transformacija mogu se navedeni problemi rešavati. Na primer, metod ekstrakcije uklanja dupliranja i kreira novi metod. Preuređenjem iskaza uslovljavanja u programu, više testova vi zamenjujete jednim testom. Možete zameniti slične metode jednim metodom u njihovoj super klasi. Okruženja za interaktivni razvoj, kao što je Eclipse, uključuje

restrukturiranje u svoje editore. To olakšava nalaženje delova programa koji treba da se promene restrukturiranjem.

*Restrukturiranje projektnog rešenja je skuplji i složeniji metod poboljšanja programa.* On se primenjuje radi utvrđivanja odgovarajućih šabloni projektnih rešenja. Ova vrsta restrukturiranja se primenjuje kada samo restrukturiranje koda nije dovoljno. Program je isuviše degradiran da se na taj način on dovoljno poboljša. Restrukturiranjem projektnog rešenja onda može da dovede i do promene projektnog rešenja (engl. **design**) softvera.

## ŠTA JE RESTRUKTIRANJE KODA (CODE REFACTORING) (VIDEO)

*Georgia Teck predavanje - Šta je restruktiranje koda?*

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

## KVIZ: PITANJE - RESTRUKTURIRANJE KODA (VIDEO)

*Georgia Teck predavanje - Kviz pitanje - restrukturiranje koda*

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

## ODGOVOR NA PITANJE KVIZA - (RESTRUKTURIRANJE VIDEO)

*Georgia Teck predavanje - Odgovor na pitanje kviza - restrukturiranje*

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

## ZAŠTO VRŠITI RESTRUKTURIRANJE KODA? (VIDEO)

*Georgia Teck predavanje - Zašto vršiti restrukturiranje koda?*

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

## ISTORIJA PRIMENE RESTRUKTURIRANJA KODA (VIDEO)

*Georgia Teck predavanje - Istorija restruktuiranja*

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

## VRSTE RESTRUKTURIRANJA KODA (VIDEO)

*Georgia Teck predavanje -vrste restrukturiranja koda*

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

## RESTRUKTURIRANJE PROMENOM HIJERARHIJE (VIDEO)

*Georgia Teck predavanje - Restrukturiranje promenom hijerarhije*

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

## RESTRUKTURIRANJE PROMENOM IZKAZA USLOVLJAVANJA (1) (VIDEO)

*Georgia Teck predavanje - Restrukturiranje promenom izkaza uslovljavanja (1)*

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

## RESTRUKTURIRANJE PROMENOM IZKAZA USLOVLJAVANJA (2) (VIDEO)

*Georgia Teck predavanje - Restrukturiranje promenom izkaza uslovljavanja (2) - Consolodate Conditional Expression (video)*

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

## RESTRUKTURIRANJE DEKOMPONICIJOM USLOVLJAVANJA (VIDEO)

*Georgia Teck predavanje - Restrukturiranje dekompozicijom uslovljavanja (1) - Decompose Conditionals*

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

## IZVLAČENJE KLASE (VIDEO)

*Georgia Teck predavanje - Izvlačenje klase - - Extract Class*

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

## IZVLAČENJE METODA (VIDEO)

*Georgia Tech predavanje - Izvaćenje metoda*

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

## DEMO RESTRUKTURIRARANJA (VIDEO)

*Georgia Tech predavanje - Demo restrukturiranja*

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

## ODGOVOR NA PITANJE KVIZA - RESTRUKTURIRANJE IZVLAČENE METODA (VIDEO)

*Georgia Tech predavanje - Odgovor na pitanje kviza - restrukturiranje izvlačene metoda (video)*

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

## KVIZ: PITANJE - RESTRUKTURIRANJE IZVLAČENJEM METODA (VIDEO)

*Georgia Tech predavanje - Kviz pitanje - izvlačenje metoda*

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

## RIZICI RESTRUKTURIRANJA (VIDEO)

*Georgia Tech predavanje - Rizici prestrukturiranja*

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

## TROŠKOVI RESTRUKTURIRANJA KODA (VIDEO)

*Georgia Tech predavanje - troškovi restrukturiranja koda*

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

KADA NE TREBA PRIMENITI RESTRUKTURIRANJE KODA? (VIDEO)

*Georgia Tech predavanje - Kada ne treba primeniti restrukturiranje*

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

LOŠI MIRISI (BAD SMELLS) (VIDEO)

*Georgia Tech predavanje - Loši mirisi*

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

PRIMER ZA "LOŠE MIRISE" (VIDEO)

*Georgia Tech predavanje - primer za loše mirise*

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

## ▼ Poglavlje 4

# Upravljanje starim softverom

## ŠTA RADITI SA STARIM SOFTVEROM?

*Da bi se izabrala pravilna strategija evolucije starog sistema, potrebno je najpre izvršiti njegovu procenu i sa poslovnog i sa tehničkog stanovišta*

Organizacije razvijaju nove softverske sisteme, povećavaju pokrivenost svog poslovanja novim aplikacijama. Sve više shvataju važnost upravljanja životnim vekom ovih sistema i potrebu integracije faze razvoja i evolucije. Međutim, skoro svaka organizacija ima isti problem: Šta raditi sa starim sistemima? Oni i dalje zadovoljavaju izvesne funkcije, na više ili manje uspešan način. Najčešće su tehnološki prevaziđeni i teško se integrišu sa novim sistemima. Koju strategiju evolucije izabrati za stare sisteme? Postoje četiri mogućnosti:

- 1. Potpuno uništiti sistem:** Ovo treba primeniti kada sistem nema više efekta na poslovne procese u organizaciji. Poslovni procesi su se promenili i više ne zavise od ovog sistema.
- 2. Ostaviti sistem kakav jeste i nastaviti njegovo regularno održavanje:** Primeniti u slučaju kada je sistem i dalje potreban, dosta stabilan, a malo ima zahteva za njegovu promenu.
- 3. Izvršiti reinženjering sistema radi poboljšanja njegove sposobnosti održavanja:** Ovo se primenjuje kada je kvalitet sistema opao zbog izvršenih promena na sistemu i kada se i dalje predlažu nove promene (npr. razvoj novih interfejsa), tako da stari i novi sistem mogu da rade paralelno.
- 4. Zameniti sve ili neke delove sistema sa novim sistemom:** Ovu opciju treba izabrati kada zbog promene hardvera sistem ne može više da radi, ili kada se pojave novi softverski proizvodi koji se mogu relativno jednostavno i sa prihvatljivim troškovima upotrebiti za razvoj novog sistema. U mnogim slučajevima se primenjuje evolutivna zamena glavnih komponenata sa novim (nabavljenim) softverskim proizvodima, dok ostale komponente nastavljaju da se koriste

Kod složenijih sistema, može se koristiti više od jedne od navedenih opcija, jer za svaki podsistem se može koristiti i različita strategija.

Da bi se izabrala pravilna strategija evolucije starog sistema, potrebno je najpre izvršiti njegovu procenu i sa poslovnog i sa tehničkog stanovišta. Sa poslovnog stanovišta – znači da treba da ocenite da li vam je taj sistem uopšte više potreban za posao koji sada radite. Sa tehničkog stanovišta – treba da ocenite kvalitet softvera i softvera i hardvera koji ga podržavaju. Na osnovu ovakve analize zaključujete šta je najbolje da uradite sa starim sistemom

## ČETIRI KLASTERA STARIH SISTEMA

*Pre donošenja odluke o sudbini starih (zatečenih) sistema, vrši se analiza njihove poslovne vrednosti i njihovog kvaliteta, te se grupišu u četiri klastera*

Ako imate više starih sistema, potrebno je za svaki od njih da napravite analizu i da odredite poslovnu vrednost sistema. Dobijene rezultate možete prikazati na dijagramu koji pokazuje poslovnu vrednost u odnosu na kvalitet sistema (slika 1 ).



Slika 4.1 Klasterizacija starih softverskih sistema [1.2]

Kao što se može videti na slici, postoji četiri klastera ovih sistema:

- Niski kvalitet, niska poslovna vrednost:** Ove sisteme treba uništiti (napustiti) jer je skup rad ovih sistema a poslovna vrednost vrlo mala.
- Niski kvalitet, visoka poslovna vrednost:** Ove sisteme ne možete odbaciti jer imaju veliki značaj za poslovanje. Zbog velikih troškova održavanja (niski kvalitet) ovaj sistem se treba zameniti novim, i to, ako je moguće, novim gotovim („sa polica“ ili engl. „off-the-shelf“) softverskim proizvodima.
- Visok kvalitet, niska poslovna vrednost:** Ovi sistemi ne doprinose mnogo poslovanju, ali nije skupo njihovo održavanje (visok kvalitet). Zbog toga, ne isplati se njihova zamena i treba nastaviti njihovo korišćenje, sem u slučaju da ima zahteva za njihovu promenu koji zahtevaju visoka ulaganja ili novi hardver, kada je nužno njihovo odbacivanje.
- Visok kvalitet, visoka poslovna vrednost:** Ove sisteme treba zadržati i nastaviti njihovo normalno održavanje.

## UTVRĐIVANJE POSLOVNE VREDNOSTI SOFTVERSKOG SISTEMA

*Poslovna vrednost softvera u nekoj organizaciji se određuje analizom frekvencije njegove upotrebe, brojem poslovnih procesa koje podržava, stepenom pouzdanosti i važnosti rezultata.*

Kako utvrditi poslovnu vrednost nekog softverskog sistema? Potrebno je prvo da identifikujete zainteresovane aktere tih sistema i onda da im postavite niz pitanja o sistemu. U tim razgovorima, možete diskutujete o sledećim temama:

1. **Upotreba sistema:** Ako se sistemi samo povremeno koriste, ili ih koristi mali broj ljudi, onda oni imaju nisku poslovnu vrednost. To se često dešava ako je prvobitna uloga sistema prevaziđena promenama u poslovanju ili se ona sada realizuje na neki drugi, efektniji način. Međutim, ako je i povremeno korišćenje vrlo važno za organizaciju, onda se takav sistem ne odbacuje.
2. **Poslovni procesi koje sistem podržava:** Svaki sistem podržava jedan ili više poslovnih procesa. Ako je sistem nefleksibilan, i ovi procesi ne mogu da se menjaju. Međutim, zbog promene okruženja i poslovnih zahteva, vrlo često poslovni procesi moraju da se menjaju. Sistem koji ne može da podrži takve promene onda treba izbaciti iz upotrebe.
3. **Pouzdanost sistema:** Pored tehničke, postoji i poslovna pouzdanost. Ako sistem nije pouzdan, te direktno time ometa poslovne partnere, ili zahteva ulaganje resursa u rešavanje problema pouzdanosti, onda sistem ima nisku poslovnu vrednost.
4. **Rezultati sistema:** Treba utvrditi važnost rezultata koje proizvodi sistem na uspešno funkcionisanje poslovanja. Ako posao zavisi od ovih rezultata, onda sistem ima visoku poslovnu vrednost. S druge strane, ako se rezultati sistema retko koriste, ili se mogu dobiti i na drugi način, onda je poslovna vrednost sistema niska.

## OCENJIVANJE OKRUŽENJA U KOME RADI SOFTVER

*Da bi ocenili okruženje, potrebno je da izvršite neka merenja sistema, njegovog okruženja i njegovog procesa održavanja*

Da bi ocenili softverski sistem s tehničkog stanovišta, treba da uzmete u obzir i aplikacioni sistem i okruženje (hardver, softver) u kom on radi. Mnogi sistemi se moraju menjati jer se okruženje promenilo.

Da bi **ocenili okruženje**, potrebno je da izvršite neka merenja sistema i njegovog okruženja i njegovog procesa održavanja. Koristite podatke, kao što su troškovi održavanja hardvera i softvera za podršku, broj otkaza hardvera u nekom periodu i frekvenciju izvršenih popravki softvera za podršku. Slika 2 prikazuje faktore koje bi trebalo da uzmete u obzir prilikom ocenjivanja okruženja u kom stari sistem radi. To nisu tehničke karakteristike okruženja. Morate takođe uzeti u obzir i pouzdanost proizvođača opreme. Ako oni više nisu u poslu, onda je moguće da dalja podrška opreme nije moguća.

Faktor	Pitanja
Stabilnost proizvođača	Da li proizvođač još postoji? Da li je finansijski stabilan i da li je verovatno da će nastaviti svoj posao? Ako proizvođač nije više u poslu, da li još neko održava njegove sisteme?
Brzina otkaza	Da li hardver ima visoku učestanost prijavljenih otkaza? Da li softver za podršku pada i prouzrokuje da i sistem pada?
Starost	Koliko je star hardver i softver? Što su hardver i softver za podršku stariji, to je manje upotrebljiv. Oni i dalje funkcioniše korektno ali su moguće značajne ekonomske i poslovne koristi od prihvatanja modernijeg sistema.
Performanse	Da li su performanse sistema odgovarajuće? Da li problemi sa performansama imaju značajan uticaj na korisnike sistema?
Zahtevi za podrškom	Koja je lokalna podrška potrebna za hardver i softver? Ako tu podršku prate visoki troškovi, onda treba razmotriti mogućnost zamene sistema.
Troškovi održavanja	Koliki su troškovi održavanja hardvera i licenci softvera za podršku? Stariji hardver može da ima visoke troškove održavanja nego moderni sistemi. Softver za podršku može imati visoke troškove godišnjih licenci.
Interoperativnost	Da li postoje problemi sprezanja sistema sa drugim sistemima? Da li kompjajleri, na primer, mogu da koriste sadašnju verziju operativnog sistema? Da li je potrebna emulacija hardvera?

Slika 4.2 Tabela-1 Ocena okruženja [1.2]

## OCENJIVANJE KVALITETA APLIKACIJE

*Najvažniji faktori za ocenjivanje kvaliteta aplikacije su vezani za pouzdanost sistema, poteškoće u održavanje sistema i dokumentaciju.*

Da bi ocenili kvalitet aplikacije (starog softverskog sistema) treba da ocenite niz faktora (slika 3 ) koji su prvenstveno vezani za pouzdanost sistema, poteškoće u održavanje sistema i dokumentaciju.

Faktor	Pitanja
Razumljivost	Koliko je teško da se razume izvorni kod sadašnjeg sistema= Koliko je složena njegova upravljačka struktura? Da li promenljive imaju smisalne nazive koji odražavaju njihovu funkciju?
Dokumentacija	Koja je dokumentacija sistema dostupna? Da li je dokumentacija kompletna, konsistentna i raspoloživa?
Podaci	Da li postoji eksplicitni model podataka sistema? Do kog stepena se duplicitiraju podaci po datotekama? Da li su podaci koje koristi sistem tačni i konsistentni?
Performanse	Da li su performanse aplikacije odgovarajuće? Da li problemi sa performansama imaju značajan uticaj na korisnike sistema?
Programski jezik	Da li su raspoloživi moderni kompjajleri za programske jezike koji se koristio u razvoju sistema? Da li postoji eksplicitan opis verzije komponenti koje koristi sadašnji sistem?
Upravljanje konfiguracijom	Da li se sve verzije svih delova sistema kontrolišu uz pomoć sistema za upravljanje konfiguracijom sistema? Da li postoji eksplicitni opis verzija komponenti koje upotrebljava sadašnji sistem?
Podaci za testiranje	Da li postoje podaci testiranja sadašnjeg sistema? Da li postoji zapis urađenih regresivnih testova kada su nova svojstva bila dodavana sistemom?
Lične veštine	Da li postoje ljudi koji imaju veštine neophodne za održavanje aplikacije? Da li postoje ljudi koji imaju iskustvo rada sa sistemom?

Slika 4.3 Tabela-2 Ocena kvaliteta aplikacije [1.2]

Da bi izvršili ocenu rada dosadašnjeg sistema (aplikacije) potrebni su vam sledeći podaci:

- 1. Broj zahteva za promenu sistema:** Promene sistema pogoršavaju strukturu sistema i otežavaju dalje promene. Što je veći broj akumuliranih promena, to je niži kvalitet sistema.
- 2. Broj korisničkih interfejsa:** Kod sistema koji se baziraju na formama koje korisnici koriste, značajan faktor je njihov broj. Što ih je više, veća je verovatnost nekonsistentnosti i ponovljivosti kod ovih interfejsa u budućnosti, a prilikom daljih promena sistema.
- 3. Količina podataka koje sistem koristi:** Što je veća količina podataka (broj datoteka, veličina baza podataka itd.), veća je verovatnoća da će doći do pojave nekonsistenih podataka koji smanjuju kvalitet softvera.

## DRUGI FAKTORI ODLUČIVANJA

*Često se odluke donose i na osnovu manje objektivnih ocena, jer se zasnivaju na organizacionim ili političkim faktorima*

Posle ocene kvaliteta samog softvera, tj. kvaliteta aplikacije, kao i uzimanja u obzir i faktora okruženja u kome softver radi, donosi se ocena softverski sistem s tehničkog stanovišta. Konačna odluka o zadržavanju ili menjanja postojećeg softverskog sistema zavisi od ocene njegove poslovne vrednosti i njegovog kvaliteta.

U idealnom slučaju, pri donošenu odluke o sudbini postojećih sistema, trebalo bi se služiti isključivo objektivnim ocenama svih navedenih faktora. Međutim, često se odluke donose i na osnovu manje objektivnih ocena, jer se zasnivaju na organizacionim ili političkim faktorima. Navešćemo nekoliko primera:

- Pri spajanju dve kompanije, najčešće se zadržavaju sistemi jačeg partnera, a napuštaju drugi sistemi.
- Ako organizacija odluči da pređe na novu hardversku platformu, onda to zahteva promenu aplikacija.
- Ako nema potrebnog budžeta za transformaciju sistema, onda se nastavlja sa održavanjem starog sistema, iako to dovodi do, dugoročno gledano, do povećanih troškova.

## LEGACY SYSTEMS (VIDEO)

*Trajanje: 7:16 minuta*

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

# VIDEO PREDAVANJE ZA OBJEKAT "UPRAVLJANJE STARIM SOFTVEROM"

*Trajanje video snimka: 16min 53sek*

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

## ▼ Poglavlje 5

### Pokazna vežba

#### OPIS VEŽBE (45 MINUTA)

##### *Uvod i rad sa CI continuous integration korišćenjem Jenkins alata*

Osnovni cilj ove vežbe jeste da demonstrira i uputi studente u Jenkins alat. Kako ga konfigurisati i podesiti i koristiti u realnom okruženju. Ova vežba neće pokriti sve detalje koje se odnose na Jenkins ali će prikazati veliku sliku, način rada i upotrebe Jenkins kao (Continuous integration) alata. Pored toga, kao osnovni primer moćiće da uradite build projekta uz pomoć Jenkins alata.

#### JENKINS

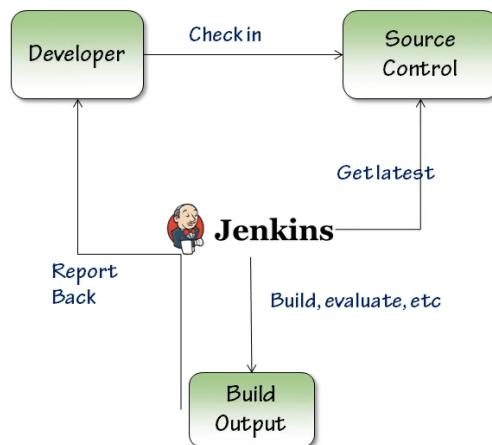
*Jenkins je softver namenjen CI (continuous integration) koji se pokreće na serveru, pomaže u build procesu i aktivnostima koji se odnose u najvećoj me*

Jenkins Vam može omogućiti automatizaciju većine poslova kako bi dobili CI u okviru nekog projekta. Jenkins je otvorenog koda i dostupan je za besplatnu upotrebu. Jenkins u suštini predstavlja Web aplikaciju baziranu na Java tehnologijama. Bez obzira što je bazirana na Java tehnologijama, Jenkins uporedo dobro radi i sa drugim programskim jezicima (npr c#). Jenkins je dosta pogodan takođe iz razloga što poseduje veliki broj dodataka koji se mogu ugraditi u njega za specifične primene. Tako je moguće preuzeti dodatak za Build Andorid ili IOS aplikacije. Takođe postoji mogućnost kreiranja sopstvenog dodatka koji radi specifičnu stvar koja je samo Vama potrebna.

Jenkins ima mogućnost nadgledanja repozitorijuma sa kodom (Source controle) kao što su SVN, GIT i drugi. Nakon toga, Jenkins se može podesiti tako da ima uvid u određenom vremenskom periodu, svaki put kada programer pošalje novu verziju koda ili na razne druge događaje u zavisnosti od potrebe. Nakon toga na Jenkinsu je da uradi šta mu je definisano. Najčešće je to puštanje build-a aplikacije koja je uzeta sa repozitorijuma, startovati testove i drugo. Nakon završavanja određenih testova dolazi do rezultata koji se šalju najčešće programerima ili određenoj osobi koja je zadužena za kontrolu nad kodom. Rezultat može biti da je sve uspešno završeno, ali i informacija da neki test nije prošao uspešno, došlo je do greške u build-u i slično. Ove informacije pomažu u otkrivanju grešaka i sigurnošću da neće biti puštena test verzija koja nije prošla proveru. Jenkins je tu da uradi automatizovanu proveru. Na slici 2 je prikazan proces rada Jenkins softvera.



Slika 5.1.1 Logo Jenkins softvera [Izvor: Autor]



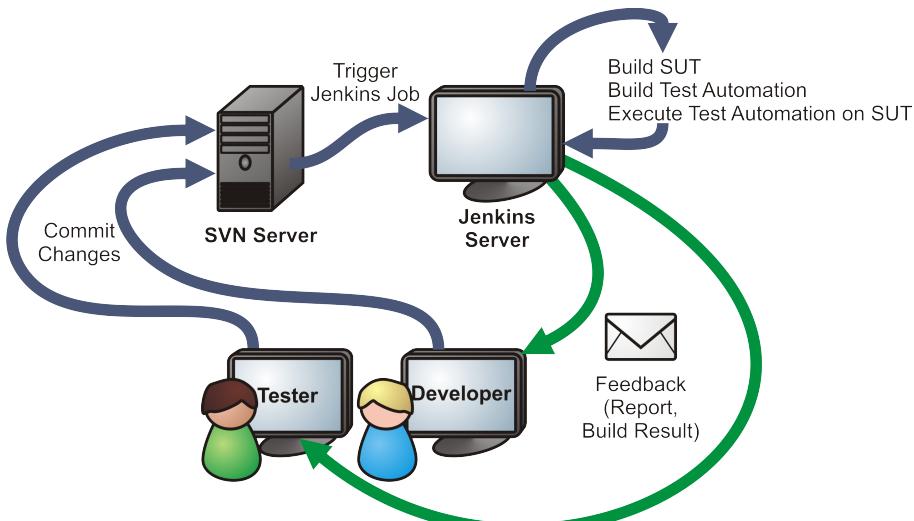
Slika 5.1.2 Proces rada Jenkins softvera [Izvor: Autor]

## CONTINUOUS INTEGRATION (KONTINUALNA INTEGRACIJA KODA)

*Šta predstavlja pojam Continuous Integration – stalno povezivanje koda*

Continuous Integration (CI) ili u slobodnom prevodu (stalno povezivanje koda) odnosi se na potrebu za učestalim povezivanjem s kodom na kom rade ostali članovi tima. Ukoliko na istim delovima koda radi više ljudi, što duže traje razvoj (bez međusobne sinhronizacije), to su veće razlike i teže je povezati kod u jednu smislenu celinu kada napokon za to dođe vreme. Takođe, veća je šansa da nastanu oku nevidljivi bugovi. Iz ove prakse proizlazi čitav niz metoda bez kojih je danas praktično nezamisliv razvoj kvalitetnog softvera:

1. Održavanje repozitorijuma koda koji služi za čuvanje i sinhronizaciju koda u timu. Alati poput CVS, SVN ili Git danas su neizostavni i na najmanjim projektima.
2. Automatizacija kreiranja aplikacijskih artefakata - često može uključivati i instalaciju aplikacije na testu okolini. Za to se tipično koriste dobro poznati Ant, Maven, IBM Rational Build Forge i sl.
3. Učestalo stavljanje koda na repozitorijum. Testiranje koda trebalo bi da se izvršava na okolini koja je vrlo slična produkcijskoj.



Slika 5.1.3 Primer CI [Izvor: Autor]

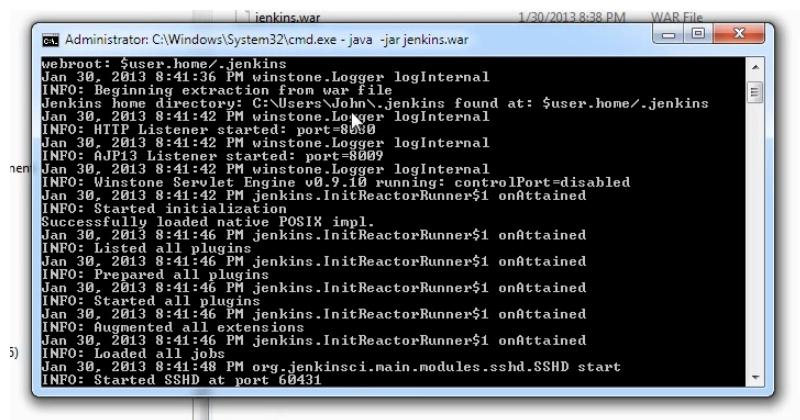
## JENKINS INSTALACIJA

### *Preduslovi za instalaciju i proces instalacije Jenkins softvera*

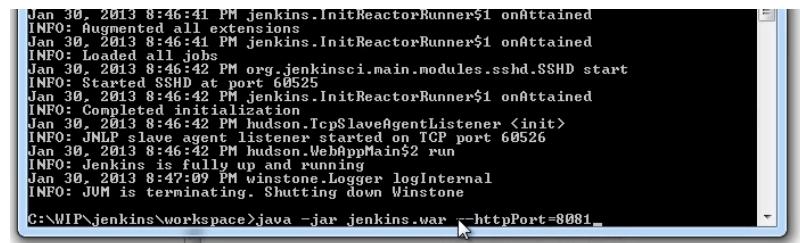
- Nepohodno je instalirati Java 1.5 verziju ili noviju, zbog toga što je Jenkins razvijan na Java platformi.
- Poželjno je konfigurisati Java Path (Ukoliko se Jenkins pokreće iz konzole kako bi se lakše pozvala java)
- Web server (samo u slučaju ukoliko zelite da hostuje aplikaciju izvan Jenkins softvera)

Nakon ispunjavanja svih preduslova moguće je preuzeti Jenkins sa zvaničnog sajta (<https://jenkins.io/download/>) koji se može instalirati na dva načina. Jedan način je u okviru .war extenzije kao standardne web aplikacije koju je moguće postaviti na neki od aplikacionih servera (glassfish, tomcat) i startovati. Takođe Jenkins ima "ugrađen" server tako da je ne zavisan od drugog aplikacionog servera. Ukoliko bi uneli komande (java -jar jenkins.war) u terminal (konzolu), Jenkins bi se automatski startovao sa svojim definisanim portom 8080 kao što je prikazano na slići 4. Takođe moguće je definisati specifičan port za startovanje (slika 5).

Drugi način je sa sajta preuzeti specifičan paket instalaciju za određenu platformu na kojoj će se Jenkins nalaziti.



Slika 5.1.4 Prikaz startovanog Jenkins-a iz konzole [Izvor: Autor]



Slika 5.1.5 Proces startovanja Jenkinsa sa specifičnog porta [Izvor: Autor]

## JENKINS INSTALACIJA (VIDEO)

*How to install Jenkins on Windows 10*

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

## DASHBOARD

*Prvi prikaz Jenkins softvera nakon instalacije*

Kada se pokrene, Jenkins (fabrički) nalazi se na portu 8080. Obzirom da je Web aplikacija, možete joj pristupiti sa linka (<http://localhost:8080>). Gde će se otvoriti Radna površina (Dashboard) Jenkins alata. Glavni prozor za podešavanje Jenkins alata nalazi se u delu ManageJenkins.



Slika 5.1.6 Jenkins Dashboard i Menage Jenkins [Izvor: Autor]

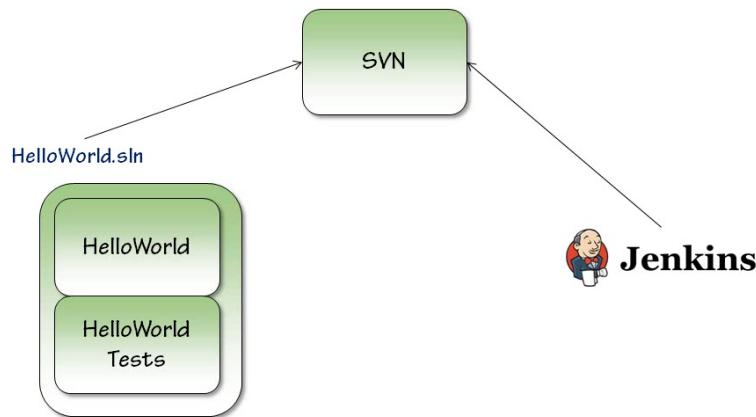
## JENKINS POSLOVI

### *Kreiranje jednostavnog Jenkins posla i aktivnosti*

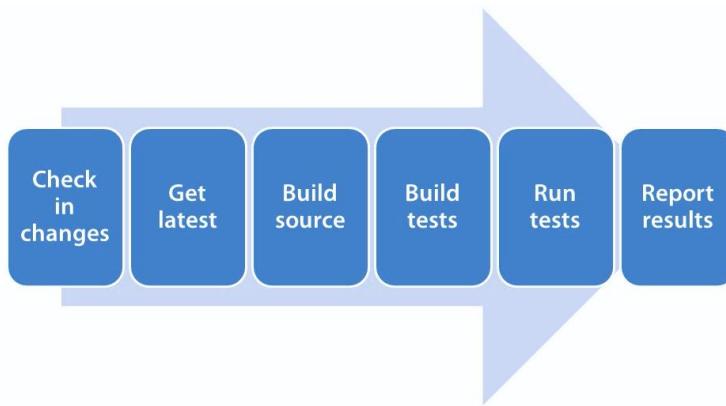
Na početku treba definisati jednostavnu strukturu projekta sistema na kome će Jenkins raditi. Na slici 7 je prikazan jednostavan projekat koji sadrži izvršnu klasu, klasu koja testira napisani kod odnosno Test klasu. Taj jednostavan projekat se prosleđuje na neki Source control u ovom primeru SVN. Dalji tok je Jenkins koji komunicira sa SVN kako bi preuzeo kod i radio na njemu.

Obzirom da će ova vežba pokazati realan primer u okviru NetBeans razvojnog okruženja napravljen je jednostavan HelloWorld primer koji će demonstrirati upotrebu Jenkins okruženja. Hello World će imati 2 jednostavna Junit testa, kao i jednu klasu. (Projekat možete preuzeti iz dodatnog materijala).

Postupak procesa postavljanja aplikacije upotrebom Jenkins-a prikazan je na slici 8 .



Slika 5.1.7 Prikaz arhitekture aplikacije [Izvor: Autor]



Slika 5.1.8 Proces rada Jenkins [Izvor: Autor]

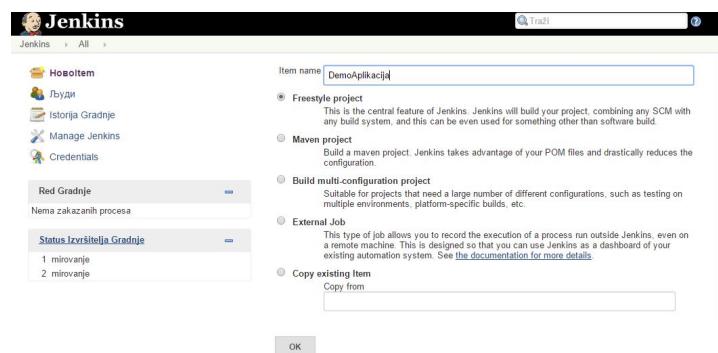
## KRIERANJE POSLA

### *Proces kreiranja posla unutar Jenkins alata*

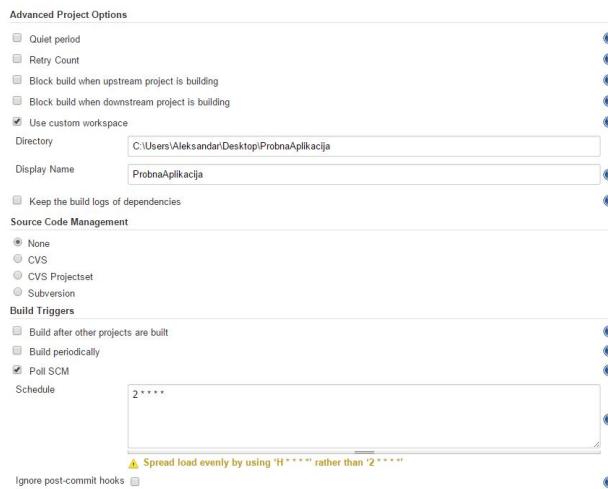
Pritiskom na dugme New Items otvara se prozor za definisanje koji tip projekta se kreira. Obzirom da je demo aplikacija koja je napravljena obična Java desktop aplikacija i ne nalazi se ni na jednom SVN,GIT repozitorijumu projekt koji će biti kreiran je FreeStyle project kao što je prikazano na slici 9 . Nakon odabira projekta otvara se nova stranica na kojoj se mogu definisati posebne pogodnosti koje Jenkins omogućava za projekat koji se kreira. Postoje vremenski okidači, šta se dešava pre, a šta posle build-ovanja projekta. Okidači (Build Triggers) u Jenkinsu imaju tri mogućnosti. Prva mogućnost je (Build after other projects are built) koja označava da će ovaj projekat biti buildovan nakon uspešnog završetka nekog drugog projekta od koga možda zavisi, drugi je (Build periodically) koji omogućava periodično pokretanje Build-a npr na svakih pet minuta, jednom dnevno i dr. i treći je (Poll SCM- Poll Source control) i ovo je najkorišćeniji metod koji omogućava podešavanje tako da Jenkins posmatra svaku SVN promenu i ukoliko je ima uradi build. Postoji još mogućnosti koje su pokrivene ovom metodom i mogu se pogledati na sledećem linku

(<https://wiki.jenkins-ci.org/display/JENKINS/Building+a+software+project>).

Putanja do projekta koji se Builduje u ovom primeru prikazana je na slici 10 .



Slika 5.1.9 Prikaz kreiranja posla [Izvor: Autor]



Slika 5.1.10 Prikaz konfiguracije projekta [Izvor: Autor]

## PREGLED PROJEKTA

### *Pregled kreiranog projekta i dobijenog workspace-a*

Od naprednih podešavanja koja su bila ponuđena neophodno je podesiti putanju do Build-a, vreme izvršavanja Builda, koje su akcije kada se Build uspešno završi ili u slučaju da Build „padne“. Nakon toga na Dashboard-u se pojavljuje projekat koji je kreiran kao što je prikazano na slici 11 .

All	+				
S	W	Name ↓	Poslednje uspešno	Poslednja greška	Poslednje trajanje
		ProbnaAplikacija	9 min 42 sec - #3	Nije dostupno	0.39 sec

ikonica: [S](#) [M](#) [L](#)

Legenda [RSS za sve](#) [RSS za neuspešne](#) [RSS samo za najnovije gradnje](#)

[Noviji opisi](#)

Slika 5.1.11 Prikaz projekta na Dashboard-u [Izvor: Autor]

Ukoliko sačekamo vreme izvršavanja Build-a ili „ručno“ pokrenemo build projekta Jenkins će ukoliko je SVN/GIT preuzeti kod ili takođe ukoliko je iz lokalnog foldera projekat kao što je u ovom slučaju. Prikaz šta je preuzeto sa Source kontrole možete pogledati pritiskom na naziv projekta na Dashboard-u i odabir Workspace. Nakon toga otvara se menadžer fajlova tog projekta. Prikaz demo projekta nalazi se na slici 12 .



Slika 5.1.12 Prikaz strukture foldera projekta [Izvor: Autor]

## REZULTAT POSLA - PRIKAZ STATUSA PROJEKTA

### *Pregled statusa projekata*

Ukoliko je projekat uspešno kreiran pored naziva projekta biće prikazan plavi krug koji označava da je build prošao uspešno. Ukoliko стоји сунце pored naziva projekta to takođe znači da su svi Build-ovi na tom projektu uspešno prošli. Ukoliko je krug crven ili nije prikazano сунце, to znači da neki od buildova nisu prošli. Prikaz uspešnog i ne uspešnog build-a dat je na slici 13 .

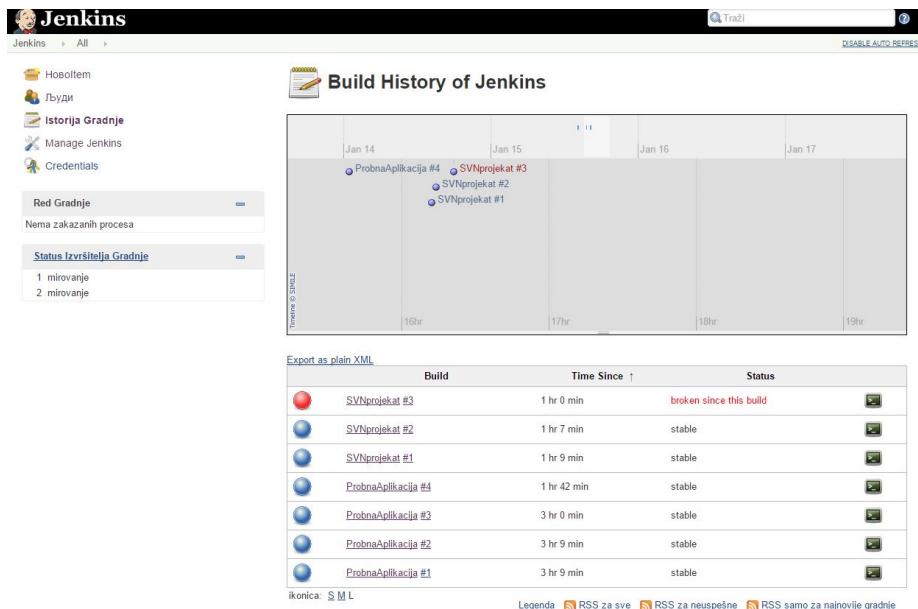
All	+	S	W	Name	Poslednje uspešno	Poslednja greška	Poslednji trajanj
		<a href="#">ProbnaAplikacija</a>			1 hr 38 min - #4	Nije dostupno	0.29 s
		<a href="#">SVNprojekat</a>			1 hr 3 min - #2	56 min - #3	0.84 s

ikonica: [S](#) [M](#) [L](#)

[Legenda](#) [RSS za sve](#) [RSS za neuspešne](#)

Slika 5.1.13 Prikaz svih statusa projekata na Dashboard-u [Izvor: Autor]

Postoji mogućnost detaljnijeg prikaza svih Build-ova projekata. Kao što je prikazano na slici 14 . Ukoliko se pojavi greška u build-u kada se klikne na konzolu pojavljuje se kompletan prikaz greške koja je nastala. Prikaz Greške



Slika 5.1.14 Grupni prikaz Build-a [Izvor: Autor]

## REZULTAT POSLA

### *Pregled statusa projekata - prikaz grešaka i prikaz istorije projekta*

Ukoliko je projekat uspešno kreiran pored naziva projekta biće prikazan plavi krug koji označava da je build prošao uspešno. Ukoliko стоји сунце поред назива пројекта то такође значи да су сви Build-ovi на том пројекту успешио прошли. Уколико је кружнице црвени или нисе приказано сунце, то значи да неки од buildova нису прошли. Prikaz uspešnog i ne uspešnog build-a dat je na slici 15 .

Postoji mogućnost detaljnijeg prikaza svih Build-ova projekata. Kao što je prikazano na slici 16 . Ukoliko se pojavi greška u build-u kada se klikne na konzolu pojavljuje se kompletan prikaz greške koja je nastala. Prikaz Greške dat je na slici 17 .

```

Started by user anonymous
Building in workspace C:\Program Files (x86)\Jenkins\jobs\SVNprojekat\workspace
Checking out a fresh workspace because there's no workspace at C:\Program Files
(86)\Jenkins\jobs\SVNprojekat\workspace
Cleaning local Directory
Checking out https://code.bmu.internal:7788/svn/se201/ProbnaAplikacija at revision '2015-01-15T15:21:21+0100'
A test
A .MainTest.java
A nbproject
A nbproject/MainTest.java
A nbproject/nbproject
A nbproject/project.properties
A nbproject/project.xml
A nbproject/genfiles.properties
A nbproject/build-impl.xml
A manifest.mf
A src
A src/demo
A src/demo/Main.java
A build.xml
U
At revision 3
no change for https://code.bmu.internal:7788/svn/se201/ProbnaAplikacija since the previous build
Recording test results
ERROR: hudson.tasks.junit.JUnitResultArchiver aborted due to exception
java.io.IOException: Failed to read C:\Program Files (x86)\Jenkins\jobs\SVNprojekat\workspace\manifest.mf
is this really a JUnit report file? Your configuration must be matching too many files
at hudson.tasks.junit.TestResult.parse(TestResult.java:290)
at hudson.tasks.junit.TestResult.getResult(TestResult.java:228)
at hudson.tasks.junit.TestResult.parsePossiblyEmpty(TestResult.java:228)
at hudson.tasks.junit.TestResult.parse(TestResult.java:146)
at hudson.tasks.junit.TestResult.<init>(TestResult.java:122)
at hudson.tasks.junit.JUnitParser$ParseResultCallable.invoke(JUnitParser.java:119)
at hudson.tasks.junit.JUnitParser$ParseResultCallable.invoke(JUnitParser.java:93)
at java.util.concurrent.FutureTask.run(FutureTask.java:266)
at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1146)
at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:615)
at java.lang.Thread.run(Thread.java:744)
at hudson.FilePath.act(FilePath.java:99)
at hudson.tasks.junit.JUnitParser.parseResult(JUnitParser.java:90)
at hudson.tasks.junit.JUnitResultArchiver.parse(JUnitResultArchiver.java:120)
at hudson.tasks.junit.JUnitResultArchiver.perform(JUnitResultArchiver.java:137)
at hudson.tasks.junit.JUnitResultArchiver$1.call(JUnitResultArchiver.java:74)
at hudson.tasks.BuildStepMonitor$1.onCompleted(BuildStepMonitor.java:28)
at hudson.model.AbstractBuild$AbstractBuildExecution.perform(AbstractBuildExecution.java:770)
at hudson.model.AbstractBuild$AbstractBuildExecution.post(AbstractBuildExecution.java:183)
at hudson.model.Run.execute(Run.java:178)
at hudson.model.FreeStyleBuild.run(FreeStyleBuild.java:683)
at hudson.model.FreeStyleBuild$FreeStyleBuildExecution.doRun(FreeStyleBuild.java:43)

```

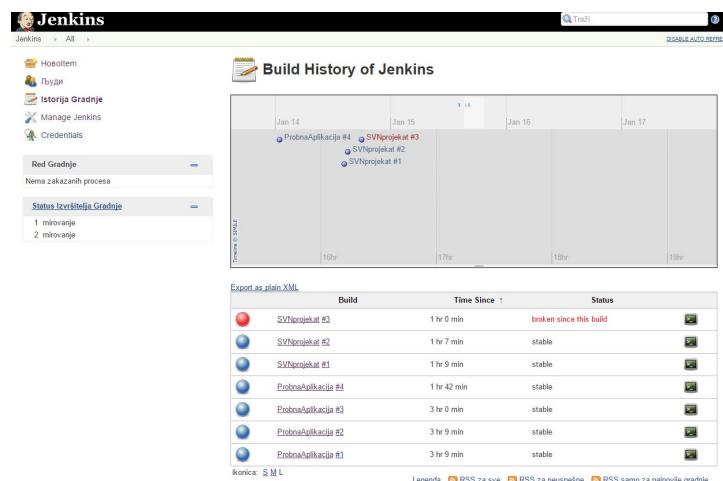
Slika 5.1.15 Prikaz greške tokom Build-a projekta [Izvor: Autor]

All	+	S	W	Name	Poslednje uspešno	Poslednja greška	Poslednji trajanj
				<a href="#">ProbnaAplikacija</a>	1 hr 38 min - #4	Nije dostupno	0.29 s
				<a href="#">SVNprojekat</a>	1 hr 3 min - #2	56 min - #3	0.84 s

ikonica: [S](#) [M](#) [L](#)

Legenda RSS za sve RSS za neuspešne

Slika 5.1.16 Prikaz grešaka projekta [Izvor: Autor]



Slika 5.1.17 Prikaz istorije Build-a projekta [Izvor: Autor]

## ❖ 5.1 Upravljanje verzijama - Git

### KREIRANJE REPOZITORIJUMA I POSTAVLJANJE VERZIJE (15 MINUTA)

#### *Kreiranje repozitorijuma i postavljanje verzije za postojeći projekat*

Git je **sistem kontrole verzija** koji se koristi za **praćenje promena** u fajlovima, kao i za **koordinisanje rada** više ljudi na istom projektu. Git je distribuirani sistem kontrole verzija, što znači da se na svakom računaru koji radi na datom projektu nalazi repozitorijum sa kompletnom istorijom izmena. Pored toga, repozitorijum se može postaviti na neki od centralnih servera (što se može učiniti na primer na <https://github.com/>), a zatim da svako ko učestvuje u razvoju sinhronizuje svoju verziju sa trenutno aktuelnom.

Git može da se preuzme sa: <https://git-scm.com/downloads>

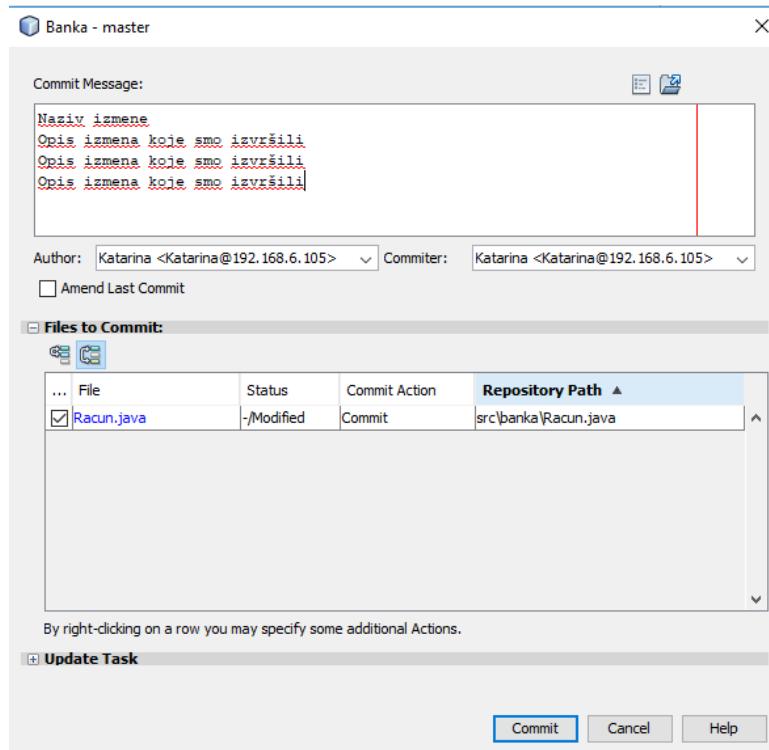
Pored toga, posebno za Windows postoje prilagođene GUI verzije alata, koje se mogu preuzeti sa: <https://gitforwindows.org/>

**Većina IDE alata za razvoj pružaju podršku za rad sa Git sistemom.** Detaljno uputstvo za rad sa Git sistemom u **NetBeans** alatu može se naći ovde: <https://netbeans.org/kb/docs/ide/git.html>

Kako bismo napravili početni repozitorijum za neki projekat u NetBeansu, treba uraditi desni klik na željeni projekat, a zatim izabratи: **Versioning > Initialize Git Repository**.

Nakon uspešnog kreiranja repozitorijuma dobijamo opciju **Git** kada uradimo desni klik na projekat.

Tokom razvoja, želimo da sačuvamo izmene i da ih kasnije vidimo u istoriji. Za to koristimo naredbu **commit**. Kada izaberemo git commit, dobijamo prozor sa slike 1, gde se vide fajlovi koji su izmenjeni (pri prvom komitu će svi fajlovi biti na listi) i gde možemo da unesemo opis izmena:



Slika 5.2.1 Prikaz prozora za commit naredbu u okviru git-a [Izvor: Autor]

## POSTAVLJANJE PROJEKTA NA UDALJENI REPOZITORIJUM

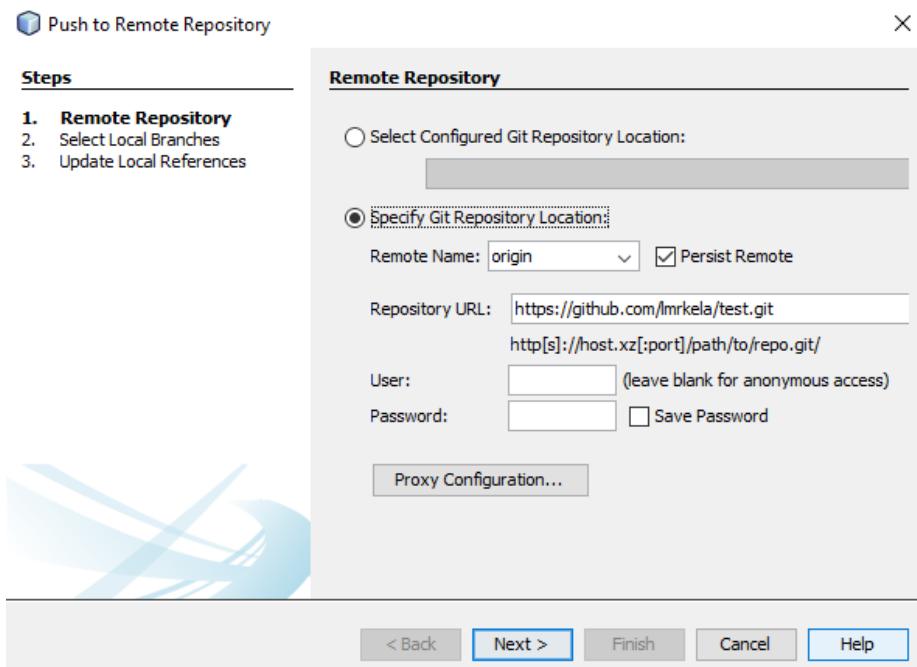
### *Postavljanje izmena pomoću git push naredbe na udaljeni repozitorijum*

Kada smo uradili prvi commit, možemo da vidimo sve promene sa **git > show changes** ili da vidimo celu istoriju promene fajlova, tj. spisak svih commit naredbi sa **git > show history**.

Kako bismo omogućili da više programera razvija jedan softver, moramo da obezbedimo **centralni udaljeni repozitorijum**. Posle toga, svaki programer može da postavi svoje lokalne izmene na centralni repozitorijum sa naredbom **git > remote > push to upstream** ili da preuzme trenutnu verziju sa **git > remote > pull from upstream**.

Prvo je potrebno da se registrujete na: <https://github.com/>. Zatim kliknite na **create new repository**. Gde unosite ime repozitorijuma i određujete prava pristupa, tj. da li je privatni ili javni. Kada odete na stranicu novog repozitorijuma imate opciju **Clone or download**, gde se nalazi link koji je potreban za povezivanje sa NetBeans Git sistemom. Primer jednog linka je: <https://github.com/lmrkela/test.git>

Prvi put kada postavljate projekat na udaljeni repozitorijum, birate opciju **git > remote > push...** i nakon toga se otvara prozor sa slike 1. Prethodno kopirani URL postavite u Repository URL i unesete korisničko ime i šifru.



Slika 5.2.2 Povezivanje na udaljeni repozitorijum [Izvor: Autor]

## KAKO KORISTITI GIT

*Tutorial za upotrebu Git sistema iz komandne linije*

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

## ▼ Poglavlje 6

### Individualna vežba

#### ZADACI ZA SAMOSTALAN RAD OD 1 DO 4

*Upotreba Jenkins softverskog alata.*

**Zadatak 1** (15 minuta)

Instalirati Jenkins korišćenjem .war fajla

**Zadatak 2** (30 minuta)

Napraviti Java aplikaciju kalkulator za dva operanda, sa metodama za sabiranje i oduzimanje. Napraviti nekoliko JUnit testova. Postaviti aplikaciju na Jenkins tako da se na svakih 5 minuta ponovo generiše (Build) aplikacija. Ukoliko Build ne prođe potrebno je da Jenkins pošalje mail na Vašu adresu.

**Zadatak 3** (20 minuta)

Naći proizvoljan projekat na GitHub-u. Napraviti novi projekat koji će se zvati GitHubTest. Povezati projekat sa tim GitHub nalogom. Uraditi generisanje projekta svako jutro u 9AM u slučaju da je bilo promena na kodu.

**Zadatak 4** (15 minuta)

Pronaći i instalirati bilo koji Plugin na Jenkins okruženju

#### ZADACI ZA SAMOSTALAN RAD OD 5 DO 6

*Upotreba Git alata.*

**Zadatak 5** (15 minuta)

Napravite nalog na GitHub-u i kreirajte jedan repozitorijum za projekat konvertora latinice u cirilicu i obrnuto.

**Zadatak 6** (20 minuta)

Implementirajte konvertor latinice u cirilicu i postavite sve izmene na Git odovarajućim komandama.

## ▼ Zaključak

### POUKE OVE LEKCIJE

*Evolucija softvera je proces izmena softvera tokom njegovog životnog veka, koje se rade u cilju uklanjanja grešaka, prilagođavanju promenama u okruženju*

Pouke ove lekcije:

1. Razvoj i evolucija softvera treba da budu integrisani i iterativni procesi koji se mogu prestaviti spiralnim modelom.
2. Troškovi održavanja sistema razvijenih po posebnom zahtevu kupca, najčešće su veći od troškova nabavke (razvoja) sistema.
3. Proces evolucije softvera je vođen zahtevima za promenama, koje obuhvataju analizu uticaja promena, planirana izdanja softvera i promenu implementacije.
4. Lemanov zakoni, kao što je onaj o kontinualnim promenama, opisuju niz iskustava koji su rezultat dugotrajnih studija o evoluciji sistema.
5. Postoji tri tipa održavanja softvera: a) popravka grešaka, b) promena softvera da bi radi sa novim okruženjem, i c) implementacija ovih ili promenjenih zahteva.
6. Reinženjering softvera se bavi promenom strukture softvera i njegove dokumentacije da bi se olakšale sprovođenje i razumevanje promena softvera.
7. Restrukturiranje softvera omogućava da se malim promenama u program održava njegova funkcionalnost, ima karakter preventivnog održavanja.
8. Poslovna vrednost starih sistema i kvalitet aplikativnog softvera i njegovog okruženja bi trebalo da bude ocenjeno da bi se donela odluka da li bi trebalo da sistem bude zamenjen, transformisan ili zadržan.

### LITERATURA

#### *Preporučena literatura*

##### **1. Obavezna literatura:**

1. Predavanja, vežbanja i dodatni materijali objavljeni na sistemu za e-učenje, 2020
2. Ian Sommerville, Software Engineering, Tenth Edition, Pearson Education Inc., 2016.

##### **2. Dopunska literatura:**

1. B. Bruegge, A. Dutoit, Object-Oriented Software Engineering – Using OML, Patterns, and Java, Thirth Edition, Prentice Hall, 2010
2. O'Reilly, Head First Design Patterns
3. Partha Kuchana, Software Architecture Design patterns in Java

4. Design Patterns - Elements of Reusable Object-Oriented Software, Eric Gamma, Richard Helm, Ralph Johnson, Jogns Viisides, 19
5. Design Patterns in Java Tutorial, tutorialspoint.com

**3. Veb lokacije :** Na ovim veb lokacijama možete naći opise mnogih šablona projektovanja sa primerima, te se prepućuje da ih proučite

1. <http://www.netobjectives.com/resources/books/design-patterns-explained>
2. <https://www.odesign.com/>