УНИВЕРЗИТЕТ У БЕОГРАДУ

ФАКУЛТЕТ ОРГАНИЗАЦИОНИХ НАУКА

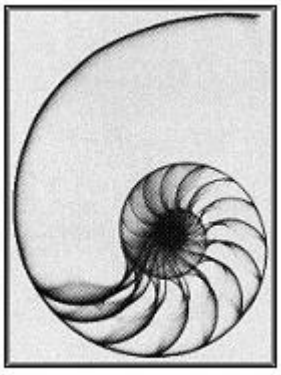
(Катедра за софтверско инжењерство)

ПРОЈЕКТОВАЊЕ СОФТВЕРА

(СКРИПТА – радни материјал)

вер. 2.0.

Аутор: др Синиша Влајић



Београд - 2025.

Аутор
проф. др Синиша Влајић

Наслов
ПРОЈЕКТОВАЊЕ СОФТВЕРА (СКРИПТА – РАДНИ МАТЕРИЈАЛ)

Издавач
проф. Др Синиша Влајић, Београд

Е-пошта издавача:
sinisa.vlajic@fon.bg.ac.rs

Напомена: *Електронска верзија скрипте из предмета Пројектовање софтвера је бесплатна. Уколико сте у могућности донирајте Универзитетску дечију клинику, Тиршова 10, Београд. Веб адреса клинике, на којој се налазе информације везано за донацију су:*
https://tirsova.rs/донације-на-рачун/

Садржај

ПРОЈЕКТОВАЊЕ СОФТВЕРА

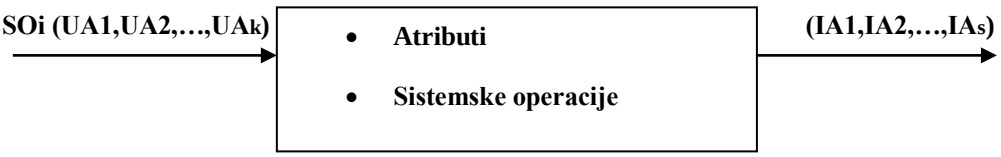
Посвећено мом унуку Лазару ♥

1. ОСНОВЕ РАЗВОЈА СОФТВЕРА

1.1 Софтверски систем[1]

Софтверски систем[2] се састоји од **атрибута** и **системских операција**[3]. Атрибути описују **структуру** система, док системске операције описују **понашање** система. Атрибути представљају концепте реалног система који описују статичке карактеристике система (нпр. *Рачун, Partner, ..*). Системске операције представљају **основне** (атомске) **функције** система, које се могу користити из окружења система (*нпр. UnesiRačun, PromeniRačun, IzracunajRacun, ...*).
Допустиви улаз у софтверски систем је дефинисан потписом (сигнатуром), који садржи назив системске операције која се позива и скупом улазних аргумената (*нпр. IzracunajRacun(Racun),...*). **Излаз** из софтверског система је представљен преко скупа излазних аргумената (*нпр. Racun, signal, ...*). Излаз се добија као резултат извршења неке од системских операција над атрибутима система.

СОФТВЕРСКИ СИСТЕМ



Atributi = {AT1, AT2,…,ATn}

Sistemske operacije = {SO1,SO2,..,SOm} - Основне функције система

SOi ∈ Sistemske operacije , i = (1,..,m)

UA1,UA2,…,UAk – Улазни аргументи

IA1,IA2,…,IAs – Излазни аргументи

---

[1] Радећи на овом материјалу, често сам долазио у наизглед нерешиве ситуације код повезивање спецификације проблема са њеном реализацијом (имплементацијом). Оно што је правило највећи проблем јесте схватање шта је софтверски систем и које су његове границе. Питање које ми је задавало највише главобоље односило се на кориснички интерфејс: *Да ли је кориснички интерфејс део софтверског система или је он изван система.*
Одговор на наведено питање, и на многа друга око схватања шаренила многих појмова и термина који се користе у софтверском инжењерству, дао ми је проф. Лазаревић Бранислав. Он ми је рекао да је кориснички интерфејс реализација улаза и/или излаза софтверског система. У првом тренутку то ми је изгледало нелогично, али након неког времена, јасна и прецизна аргументација проф. Лазаревића ме је уверила у исправност његових ставова. Наведени став смо представили преко следеће дефиниције:

Деф. Лаз1: Кориснички интерфејс представља реализацију улаза и/или излаза софтверског система.

Наведена дефиниција, колико год једноставно изгледала, је отворила пут да софтверски систем схватимо као систем у правом смислу дефиниције система (Петровић, 1998). Надам се да ће системски приступ у схватању развоја софтвера, коначно од софтверског инжењерства направити нешто што ће бити вредно научног а не само технолошког поштовања.

[2] У даљем тексту систем.

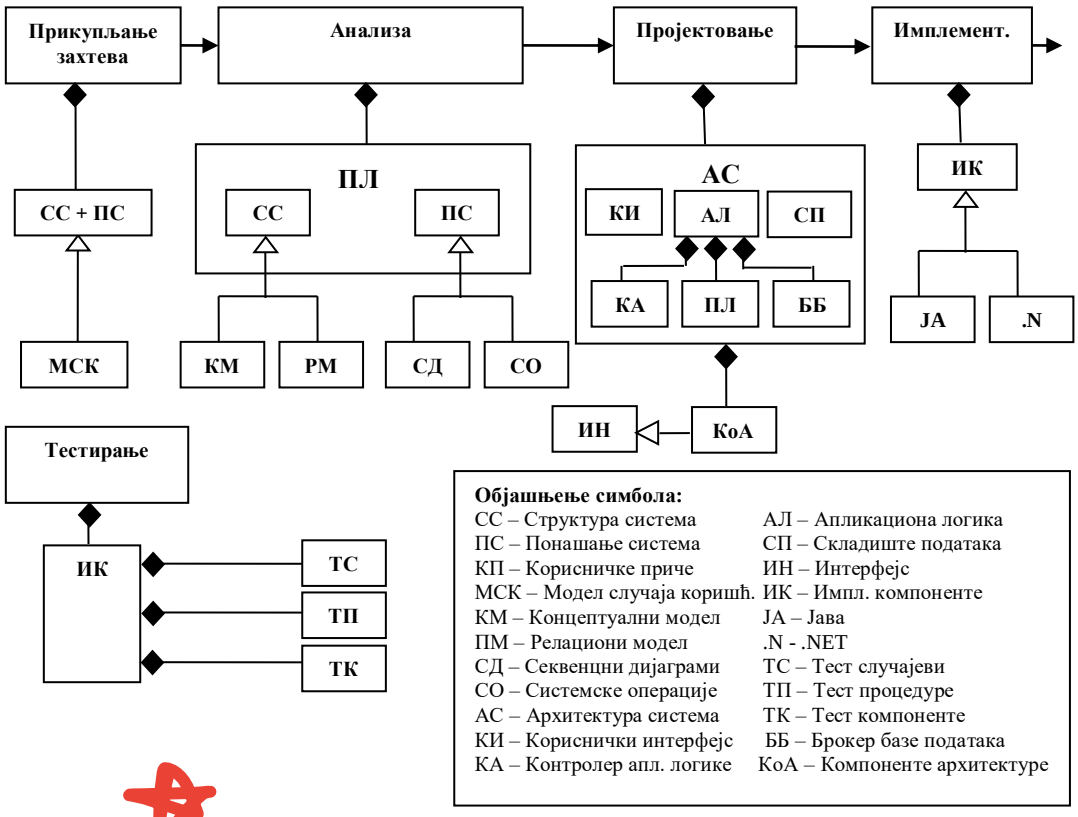
[3] Јединствени процес развоја софтвера (Jacobson et al., 1999) користи термин системска операција када жели да нагласи разлику између операција софтверског система и операција које се налазе изван софтверског система.

1.2 Развој (животни циклус) софтверског система

Развој (животни циклус) софтверског система[4] се састоји из следећих фаза (Слика RASS):

- Прикупљања захтева од корисника
- Анализе
- Пројектовања
- Имплементације
- Тестирања

У фази прикупљања захтева се дефинишу својства и услови које софтверски систем или шире гледајући пројекат треба да задовољи (Larman, 1998). Захтеви се могу поделити као функционални и нефункционални. Функционални захтеви дефинишу захтеване функције система, док нефункционални захтеви дефинишу све остале захтеве. Нефункционални захтеви као што су *употребљивост, поузданост, перформансе* и *подржаност система* представљају атрибуте квалитета (*quality attributes*) софтверског система. Функционални захтеви се описују преко модела случаја коришћења. Наведени модел садржи елементе структуре и понашања софтверског система. Елементи структуре софтверског система се описују помоћу *именица* и *придева* док се елементи понашања описују помоћу *глагола*. У фази анализе се описује логичка структура и понашање софтверског система односно *пословна логика* софтверског система.



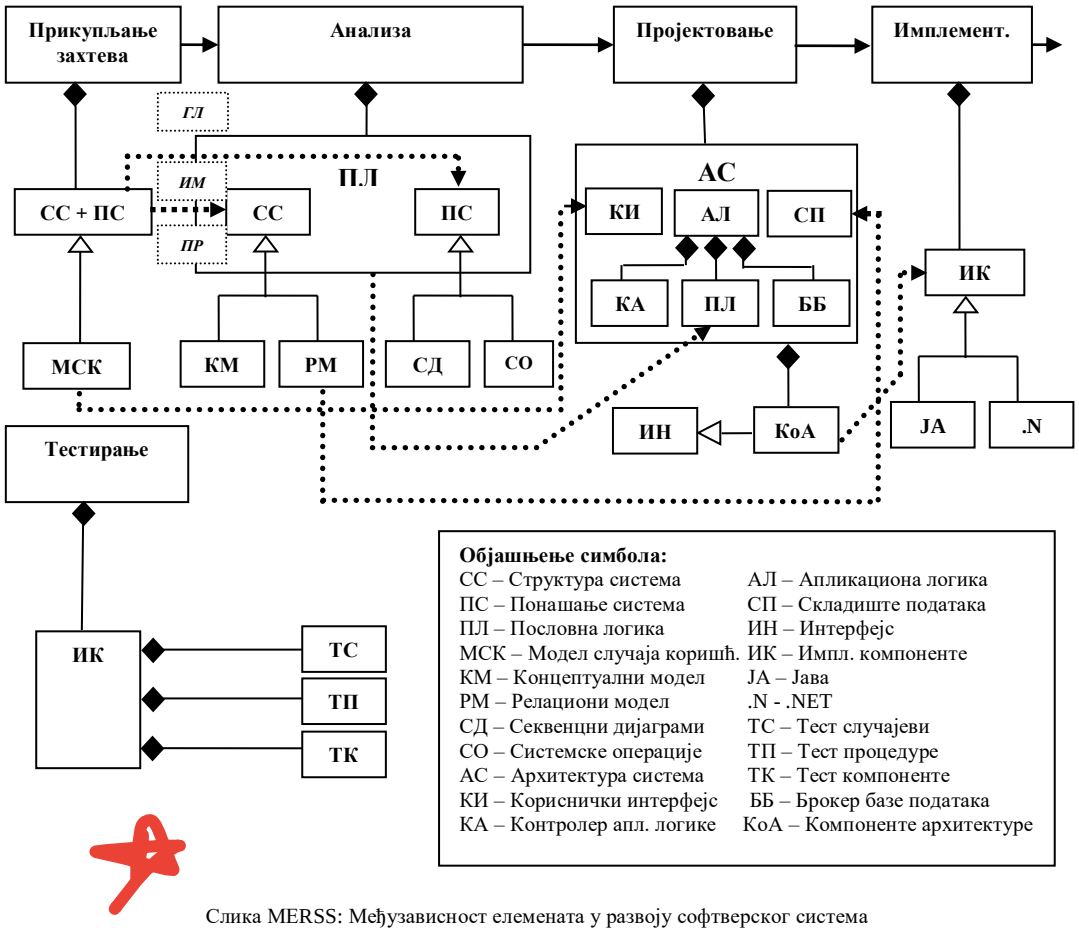
Слика RASS: Развој софтверског система

Структура софтверског система се описује преко *концептуалног (доменског)* и *релационог модела* док се понашање софтверског система описује помоћу *секвенцних дијаграма* и *системских операција*. У фази пројектовања се описује *архитектура* софтверског система која је *тронивојска* и састоји се од: а) *корисничког интерфејса* б) *апликационе логике* и ц) *складишта података*. Апликациона логика се састоји од: б1) *контролера апликационе логике* б2) *пословне логике* и б3) *брокера базе података*. Сваки од наведених нивоа тронивојске архитектуре је реализован преко скупа *класа* које представљају *компоненте* архитектуре софтверског система. Компоненте архитектуре су дефинисане преко *интерфејса*. У фази

[4] Развој софтверског система је објашњен помоћу упрошћене и делимично измењене Ларманове методе развоја софтвера (Larman, 1998) која се користи на ФОН-у, у извођењу наставе на предмету Пројектовања програма и Пројектовање софтвера, од 2003. године.

имплементације се праве *имплементационе компоненте* које се реализују у некој од постојећих технологија (*Јава, .NET,...*). У фази тестирања, свака од компоненти се тестира тако што се за сваку од њих праве *тест случајеви*, *тест процедуре* и *тест компоненте*.

Развој софтверског система код Ларманове методе има јасан логички след *(Слика* MERSS*)*. У фази прикупљања захтева елементи структуре и понашања софтверског система се налазе заједно у случајевима коришћења и они су представљени преко именица, глагола и придева. Структура софтверског система (концепти и атрибути концепата), из фазе анализе, се добија на основу именица и придева који су дефинисани у случајевима коришћења. Системске операције, из фазе анализе, се добијају на основу глагола који су дефинисани у случајевима коришћења. Секвенцни дијаграми, из фазе анализе, се добијају на основу сценарија случајева коришћења.



Слика MERSS: Међузависност елемената у развоју софтверског система

У фази пројектовања се прави архитектура софтверског система која је тронивојска (кориснички интерфејс, апликациона логика и складиште података). Кориснички интерфејс је дефинисан преко скупа екранских форми. Сценарија коришћења екранских форми је директно повезан са сценаријима случајева коришћења[5]. Пословна логика, која се добија у фази анализе, се преноси у фазу пројектовања и постаје саставни део апликационе логике. Складиште података се пројектује на основу релационог модела. Имплементационе компоненте, из фазе имплементације, треба да реализују компоненте које су добијене у фази пројектовања. Свака од имплементационих компоненти се тестира у фази тестирања.

[5] Сценарија коришћења екранских форми у суштини представља упуство за крајњег корисника како се користи програм (корисничко упутство).

1.3 СИЛАБ метода развоја софтвера

СИЛАБ[6] метода развоја софтвера представља **агилни приступ** у развоју софтверских система који се заснива на **итеративно-инкременталном процесу изградње и унапређивања (еволуцији) артефаката** који настају током животног циклуса развоја софтвера.

СИЛАБ метода **аутоматизује**[7] добијање појединих артефаката на основу **типских мета-концептуалних модела**, што је у складу са приступима (стратегијама) развоја софтвера који су вођени моделом (**model-driven approaches**) Приступи вођени моделом омогућавају аутоматско генерисање софтверских артефаката из апстрактних (мета) модела, чиме се побољшава конзистентност (у развоју софтверских система) и смањује ручни рад.“ (Schmidt, 2006).

У складу са **основним принципима агилног развоја**, нагласак се ставља на континуирану сарадњу са корисником, флексибилност у односу на промене и честу испоруку функционалних целина софтверског система (Beck et al., 2001; Highsmith, 2002).

Итеративни аспект у СИЛАБ методи подразумева вишеструки пролазак кроз фазе развоја у кратким циклусима, где се у свакој итерацији врши поновна анализа и унапређење претходно изграђених артефаката. **Инкрементални аспект** подразумева да се систем гради постепено, додавањем функционалних делова који су у свакој фази употребљиви и тестирани у реалним условима (Larman, 2004; Boehm & Turner, 2004).

СИЛАБ метода организује процес развоја кроз стандардне фазе: **прикупљање захтева, анализа, пројектовање, имплементација и тестирање**, али их повезује у цикличну структуру која омогућава континуирану евалуацију и побољшање решења. На тај начин, метода задржава структуру традиционалног инжењерског приступа развоју софтвера, док истовремено интегрише принципе агилности и итеративно-инкременталног развоја (Sommerville, 2016; Pressman & Maxim, 2014).

СИЛАБ метода биће објашњена кроз фазе развоја софтвера (Слика СИЛАБМет):

1. ПРИКУПЉАЊЕ ЗАХТЕВА ОД КОРИСНИКА

У фази **прикупљања корисничких захтева** (**User Requirements Gathering**) основни циљ је да се на јасан и прецизан начин идентификују потребе корисника и функционалности које софтверски систем треба да обезбеди.

У оквиру ове фазе реализују се следећи кораци:
а) **Одређивање мета-концептуалног модела** – бира се мета-концептуални модел који одговара конкретном концептуалном моделу који описује домен софтверског система који ће бити развијен.
б) **Дефинисање конкретног концептуалног модела** - дефинише се конкретни концептуални модела који се састоји из концепата (нпр. Продавац, Купац, Рачун, ...) и њихових веза. Везе су дефинисане преко типских веза (композиција, агрегација, генерализација-специјализација, асоцијативни концепт) и пресликавања између концепата.
ц) **Додела функционалности концептима** - сваком концепту се придружује скуп функција које се извршавају над тим концептом. Стандардне функције су: *Креирај, Убаци, Претражи, Промени и Обриши.*
д) **Дефинисање граматичких категорија концепата** - за сваки од концепата се дефинишу граматичке категорије, које су потребне за аутоматско генерисање случајева коришћења.
е) **Генерисање граматичких категорија случајева коришћења** – за сваку функцију свих концепата се аутоматски генерише случај коришћења са придруженим граматичким категоријама, које се добијају на основу граматичких категорија концепата.
е) **Одређивање детаљних случајева коришћења** - одређују се случајеви коришћења који ће бити детаљно описани.

---

[6] СИЛАБ је скраћеница од Лабораторије за софтверско инжењерство (као дела Катедре за софтверско инжењерство) на Факултету организационих наука, Универзитет у Београду, Република Србија. СИЛАБ метода развоја софтвера је развијена у оквиру Лабораторије и Катедре за софтверско инжењерство на Факултету организационих наука. Аутор СИЛАБ методе развоја софтвера је проф. др Синиша Влајић. СИЛАБ метода се једним делом наслања на упрошћену и делимично измењену Ларманову методу развоја софтвера.

[7]. Аутоматизација је извршена помоћу СИЛАБ генератора. Тренутана верзија СИЛАБ генератора је 2.9.

ф) **Опис специфичности софтверског система** - описују се специфичности везане за софтверски систем који треба да се развије (назив система, аутор система, опис веза између концепата,...).
г) **Генерисање артефаката везаних за фазу прикупљања захтева** – генеришу се артефакти софтверског система који су везани за фазу прикупљања захтева:

- **вербални опис корисничких захтева** на основу постојећег шаблона вербалног описа коме се додаје опис специфичности софтверског система.
- табела која **повезује концепте и њихове функционалности**
- **изглед главног менија** преко кога се позивају основне функционалности система
- табеле у којој се налазе **случајеви коришћења**, **предуслови** за извршење случајева коришћења и **критеријуми** по којима се врши претраживање концепта који је описан случајем коришћења.
- табела **случајева коришћења** за које ће бити дат **детаљан опис**.
- **детаљан опис изабраних случајева коришћења** који се добија на основу шаблона стандардних случајева коришћења (*Kreiraj, Ubaci, Promeni, Obrisi, Pretrazi, Prijavi, IzborFormeSaMeniј*а) и граматичких категорија концепата.

2. АНАЛИЗА

У фази анализе се описује логичка структура и понашање софтверског система односно *пословна логика* софтверског система. Структура софтверског система се описује преко *концептуалног (доменског)* и *релационог модела* док се понашање софтверског система описује помоћу *секвенцних дијаграма* и *системских операција*.

У оквиру ове фазе реализују се следећи кораци везани за логичко **понашање** софтверског система:
а) **Генерисање свих системских операција свих случајева коришћења** – генерише се табела која повезује случајеве коришћења са њеним системским операцијама.
б) **Генерисање јединствених системских операција –** генерише се табела која садржи све јединствене системске операције.
ц) **Прављење секвенцних дијаграм за случајеве коришћења** - праве се секвенцни дијаграми за сваки случај коришћења и одређују се системске операције које су уочене на секвенцним дијаграмима.
д) **Дефинисање уговора за системске операције –** за системске операцију се праве уговори који имају стандардне одељке (*потпис операције, веза са случајевима коришћења, предуслови и посуслови*)**.**

У оквиру ове фазе реализују се следећи кораци везани за логичку **структуру** софтверског система:
а) **Генерисање концептуалног модела** – генерише се концептуални модел који има само називе концепата и везе између њих. Атрибуте се додају до концепата.
б) **Генерисање релационог модела –** генерише се релациони модел који има атрибуте који су примарни и/или спољни кључеви. Остали атрибути се додају до релација.
ц) **Генерисање табела структурних и вредносних ограничења релација** – За сваку релацију се генерише табела структурних и вредносних ограничења. Генерисани су атрибути који су примарни кључеви и структурна правила интегритета релације. Остали атрибути се додају у табелу. У табели се дефинишу проста и сложена вредносна ограничења релације.

3. ФАЗА ПРОЈЕКТОВАЊА

У фази пројектовања се описује *архитектура* софтверског система која је *тронивојска* и састоји се од: а) *корисничког интерфејса* б) *апликационе логике* и ц) *складишта података*. Кориснички интерфејс се састоји из: а1) *екранских форми* и а2) *контролера корисничког интерфејса*. Апликациона логика се састоји од: б1) *контролера апликационе логике* б2) *пословне логике* и б3) *брокера базе података*. Сваки од наведених нивоа тронивојске архитектуре је реализован преко скупа *класа* које представљају *компоненте* архитектуре софтверског система. Компоненте архитектуре су дефинисане преко *интерфејса*.

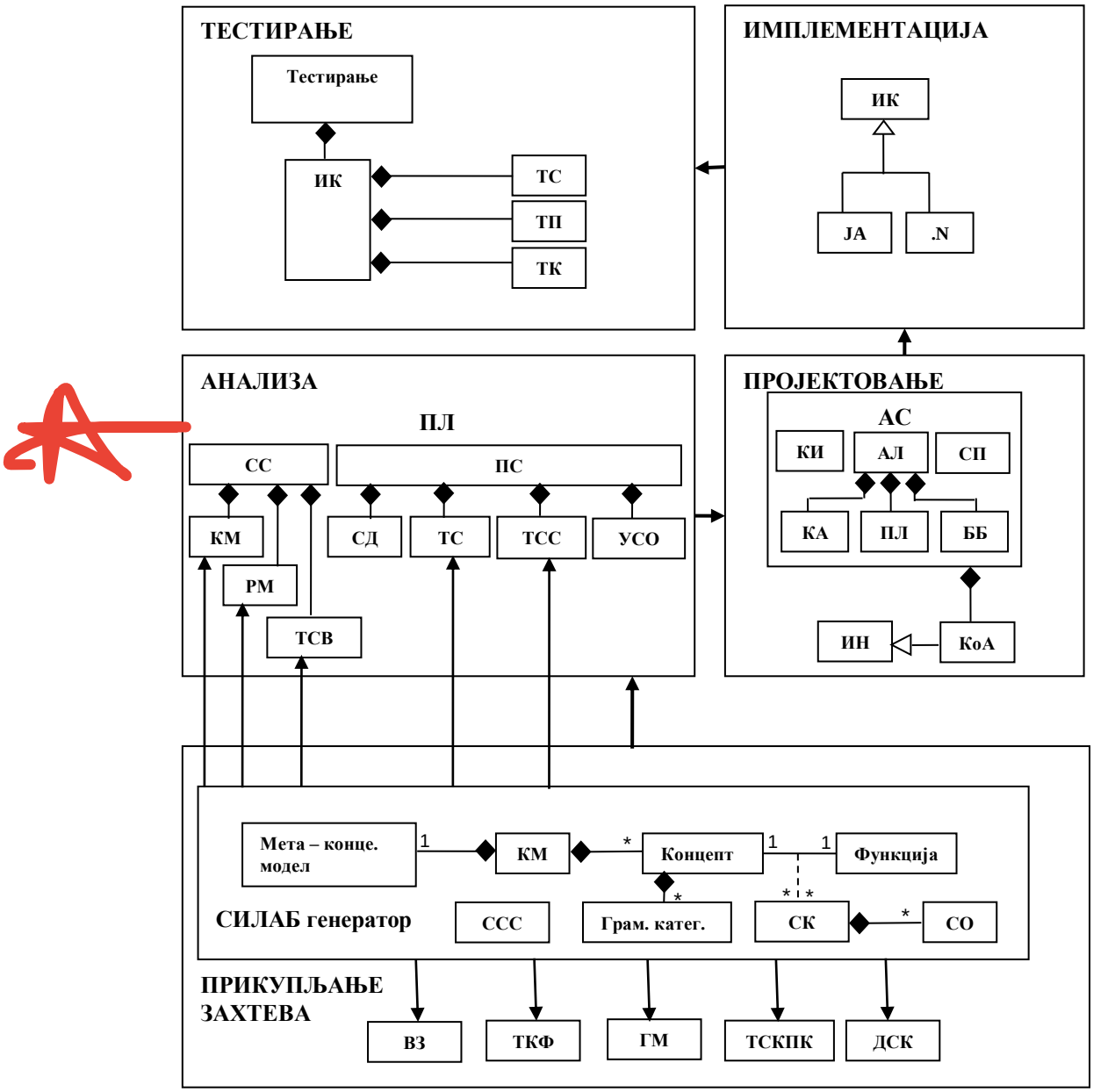
У оквиру ове фазе реализују се следећи кораци:

а) **пројектовање корисничког интерфејса** – пројектује се кориснички интерфејс, односно пројектују се екранске форме[8] и контролер корисничког интерфејса.
б) **пројектовање апликационе логике –** пројектује се апликациона логика, односно контролер апликационе логике, пословна логика и брокер базе података**.** Пројектовање пословне логике обухвата пројектовање физичке структуре и понашања софтверског система.
ц) **пројектовање складишта података –** пројектује се складиште података на основу релационог модела и табела вредносних и структурних ограничења релација из фазе анализе.

4. ФАЗА ИМПЛЕМЕНТАЦИЈЕ
У фази имплементације се праве *имплементационе компоненте* које се реализују у некој од постојећих технологија (*Јава, .NET,...*).

5. ФАЗА ТЕСТИРАЊА
У фази тестирања, свака од компоненти се тестира тако што се за сваку од њих праве *тест случајеви*, *тест процедуре* и *тест компоненте*.

[8]. Сценарија коришћења екранских форми су директно повезана са сценаријима случајева коришћења. Сценарија коришћења екранских форми у суштини представљају упуство за крајњег корисника како се користи програм (корисничко упутство).



Објашњење симбола:

СС – Структура система	**АЛ** – Апликациона логика
ПС – Понашање система	**СП** – Складиште података
ИК – Импл. компоненте	**ИН** – Интерфејс
КМ – Концептуални модел	**JA** – Јава
РМ – Релациони модел	**.N** - .NET
СД – Секвенцни дијаграми	**ТС** – Тест случајеви
СО – Системске операције	**ТП** – Тест процедуре
АС – Архитектура система	**ТК** – Тест компоненте
КИ – Кориснички интерфејс	**ББ** – Брокер базе података
КА – Контролер апл. логике	**КоА** – Компоненте арх.
СК – Случајеви коришћења	**ТКФ** – Табела конц.- функц.
ВЗ – Вербални опис. кор. зах.	**ГМ** – Главни мени
ДСК – Детаљни случ. кор.	**ТС** – Табела сист. опер.

ТСКПК – Табела случ. кор. – предуслови - критеријуми
ТСС – Табела случ. коришћења – сист. операција
ТСВ – Табела структурних и вредносних ограничења
ПЛ – Пословна логика **УСО** – Уговор за С.О.
ССС – Специф. софт. система

Слика SILABMet: СИЛАБ метода развоја софтвера

2. РАЗВОЈ СОФТВЕРСКОГ СИСТЕМА

2.1 Прикупљања захтева од корисника

2.1.1 Захтеви (Requirements)

Захтеви представљају својства и услове које систем или шире гледајући пројекат мора да задовољи (Larman, 1998) . Постоје различити типови захтева које систем мора да задовољи и они су категоризовани према **FURPS+** (**Functional** - *Функционалност*, **Usability** - *Употребљивост*, **Reliability** - *Поузданост*, **Performance** - *Перформансе*, **Supportability** - *Подржљивост*) моделу:

- *Функционалност* представља способност (**capabilities**) система да обезбеди захтеване функције (понашање система). Заштита (**security**) система представља једну од основних функција коју систем треба да обезбеди.
- *Употребљивост* представља способност система да се може једноставно користити. То се постиже помоћу разних упутстава и документције који описују начин његовог коришћења.
- *Поузданост* представља способност система да може успешно обрадити проблем (**failure**) који се дешава у току извршења система. У том смислу систем мора да обезбеди начин опоравка (**recoverability**) података у случају насилног прекида рада система. Такође систем треба да омогући предвиђање (**predictability**) могућих понашања система.
- *Перформансе* система се односе на време одзива (**response time**) захтеваних функција, пропусну моћ (**throughput**) мреже кроз коју пролазе подаци, тачност (**accuracy**) извршења функција, могућност коришћења односно расположивост (**availability**) функција система и начин коришћење расположивих ресурса(**resource usage**) система.
- *Подржљивост* система се односи на лакоћу његовог прилагођавања (**adaptability**) и одржавања (**maintainability**), интернационализацију (**internationalization**) у смислу његове прилагодљивости различитим знаковним системима који се користе у свету и начину конфигурисања (**configurability**) система.

Захтеви се често категоризују као функционални и не функционални захтеви. Функционални захтеви дефинишу захтеване функције система, док не функционални захтеви дефинишу све остале захтеве. У том смислу не функционални захтеви (употребљивост, поузданост, перформансе и подржљивост система) представљају атрибуте квалитета (**quality attributes**) софтверског система.

У ФУРПС+ моделу знак ‘+’ указује на помоћне захтеве који се односе на:

- *Имплементацију* (**Implementation**) система – до којих граница се могу користити расположиви ресурси (**resource limitations**). Који се програмски језици (**programming languages**) и алати (**tools**) могу користити. Поред тога имплемантациони захтеви се односе и на хардвер (**hardware**) који ће се користити.
- *Интерфејс* (**Interface**) система – ограничења која постоје у комуникацији система са његовим окружењем (екстерним системима).
- *Операције* (**Operations**) система – управљање системом и његовим операцијама.
- *Паковање* (**Packaging**) система – начин физичког организовања система у пакете, који представљају управљиве јединице система.
- *Легалност* (**Legal**) – могућност употребе система у смислу његове легалности (лиценце и права коришћења система).

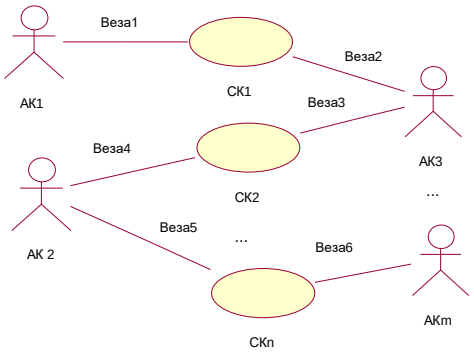
У даљем тексту ми ћемо главни нагласак ставити на разматрање функционалних захтева, од њиховог прикупљања до имплементације. Сматрамо да истовремено објашњење функционалних и не-функционалних захтева, које прати студијски пример, може доста да усложи схватање суштине развоја једног софтверског система из угла његове основне функционалности. У студијском примеру ми ћемо увести неки од не-функционалних захтева који су директно повезани са функционалношћу система (*поузданост и подржљивост система*) и неке од помоћних захтева (*имплементација, операције и паковање система*) Остали не-функционални и помоћни захтеви неће бити разматрани, будући да су они у највећој мери ортогонално

постављени у односу на функције система. То значи да они не утичу на схватање и објашњење развоја функција система.

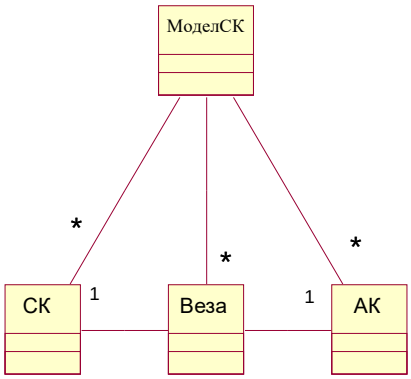
2.1.2 Опис захтева помоћу модела случаја коришћења

Захтеви се код Лармана описују помоћу UML *Модела Случаја Коришћења* (**Use-Case Model**).

Деф. ЗА1: Модел СК се састоји од скупа случаја коришћења (СК), актора (АК) и веза између случаја коришћења и актора.

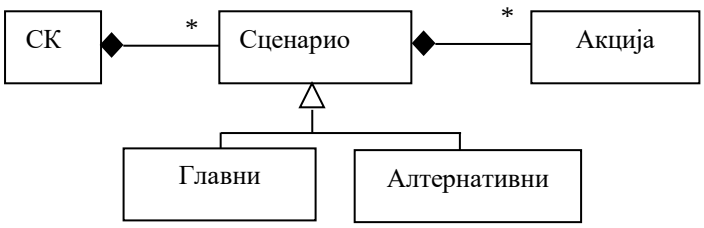


Модел СК се може представити са:



Модел СК је везан за више а) СК, б) веза између СК и АК и ц) АК. Један СК може да има више веза са АК. Један АК може да има више веза са СК. Једна веза се односи на један пар СК-АК.

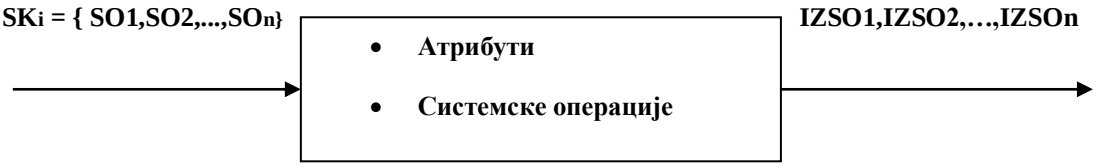
Деф. ЗА2: **Случај коришћења** описује скуп сценарија (use-case појављивања), односно скуп жељених коришћења система од стране актора.



Сценарио описује једно жељено коришћење система од стране актора. Случај коришћења има један главни и више алтернативних сценарија. Сценарио је описан преко: а) секценце акција и б) интеракција између актора и система. СК се састоји из главног и алтернативних сценарија.

У току интеракције између актора и софтверског система, актор позива системске операција софтверског система. То значи да се у току неког (једног) сценариа *Ski* случаја коришћења позива једна или више системских операција (*SO1, SO2,...,SOn*).

СОФТВЕРСКИ СИСТЕМ



Као резултат извршења системске операције над атрибутима софтверског система се добијају излазни резултати IZSO1, IZSO2,...,IZSOn.

Системске операције = {SO1,SO2,..,SOn} - Основне функције система (атомске функције)

Случајеви коришћења = {SK1,SK2,..,SKp} - Жељене функције система (молекулске функције)

SKi ∈ Slučajevi korišćenja, i =(1..p)

СК, односно сценарија СК дефинишу жељене функције система. Жељене функције система, када се извршавају, позивају по одређеном редоследу основне функције система.

Једну акцију сценариа изводи или актор или систем. У том смислу:
Актор изводи једну од три врсте акција:
- а) **А**ктор **П**рипрема **У**лаз (Улазне Аргументе) за **С**истемску **О**перацију (**АПУСО**).
- б) **А**ктор **П**озива систем да изврши **С**истемску **О**перацију (**АПСО**).
- ц) **А**ктор извршава **НеС**истемску **О**перацију (**АНСО**).

Систем изводи две акције у континуитету:
- а) Систем извршава **С**истемску **О**перацију(**СО**):
- б) Резултат системске операције (Излазни аргументи (**ИА**)) се прослеђује до актора.

Деф. ЗА3: Актор (учесник) представља спољног корисника система. Он поставља захтев систему да изврши једну или више системских операција, по унапред дефинисаном сценарију[9]. Систем одговара на постављени захтев актора, шаљући му вредност излазних аргумената као резултат извршења операција. Актор се обично дефинише као неко или нешто (нпр: људи, рачунарски системи или организациона јединица) што има понашање.

Правила код прикупљања захтева:

Правило ЗА1 (Независност сценарија СК): Сценарија СК не треба да буду у међусобној интеракцији. Она се требају дефинисати као атомска, у смислу да се извршавају у потпуности самостално. На тај начин се олакшава њихов развој и одржавање (Jacobson et al., 1999)[10].

Правило ЗА2 (Glass' low) (Endres, A., & Rombach, 2003) **:** Недостаци код дефинисања захтева су основни разлог могућег неуспеха у развоју пројекта (програма).

[9] Захтев за извршење једне или више системских операција се не одиграва континуално него дискретно. То значи да корисник интерактивно позива једну по једну системску операцију, у дискретним временским интервалима. Из наведеног може да се закључи да сценарио описује интерактивно коришћење софтверског система.

[10] У разговору са проф. Братиславом Петровићем рекао сам да независност сценарија, повећава редудансу у опису понашања система али истовремено знатно олакшава одржавање система. Рекао сам да су редуданса и одржавање обрнуто сразмерни. Проф. Петровић је рекао да је њихов производ вероватно неки коефицијент. ***Било би веома интересантно када би неко успео да формално објасни тај однос и да пронађе наведени коефицијент.***

Правило ЗА3 (Boehm's first low) (Endres, A., & Rombach, 2003)**:** Уколико се не уоче грешке у току дефинисања захтева, исте се веома тешко могу уклонити у каснијим фазама развоја програма.

Правило ЗА4 (Boehm's second low) (Endres, A., & Rombach, 2003)**:** Прављење прототипова значајно смањује могуће грешке код дефинисања захтева и његовог развоја, нарочито код дефинисања корисничког интерфејса.

Начин представљања модела СК

СК се у почетним фазама развоја софтвера представљају текстуално док се касније они представљају преко секвенцних дијаграма, дијаграма сарадње, дијаграма прелаза стања или дијаграма активности.
Тексуални опис СК има следећу структуру:

- Назив СК.
- Акторе СК.
- Учеснике СК.
- Предуслови који морају бити задовољени да би СК почео да се извршава.
- Основни сценарио извршења СК.
- Постуслови који морају бити задовољени да би се потврдило да је СК успешно извршен.
- Алтернативна сценарија извршења СК.
- Специјални захтеви.
- Технолошки захтеви.
- Отворена питања.

У нашем примеру имамо следеће СК-а:

1. *Креирање новог рачуна.*
2. *Претраживање рачуна.*
3. *Сторнирање рачуна.*
4. *Промена рачуна.*

Наведене СК користи Продавац (актор).
Модел СК се може представити преко следећег дијаграма СК:

СК1: Случај коришћења – Креирање новог рачуна

Назив СК
Креирање новог рачуна

Актори СК
Продавац

Учесници СК
Продавац и систем (програм)

Предуслов: Систем је укључен и продавац је улогован под својом шифром. Систем приказује форму за рад са рачуном.

Основни сценарио СК

1. Продавац **позива** систем да креира нови рачун. (АПСО)
2. Систем **креира** нови рачун. (СО)
3. Систем **приказује** продавцу нови рачун и поруку: "Систем је креирао нови рачун". (ИА)
4. Продавац **уноси** податке у нови рачун. (АПУСО)
5. Продавац **контролише** да ли је коректно унео податке у нови рачун. (АНСО)
6. Продавац **позива** систем да запамти податке о рачуну. (АПСО)
7. Систем **памти** податке о рачуну. (СО)
8. Систем **приказује** продавцу запамћени рачун и поруку: "Систем је запамтио рачун. (ИА)
9. Продавац **позива** систем да обради рачун. (АПСО)
10. Систем **обрађује** рачун.(СО)
11. Систем **приказује** продавцу обрађен рачун и поруку: "Систем је обрадио рачун".(ИА)

Алтернативна сценарија
3.1 Уколико систем не може да креира рачун он приказује продавцу поруку: "Систем не може да креира нови рачун". Прекида се извршење сценариа. (ИА)
8.1 Уколико систем не може да запамти податке о рачуну он приказује продавцу поруку "Систем не може да запамти рачун". Прекида се извршење сценариа. (ИА)
11.1 Уколико систем не може да обради рачун он приказује продавцу поруку: "Систем не може да обради рачун". (ИА)

СК2: Случај коришћења – Претраживање рачуна

Назив СК
Претраживање рачуна

Актори СК
Корисник

Учесници СК
Корисник и систем (програм)

Предуслов: Систем је укључен и корисник је улогован под својом шифром. Систем приказује форму за рад са рачуном.

Основни сценарио СК

1. Корисник **уноси** вредност по којој претражује рачун. (АПУСО)
2. Корисник **позива** систем да нађе рачун по задатој вредности. (АПСО)
3. Систем **тражи** рачун по задатој вредности. (СО)
4. Систем приказује кориснику податке о рачуну и поруку: "Систем је нашао рачун по задатој вредности". (ИА)

Алтернативна сценарија
4.1 Уколико систем не може да нађе рачун он приказује кориснику поруку: "Систем не може да нађе рачун по задатој вредности". (ИА)

СК3: Случај коришћења – Сторнирање рачуна

Назив СК
Сторнирање рачуна

Актори СК
Корисник

Учесници СК
Корисник и систем (програм)

Предуслов: Систем је укључен и корисник је улогован под својом шифром. Систем приказује форму за рад са рачуном.

Основни сценарио СК

1. Корисник **уноси** вредност по којој претражује рачун. (АПУСО)
2. Корисник **позива** систем да нађе рачун по задатој вредности. (АПСО)
3. Систем **тражи** рачун по задатој вредности. (СО)
4. Систем приказује кориснику рачун и поруку: "Систем је нашао рачун по задатој вредности". (ИА)
5. Корисник **позива** систем да сторнира задати рачун. (АПСО)
6. Систем **сторнира** рачун. (СО)
7. Систем **приказује** кориснику сторниран рачун и поруку: "Систем је сторнирао рачун ". (ИА)

Алтернативна сценарија

4.1 Уколико систем не може да нађе рачун он приказује кориснику поруку: "Систем не може да нађе рачун по задатој вредности". Прекида се извршење сценариа. (ИА)
7.1 Уколико систем не може да сторнира рачун он приказује кориснику поруку: "Систем не може да сторнира рачун".

СК4: Случај коришћења – Промена рачуна

Назив СК
Промена рачуна

Актори СК
Продавац

Учесници СК
Продавац и систем (програм)

Предуслов: Систем је укључен и продавац је улогован под својом шифром. Систем приказује форму за рад са рачуном.

Основни сценарио СК

1. Продавац **уноси** вредност по којој претражује рачун. (АПУСО)
2. Продавац **позива** систем да нађе рачун по задатој вредности. (АПСО)
3. Систем **тражи** рачун по задатој вредности. (СО)
4. Систем приказује продавцу рачун и поруку: "Систем је нашао рачун по задатој вредности". (ИА)
5. Продавац **уноси (мења)** податке о рачуну. (АПУСО)
6. Продавац **контролише** да ли је коректно унео податке о рачуну. (АНСО)
7. Продавац **позива** систем да запамти податке о рачуну. (АПСО)
8. Систем **памти** податке о рачуну. (СО)
9. Систем **приказује** продавцу поруку: "Систем је запамтио рачун." (ИА)
10. Продавац **позива** систем да обради рачун. (АПСО)
11. Систем **обрађује** рачун.(СО)
12. Систем **приказује** продавцу обрађен рачун и поруку: "Систем је обрадио рачун.(ИА)

Алтернативна сценарија

4.1 Уколико систем не може да нађе рачун он приказује продавцу поруку: "Систем не може да нађе рачун по задатој вредности". Прекида се извршење сценариа. (ИА)
9.1 Уколико систем не може да запамти податке о рачуну он приказује продавцу поруку "Систем не може да запамти рачун". Прекида се извршење сценариа. (ИА)
12.1 Уколико систем не може да обради рачун он приказује продавцу поруку: "Систем не може да обради рачун". (ИА)

2.2 Анализа

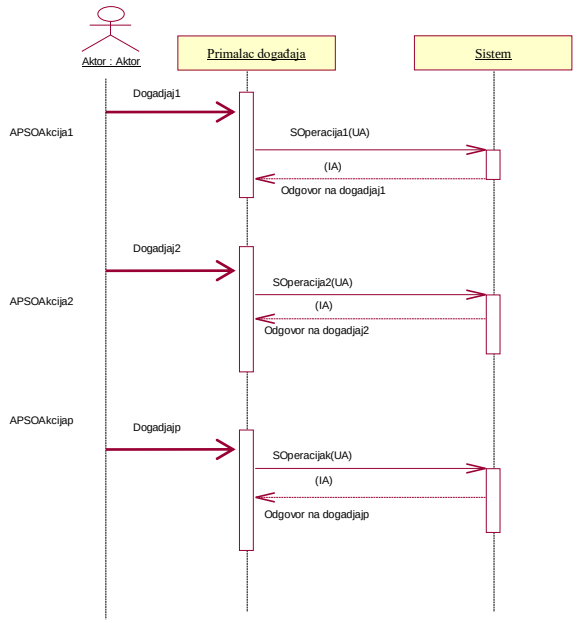
Фаза анализе описује логичку структуру и понашање софтв. система (пословну логику софт. система). Понашање софтверског система је описано помоћу системских дијаграма секвенци и преко системских операција. Структура софтерског система се описује помоћу концептуалног и релационог модела.

2.2.1 Понашање софтверског система - Системски дијаграми секвенци

Понашање система се може описати преко УМЛ-ових **секвенцних дијаграма** (Larman, 1998)**,** односно преко **дијаграма сарадње** (Jacobson et al., 1999)**.**

Деф. АН1: Системски дијаграм секвенци приказује, за издвојени сценарио СК, догађаје у одређеном редоследу, који успостављају интеракцију између актора и софтверског система.

Деф. АН2: Догађај који **направи** актор је побуда за позив системске операције. То значи да актор не позива системску операцију непосредно већ то чини преко посредника (примаоца догађаја). Позив системске операције указује на интеракцију између актора и система. За догађај који представља побуду за позив СО се често каже да је то **системски догађај.**



Догађаје праве актори (*нпр. клик на дугме, које се налази на екранској форми*), у оквиру APSO акција, над примаоцем догађаја (*нпр. дугме*). Прималац догађаја прихвата догађај и позива системску операцију (*нпр. дугме прима клик (догађај) и покреће методу која позива системску операцију*) која се налази на страни система. Након извршења системске операције систем враћа неки резултат као одговор на догађај (*то може да буде сигнал о успешности извршења операције и/или неки податак*).

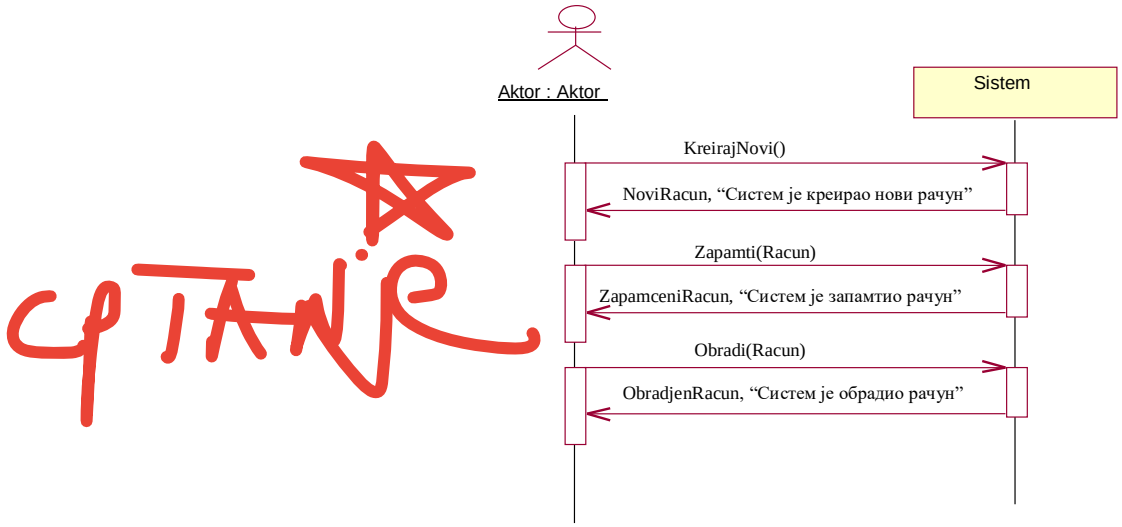
Као резултат анализе сценарија СК добијају се захтеви за извршење системских операција. За сваку системску операцију се праве уговори(контракти).

За сваки СК, прецизније речено за сваки сценарију СК, праве се системски дијаграми секвенци и то само за АПСО и ИА акције сценарија.

ДС1: ***Дијаграми секвенци случаја коришћења – Креирање новог рачуна***

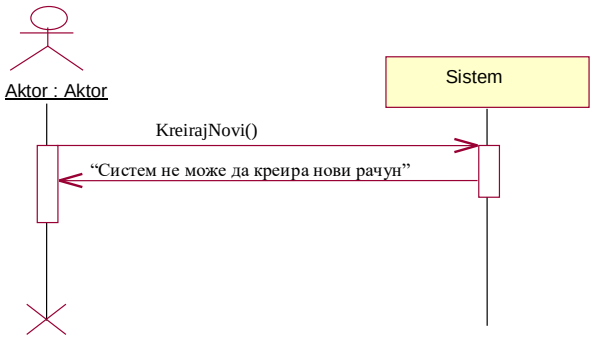
1. Продавац **позива** систем да креира нови рачун. (АПСО)
2. Систем **приказује** продавцу нови рачун и поруку: "Систем је креирао нови рачун". (ИА)
3. Продавац **позива** систем да запамти податке о рачуну. (АПСО)
4. Систем **приказује** продавцу запамћени рачун и поруку: "Систем је запамтио рачун. (ИА)
5. Продавац **позива** систем да обради рачун. (АПСО)
6. Систем **приказује** продавцу обрађен рачун и поруку: "Систем је обрадио рачун".(ИА)

Из наведеног може да се закључи да се на системском дијаграму секвенци не виде АПУСО, СО и АНСО акције.

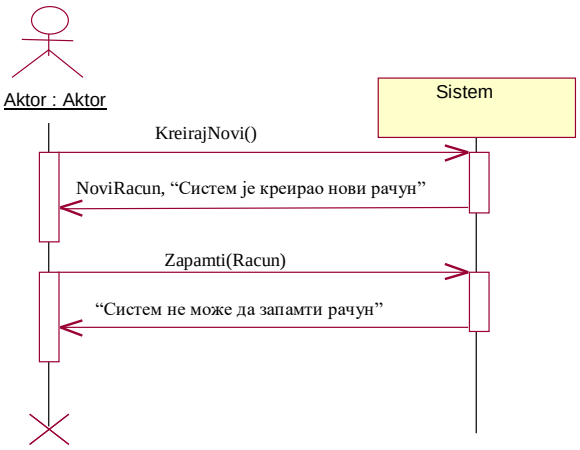


Алтернативна сценарија

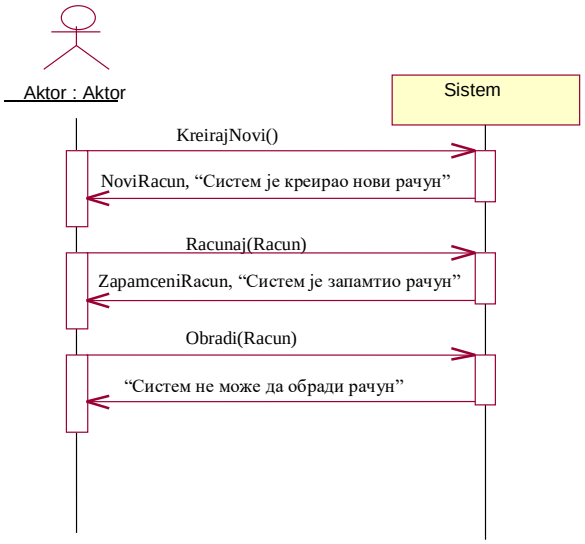
2.1 Уколико систем не може да креира рачун он приказује продавцу поруку: "Систем не може да креира нови рачун". Прекида се извршење сценариа. (ИА)



4.1 Уколико систем не може да запамти податке о рачуну он приказује продавцу поруку “Систем не може да запамти рачун”. Прекида се извршење сценариа. (ИА)



6.1 Уколико систем не може да обради рачун он приказује продавцу поруку: “Систем не може да обради рачун”. (ИА)

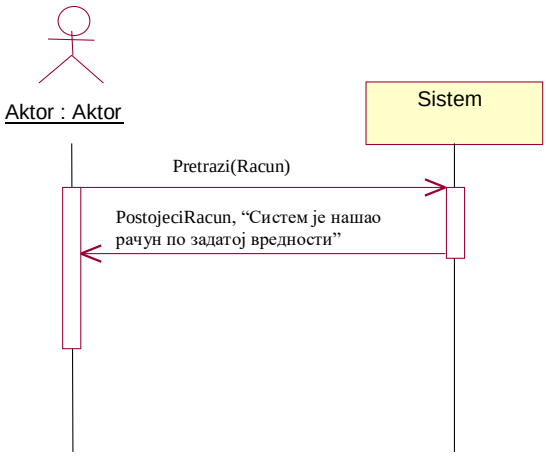


Са наведених секвенцних дијаграма уочавају се 3 системске операције које треба пројектовати:

1. *signal* **KreirajNovi** (*Racun*);
2. *signal* **Zapamti**(*Racun*);
3. *signal* **Obradi**(*Racun*);

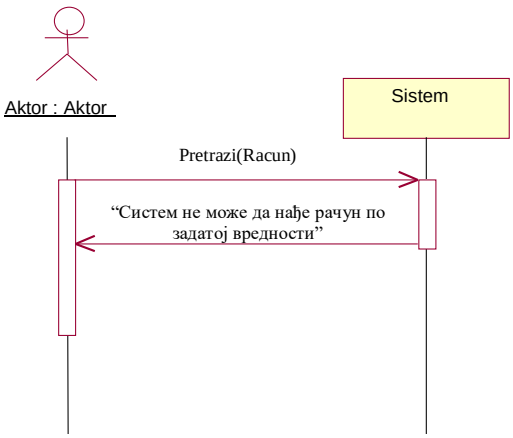
ДС2: *Дијаграми секвенци случаја коришћења – Претраживање рачуна*

1. Корисник **позива** систем да нађе рачун по задатој вредности. (АПСО)
2. Систем приказује кориснику податке о рачуну и поруку: “Систем је нашао рачун по задатој вредности”. (ИА)



Алтернативна сценарија

2.1 Уколико систем не може да нађе рачун он приказује кориснику поруку: “Систем не може да нађе рачун по задатој вредности”. (ИА)

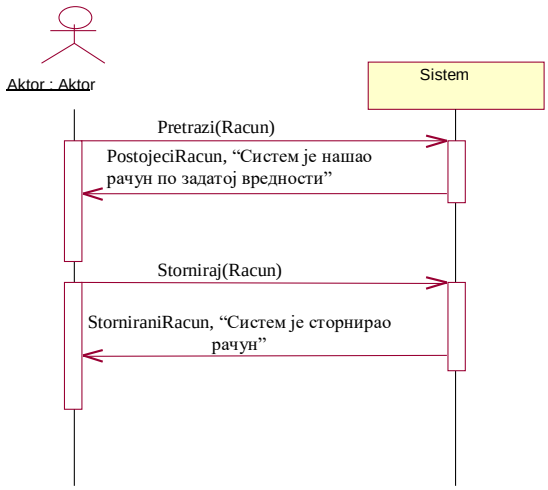


Са наведених секвенцних дијаграма уочава се још једна системска операција коју треба пројектовати:
1. signal **Pretrazi** (*Racun*);

ДС3: *Дијаграми секвенци случаја коришћења – Сторнирање рачуна*

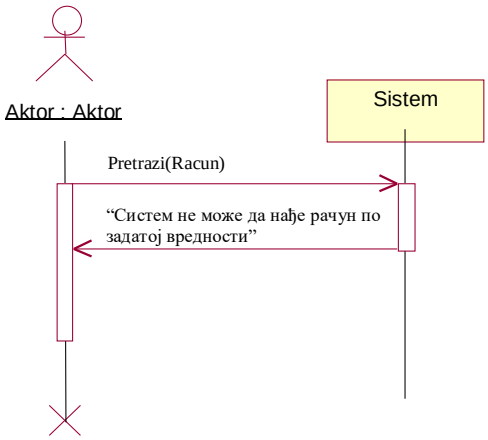
Основни сценарио СК

1. Корисник **позива** систем да нађе рачун по задатој вредности. (АПСО)
2. Систем приказује кориснику рачун и поруку: "Систем је нашао рачун по задатој вредности". (ИА)
3. Корисник **позива** систем да сторнира задати рачун. (АПСО)
4. Систем **приказује** кориснику сторниран рачун и поруку: "Систем је сторнирао рачун ". (ИА)

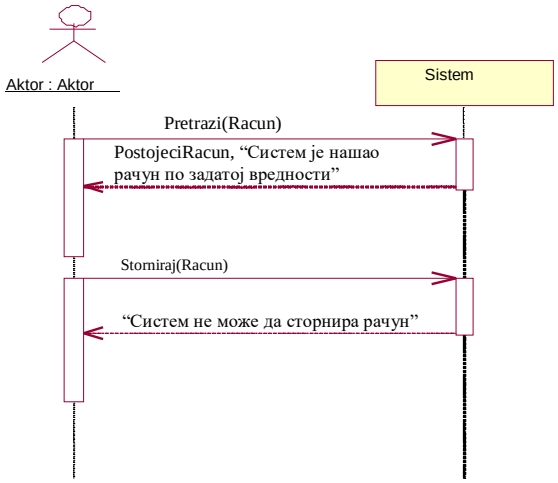


Алтернативна сценарија

2.1 Уколико систем не може да нађе рачун он приказује кориснику поруку: "Систем не може да нађе рачун по задатој вредности". Прекида се извршење сценариа. (ИА)



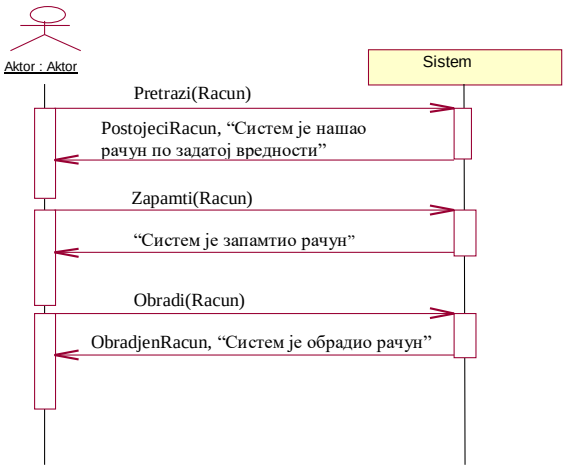
4.1 Уколико систем не може да сторнира рачун он приказује кориснику поруку: "Систем не може да сторнира рачун".



Са наведених секвенцних дијаграма уочавају се следеће системске операције које треба пројектовати:
1. signal **Pretrazi** (*Racun*);
2. signal **Storniraj** (*Racun*);

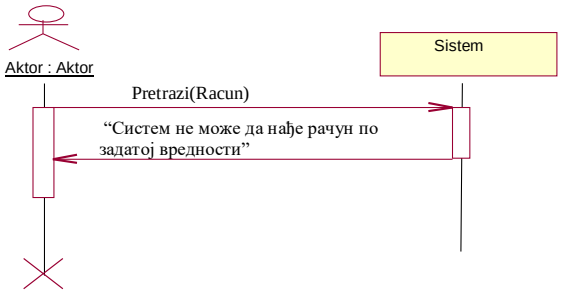
ДС4: *Дијаграми секвенци случаја коришћења – Промена рачуна*

1. Продавац **позива** систем да нађе рачун по задатој вредности. (АПСО)
2. Систем приказује продавцу рачун и поруку: "Систем је нашао рачун по задатој вредности". (ИА)
3. Продавац **позива** систем да запамти податке о рачуну. (АПСО)
4. Систем **приказује** продавцу поруку: "Систем је запамтио рачун." (ИА)
5. Продавац **позива** систем да обради рачун. (АПСО)
6. Систем **приказује** продавцу обрађен рачун и поруку: "Систем је обрадио рачун.(ИА)

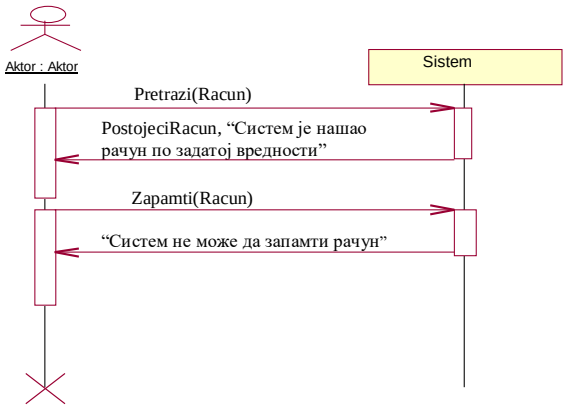


Алтернативна сценарија

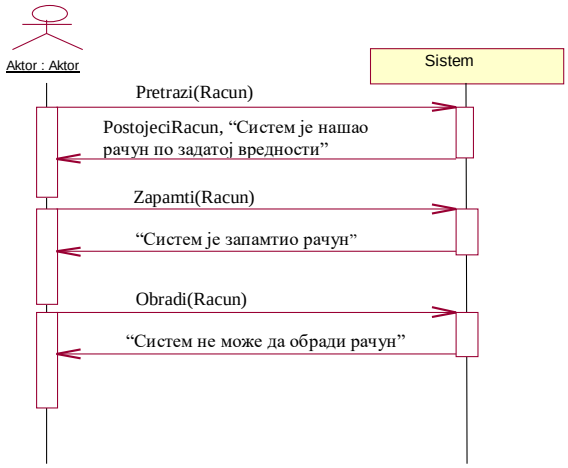
2.1 Уколико систем не може да нађе рачун он приказује продавцу поруку: "Систем не може да нађе рачун по задатој вредности". Прекида се извршење сценариа. (ИА)



4.1 Уколико систем не може да запамти податке о рачуну он приказује продавцу поруку "Систем не може да запамти рачун". Прекида се извршење сценариа. (ИА)



6.1 Уколико систем не може да обради рачун он приказује продавцу поруку: “Систем не може да обради рачун”. (ИА)



Са наведених секвенцних дијаграма уочавају се 3 системске операције које треба пројектовати:

1. *signal* **Pretrazi** (*Racun*);
2. *signal* **Zapamti**(*Racun*);
3. *signal* **Obradi**(*Racun*);

Као резултат анализе сценарија свих случајева коришћења добијено је укупно 5 системских операција које треба пројектовати:

1. *signal* **KreirajNovi** (*Racun*);
2. *signal* **Zapamti**(*Racun*);
3. *signal* **Obradi**(*Racun*);
4. *signal* **Pretrazi** (*Racun*);
5. *signal* **Storniraj** (*Racun*);

2.2.2 Дефинисање уговора о системским операцијама

За сваку од уочених системских операција праве се уговори.

Деф. АН3: Системска операција описује понашање софтверског система. Системска операција има свој потпис, који садржи име методе и опционо улазне и/или излазне аргументе.

Деф. АН4: Уговори се праве за системске операције и они описују њено понашање. Уговори описују шта операција треба да ради, без објашњења како ће то да ради. Један уговор је везан за једну системску операцију.
Уговори се састоје из следећих секција:

- **Операција –** име операције и њени улазни и излазни аргументи
- **Веза са СК –** имена СК у којима се позива системска операција
- **Предуслов** – пре извршења системске операције морају бити задовољени одређени предуслови (систем мора бити у одговарајућем стању).
- **Постуслови –** после извршења системске операције у систему морају бити задовољени одређени постуслови (систем мора бити у одговарајућем стању или се поништава резултат операције).

Деф. АН5: Постуслови СО указују на то шта треба да се деси (ефекти извршења СО), након извршења СО, а не како ће то да се деси.
Постуслови се изражавају у прошлом времену, како би се нагласило да је објекат дошао у ново стање, а не да ће доћи у ново стање. Нпр. *Ставка рачуна је креирана.* Не би било добро да се напише: *Креирање ставке рачуна.*

Деф. АН6: Предуслови СО указују на то шта је требало да се деси, како би СО могла да се изврши, а не како се то десило.

1. Уговор UG1: *KreirajNovi*

Операција: KreirajNovi (*Racun*):signal;
Веза са СК: СКП31
Предуслови: *Структурна и вредносна ограничење над Racun објектом морају бити задовољена.*
Постуслови: *Направљен је нови рачун.*

2. Уговор UG2: *Zapamti*

Операција: Zapamti(*Racun*):signal;
Веза са СК: *СК1, СК4*
Предуслови: *Структурна и вредносна ограничење над Racun објектом морају бити задовољена. Ако је рачун обрађен или сторниран не може се извршити системска операција.*

Постуслови:

- *Рачун је запамћен.*

3. Уговор UG3: *Obradi*

Операција: Obradi(*Racun*):signal;
Веза са СК: *СК1, СК4*
Предуслови: *Структурна и вредносна ограничење над Racun објектом морају бити задовољена. Ако је рачун обрађен или сторниран не може се извршити системска операција.*

Постуслови:

- *Рачун је обрађен.*

4. Уговор UG4: *Pretraži*

Операција: Pretraži (*Racun*):signal;
Веза са СК: *СК2, СК3, СК4*
Предуслови:
Постуслови: Пронађен је тражени рачун.

5. Уговор UG5: *Storniraj*

Операција: Storniraj(*Racun*):signal;
Веза са СК: *СК3*
Предуслови: *Структурна и вредносна ограничење над Racun објектом морају бити задовољена. Ако је рачун сторниран не може се извршити системска операција.*
Постуслови: *Рачун је сторниран.*

2.2.3 Ограничења при извршењу системских операција

Системске операције се могу састоји од: а) **операција одржавања базе података** (убаци, избаци и промени) и/или б) **операција извештавања**.
У току извршења системске операције над структуром софтверског система (односно над базом података) подаци (објекти у оперативној меморији и слогови у бази података) морају да остану конзистентни, односно морају да буду задовољена вредносна и структурна органичења дефинисана над подацима.

Вредносна ограничења се односе на дозвољене вредности атрибута доменских класа (табела) и она се деле на: а) проста вредносна ограничења - ограничења на домен (тип) атрибута и вредност атрибута. б) сложена вредносна ограничења - међузависност атрибута. **Структурна органичења** су дефинисана преко кардиналности пресликавања између доменских класа (табела).

При извршењу операција **убаци** и **промени** објекат (слог) проверавају се и вредносна и структурна ограничења. Код уговора за системске операције у одељку **постуслови** се наводи резултат операција убаци (нпр. *Направљен је нови рачун*), и промени (*нпр. Рачун је сторниран, Израчуната је укупна вредност рачуна,...*). Код уговора за системске операције у одељку **предуслови** и/или **постуслови** се наводе вредносна и структурна ограничења. Сложена вредносна ограничења се могу навести у одељку предуслови или се могу навести кроз реализацију системске операције.

При извршењу операције **обриши** објекат (слог) проверавају се структурна и вредносна ограничења. Код уговора за системске операције у одељку **постуслови** се наводи резултат операција обриши (нпр. *Обрисан је предмет*), док се у одељку **предуслови** и/или **постуслови** наводе вредносна и структурна ограничења.

При извршењу операције **извештавања** не проверавају се вредносна и структурна ограничења. Код уговора за системске операције у одељцима **постуслови** и **предуслови** ништа се не наводи везано за ограничења. У постусловима је могуће навести резултат претраживања.

2.2.4 Структура софтверског система - Концептуални (доменски) модел

Структура софтверског система се описује помоћу концептуалног модела. Наводимо дефиниције концептуалног модела и његових елемената.

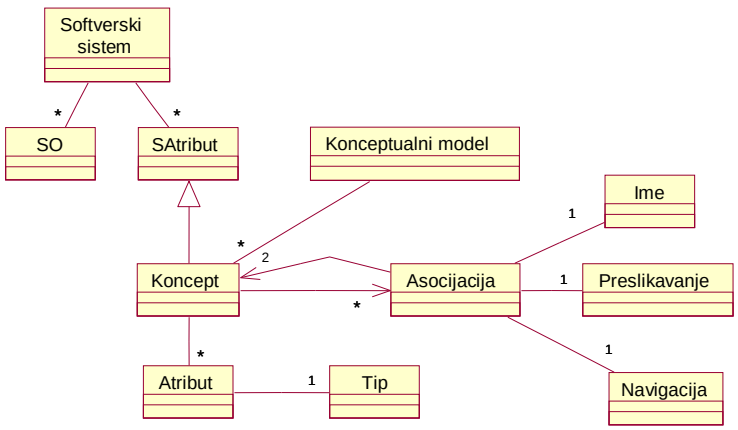
Дефиниција АН7: Концептуални модел описује концептуалне класе домена проблема. Концептуални модел садржи концептуалне класе (доменске објекте) и асоцијације између концептуалних класа.Често се за концептуалне моделе каже да су то **доменски модели** или **модели објектне анализе.**

Дефиниција АН8: Концепти (концептуалне класе) представљају атрибуте софтверског система. То значи да концепти описују структуру софтверског система. Концептуалне класе састоје се од атрибута, који описују особине класе. Концептуалне класе треба разликовати од софтверских класа.

Дефиниција АН9: Атрибути представљају особине која се придружују до концептуалних класа. Сваки од атрибута је везан за одређени тип податка.
Атрибут има конкретну вредност за конкретно појављивање концептуалне класе.

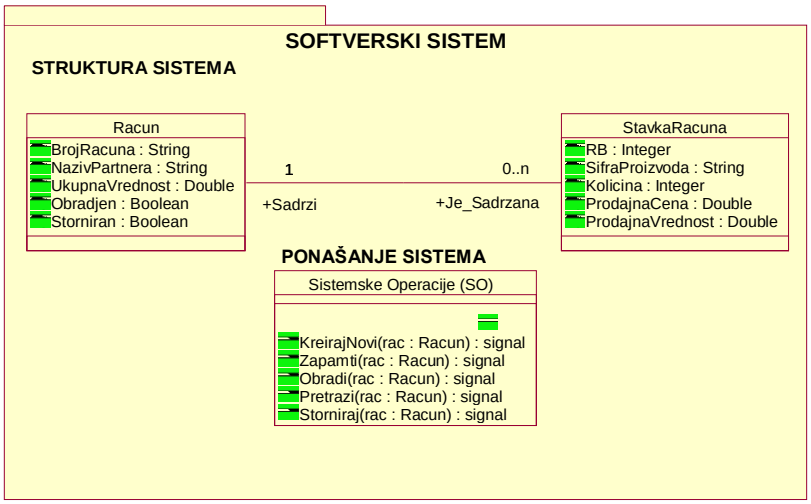
Дефиниција АН10: Асоцијација је веза између концептуланих класа. Сваки крај асоцијације представља **улогу** концепта који учествује у асоцијацији. Улога садржи име, пресликавање и навигацију.
Име улоге је засновано на формату: ИмеКонцКласе1 – Глагол – ИмеКонцКласе2, где глагол описује однос између концептуалних класа у датом контексту.
Пресликавање дефинише колико много појављивања концептуалне класе А може бити придружено једном појављивању концептуалне класе Б.
Навигација указује на једносмерне везе између концептуалних класа.
Између концептуалних класа може постојати више асоцијација.

Објашњење дефиниција концептуалног модела и његових елемената
Софтверски систем се састоји од атрибута (САтрибут) и системских операција (СО). Концепти представљају реализацију атрибута софтверског система.



Конкретан пример концептуалног модела

Као резултат анализе сценарија СК и прављења концептуалног модела добија се **логичка структура и понашање софтверског система**:



2.2.5 Структура софтверског система - Релациони модел

На основу концептуалног модела може се направити релациони модел, који ће да представља основу за пројектовање релационе базе података[Ullman].

На основу датог концептуалног модела (Racun, StavkeRacuna) прави се **релациони модел**:

Racun(BrojRacuna, NazivPartnera,UkupnaVrednost,Obradjen,Storniran);

StavkaRacuna(BrojRacuna,RB,SifraProizvoda,Kolicina,ProdajnaCena,ProdajnaVrednost)

Табела Racun		**Просто вредносно ограничење**		**Сложено вредносно ограничење**		**Структурно ограничење**
Atributi	**Име**	**Тип атрибута**	**Вредност атрибута**	**Међузав. атрибута једне табеле**	**Међузав.атрибута више табела**	INSERT / UPDATE CASCADES StavkeRacuna DELETE CASCADES StavkeRacuna
	BrojRacuna	String	not null			
	NazivPartnera	String				
	UkupnaVrednost	Double	(default:0)		UkupnaVrednost= SUM (StavkaRacuna.ProdajnaVrednost)	
	Obradjen	Boolean	(default: false)			
	Storniran	Boolean	(default: false)			

Код сложених објеката (као што је Racun) треба узети следеће ограничење у обзир:
UPDATE Racun ----> DELETE StavkeRacuna (у бази) ----> INSERT StavkeRacuna (из оперативне меморије)

Табела StavkaRacuna		**Просто вредносно ограничење**		**Сложено вредносно ограничење**		**Структурно ограничење**
Atributi	**Име**	**Тип атрибута**	**Вредност атрибута**	**Међузав. атрибута једне табеле**	**Међузав. атрибута више табела**	INSERT RESTRICTED Racun UPDATE RESTRICTED Racun DELETE /
	BrojRacuna	String	Not null			
	RB	Integer	Not null and > 0			
	SifraProizvoda	String				
	Kolicina	Integer	>0 (default:0)			
	ProdajnaCena	Double	>0 (default:0)			
	ProdajnaVrednost	Double	(default:0)	ProdajnaVrednost = Kolicina*ProdajnaCena		

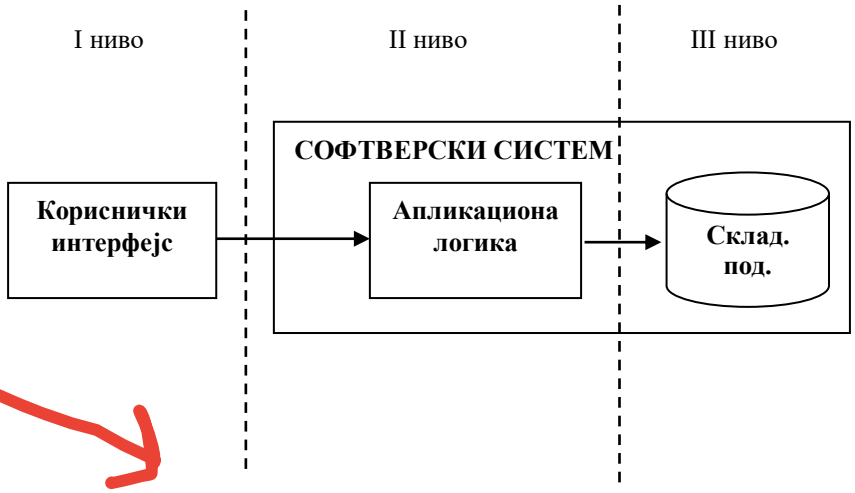
2.3 Пројектовања

Фаза пројектовања описује физичку структуру и понашање софтверског система (архитектуру софтверског система). Пројектовање архитектуре софтверског система обухвата пројектовање корисничког интерфејса, апликационе логике и складишта података. Пројектовање корисничког интерфејса обухвата пројектовање екранских форми и контролера корисничког интерфејса. У оквиру апликационе логике се пројектују контролер апликационе логике, пословна логика и брокер базе података. Пројектовање пословне логика обухвата пројектовање логичке структуре и понашања софтверског система.

АРХИТЕКТУРА СОФТВЕРСКОГ СИСТЕМА
Пре него што кренемо на пројектовање структуре и понашања софтверског система потребно је да се дефинише архитектура софтверског система. Ми ћемо користити класичну тронивојску архитектуру (Tsichritzis & Klug (1978), Schulte, 1995).

Дефиниције и правила архитектуре софтверског систа,

Деф. PRAR1: Тронивојска архитектура се састоји из следећих ниво:
1. **Корисничког интерфејса** који представља улазно – излазну репрезентацију софтверског система.
2. **Апликационе логике** која описује структуру и понашање софтверског система.
3. **Складишта података** који чува стање атрибута софтверског система.



Слика ТНА: Тронивојска архитектура

Правило PRpARH1: Апликациона логика се пројектује независно од корисничког интерфејса и обрнуто.

Правило PRpARH2: Апликациона логика може да има различите улазно-излазне репрезентације.

Правило PRpARH3 (Model-View Separation Principle): Апликациона логика (модел) нема знања о томе где се налази кориснички интерфејс (поглед).

Деф. ПРАР2: На основу **тронивојске архитектуре** су направљени савремени **апликациони сервери**.

Деф. ПРАР3: Апликациони сервери су одговорни да обезбеде сервисе који ће да омогуће реализацију апликационе логике софтверског система. Сваки апликациони сервер се састоји из три основна дела:
1. део за комуникацију са клијентим (контролер)
2. део за комуникацију са неким складиштем података (база података или датотечни систем)
3. део који садржи пословну логику

Деф. ПРАР4: Контролер је одговоран да прихвати захтев за извршење системске операције од клијента и да га проследи до пословне логике која је одговорна за извршење системске операције.

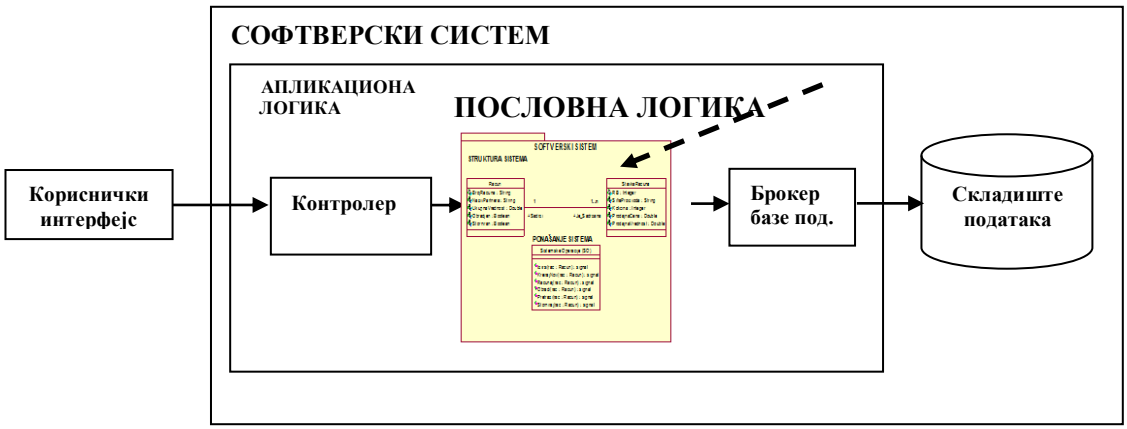
Деф. ПРАР5: Пословна логика је описана са структуром (доменским класама) и понашањем (системским операцијама).

Деф. ПРАР6: Брокер базе података је одговоран за комуникацију између пословне логике и складишта података.

У даљем тексту ћемо пројектовати сваки од наведених елемената тронивојске архитектуре:
- контролер
- пословна логика – доменске класе
- пословна логика – системске операције
- датабасе брокер
- складиште података
- кориснички интерфејс

Из наведеног можемо да закључимо да смо у фазама прикупљања захтева и анализе дали спецификацију структуре и понашања софтверског система, односно **спецификацију пословне логике софтверског система** (Slika ASSPL).

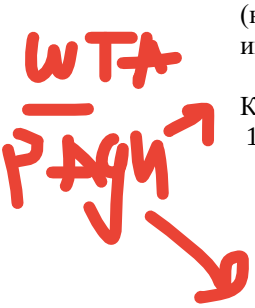
Из наведеног можемо да закључимо да смо у фазама прикупљања захтева и анализе дали спецификацију структуре и понашања софтверског система, односно **спецификацију пословне логике софтверског система**.



Слика **АССПЛ**: Архитектура софт. система – Пословна логика

2.3.1 Пројектовање корисничког интерфејса

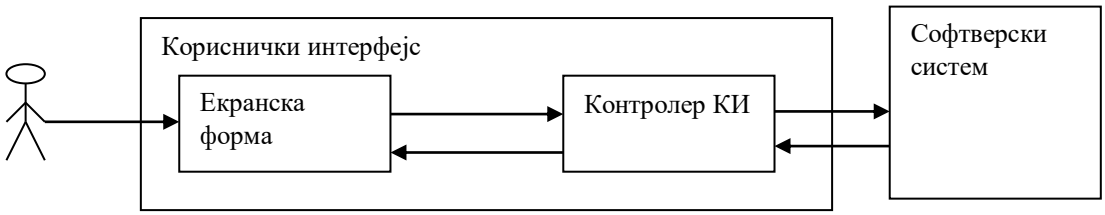
Кориснички интерфејс, сходно Деф. Лаз1, представља реализацију улаза и/или излаза софтверског система. Пре пројектовања корисничког интерфејса објаснићемо делове корисничког интерфејса (његову структуру), начин њихове међусобне комуникације и начин комуникације корисничког интерфејса са софтверским системом (Слика СКИ).



Кориснички интерфејс се састоји од:

1. Екранске форме која је одговорна да:
   а) прихвата податке које уноси актор,
   б) прихвата догађаје које прави актор,
   ц) позива контролера графичког интерфејса, прослеђујући му прихваћене податке
   д) приказује податке које је добио од контролера графичког интерфејса.

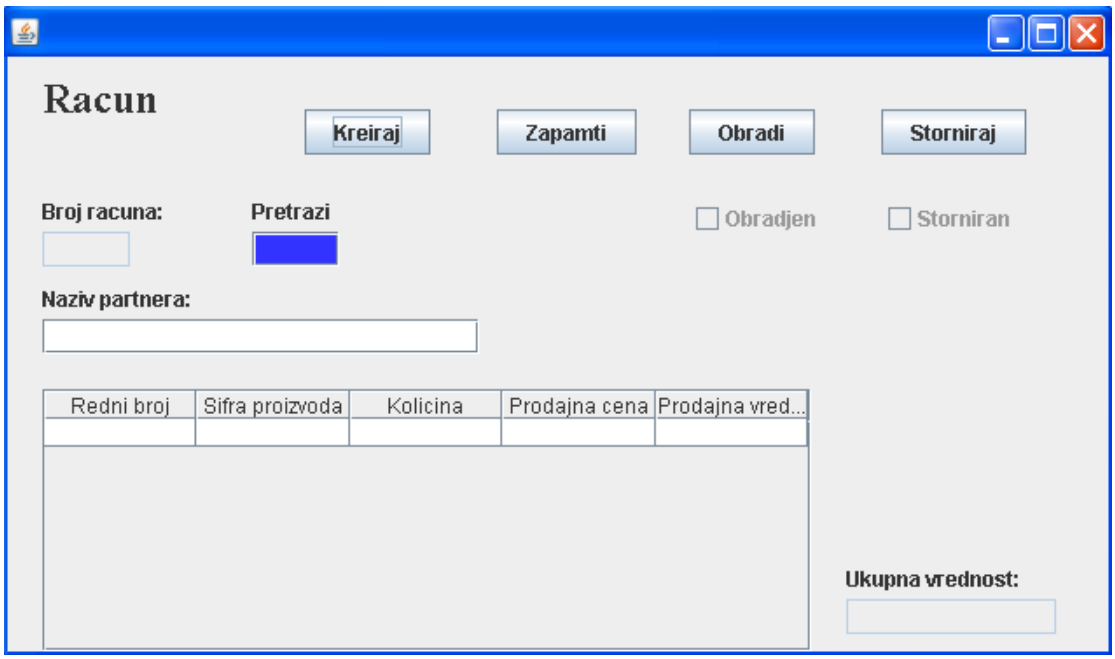
2. Контролера корисничког интерфејса који је одговоран да:
   а) прихвати податке које шаље екранска форма,
   б) конвертује податке (који се налазе у графичким елементима) у објекат који представља улазни аргумент СО која ће бити позвана,
   ц) шаље захтев за извршење СО до апликационог сервера (софтверског система),
   д) прихвата објекат (излаз) софтверског система који настаје као резултат извршења СО и
   е) конвертује објекат у податке графичких елемената.



Слика **СКИ:** Структура корисничког интерфејса

2.3.1.1 Пројектовање екранске форме

Екранска форма треба, за наведени пример, да има следећи изглед:



Кориснички интерфејс је дефинисан преко скупа екранских форми. Сценарија коришћења екранских форми је директно повезан са сценаријима случајева коришћења.
Постоје два аспекта пројектовања екранске форме:

a) Пројектовање сценарија СК који се изводе преко екранске форме.
b) Пројектовање метода екранске форме.

2.3.1.1.1 ПројектовањесценаријаСК

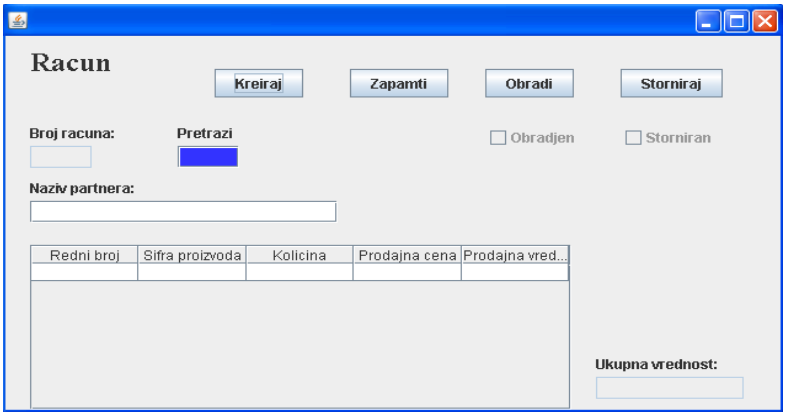
СК1: Случај коришћења – Креирање новог рачуна

Назив СК
Креирање новог рачуна

Актори СК
Продавац

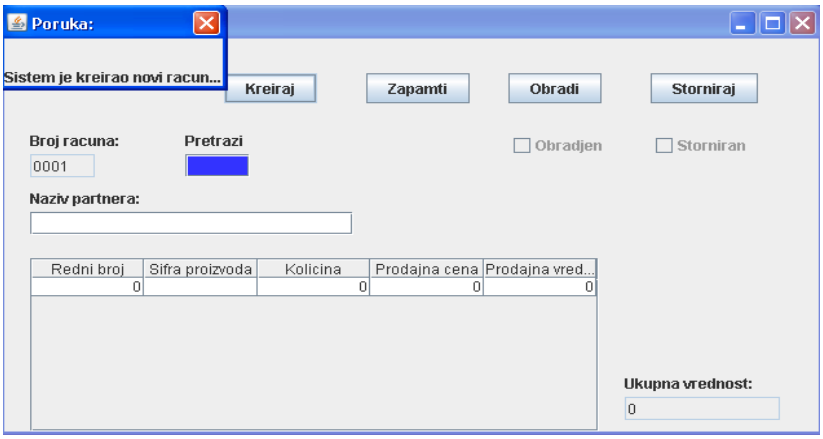
Учесници СК
Продавац и систем (програм)

Предуслов: Систем је укључен и продавац је улогован под својом шифром. Систем приказује форму за рад са рачуном.

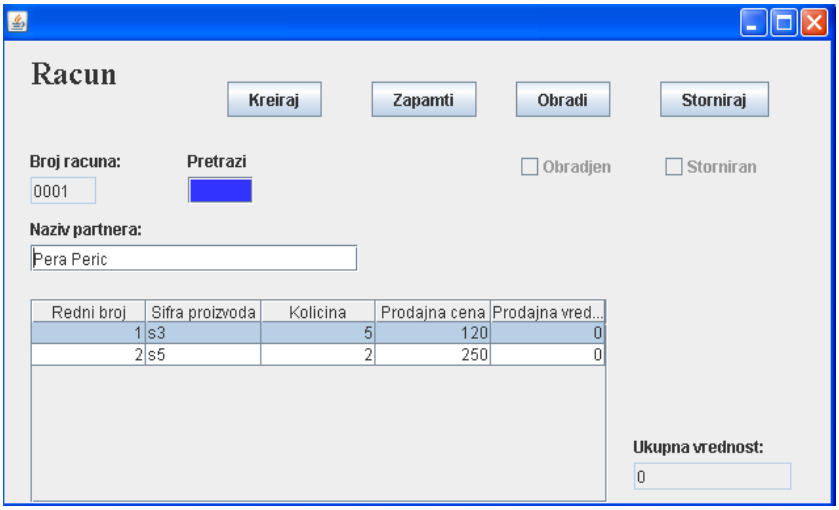


Основни сценарио СК

1. Продавац **позива** систем да креира нови рачун. (АПСО)
*Опис акције: Продавац кликом на дугме "Kreiraj" позива системску операцију **kreirajNovi (Racun)** која прави нови рачун.*
2. Систем **креира** нови рачун. (СО)
3. Систем **приказује** продавцу нови рачун и поруку: "Систем је креирао нови рачун". (ИА)



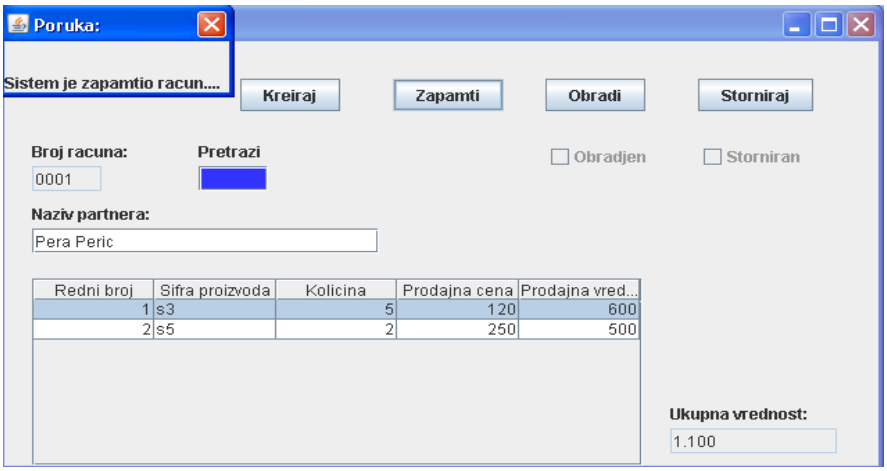
4. Продавац **уноси** податке у нови рачун. (АПУСО)



5. Продавац **контролише** да ли је коректно унео податке у нови рачун. (АНСО)
6. Продавац **позива** систем да запамти податке о рачуну. (АПСО)

*Опис акције: Продавац кликом на дугме "Zapamti" позива системску операцију **Zapamti (Racun)** која памти нови рачун.*

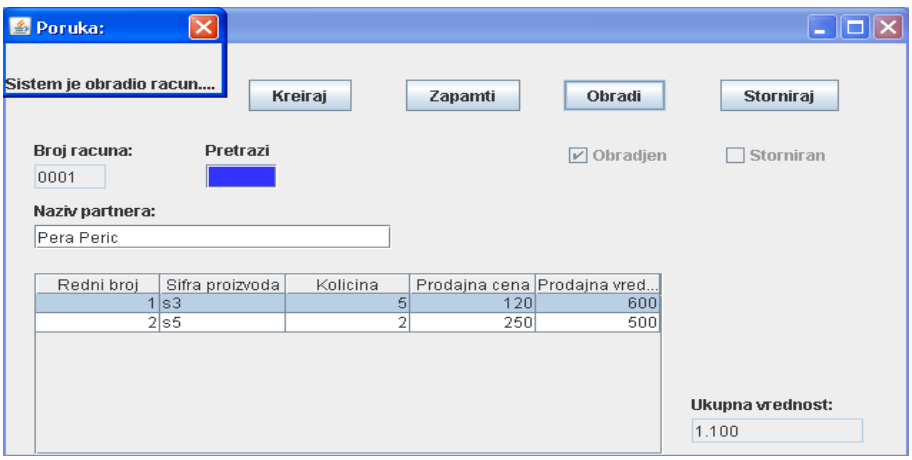
7. Систем **памти** податке о рачуну. (СО)
8. Систем **приказује** продавцу запамћени рачун и поруку: "Систем је запамтио рачун". (ИА)



9. Продавац **позива** систем да обради рачун. (АПСО)

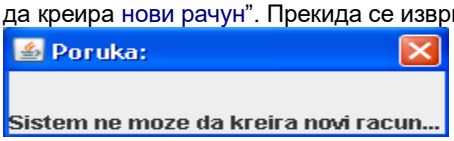
*Опис акције: Продавац кликом на дугме "Obradi" позива системску операцију **Obradi (Racun)** која обрађује нови рачун.*

10. Систем **обрађује** рачун.(СО)
11. Систем **приказује** продавцу обрађен рачун и поруку: "Систем је обрадио рачун".(ИА)

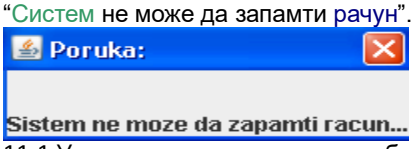


Алтернативна сценарија

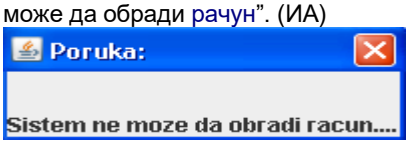
3.1 Уколико систем не може да креира рачун он приказује продавцу поруку: “Систем не може да креира нови рачун”. Прекида се извршење сценариа. (ИА)



8.1 Уколико систем не може да запамти податке о рачуну он приказује продавцу поруку “Систем не може да запамти рачун”. Прекида се извршење сценариа. (ИА)



11.1 Уколико систем не може да обради рачун он приказује продавцу поруку: “Систем не може да обради рачун”. (ИА)



На сличан начин за наведени пример унећемо још 2 рачуна.

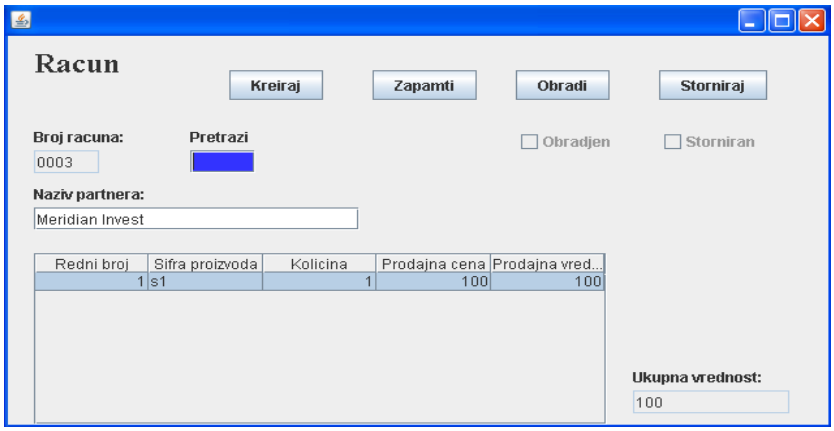
СК2: Случај коришћења – Претраживање рачуна

Назив СК
Претраживање рачуна

Актори СК
Корисник

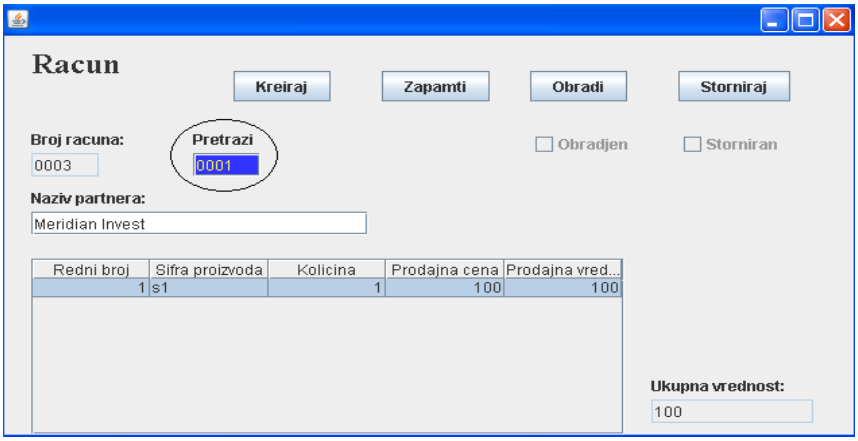
Учесници СК
Корисник и систем (програм)

Предуслов: Систем је укључен и корисник је улогован под својом шифром. Систем приказује форму за рад са рачуном.



Основни сценарио СК

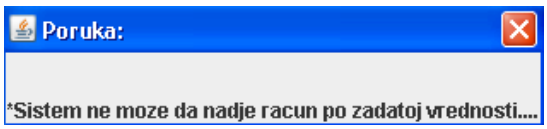
1. Корисник **уноси** вредност по којој претражује рачун. (АПУСО)
Опис акције: Корисник уноси вредност у поље под називом Pretrazi.



2. Корисник **позива** систем да нађе рачун по задатој вредности. (АПСО)
Опис акције: *Корисник након уноса вредности у поље под називом Pretrazi притиска типку <Enter> и позива системску операцију* ***Pretrazi (Racun)****.*
3. Систем **тражи** рачун по задатој вредности. (СО)
4. Систем приказује кориснику податке о рачуну и поруку: "Систем је нашао рачун по задатој вредности". (ИА)

Алтернативна сценарија

4.1 Уколико систем не може да нађе рачун он приказује кориснику поруку: "Систем не може да нађе рачун по задатој вредности". (ИА)



СК3: Случај коришћења – Сторнирање рачуна

Назив СК
Сторнирање рачуна

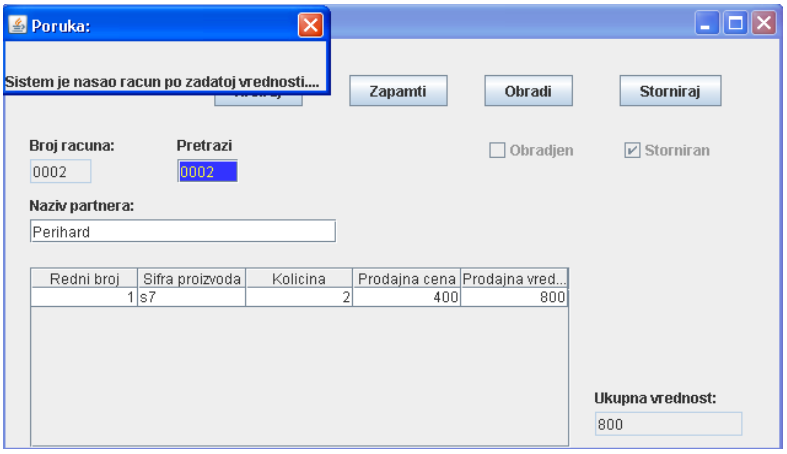
Актори СК
Корисник

Учесници СК
Корисник и систем (програм)

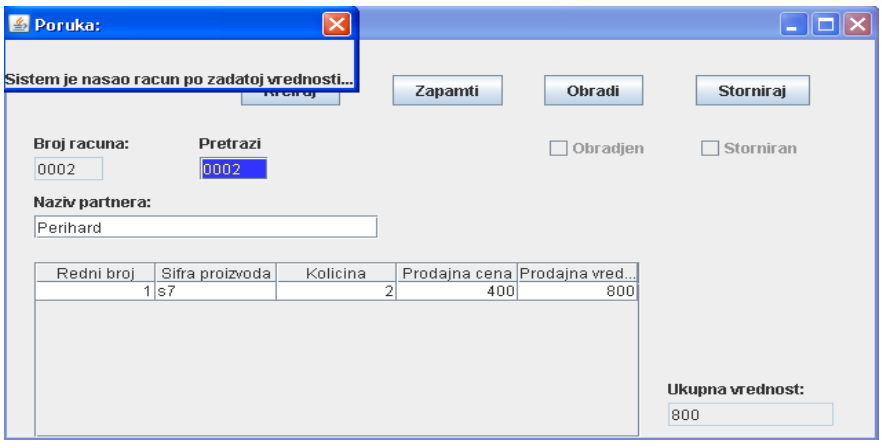
Предуслов: Систем је укључен и корисник је улогован под својом шифром. Систем приказује форму за рад са рачуном.

Основни сценарио СК

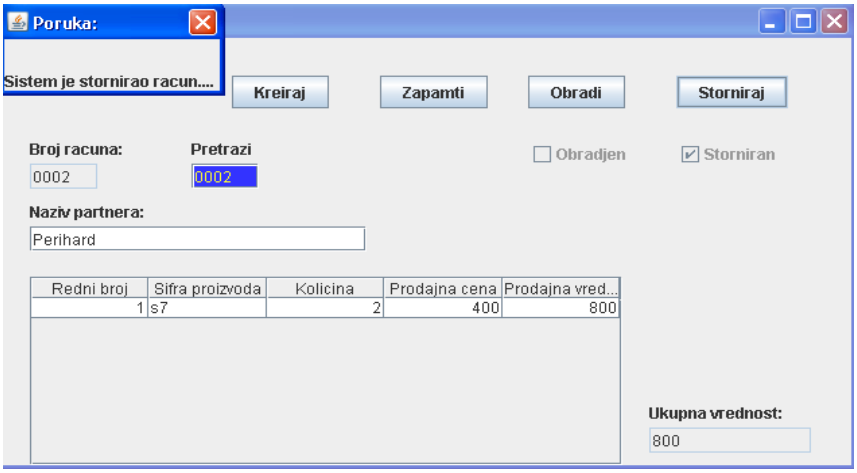
1. Корисник **уноси** вредност по којој претражује рачун. (АПУСО)
   Опис акције: Корисник уноси вредност у поље под називом Pretrazi.



2. Корисник **позива** систем да нађе рачун по задатој вредности. (АПСО)
   *Опис акције: Корисник након уноса вредности у поље под називом Pretrazi притиска типку <Enter> и позива системску операцију **Pretrazi (Racun)**.*
3. Систем **тражи** рачун по задатој вредности. (СО)
4. Систем приказује кориснику рачун и поруку: "Систем је нашао рачун по задатој вредности". (ИА)

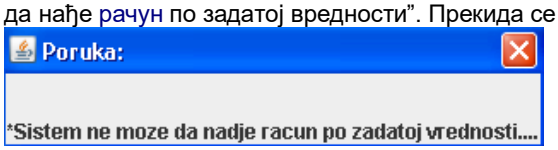


5. Корисник **позива** систем да сторнира задати рачун. (АПСО)
   *Опис акције: Корисник кликом на дугме "Storniraj " позива системску операцију **Storniraj (Racun)**.*
6. Систем **сторнира** рачун. (СО)
7. Систем **приказује** кориснику сторниран рачун и поруку: "Систем је сторнирао рачун ". (ИА)

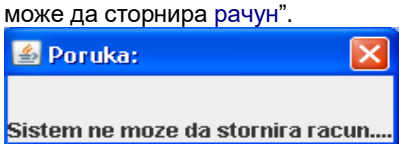


Алтернативна сценарија

4.1 Уколико систем не може да нађе рачун он приказује кориснику поруку: “Систем не може да нађе рачун по задатој вредности”. Прекида се извршење сценариа. (ИА)



7.1 Уколико систем не може да сторнира рачун он приказује кориснику поруку: “Систем не може да сторнира рачун”.



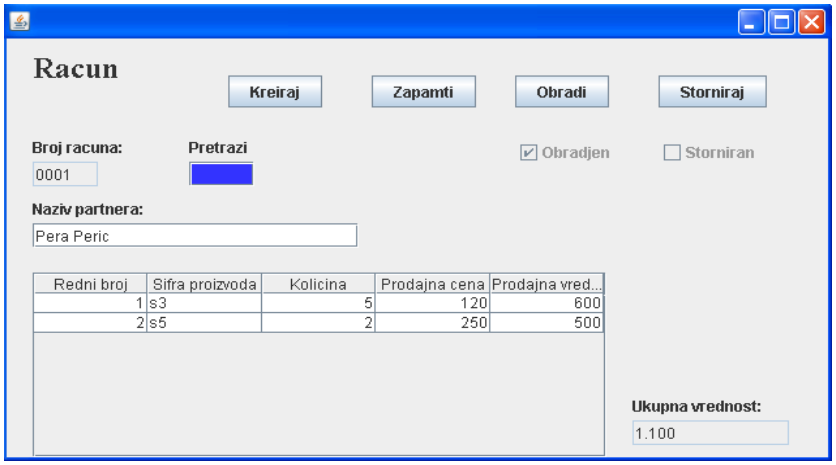
СК4: Случај коришћења – Промена рачуна

Назив СК
Промена рачуна

Актори СК
Продавац

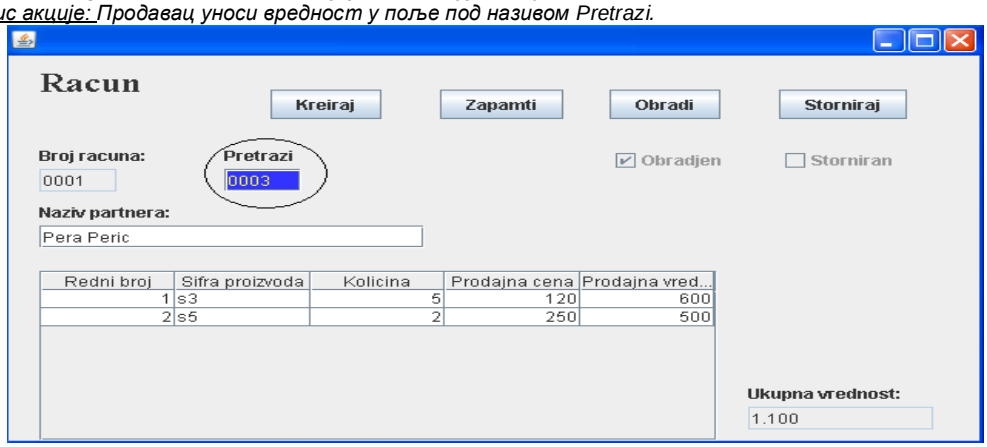
Учесници СК
Продавац и систем (програм)

Предуслов: Систем је укључен и продавац је улогован под својом шифром. Систем приказује форму за рад са рачуном.

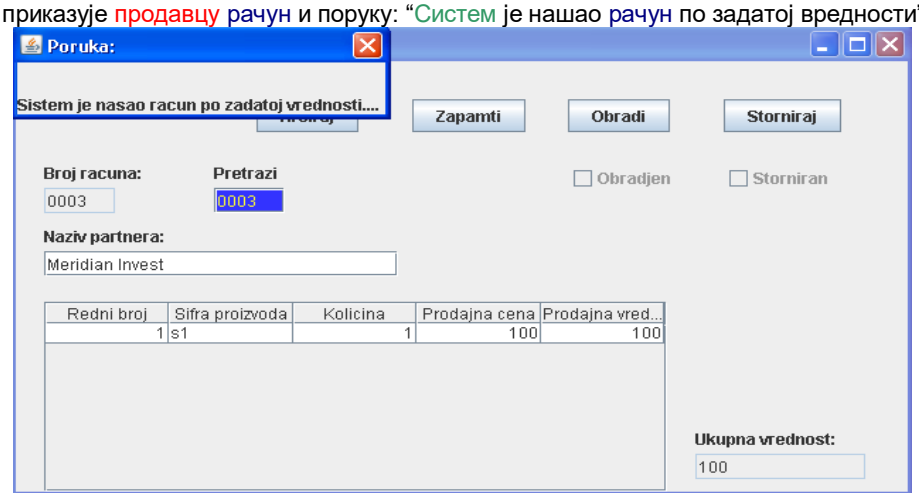


Основни сценарио СК

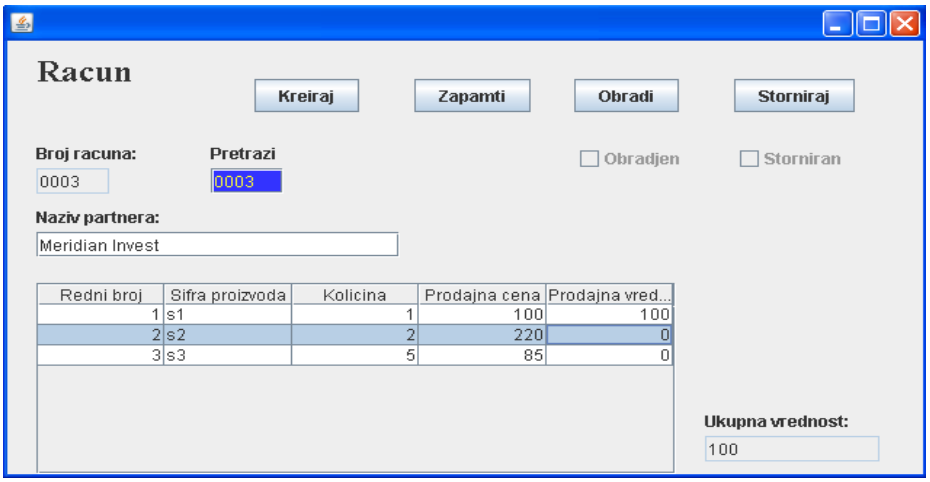
1. Продавац **уноси** вредност по којој претражује рачун. (АПУСО)
   Опис акције: Продавац уноси вредност у поље под називом Pretrazi.



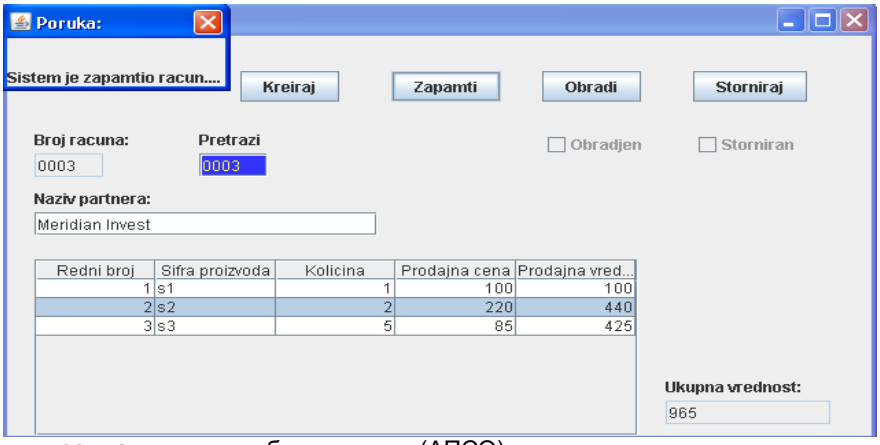
2. Продавац **позива** систем да нађе рачун по задатој вредности. (АПСО)
3. Систем **тражи** рачун по задатој вредности. (СО)
4. Систем приказује продавцу рачун и поруку: "Систем је нашао рачун по задатој вредности". (ИА)



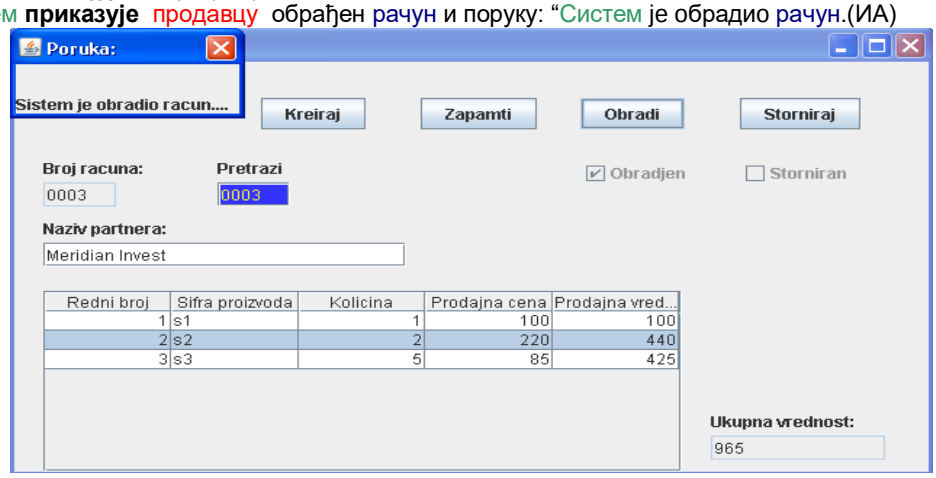
5. Продавац **уноси (мења)** податке о рачуну. (АПУСО)



6. Продавац **контролише** да ли је коректно унео податке о рачуну. (АНСО)
7. Продавац **позива** систем да запамти податке о рачуну. (АПСО)
   *Опис акције: Продавац кликом на дугме "Zapamti " позива системску операцију **Zapamti (Racun)**.*
8. Систем **памти** податке о рачуну. (СО)
9. Систем **приказује** продавцу запамћени рачун и поруку: "Систем је запамтио рачун." (ИА)

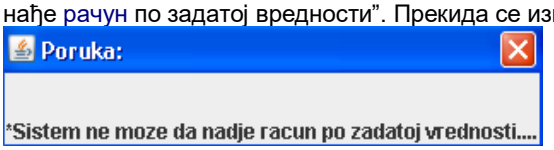


10. Продавац **позива** систем да обради рачун. (АПСО)
    *Опис акције: Продавац кликом на дугме "Obradi " позива системску операцију **Obradi (Racun)** која обрађује нови рачун.*
11. Систем **обрађује** рачун.(СО)
12. Систем **приказује** продавцу обрађен рачун и поруку: "Систем је обрадио рачун.(ИА)

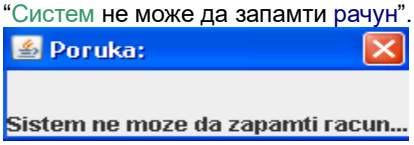


Алтернативна сценарија

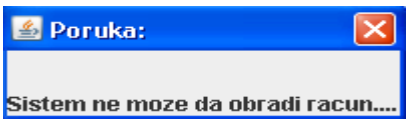
4.1 Уколико систем не може да нађе рачун он приказује продавцу поруку: "Систем не може да нађе рачун по задатој вредности". Прекида се извршење сценариа. (ИА)



9.1 Уколико систем не може да запамти податке о рачуну он приказује продавцу поруку "Систем не може да запамти рачун". Прекида се извршење сценариа. (ИА)



12.1 Уколико систем не може да обради рачун он приказује продавцу поруку: "Систем не може да обради рачун". (ИА)



2.3.1.1.2 Пројектовање метода екранске форме

Постоје две класе које се користе у пројектовању метода екранске форме: **OpstaEkranskaForma** и **EkranskaFormaRacun.** Абстрактна класа OpstaEkranskaForma садржи опште методе које су независне од екранске форме која ће да репрезентује неку доменску класу. Док класа EkranskaFormaRacun, садржи методе које приказују екранску форму која репрезентује конкретну доменску класу (у нашем примеру рачун) са свим припадајућим графичким елементима. Класа EkranskaFormaRacun је повезана са класом KontrolerKIRacun којој прослеђује графички објекат (при иницијализацији) и захтев за извршење системске операције.

```
abstract class OpstaEkranskaForma extends … // (navodi se ime klase koja omogućava kreiranje
                                              //  ekranske forme)
{
   … // navode se opšte metode ekranske forme.
  abstract OpstiDomenskiObjekat kreirajObjekat();
}


class EkranskaFormaRacun extends OpstaEkranskaForma
 { KontrolerKIRacun kkir;


   // Glavni program
   public static void main(String args[])
     { EkranskaFormaRacun EF = new EkranskaFormaRacun();
     }

   // 1. Konstruktor ekranske forme
   public EkranskaFormaRacun ()
     {  kreirajKomponenteEkranskeForme();          // 1.1
        pokreniMenadzeraRasporedaKomponeti();   // 1.2
        postaviImeForme();                        // 1.3
        postaviTextFieldBrojRacuna();             // 1.4
        postaviTextFieldNazivPartnera();          // 1.5
        postaviTextFieldUkupnaVrednost();         // 1.6
       postaviTextFieldPretrazivanje();           // 1.7
       postaviCheckBoxObradjen();                 // 1.8
       postaviCheckBoxStorniran();                // 1.9
       postaviDugmeKreiraj();                     // 1.10
      postaviDugmeObradi();                       // 1.11
      postaviDugmeStorniraj();                    // 1.12
      postaviDugmeZapamti();                      // 1.13
      postaviTabelu();                            // 1.14
      inicijalizacijaKontrolera();                // 1.15
    }


…
void postaviDugmeKreiraj()
 {   …
     // Kada se klikne na dugne poziva se:
    String signal = kkir.SOKreirajNovi();
     …
}


void postaviDugmeObradi()
 { …
    // Kada se klikne na dugne poziva se:
    String signal = kkir.SOObradi();
    …
}
```

```
void  postaviDugmeStorniraj()
{ ...
   // Kada se klikne na dugne poziva se:
   String signal = kkir.SOStorniraj();
  …
 }


void postaviDugmeZapamti()
 { …
   // Kada se klikne na dugne poziva se:
   String signal = kkir.SOZapamti();
   …
}


void postaviTabelu()
{  ...
   // Kada se pritisne neka od tipki na tabeli poziva se:
         String signal = kkir.pritisakTipke(evt);
  …
 }


// 1.15 Inicijalizacija KontroleraKI
// Pri inicijalizaciji, kontroler dobija referencu na graficki objekat (this).
void inicijalizacijaKontrolera()
      {  kkir = new KontrolerKIRacun (this); }

OpstiDomenskiObjekat kreirajObjekat() {return new Racun();}

}

}
```

2.3.1.2 Пројектовање контролера корисничког интерфејса

Контролер корисничког интерфејса треба пројектовати тако да има општи део (***OpstiKontrolerKI***) који је независтан од екранске форме преко које се извршава сценаријо случаја коришћења и конректни део који је везан за домен екранске форме (**KontrolerKIRacun**).

Општи контролер:

a) успоставља везу између екранске форме и апликационе логике.
b) прихвата од екранске форме захтев за извршење системске операције.
c) креира доменски објекат.
d) прослеђује захтев за извршење системске операције и доменске објекте до апликационог сервера (апликационе логике).
e) прихвата доменске објекте и сигнале (о успешности извршења СО) које је вратио апликациони сервер као резултат извршења системске операције.

Конкретни контролер:

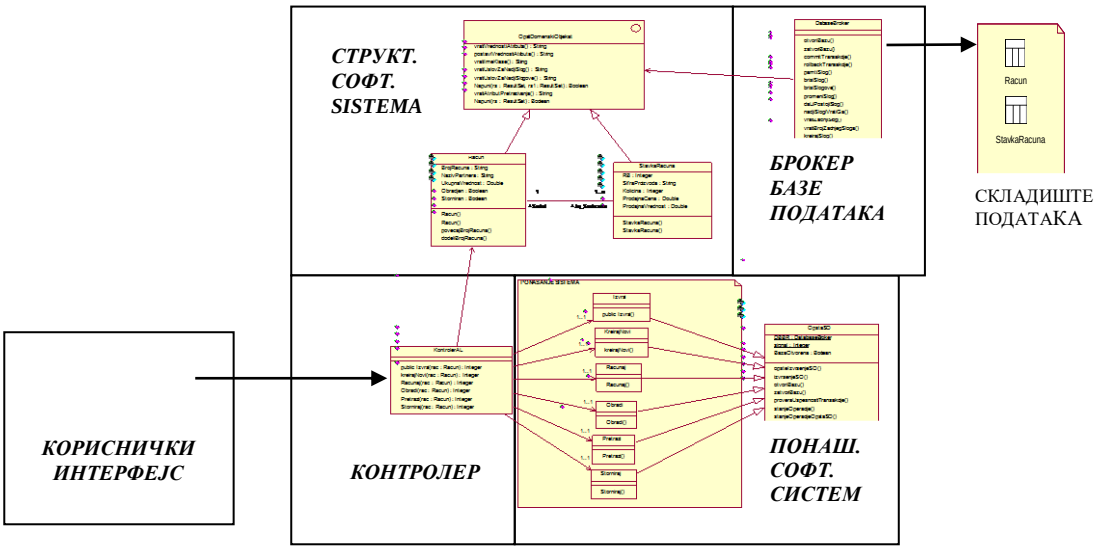
a) прихвата од екранске форме графичке објекте.
b) конвертује графичке објекте у доменске објекте који ће бити прослеђени преко мреже до апликационог сервера.
c) конвертује доменске објекте у графичке објекте и прослеђује их до екранске форме.

```
abstract class OpstiKontrolerKI
 { AplikacionaLogika al;
   String signal;
   OpstiDomenskiObjekat odo;
   OpstaEkranskaForma oef;

// a) успоставља везу између екранске форме и апликационе логике преко сокета.
   OpstiKontrolerKI()
     { UspostaviVezuIzmeđuEkranskeFormeIAplikacioneLogike();  }

   public String pritisakTipke(KeyEvent evt)
    {  OdredjujeSeNacinObradiTipkiPriRaduSaTabelom();  }

// b) прихвата од екранске форме захтев за извршење системске операције.
   public String SOPretrazi()
   {  // c) креира доменски објекат.
      odo = oef.kreirajObjekat();
      KonvertujGrafickiObjekatUDomenskiObjekat();
      signal = pozivSO("Pretrazi");
      KonvertujDomenskiObjekatUGrafickiObjekat();
      return signal;
  }

  public String SOKreirajNovi()
  {  odo = oef.kreirajObjekat();
      signal = pozivSO("kreirajNovi");
      KonvertujObjekatUGrafickeKomponente();
      return signal;
  }

  public String SOZapamti()
  {  odo = oef.kreirajObjekat();
      KonvertujGrafickiObjekatUDomenskiObjekat ();
      signal = pozivSO("Zapamti");
      KonvertujDomenskiObjekatUGrafickiObjekat();
      return signal;
  }

  public String SOStorniraj()
  {  odo = oef.kreirajObjekat();
      KonvertujGrafickiObjekatUDomenskiObjekat();
      signal = pozivSO("Storniraj");
      KonvertujDomenskiObjekatUGrafickiObjekat();
      return signal;
  }
```

```
  public String SOObradi()
   {   odo = oef.kreirajObjekat();
       KonvertujGrafickiObjekatUDomenskiObjekat();
       signal = pozivSO("Obradi");
       KonvertujDomenskiObjekatUGrafickiObjekat();
       return signal;
    }

// d) прослеђује захтев за извршење системске операције и доменске објекте до апликационог
// сервера (апликационе логике).
// e) прихвата доменске објекте и сигнале (о успешности извршења СО) које је вратио
// апликациони сервер као резултат извршења системске операције.
String pozivSO(String nazivSO)
{  signal = PozivAplikacionogServera(odo);
   return signal;
}

abstract public void KonvertujGrafickiObjekatUDomenskiObjekat();
abstract public void KonvertujDomenskiObjekatUGrafickiObjekat();
}



class KontrolerKIRacun extends OpstiKontrolerKI
{

// a) прихвата од екранске форме графичке објекте.
  KontrolerKIRacun(EkranskaFormaRacun efr)  {oef = efr;}

// b) конвертује графичке објекте  у доменске објекте који ће бити прослеђени преко мреже до
// апликационог сервера.
public void KonvertujGrafickiObjekatUDomenskiObjekat()
   {
      Racun rac = (Racun) odo;
      EkranskaFormaRacun efr = (EkranskaFormaRacun) oef;
      KonvertujeElementeGrafickogObjektauAtributeDomenskogObjekta();
      KonvertujeRedoveTabeleStavkeRacunaUNizObjekataStavkeRacuna();
   }


// c) конвертује доменске објекте у графичке објекте и прослеђује их до екранске форме.
 public void KonvertujDomenskiObjekatUGrafickiObjekat()
   {  Racun rac = (Racun) odo;
      EkranskaFormaRacun efr = (EkranskaFormaRacun) oef;
      KonvertujeAtributeDomenskogObjektaUElementeGrafickogObjekta();
      KonvertujeNizObjekataStavkeRacunaURedoveTabeleStavkeRacuna();
   }
}
```

Након пројектовања корисничког интерфејса добија се следећи дијаграм класа:

Кориснички интерфејс у контексту архитектуре софтверског система може се представити на следећи начин:



2.3.2 Пројектовање апликационе логике

2.3.2.1 Контролер апликационе логике

Контролер апликационе логике треба да подигне серверски сокет који ће да ослушкује мрежу. Када клијент (клијентски сокет) успостави конекцију са контролером (серверским сокетом), тада контролер треба да генерише нит која ће успоставити двосмерну везу са клијентом (улазну и излазну). Слање и примање података од клијента се остварује преко сокета.
Клијент шаље захтев за извршење неке од СО до одговарајуће нити (коју смо назвали "нит клијента"), која је повезана са тим клијентом. "Нит клијента" прима захтев и даље га преусмерава до класа које су одговорне за извршење СО. Након извршења СО резултат се враћа до апликационе логике, односно до "нити клијента", која тај резултат шаље назад до клијента.

Дајемо приказ пројектоване класа *KontrolerAL* и *NitKlijenta*.

```
class KontrolerPL // Kontroler poslovne logike
{
   void main(String[] args)
            {
            serverskiSoket = podizanjeServerskogSoketa();
            for(…)
              {  klijentskiSoket = serverskiSoket.osluskivanjeMreze();
                 NitKlijenta.kreiranjeNitiKlijenta(klijentskiSoket);

               }

            }


class NitKlijenta {

void kreiranjeNitiKlijenta (klijentskiSoket)
  {
    povezivanjeNitiSaKlijentskimSoketom();
    Ulaz ul = kreiranjeUlazaPrekoKlijentskogSoketa();
    Izlaz iz = kreiranjeIzlazaPrekoKlijentskogSoketa();

     ImeOperacije = ul.PrihvatiImeOperacijeOdKlijenta();
     OpstiDomenskiObjekat odo = ul.PrihvatiDomenskiObjekatOdKlijenta();

     if  (ImeOperacije = "kreirajNovi")   signal = KreirajNovi.kreirajNovi(odo);
     if  (ImeOperacije = "Pretrazi")      signal = Pretrazi.Pretrazi(odo);
     if  (ImeOperacije = "Zapamti")       signal = Racunaj.Zapamti(odo);
     if  (ImeOperacije = "Obradi")        signal = Obradi.Obradi(odo);
     if  (ImeOperacije = "Storniraj")     signal = Storniraj.Storniraj(odo);

      izl.SlanjeDomenskogObjektaDoKlijenta(rac);
      izl.SlanjeSignalaDoKlijenta(signal);
  }
```

2.3.2.2 Пословна логика

2.3.2.2.1 Пројектовање понашања софтверског система – системске операције

Препорука ПР1: У почетку пројектовања СО треба треба направити концептуалне реализације (решења) за сваку СО.Концептуалне реализације требају да буду директно повезане са логиком проблема.

Препорука ПР2: Може се предпоставити да се подаци (стање софтверског система) чувају у бази. У том смислу могу се позвати неке од основних операција базе (insert,update,delete,find,...), без улажења у начин њихове реализације.

Препорука ПР3: Аспекти реализације који се односе на конекцију са базом, перзистентност и трансакције треба избећи у почетку пројектовања СО. У каснијим фазама наведени аспекти требају ортогонално да се повежу са пројектованим решењима СО, како би се логика решења проблема независно од њих развијала.

Препорука ПР4: Концептуалне реализација се могу описати преко објектног псеудокода, дијаграма сарадње (Larman, 1998; Jacobson et al., 1999), секвенцних дијаграма[11] (Larman, 1998; Jacobson et al., 1999), дијаграма активности, дијаграма прелаза стања или дијаграма структура (Budgen, 2003).

За сваки од уговора пројектује се концептуално решење.

Пројектовање концептуалних решења СО

1. Уговор UG1: *KreirajNovi*

Операција: KreirajNovi (*OpstiDomenskiObjekat*):signal;
Веза са СК: СКП31
Предуслови: *Структурна и вредносна ограничење над доменским објектом морају бити задовољена.*
Постуслови: *Направљен је нови доменски објекат.*.

```
class KreirajNovi extends OpstaSO
{
 public static String kreirajNovi(OpstiDomenskiObjekat odo)
   { KreirajNovi kn = new KreirajNovi();
     OpstaSO.transakcija = true;
     return OpstaSO.opsteIzvrsenjeSO(odo,kn);
   }

 // Prekrivanje metode klase OpstaSO
 boolean izvrsenjeSO(OpstiDomenskiObjekat odo)
  {  if (!Preduslov(odo))   { return false;}
     if (!BBP.kreirajSlog(odo))
       { PorukaIzvrsenjeSO("Sistem ne moze da kreira " + odo.vratiNazivNovogObjekta() + ".");
                 return false;
       }
     PorukaIzvrsenjeSO("Sistem je kreirao " + odo.vratiNazivNovogObjekta() + ".");
     return true;
  }

 private boolean Preduslov(OpstiDomenskiObjekat odo)
   {  if (!odo.prostaVrednosnaOgranicenja())
            { PorukaIzvrsenjeSO("Sistem ne moze da zapamti " + odo.vratiNazivRacuna() + ".
                                Naruseno je prosto vrednosno ogranicenje.");return false; }
      if (BBP.strukturnaOgranicenja(odo))
            { PorukaIzvrsenjeSO("Sistem ne moze da zapamti " + odo.vratiNazivRacuna() + ".
                                Naruseno je strukturno ogranicenje."); return false;}
            }       return true;}}
```

[11] И секвенцни и дијаграми сарадње чувају исту семантику интеракције између објеката. Разлика између њих се огледа у начину њиховог представљања. Дијаграми сарадње описују интеракцију између објеката у графичком или мрежном формату, у коме објекти могу да се поставе било где на дијаграму.
Секвенцни дијаграм описује интеракцију у fence (ограда) формату, у коме се сваки следећи објекат додаје десно у односу на претходни објекат.
Наведени дијаграми имају одређене предности и недостатке:
• Секвенцни дијаграм:

- предности: јасно се приказује секвенца порука у времену, једноставно описивање.
- недостатак: када се додаје нови објекат, заузима се простор удесно.

• Дијаграм сарадње:

- предност: једноставније се додају нови објекти на дијаграму. Боље се илуструју комплексне гране, циклуси и конкурентно понашање.
- недостатак: тешкоће код посматрања секвенце акција. Сложенија нотација.

2. Уговор UG2: *Zapamti*

Операција: Zapamti(*OpstiDomenskiObjekat*):signal;
Веза са СК: *СК1, СК4*
Предуслови: *Структурна и вредносна ограничење над доменским објектом морају бити задовољена. Ако је доменски објекат обрађен или сторниран не може се извршити системска операција.*

Постуслови: *Запамћен је доменски објекат..*

```
class Zapamti extends OpstaSO
{
  public static String Zapamti(OpstiDomenskiObjekat odo)
   { Zapamti r = new Zapamti();
     OpstaSO.transakcija = true;
     return OpstaSO.opsteIzvrsenjeSO(odo,r);
   }

  // Prekrivanje metode klase OpstaSO
  boolean izvrsenjeSO(OpstiDomenskiObjekat odo)
   { if (!Preduslov(odo))
                {
                  return false;
                }

     if (!odo.slozenaVrednosnaOgranicenja())
                { PorukaIzvrsenjeSO("Sistem ne moze da zapamti " + odo.vratiNazivRacuna() + ".
                                                Naruseno je vrednosno ogranicenje.");
                  return false;
                }

      if (!BBP.brisiSlog(odo))
                { PorukaIzvrsenjeSO("Sistem ne moze da zapamti " + odo.vratiNazivRacuna() + ".");
                  return false;
                }

      if (!BBP.pamtiSlozeniSlog(odo))
                { PorukaIzvrsenjeSO("Sistem ne moze da zapamti " + odo.vratiNazivRacuna() + ".");
                   return false;
                 }

      PorukaIzvrsenjeSO("Sistem je zapamtio " + odo.vratiNazivRacuna() + ".");
      return true;
    }

  private boolean Preduslov(OpstiDomenskiObjekat odo)
    {  if (!odo.prostaVrednosnaOgranicenja())
                { PorukaIzvrsenjeSO("Sistem ne moze da zapamti " + odo.vratiNazivRacuna() + ".
                                        Naruseno je prosto vrednosno ogranicenje.");
                  return false;
                }

      if ((BBP.vratiLogickuVrednostAtributa(odo,"Obradjen") == true) ||
         (BBP.vratiLogickuVrednostAtributa(odo,"Storniran") == true))
           {  PorukaIzvrsenjeSO("Sistem ne moze da zapamti " + odo.vratiNazivRacuna() + "." +
                                    odo.vratiNazivRacuna() + " je vec obradjen ili storniran.");

             return false;
           }
       if (BBP.strukturnaOgranicenja(odo))
                { PorukaIzvrsenjeSO("Sistem ne moze da zapamti " + odo.vratiNazivRacuna() + ".
                                      Naruseno je strukturno ogranicenje.");
                  return false;
                }


      return true;
    }
}
```

3. Уговор UG3: *Obradi*

Операција: Obradi(*OpstiDomenskiObjekat*):signal;
Веза са СК: *СК1, СК4*
Предуслови: *Структурна и вредносна ограничење над доменским објектом морају бити задовољена. Ако је доменски објекат обрађен или сторниран не може се извршити системска операција.*
Постуслови: *Доменски објекат је обрађен.*

```
class Obradi extends OpstaSO
 {
   public static String Obradi(OpstiDomenskiObjekat odo)
    { Obradi ob = new Obradi();
      OpstaSO.transakcija = true;
      return OpstaSO.opsteIzvrsenjeSO(odo,ob);
    }


    // Prekrivanje metode klase OpstaSO
  boolean izvrsenjeSO(OpstiDomenskiObjekat odo)
   { if (!Preduslov(odo))
                {
                  return false;
                }

     if (!odo.slozenaVrednosnaOgranicenja())
                { PorukaIzvrsenjeSO("Sistem ne moze da obradi " + odo.vratiNazivRacuna() + ".
                                    Naruseno je vrednosno ogranicenje.");
                  return false;
                }

     odo.obradi();

      if (!BBP.brisiSlog(odo))
                { PorukaIzvrsenjeSO("Sistem ne moze da obradi " + odo.vratiNazivRacuna() + ".");
                  return false;
                }

       if (!BBP.pamtiSlozeniSlog(odo))
                { PorukaIzvrsenjeSO("Sistem ne moze da obradi " + odo.vratiNazivRacuna() + ".");
                   return false;
                  }

       PorukaIzvrsenjeSO("Sistem je obradio " + odo.vratiNazivRacuna() + ".");
       return true;
    }

  private boolean Preduslov(OpstiDomenskiObjekat odo)
    {  if (!odo.prostaVrednosnaOgranicenja())
                { PorukaIzvrsenjeSO("Sistem ne moze da obradi " + odo.vratiNazivRacuna() + ".
                                    Naruseno je prosto vrednosno ogranicenje.");
                  return false;
                }

       if ((BBP.vratiLogickuVrednostAtributa(odo,"Obradjen") == true) ||
          (BBP.vratiLogickuVrednostAtributa(odo,"Storniran") == true))
            {  PorukaIzvrsenjeSO("Sistem ne moze da obradi " + odo.vratiNazivRacuna() + "." +
                                 odo.vratiNazivRacuna() + " je vec obradjen ili storniran.");

              return false;
            }
        if (BBP.strukturnaOgranicenja(odo))
                { PorukaIzvrsenjeSO("Sistem ne moze da obradi " + odo.vratiNazivRacuna() + ".
                                     Naruseno je strukturno ogranicenje.");
                  return false;
                }


       return true;
    }
}
```

4. Уговор UG4: *Pretraži*

Операција: Pretraži (*OpstiDomenskiObjekat*):signal;
Веза са СК: *СК2, СК3, СК4*
Предуслови:
Постуслови: Пронађен је тражени доменски објекат.

```
class Pretrazi extends OpstaSO
{
  public static String Pretrazi(OpstiDomenskiObjekat odo)
   { Pretrazi p = new Pretrazi();
            OpstaSO.transakcija = false;
            return OpstaSO.opsteIzvrsenjeSO(odo,p);
   }

  // Prekrivanje metode klase OpstaSO
  boolean izvrsenjeSO(OpstiDomenskiObjekat odo)
    { signal = BBP.nadjiSlogiVratiGa(odo);
        if (!signal)
            { PorukaIzvrsenjeSO("*Sistem ne moze da nadje " + odo.vratiNazivRacuna() + " po
                                    zadatoj vrednosti.");
              return false;
            }
       PorukaIzvrsenjeSO("Sistem je nasao " + odo.vratiNazivRacuna() + " po zadatoj
                                    vrednosti.");
      return true;
   }
}
```

5. Уговор UG5: *Storniraj*

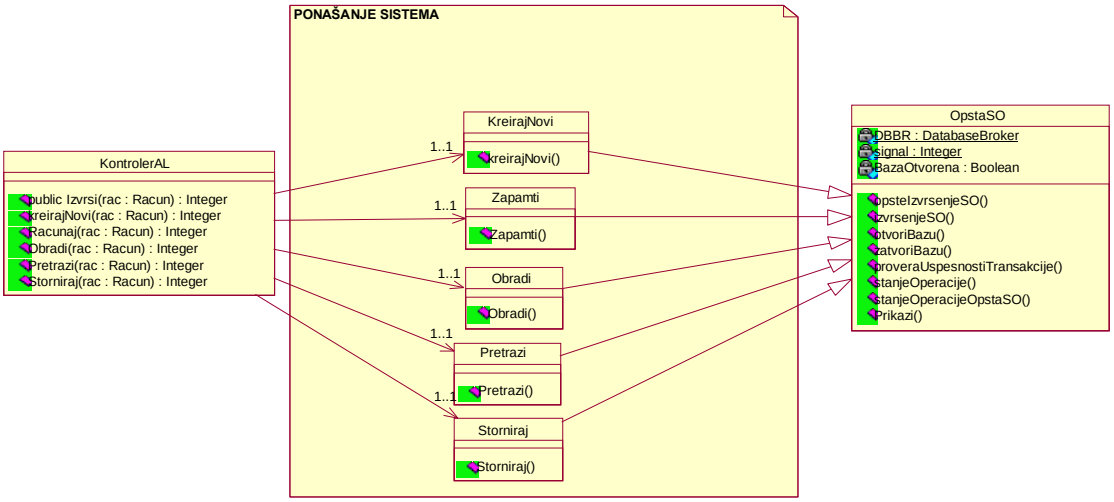
Операција: Storniraj(*OpstiDomenskiObjekat*):signal;
Веза са СК: *СК3*
Предуслови: *Структурна и вредносна ограничење над доменским објектом морају бити задовољена. Ако је доменски објекат сторниран не може се извршити системска операција.*
Постуслови: *Доменски објекат је сторниран.*

```
class Storniraj extends OpstaSO
 {
   public static String Storniraj(OpstiDomenskiObjekat odo)
    { Storniraj st = new Storniraj();
      OpstaSO.transakcija = true;
     return OpstaSO.opsteIzvrsenjeSO(odo,st);
    }


// Prekrivanje metode klase OpstaSO
  boolean izvrsenjeSO(OpstiDomenskiObjekat odo)
    { if (!Preduslov(odo))
                {
                  return false;
                }

      if (!odo.slozenaVrednosnaOgranicenja())
                { PorukaIzvrsenjeSO("Sistem ne moze da stornira " + odo.vratiNazivRacuna() + ".
                                    Naruseno je vrednosno ogranicenje.");
                  return false;
                }

      odo.storniraj();

      if (!BBP.brisiSlog(odo))
                { PorukaIzvrsenjeSO("Sistem ne moze da stornira " + odo.vratiNazivRacuna() + ".");
                  return false;
                }

       if (!BBP.pamtiSlozeniSlog(odo))
                { PorukaIzvrsenjeSO("Sistem ne moze da stornira " + odo.vratiNazivRacuna() + ".");
                    return false;
                  }

       PorukaIzvrsenjeSO("Sistem je stornirao " + odo.vratiNazivRacuna() + ".");
       return true;
    }

  private boolean Preduslov(OpstiDomenskiObjekat odo)
    {  if (!odo.prostaVrednosnaOgranicenja())
                { PorukaIzvrsenjeSO("Sistem ne moze da stornira " + odo.vratiNazivRacuna() + ".
                                    Naruseno je prosto vrednosno ogranicenje.");
                  return false;
                }

       if (BBP.vratiLogickuVrednostAtributa(odo,"Storniran") == true))
            {  PorukaIzvrsenjeSO("Sistem ne moze da stornira " + odo.vratiNazivRacuna() + "." +
                                 odo.vratiNazivRacuna() + " je vec storniran.");
              return false;
            }
        if (BBP.strukturnaOgranicenja(odo))
                { PorukaIzvrsenjeSO("Sistem ne moze da stornira " + odo.vratiNazivRacuna() + ".
                                    Naruseno je strukturno ogranicenje.");
                  return false;
                }
       return true;
    }
}
```

Након пројектовања сваке од СО прелази се на пројектовање класе која је одговорна за конекцију са базом и за контролу извршења трансакције. Метода која обезбеђује наведене захтеве се зове *opsteIzvrsenjeSO()*.
Свака од СО треба да наследи класу *OpstaSO* како би могла да се повеже са базом и како би се њено извршење пратило као трансакција:

```
class X extends OpstaSO
{
  public static String X(OpstiDomenskiObjekat odo)
    { X x = new X();
      OpstaSO.transakcija = true;
      return OpstaSO.opsteIzvrsenjeSO(odo,x);
    }
...
}
```

Класе које су одговорне за СО наслеђују класу OpstaSO (Слика OSO).



Слика OCO: Класе које су одговорне за извршење СО наслеђују класу OpstaSO

Наводимо класу OpstaSO:

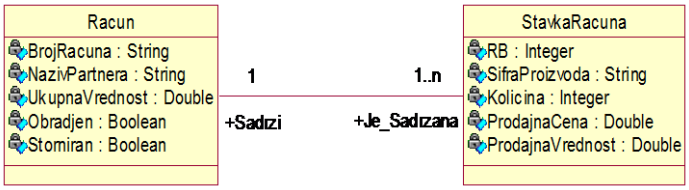
```
abstract class OpstaSO
{ static BrokerBazePodataka BBP;
  static boolean signal;
  static boolean BazaOtvorena = false;
  static boolean transakcija = false;
  String porukaUspesnostiSO;
 // Ova metoda je sinhronizovana kako bi se onemogućilo da 2 iil više klijenata u isto vreme izvršavaju
 // SO koja je pod transakcijom.

synchronized static String opsteIzvrsenjeSO(OpstiDomenskiObjekat rac, OpstaSO os)
   { if (!os.otvoriBazu()) return os.vratiPorukuMetode();

      if (!os.izvrsenjeSO(rac) && transakcija)
           { signal = os.rollbackTransakcije();
             return os.vratiPorukuMetode();
            }

     if (transakcija) os.commitTransakcije();
        return os.vratiPorukuMetode();
  }

   abstract boolean izvrsenjeSO(OpstiDomenskiObjekat rac);

   boolean otvoriBazu()
  { if (BazaOtvorena == false)
          { BBP = new BrokerBazePodataka();
            BBP.isprazniPoruku();
           signal = BBP.otvoriBazu("RACUN");
           if (!signal) return false;
          }
     BBP.isprazniPoruku();
     BazaOtvorena = true;
     return true;}
```

```
  boolean commitTransakcije()
   { return BBP.commitTransakcije();}

  boolean rollbackTransakcije()
   { return BBP.rollbackTransakcije();}

  String vratiPorukuMetode()
   { System.out.println(porukaUspesnostiSO);
     return porukaUspesnostiSO;
   }
}
```

2.3.2.2.2 Пројектовање структуре софтверског система

На основу концептуалних класа праве се софтверске класе структуре (Слика АССКС).

Концептуалне класе:



Софтверске класе структуре:

```
class Racun
{
 String BrojRacuna;
 String NazivPartnera;
 Double UkupnaVrednost;
 boolean Obradjen;
 boolean Storniran;
 StavkaRacuna []sracun;

Racun()
  { BrojRacuna = "";
    NazivPartnera = "";
    UkupnaVrednost = new Double(0);
    Obradjen = false;
    Storniran = false;
    sracun = null;
  }

}

class StavkaRacuna
{ Integer RB;
  String SifraProizvoda;
  Integer Kolicina;
  Double ProdajnaCena;
  Double ProdajnaVrednost;
  Racun rac;

   StavkaRacuna(Racun rac1)
    { RB = new Integer(0);
      SifraProizvoda = new String("");
      Kolicina = new Integer(0);
      ProdajnaCena = new Double(0);
      ProdajnaVrednost = new Double(0);
      rac = rac1;
    }
}
```

2.3.2.3 Брокер базе података

Пре пројектовања датабасе брокера наводи се његова дефиниција и дефиниција свих оних концепата који су потребни да би се јасно схватила комуникација између датабасе брокера и базе података.

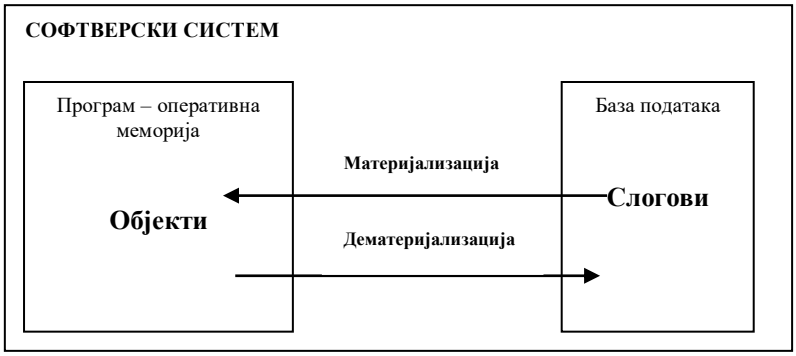
Дефиниција брокера базе података

Деф ПРПО1 (Booch et al., 2007) : Објекат је **перзистентан** уколико настави да постоји и након престанка рада програма који га је створио.

Деф ПРПО2: Објекат је **перзистентан** уколико се може **материјализовати** и **дематеријализовати** (Слика МД).

Деф ПРПО3: Материјализација представља процес трансформације слогова из базе података у објекте програма[12].

Деф ПРПО4: Дематеријализација (Пасивизација) представља процес трансформације објеката из програма у слогове базе података.



Слика МД: Материјализација и дематеријализација објекта

Деф ПРПО5: Перзистентни оквир је скуп интерфејса и класа који омогућава перзистентност објектима различитих класа (Перзистентни оквир омогућава **перзистентни сервис**[13] објектима). Он се може проширити са новим интерфејсима и класама.

Деф ПРПО6: Перзистентни оквири су засновани на **Холивудском принципу**: *"Don't call us, we'll call you"*. То значи да кориснички дефинисане класе прихватају поруке од предефинисаних класа оквира.

Деф ПРПО7: Уколико у оквиру неке **трансакције**, операције мењају стање перзистентних објеката, оне не чувају то стање одмах у бази података. Уколико се жели запамтити стање које је настало као резултат извршених операција над перзистентним објектима позива се **commit операција**. Уколико се не жели запамтити стање које је настало као резултат извршених операција над перзистентним објектима позива се **roolback операција**. Commit и rollback операције представљају **трансакционе операције**.

Пример брокера базе података

У нашем примеру ми смо пројектовали перзистентни оквир (класа BrokerBazePodataka[14]) који ће да реализује следеће методе:

1. ***int otvoriBazu(String imeBaze)***
2. ***int commitTransakcije()***

[12] Термине материјализација и дематеријализација смо објаснили у ужем смислу као процесе који трансформишу објекте програма у слогове база података. У ширем смислу би се подразумевало да објекти могу бити сачувани у било ком перзистентном складишту података.

[13] Уколико **перзистентни сервис** омогућава памћење објеката у релационој бази података за њега се каже да је он **Object-Relation сервис пресликавања (mapping service).**

[14] Брокер базе података (Larman, 1998) патерн представља једну могућу реализацију перзистентног оквира. Брокер базе података (Larman, 1998) патерн је одговоран за материјализацију, дематеријализацију и кеширање објеката у меморији. Он се често назива и **Database Mapper** патерн.

3. ***boolean rollbackTransakcije()***
4. ***boolean pamtiSlog(OpstiDomenskiObjekat)***
5. ***boolean brisiSlog(OpstiDomenskiObjekat)***
6. ***boolean promeniSlog(OpstiDomenskiObjekat)***
7. ***boolean daLiPostojiSlog(OpstiDomenskiObjekat odo)***
8. ***boolean kreirajSlog(OpstiDomenskiObjekat)***
7. ***boolean nadjiSlogiVratiGa(Objekat,Objekat)***
8. ***boolean vratiLogickuVrednostAtributa(OpstiDomenskiObjekat odo, String nazivAtributa)***
9. ***boolean pamtiSlozeniSlog(OpstiDomenskiObjekat odo)***

Dajemo detaljno objašnjenje navedenih metoda.

```
class DatabaseBroker
{
  static Connection con;
  static Statement st;
```

/*Уговор DB1: **otvoriBazu**(String imeBaze) : boolean
Постуслов: Успостављена је веза (конекција) са базом података. Уколико је успешно остварена веза са базом података метода враћа вредност true и памти поруку: „*Uspostavljena je konekcija sa bazom podataka”*, иначе метода враћа false и памти поруке у зависности од следећих ситуација:
а) ако драјвер није учитан метода памти поруку: „Drajver nije učitan”
b) ако се десила грешка код конекције метода памти поруку: „Greška kod konekcije”
c) ако се десила грешка код заштите базе података. метода памти поруку: „Greška zaštite” */

```
public boolean otvoriBazu(String imeBaze)
  {  String Urlbaze;
     try {  Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
            Urlbaze = "jdbc:odbc:" + imeBaze;
            con = DriverManager.getConnection(Urlbaze);
            con.setAutoCommit(false); // Ako se ovo ne uradi nece moci da se radi roolback.
          } catch(ClassNotFoundException e)
                { porukaMetode = "Drajver nije ucitan:" + e; return false;}
            catch(SQLException esql)
                { porukaMetode = "Greska kod konekcije:" + esql; return false;}
            catch(SecurityException ese)
                { porukaMetode = "Greska zastite:" + ese; return false;}
       porukaMetode = "Uspostavljena je konekcija sa bazom podataka."; return true;
  }
```

/*Уговор DB2: **commitTransakcije**() : boolean
Постуслов: Све операције које су мењале стање базе, од задњег позива commit или rollback transakcije, су успешно извршене (промене су успешно запамћене у бази). У том случају метода враћа true и памти поруку: „Uspešno urađen commit transakcije“, иначе враћа false и памти поруку: “Nije uspešno urađen commit transakcije”. */

```
 public boolean commitTransakcije()
  {  try { con.commit();
         } catch(SQLException esql)
                { porukaMetode = porukaMetode + "\nNije uspesno uradjen commit transakcije " + esql;
                  return false;
                }
    porukaMetode = porukaMetode + "\nUspesno uradjen commit transakcije ";
    return true;
}
```

/*Уговор DB3: **rollbackTransakcije**() : boolean
Постуслов: Ефекти свих операција које су мењале стање базе, од задњег poziva commit или rollback трансакције, су поништени (промене нису запамћене у бази). У том случају метода враћа true и поруку: “Uspesno uradjen rollback transakcije”, иначе враћа false и поруку: „Nije uspešno uradjen rollback transakcije“. */

```
public boolean rollbackTransakcije()
  {  try{ con.rollback();
        } catch(SQLException esql)
              { porukaMetode = porukaMetode + "\nNije uspesno uradjen rollback transakcije" + esql;
                return false;
              }
    porukaMetode = porukaMetode + "\nUspesno uradjen rollback transakcije";
    return true;
  }
```

/*Уговор DB4: **pamtiSlog(OpstiDomenskiObjekat)**: boolean
Постуслов: Извршено је памћење домснског објекта у базу података (материјализација). Уколико је метода успешно извршена враћа труе и памти поруку: "Uspeštno zapamćen slog u bazi", иначе враћа false и памти поруку: „Nije uspešno zapamćen slog u bazi".
Напомена: Наведена метода је генеричка јер омогуцава памћење објеката било које класе у бази, уколико те класе наслеђује класу *OpstiDomenskiObjekat.*

```
public boolean pamtiSlog(OpstiDomenskiObjekat odo)
   {   String upit;
       try{ st = con.createStatement();
            upit = "INSERT INTO " + odo.vratiImeKlase()  +
                    " VALUES (" + odo.vratiVrednostiAtributa() + ")";
           st.executeUpdate(upit);
           st.close();
         } catch(SQLException esql)
              { porukaMetode = porukaMetode + "\nNije uspesno zapamcen slog u bazi. " + esql;
                return false;
               }
        porukaMetode = porukaMetode + "\nUspesno zapamcen slog u bazi. ";
       return true;
   }
```

/**
ОБЈАШЊЕЊЕ ПОСТУПКА ПРОЈЕКТОВАЊА ГЕНЕРИЧКЕ МЕТОДЕ
Уколико би метода била специфично везана за конкретну доменску класу нпр. Račun она би имала следећи изглед (болдовали смо зависне делове методе од доменске класе) :

```
public boolean pamtiSlog(Racun rac)
   {   String upit;
       try{ st = con.createStatement();
            upit = "INSERT INTO " + rac.vratiImeKlaseRacun()  +
                    " VALUES (" + rac.vratiVrednostiAtributaRacun() + ")";
            st.executeUpdate(upit);
            st.close();
         } catch(SQLException esql)
              { porukaMetode = porukaMetode + "\nNije uspesno zapamcen slog u bazi. " + esql;
                return false;
               }
            porukaMetode = porukaMetode + "\nUspesno zapamcen slog u bazi. ";
           return true;
   }

class Racun
{ ...
public String vratiVrednostiAtributaRacuna()
  { return """+ BrojRacuna + "', '" + NazivPartnera + "', " +  UkupnaVrednost.doubleValue() + ", " + Obradjen + ", " +
    Storniran;}
public String vratiImeKlaseRacun() { return "Racun";}
}
```

Уколико би користили наведени приступ морали би за сваку доменску класу да имплементирамо методу *pamtiSLog(),* као и све остале методе које ћемо ниже навести (*brisiSlog(),promeniSlog(),...)* у класи BrokerBazePodataka. То није практично јер би број операција BrokerBazePodataka растао са појавом нових доменских класа. Због тога смо сваку од ниже наведених метода пројектовали као генеричку методу. На примеру методе *pamtiSlog()* објаснићемо општи поступак за пројектовање било које од ниже наведених генеричких метода.

Поступак пројектовања генеричке методе:
1. Одредити специфичне класе *(Racun)* у методи (*PamtiSlog*) која треба да постане генерална. Именовати специфичне класе у општем смислу (нпр. класу Račun са класом OpstiDomenskiObjekat). Такође методе специфичних класа именовати у општем смислу, ако је њихов назив повезан са нечим што је специфично за класу којој оне припадају (нпр. назив методе *vratiImeKlaseRacun()* са називом *VratiImeKlase()).*

```
public boolean pamtiSlog(OpstiDomenskiObjekat odo)
   {   String upit;
       try{ st = con.createStatement();
            upit = "INSERT INTO " + odo.vratiImeKlase()  +
                    " VALUES (" + odo.vratiVrednostiAtributa() + ")";
```

```
            st.executeUpdate(upit);
            st.close();
        } catch(SQLException esql)
              { porukaMetode = porukaMetode + "\nNije uspesno zapamcen slog u bazi. " + esql;
                return false;
              }
     porukaMetode = porukaMetode + "\nUspesno zapamcen slog u bazi. ";
    return true;
  }
```

2. Направити генералну класу (апстрактну класу или интерфејс) за специфичне класе.

```
interface OpstiDomenskiObjekat
{ vratiVrednostiAtributa();
  vratiImeKlase(); }
```

Из наведеног може да се закључи да метода *pamtiSlog()*, може да прихвати различите доменске објекте преко параметра, ако доменски објекти наследе класу *OpstiDomenskiObjekat* и имплементирају њене методе *vratiVrednostiAtributa() i vratiImeKlase().*

У том смислу доменска класа Рацун и СтавкаРачуна ће добити следећи изглед:

```
class Racun implements OpstiDomenskiObjekat
{ ...
public String vratiVrednostiAtributa()
  { return """+ BrojRacuna + "', '" + NazivPartnera + "', " +  UkupnaVrednost.doubleValue() + ", " + Obradjen + ", " +
    Storniran;}
public String vratiImeKlase()
  { return "Racun";}
}

class StavkaRacuna implements OpstiDomenskiObjekat
{
 public String vratiVrednostiAtributa()
  { return "'" + rac.BrojRacuna + "' , " + RB.intValue() + ", '" + SifraProizvoda + "', " +  Kolicina.intValue() + ", " +
ProdajnaCena.doubleValue() + ", " + ProdajnaVrednost.doubleValue();}

public String vratiImeKlase() { return "StavkaRacuna";}
}
```

Код доменских класа смо болдовали све делове програма који су промењени како би они могли да буду прихваћени као параметар методе *pamtiSlog().*

КРАЈ ОБЈАШЊЕЊА ПОСТУПКА ПРОЈЕКТОВАЊА ГЕНЕРИЧКЕ МЕТОДЕ

```
/*****************************************************************************************
```

/*Уговор DB5: **brisiSlog**(OpstiDomenskiObjekat odo) : boolean
Постуслов: Обрисан је слог у бази података (такав објекат се не може више материјализовати). Уколико је метода успешно извршена враћа true и поруку: „Успешно obrisan slog u bazi.“, иначе враћа false и памти поруку: “Nije uspešno obrisan slog u bazi.”.
Напомена: Наведена метода је генерицка јер омогуцава да се преко ње мозе обрисати објекат било које класе која наслеђује класу *OpstiDomenskiObjekat.*
/*

```
public boolean brisiSlog(OpstiDomenskiObjekat odo)
    {  String upit;
       try {  st = con.createStatement();
              upit ="DELETE *  FROM " + odo.vratiImeKlase() + " WHERE " + odo.vratiUslovZaNadjiSlog();
              st.executeUpdate(upit);
              st.close();
          } catch(SQLException esql)
              { porukaMetode = porukaMetode + "\nNije uspesno obrisan slog u bazi: " + esql;
                return false;
              }
       porukaMetode = porukaMetode + "\nUspesno obrisan slog u bazi.";
       return true;
    }
```

/*Уговор DB6: **promeniSlog**(OpstiDomenskiObjekat odo) : boolean
Постуслов: Промењен је слог у бази података по задатом услову. Уколико је метода успешно извршена враћа true и памти поруку: "Uspešno promenjen slog u bazi podataka", иначе враћа false и памти поруку: "Nije uspešno promenjen slog u bazi podataka".
Напомена: Наведена метода је генерицка јер омогуцава промену објекта било које класе која наслеђује класу *OpstiDomenskiObjekat*. /*

```
public boolean promeniSlog(OpstiDomenskiObjekat odo)
   {  String upit;
      try {  st = con.createStatement();
             upit =  "UPDATE " + odo.vratiImeKlase() +
                     " SET " + odo.postaviVrednostiAtributa() +
                     " WHERE " + odo.vratiUslovZaNadjiSlog();
             System.out.println("PROMENI SLOG" + upit);
             st.executeUpdate(upit);
             st.close();
           } catch(SQLException esql)
             { porukaMetode = porukaMetode + "\nNije uspesno promenjen slog u bazi podataka: " +
                                 esql;
               return false;
             }
        porukaMetode = porukaMetode + "\nUspesno promenjen slog u bazi podataka: ";
        return true;
    }
```

/*Уговор DB7: **daLiPostojiSlog**(OpstiDomenskiObjekat odo) : boolean
Постуслов: Уколико постоји слог у бази података метода враћа true и памти поруку: "Slog postoji u bazi podataka." иначе враћа false и памти поруку: „Slog ne postoji u bazi podataka." или "Nije uspesno pretrazena baza."
Напомена: Наведена метода је генерицка јер омогућава претраживање објекта било које класе која наслеђује класу *OpstiDomenskiObjekat*. */

```
public boolean daLiPostojiSlog(OpstiDomenskiObjekat odo)
   {  String upit;
      ResultSet RSslogovi;
      try { st = con.createStatement();
            upit =  "SELECT *" +
                    " FROM " + odo.vratiImeKlase() +
                    " WHERE " + odo.vratiUslovZaNadjiSlog();
            RSslogovi = st.executeQuery(upit);
            boolean signal = RSslogovi.next();
            RSslogovi.close();
            st.close();
            if (signal == false)
             { porukaMetode = porukaMetode + "\nSlog ne postoji u bazi podataka.";
               return false; // Slog ne postoji u bazi.
             }
          } catch(SQLException esql)
              { porukaMetode = porukaMetode + "\nNije uspesno pretrazena baza: " + esql;
                return false;
              }
       porukaMetode = porukaMetode + "\nSlog postoji u bazi.";
       return true;
     }
```

/*Уговор DB8: **kreirajSlog**(OpstiDomenskiObjekat odo): boolean
Постуслов: Креира нови слог у бази података метода. Уколико је метода успешно враћа true и памти поруку: "Kreiran je novi slog." иначе враћа false и памти поруку: „Ne moze da se kreira novi slog."
Напомена: Наведена метода је генеричка јер омогућава креирање објекта било које класе која наслеђује класу *OpstiDomenskiObjekat*. */

```
public boolean kreirajSlog(OpstiDomenskiObjekat odo)
    {  String upit;
       ResultSet rs;
       upit = "SELECT Max(" + odo.vratiAtributPretrazivanja()+ ") AS Max" +
              " FROM " + odo.vratiImeKlase();
```

```
      try { st = con.createStatement();
            rs = st.executeQuery(upit);


            if (rs.next() == false)
               odo.postaviPocetniBroj();
            else
               odo.povecajBroj(rs);

            upit = "INSERT INTO " + odo.vratiImeKlase()  +
                   " VALUES (" + odo.vratiVrednostiAtributa() + ")";
            st.executeUpdate(upit);
            st.close();
            } catch(SQLException esql)
              {  porukaMetode = porukaMetode + "\nNe moze da se kreira novi slog: " + esql;
                 return false;
              }
          porukaMetode = porukaMetode + "\nKreiran je novi slog: ";
          return true;
    }
```

/*Уговор DB9: **nadjiSlogiVratiGa**(OpstiDomenskiObjekat odo, OpstiDomenskiObjekat odo1) : integer
/*

```
public boolean nadjiSlogiVratiGa(OpstiDomenskiObjekat odo)
      {  ResultSet RS;
         String nazivVezanogObjekta;
         int brojStavki;
        String upit;
        Statement st;
         try {
                st= con.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,ResultSet.CONCUR_READ_ONLY);
                upit = "SELECT *" +   " FROM " + odo.vratiImeKlase() +
                       " WHERE " + odo.vratiUslovZaNadjiSlog();
               RS = st.executeQuery(upit);
               System.out.println("Upit-nadji: " + upit);
               boolean signal = RS.next();
               if (signal == false)
                  { porukaMetode = porukaMetode + "\nNe postoji slog u bazi podataka.";
                    return false;
                  }
               porukaMetode = porukaMetode + "\nUspesno je procitan slog iz baze podataka.";

               if (odo.Napuni(RS))  // if1
                   { for (int j=0;j<odo.vratiBrojVezanihObjekata();j++)
                        { OpstiDomenskiObjekat vezo = odo.vratiVezaniObjekat(j);
                          if (vezo == null)
                              { porukaMetode = porukaMetode + "\nNe postoji vezani objekat a navedeno je
                                  da postoji.";
                                return false;
                              }
                          else  // else1
                             { upit = "SELECT COUNT(*) as brojStavki" +   " FROM " + vezo.vratiImeKlase() +
                                      " WHERE " + vezo.vratiUslovZaNadjiSlogove();
                               RS = st.executeQuery(upit);
                               if (RS.next() == false)
                                   { porukaMetode = porukaMetode + "\nNe postoje slogovi vezanog objekta";
                                     return true;
                                   }
                               brojStavki = RS.getInt("brojStavki");
                               odo.kreirajVezaniObjekat(brojStavki,j);
                               upit = "SELECT *" +  " FROM " + vezo.vratiImeKlase() +
                                      " WHERE " + vezo.vratiUslovZaNadjiSlogove();
                               RS = st.executeQuery(upit);
                               int brojSloga = 0;
                               while(RS.next())
                                     { odo.Napuni(RS,brojSloga,j);
```

```
                        brojSloga ++;
                      }
                porukaMetode = porukaMetode + "\nUspesno su procitani slogovi vezanog objekta";
                  } // end else1
                } // end for
            } // end if1
        RS.close();
        st.close();
           } catch(Exception e)
               { porukaMetode = porukaMetode + "\nGreska kod citanja sloga iz baze podataka." + e;
                 return false;
               }
        return true;
    }/*
```

/*Уговор DB10: **vratiLogickuVrednostAtributa**(OpstiDomenskiObjekat odo, String nazivAtributa) : boolean
/*

```
public  boolean vratiLogickuVrednostAtributa(OpstiDomenskiObjekat odo, String nazivAtributa)
  { String upit;
    ResultSet rs;
     boolean s = false;
     try { st = con.createStatement();
          upit = " SELECT *" +
                 " FROM " + odo.vratiImeKlase() +
                 " WHERE " + odo.vratiUslovZaNadjiSlog();
          System.out.println("upit: " + upit);
          rs = st.executeQuery(upit);
          rs.next();
          s = KonverterTipova.Konvertuj (rs, s, nazivAtributa);
          rs.close();
          st.close();
         } catch(Exception e)
           {  porukaMetode = porukaMetode + "\nGreska kod citanja logicke vrednosti atributa sloga." + e;
              return false;
           }
      return s;
     }
```

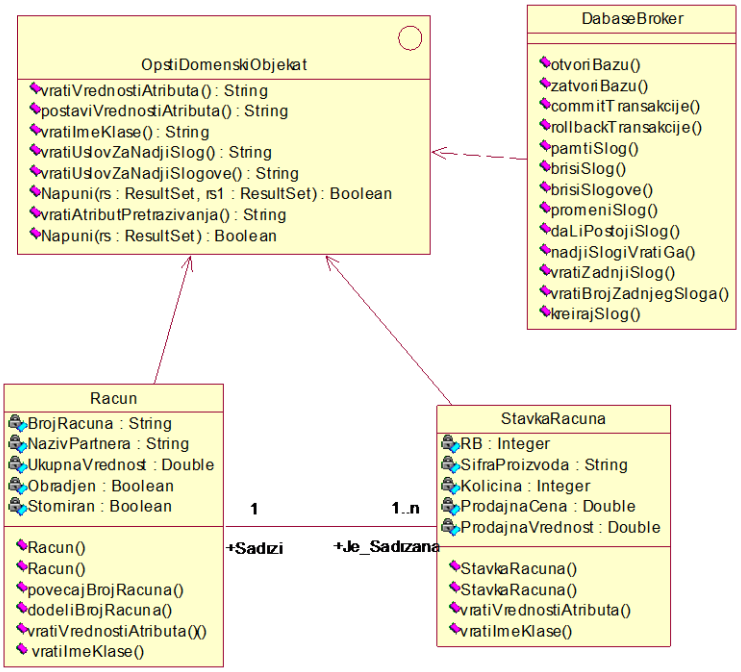
Ugovor DB11: **pamtiSlozeniSlog(OpstiDomenskiObjekat odo):** boolean

```
  public boolean pamtiSlozeniSlog(OpstiDomenskiObjekat odo)
    { String upit;
      try { st = con.createStatement();
            upit = " INSERT INTO " + odo.vratiImeKlase()  +
                   " VALUES (" + odo.vratiVrednostiAtributa() + ")";
            st.executeUpdate(upit);
            for(int j=0;j<odo.vratiBrojVezanihObjekata();j++)
                { OpstiDomenskiObjekat vezo;
                  for(int i=0; i < odo.vratiBrojSlogovaVezanogObjekta(j);i++)
                      { vezo = odo.vratiSlogVezanogObjekta(j,i);
                        upit = " INSERT INTO " + vezo.vratiImeKlase()  +
                               " VALUES (" + vezo.vratiVrednostiAtributa() + ")";
                        st.executeUpdate(upit);
                      }
                }

            st.close();
           } catch(SQLException esql)
             {  porukaMetode = porukaMetode + "\nNije zapamcen slozeni slog: " + esql;
                return false;
             }
      porukaMetode = porukaMetode + "\nZapamcen je slozeni slog.";
      return true;
    }
```

U procesu pravljenja generičkih metoda DatabaseBroker klase dobili smo metode interfejsa OpstiDomenskiObjekat:
// Operacije navedenog interfejsa je potrebno da implementira svaka od domenskih klasa,
// koja zeli da joj bude omogucena komunikacija sa Database broker klasom.

```
interface OpstiDomenskiObjekat
{ String vratiVrednostiAtributa();
 String postaviVrednostiAtributa();
 String vratiImeKlase();
 String vratiUslovZaNadjiSlog();
 String vratiUslovZaNadjiSlogove();
 boolean Napuni(ResultSet rs, ResultSet rs1);
 String vratiAtributPretrazivanja();
 boolean Napuni(ResultSet rs);
}
```

Kao rezultat projektovanja klase DatabaseBrokera i interfejsa OpstiDomenskiObjekat dobijaki se sledeći dijagrami klasa (Slika DBBR, ASSDBBR)



Слика DBBR: Database broker класа се повезује са класом OpstiDomenskiObjekat

2.3.3 Пројектовање складишта података

На основу софтверских класа структуре пројектовали смо табеле (складишта података) релационог система за управљање базом података (Слика АССТБП).

Table: Racun

Columns

Name	Type	Size
BrojRacuna	Text	10
NazivPartnera	Text	50
UkupnaVrednost	Number (Double)	8
Obradjen	Yes/No	1
Storniran	Yes/No	1

Table Indexes

Name	Number of Fields
PrimaryKey	1
Fields:	BrojRacuna, Ascending

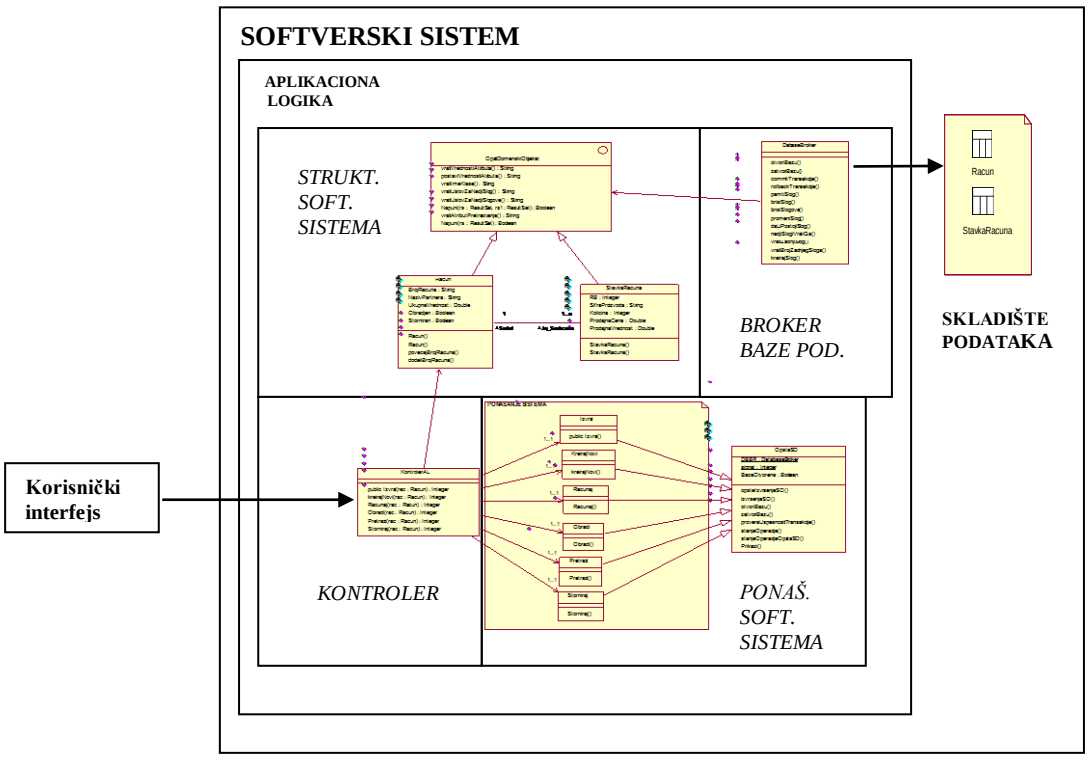
Table: StavkaRacuna

Columns

Name	Type	Size
BrojRacuna	Text	10
RB	Number (Integer)	2
SifraProizvoda	Text	20
Kolicina	Number (Integer)	2
ProdajnaCena	Number (Double)	8
ProdajnaVrednost	Number (Double)	8

Table Indexes

Name	Number of Fields
PrimaryKey	**2**
Fields:	BrojRacuna, Ascending RB, Ascending



Слика АССТБП: Архитектура софт. система након пројектовања табела базе података

2.3.4. Принципи, методе и стратегије пројектовања софтвера

2.3.4.1 Принципи (технике) пројектовања софтвера

2.3.4.1.1: Апстракција

Апстракција је "***процес*** *свесног заборављања информација тако да ствари које су различите могу бити третиране као да су исте*"[Lis01]. Под "*заборављањем информације*" се подразумева **издвајање** општих од специфичних информација . Свесно се **издвајају** опште информације, како би се нагласила суштина неке појаве, док се детаљи о самој појави избегавају ("заборављају").

Наводимо неколико објашњења абстракције:

- То је поглед на проблем где се **екстракују (издвајају)** суштинске информације важне за неку појаву док се остатак информације игнорише [IEEE, 1983].
- Суштина апстракције је **издвајање** суштинских својстава при чему се избегавају неважни детаљи [Ross et al, 1975].
- Апстракција је **процес** где се **идентификују** (**издвајају**) важни аспекти феномена а игноришу се његови детаљи [Ghezzi et al, 1991].
- Апстакција је генерално дефинисана као **процес** формулисања генерализованих концепата **екстраковањем** (**издвајањем**) заједничких квалитета из специфичних примера [Blair et al, 1991].
- Апстракција је селективно испитивање извесних аспеката проблема. Циљ апстракције је **изоловање** (**издвајање**) оних аспеката који су важни за неку појаву и потискивање оних аспеката који су неважни [Rumbaugh et al, 1991].
- Значење апстракције је дато преко Окфордског енглеског речника (Oxford English Dictionary - OED): *То је чин* ***одвајања*** *у размишљању.* Боља дефиниција може бити: ***Репрезентовање*** *(****Издвајање****) суштинских особина нечега без укључивања позадине и несуштинских детаља*[Graham, 1991].
- Апстракција **означава** (**издваја**) суштинске карактеристике објекта које га разликују од свих других врста објеката. На овај начин се обезбеђују његове концептуалне границе, које су релативне у односу на перспективу посматрача [Booch, 1991].
- У психологији апстракција је мисаони **процес** где се идеје **одвајају** од објекта [WikiAbs].
- У филозофији апстракција је **процес** обликовања концепта (concept-formation). То је процес **издвајања** заједничких особина неког скупа индивидуа [WikiAbs].

Дефиниција апстракције:

Из наведених објашњења апстракције изводимо следећу дефиницију апстракције[15]:

Апстракција је ***процес*** *који* ***издваја***[16] *из неке појаве* ***нешто***[17] *што је суштинско и општије од* ***нечега***[18] *што је детаљно и конкретније.*

---

[15] Овако добијена дефиниција апстракције је добијена коришћењем апстракције, где су из скупа различитих објашњења апстракције издвојена заједничка својства свих различитих објашњења - ***процес*** *који* ***издваја нешто*** *од* ***нечега***. Пошто се у свим објашњењима апстракције *издваја нешто од нечега* може се рећи да је то **процес издвајања,** без обзира што у неким објашњењима апстракције појам **процес** није непосредно наведен. Појам **издвајање** је**,** кроз различите облике, дат у свим објашњењима апстракције.

[16] **екстракује, идентификује, изолује, одваја, означава**

[17] **суштинске информације, суштинска својства,важни аспекти феномена, заједнички квалитет, важни аспекти, суштинске карактеристике, идеја, заједничке особине**

[18] не-суштинске информације, неважни детаљи, детаљи феномена, специфични примери, несуштински детаљи, не-суштинске карактеристике објекта, објекат, скуп индивидуа

APST1.1: Механизми апстракције

У контексту софтверског пројектовања, постоје два кључна механизма апстракције:
a) параметризација и
b) спецификација.

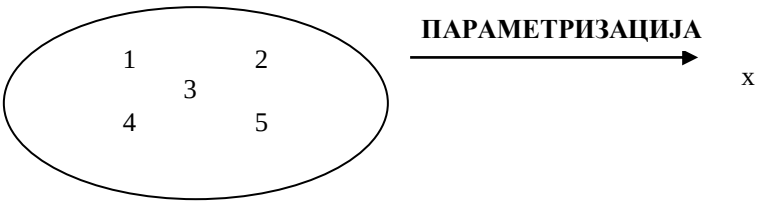
Апстракција спецификацијом води до три главне врсте апстракција:
a) процедурална апстракција
b) апстракција података
ц) апстракција контролом

PAR1: Параметризација

Параметризација је апстракција *која* ***издваја*** *из неког скупа елемената њихова општа својства која су представљена преко параметара.*
Размотрићемо параметризацију у пет случаја:
a) параметризација скупа елемената простог типа[19]
b) параметризација скупа елемената сложеног типа[20]
c) параметризација скупа операција
d) параметризација скупа процедура
e) параметризација скупа наредби

ПАР1-1: Параметризација скупа елемената простог типа

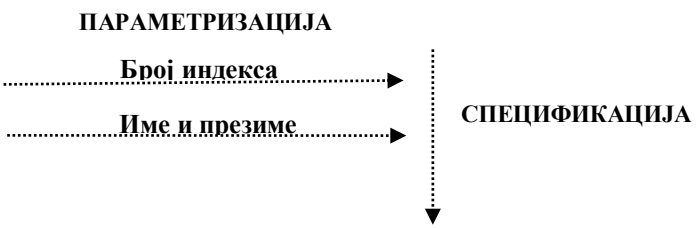
Уколико имамо нпр. скуп целих бројева *1,2,3,...*, они се могу представити нпр. преко параметра *x,* који је тада општи представник свих целих бројева.
Програмски то се може представити:
int x;
То значи да је декларисана променљива x која је целобројног типа. Декларисањем променљиве простог типа имплицитно се указује на скуп операција које се могу извршити над том променљивом.



ПАР1-2: Параметризација скупа елемената сложеног типа

Уколико имамо нпр. скуп студената ((123-00, Пера Перић),(124-00,Милош Савић),(125-00,Дејан Милић)), тада се параметризацијом добијају њихова општа својства (атрибути): *Број индекса* и *Име и презиме.* Наведени скуп студената је означен са *st1, st2, st3.*
Параметризација се ради за сваки скуп вредности атрибута. Скуп индекса студента(123-00, 124-00,125-00) се параметризује преко својства *Број индекса.* Док се скуп имена и презимена студената (Пера Перић, Милош Савић,Дејан Милић) параметризују преко својства *Име и презиме.*

st1	st2	st3
124-00	125-00	126-00
Пера Перић	Милош Савић	Дејан Милић



[19] Уколико су елементи недељиви.
[20] Уколико се елементи састоје од других елемената.

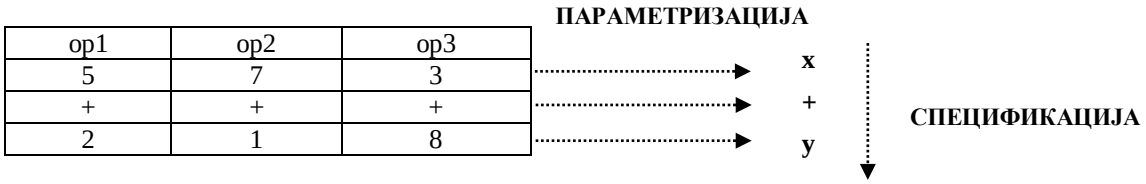
Параметризацијом долазимо до општих својства елемената скупа. Навођење општих својстава елемената скупа представља спецификацију скупа.

Закључак 1: *Параметризацијом се* ***добијају*** *општа својства неког скупа елемената. Спецификацијом се* ***набрајају (наводе)*** *општа својства неког скупа елемената.*

Закључак 2: *Параметризација предходи спецификацији, јер спецификација користи резултате параметризације.*

Апстракција података је дефинисана именом и спецификацијом скупа: *Студент (Број индекса, Име и презиме)*

ПАР1-3: Параметризација скупа операција
Уколико имамо нпр. скуп неких конкретних операција ((5+2),(7+1),(3+8)) тада се параметризацијом добијају општа својства наведених операција, односно параметри: *x, +,* и *y*.
Елементи неке операције су операнди *x* и *y* и оператор +. Оператор + указује на то **шта** операција ради, док операнди *x* и *y* указују **над чим** се изводи операција.
Наведени скуп конкретних операција је означен са *op1, op2, op3.*



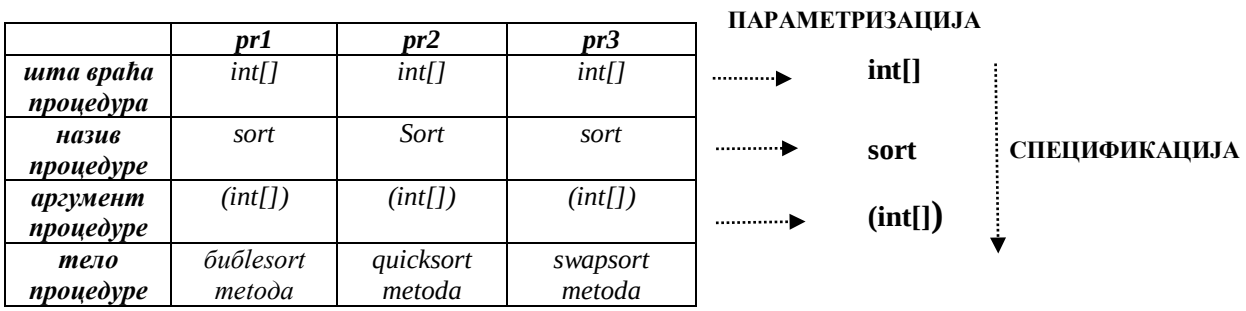
Навођење општих својстава операције представља спецификацију операције. Спецификација операције набраја параметре (елементе) операције (операнде и оператор). Спецификацијом операције, за наведени пример, се добија општа операција за сабирање два броја.

У општем слушају, спецификацијом операције се добија општа операција из неког скупа операција.

ПАР1-4: Параметризација скупа процедура
Уколико имамо нпр. скуп процедура:

1. *int[] sort(int[] niz){bublesort metoda}*
2. *int[] sort(int[] niz){quicksort metoda}*
3. *int[] sort(int[] niz){swapsort metoda}*

које сортирају низ бројева, тада се параметризацијом добијају општа својства наведених процедура, односно параметри:
int[], sort и *(int[] niz).*
Елементи неке процедуре су: *шта процедура враћа, назив процедуре, аргументи (параметри) процедуре и тело процедуре.* Наведени скуп процедура је означен са *pr1,pr2,pr3.*



Навођење општих својстава процедуре представља спецификацију процедуре. Спецификацијом процедуре се долази се до потписа процедуре. Као резултат спецификације процедуре, за наведени пример, се добија следећи потпис процедуре:
int[] sort(int[])

У општем случају, спецификацијом процедуре се добија општа процедура из неког скупа процедура. Потпис процедуре преставља **процедуралну апстракцију** неког скупа процедура.

ПАР1-5: Параметризација скупа наредби
Уколико имамо, нпр. скуп наредби:

System.out.println(1 + ". element skupa je " + x[0]);
System.out.println(2 + ". element skupa je " + x[1]);
System.out.println(3 + ". element skupa je " + x[2]);

које приказују елементе низа, тада се параметризацијом добијају општа својства наведених наредби, односно параметри
а) System.out.println(
б) i
ц) + ". element skupa: je " +
д) x[i]
е));

Наведени скуп наредби је означен са *na1,na2,n3.*

ПАРАМЕТРИЗАЦИЈА

na1	***na2***	***na3***		
System.out.println(	*System.out.println(*	*System.out.println(*	⋯⋯▶	***System.out.println(***
1	*2*	*3*	⋯⋯▶	***i***
+" element skupa je " +	*+" element skupa je " +*	*+" element skupa je " +*	⋯⋯▶	***+" element skupa je " +***
x[0]	*x[1]*	*x[2]*	⋯⋯▶	***x[i]***
);	*);*	*);*	⋯⋯▶	***);***

Навођење општих својстава наредбе представља спецификацију наре
спецификације наредбе добија се:

For (int i=0;i<3;i++) *// три пута понавља наредбу*
System.out.println(i + " element skupa: je " + x[i]);

У општем случају, спецификацијом наредби се добија општа наредба из неког скупа наредби.
Спецификација наредби представља **апстракацију наредби,** један од облика **апстракције контролом**.

СПЕ1: Спецификација
Спецификација је апстракција *која **издваја** из неког скупа елемената*[21] *њихова општа својства која могу бити представљена преко процедуре, података или контроле.*

СПЕ1-1: Процедурална апстракција
Процедуралном апстракцијом се издваја из неког скупа процедура оно што су њихова општа својства:

a) *тип онога што враћа процедура*
b) *име процедуре*
c) *аргументи процедуре*

Процедуралном апстракцијом се добија **потпис** процедуре[22]. Поред потписа процедурална апстракција може да обухвати и допунске секције које детаљно описују **шта** ради процедура и

[21] Елементи скупа могу бити или процедуре или подаци или контроле.

који су **услови** извршења процедуре. Општи случај процедуралне апстракције има следеће елементе, сходно упрошћеној Лармановој методи: **Потпис процедуре, Веза са СК, Предуслови и Подстуслови**

У случају примера ПАР1-4 имамо:
Потпис процедуре: *int[] sorti(int[] x)*
Веза са СК: *Нису коришћени СК у наведеном примеру.*
Предуслов*: Низ x треба да се састоји од целих бројева*
Постуслов*: добија се нови сортирани низ*

Предности процедуралне апстракције:

- Генерално се размишља о процедурама **шта** оне раде а не **како** раде**.**
- Сакривају се детаљи о томе **како** је процедура реализована или како ће бити реализована.
- Омогућава се лака промена постојеће имплементације процедуре са бољом имплементацијом.

СПЕ1-2: Апстракција података
Апстракцијом података се издваја из неког скупа података оно што су њихова општа својства.

Уколико имамо нпр. скуп студената ((123-00, Пера Перић),(124-00,Милош Савић),(125-00,Дејан Милић)), тада се параметризацијом добијају њихова општа својства (атрибути): *Број индекса* и *Име и презиме.*

Навођење општих својстава елемената скупа (*Број индекса* и *Име и презиме*) представља спецификацију скупа. **Апстракција података** је дефинисана **именом** (*Студент*) скупа и спецификацијом скупа[23] (*Број индекса* и *Име и презиме*).

Уколико елементи скупа имају и структуру и понашање (објекти) тада се над њима ради и процедурална апстракција и апстракција података. Као резултат наведених апстракција се добија **класа**.Класа се састоји из атрибута и процедура (метода), који су учаурени у класи.

Уколико елементи скупа имају само понашање тада се над њима ради процедурална апстракција која као резултат даје **интерфејс.** Интерфејс се састоји од скупа потписа процедура (операција).

СПЕ1-3: Апстракција контролом
Постоје два облика апстракције контроле:

1. Апстракција наредби
2. Апстракција структура података

СПЕ1-3-1:Апстракција наредби
Апстракцијом наредби се издваја из неког скупа наредби оно што су њихова општа својства и представљају се преко контролне структуре и опште наредбе.
Погледати пример ПАР1-5.

СПЕ1-3-2: Апстракција структура података
Апстракцијом структура података се издваја из неког скупа структура података оно што су њихова општа својства и представљају се преко итератора, који у општем смислу контролише кретање кроз структуру података.

---

[22] За процедуралну апстракцију се каже да је то уговор (contract) [Laman] који се прави између корисника процедуре и онога ко ће да је реализује.
[23] Спецификација скупа је математички гледајући интензија скупа.

2.3.4.1.2 Спојеност (coupling) и кохезија (cohesion)

У развоју објектно-оријентисаног софтвера потребно је постићи два циља:
а) треба изградити класе које имају високу кохезију (**high cohesion**).
б) класе треба да буду слабо повезане (**weak coupling**).

На примеру ћемо објаснити наведене принципе.

SPOJ1.1: Кохезија

Класа X је одговорна да обезбеди неко понашање m1().

```
class X
{  public
   m1(){}
}
```

Уколико у реализацији наведеног понашања учествују остале методе (*m11,m12 и m13*) класе X,

```
class X
{  public
   m1(){m11();m12();m13();…}
  private
  m11(){…}
  m12(){…}
 m13(){…}
}
```

онда за такву класу можемо да кажемо да има високу кохезију.

На основу наведеног примера можемо да закључимо да је *кохезија мера којом се утврђује колико су методе унутар класе међусобно повезане*. У наведеном примеру можемо видети да су методе *m11,m12 и m13* међусобно повезане јер заједно обезбеђују неко понашање које је дефинисано методом *m1*. Такође метода *m1* је повезана са методама *m11,m12 и m13* јер оне обезбеђују њено понашање. У наведеном примеру не постоји ниједна метода која није у функцији реализацији методе *m1*.
У примеру који је објашњен код Ларманове методе може се закључити да класе које су одговорне за реализацију системских операција имају високу кохезију, као нпр. код апстрактне класе *OpstaSO* која је одговорна за извршење системске операције *opsteIzvrsenjeSO():*

```
abstract class OpstaSO
{
 ...
 synchronized static String opsteIzvrsenjeSO(OpstiDomenskiObjekat rac, OpstaSO os)
  { if (!os.otvoriBazu()) return os.vratiPorukuMetode();

    if (!os.izvrsenjeSO(rac) && transakcija)
        { signal = os.rollbackTransakcije();
          return os.vratiPorukuMetode();
        }

   if (transakcija) os.commitTransakcije();
     return os.vratiPorukuMetode();
  }

  abstract boolean izvrsenjeSO(OpstiDomenskiObjekat rac);


  boolean otvoriBazu()
  { if (BazaOtvorena == false)
        { BBP = new BrokerBazePodataka();
          BBP.ispraznіPoruku();
         signal = BBP.otvoriBazu("RACUN");
         if (!signal) return false;
        }
    BBP.ispraznіPoruku();
    BazaOtvorena = true;
    return true;
  }
```

```
  boolean commitTransakcije()
   { return BBP.commitTransakcije();}

  boolean rollbackTransakcije()
   { return BBP.rollbackTransakcije();}

  String vratiPorukuMetode()
   { System.out.println(BBP.vratiPorukuMetode());
     return BBP.vratiPorukuMetode();
   }
}
```

У наведеном примеру све методе класе *OpstaSO* су у функцији реализације методе *opsteIzvrsenjeSO()*.

Губитак кохезије значи да је нека класа *X* одговорна да обезбеди више различитих понашања m1, m2 и m3 која нису између себе повезана. Оваква класа је је **тешка за одржавање и надоградњу**.

```
class Y
{  …
   public
  m1(){m11();m12();m13();…}
  m2(){m21();m22(); …}
  m3(){m31();m32(); …}

  private
  m11(){…}
  m12(){…}
 m13(){…}
…
m21(){…}
m22() {…}
…
m31(){…}
m32() {…}
…
}
```

Из наведеног примера може се закључити да су методе које су груписане (по понашању које треба да обезбеде) повезане. Тако су методе m11,m12,m13 повезане преко методе m1. Методе m21,m22 су повезане преко методе m2, док су методе m31,m32 повезане преко методе m3. Међутим, између метода различитих група не постоји повезаност (нпр. између m11 и m21).

Закључак:
Треба правити класе које имају јаку кохезију како би њима лакше управљало (како би се оне лако одржавале и надограђивале).

SPOJ1.2: Спојеност

Спојеност (*купловање*) значи да су класе у међусобној зависности. То значи да класа не може да обави неку операцију ако не постоји нека друга класа.
На пример класа X је одговорна да обезбеди неко понашање дефинисано методом m1() тако што позива методу m2() класе Y. Класа X је зависна од класе Y.

```
class X
{ Y y;
  X(){y=new Y();}
  public
   m1(){y.m2();}
}

class Y
{
m2(){...}
}
```

Куповање је мера везаности класе са другим класама, којом се утврђује колико је јако класа повезана са другим класама (*зависна од њих*). *Класа са високом (**high coupling**) или снажном (**strong coupling**) повезаношћу зависи од других класа.* Таква класа није пожељна, јер доводи до следећих проблема:

- Промене повезаних класа утичу на промену класа које су зависне од њих.
- Тешко је такву класу посматрати изоловану.
- Тешко је поново користити такву класу јер она захтева присуство других класа од којих зависи.

Класа са ниском (***low coupling***) или слабом (***weak coupling***) повезаношћу не зависи "много" од других класа. Класе које су слабо повезане могу се лакше поново користити у развоју других апликација.

Куповање је у програмирању неизбежно, али треба радити на томе да се зависности између класа сведу на најмању могућу меру.

У примеру који је објашњен код Ларманове методе може се закључити да су класе које су одговорне за реализацију системских операција између себе слабо повезане. [24]

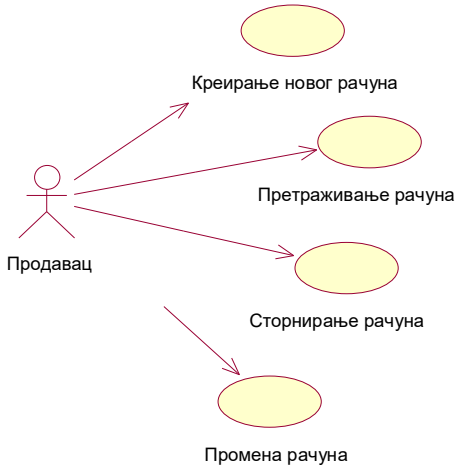
Закључак:
Софтверски систем треба да садржи класе које су независне или су међусобно слабо зависне.

[24] Повезаност се огледа преко класе OpstaSO која обезбеђује генеричко понашање које има свака од класа које су задужене за реализацију системских операција.

2.3.4.1.3 Декомпозиција и модуларизација (Decomposition and modularization)

Декомпозиција (рашчлањивање), у најопштијем смислу, је процес који почетни проблем дели у скуп подпроблема, који се независно решавају. Тако решени подпроблеми омогућавају да се лакше реши почетни проблем. Уколико декомпозицију посматрамо у контексту развоја софтверског система може се рећи да се декомпозицијом, софтверски систем дели у више модула. То значи да **модуларизација** софтверског система настаје као резултат процеса декомпозиције[25].

Декомпозицију ћемо објаснити из неколико различитих аспеката развоја софтвера:
а) *Декомпозиција код прикупљања захтева*
Кориснички захтев се декомпонује на скуп захтева који се описују преко случајева коришћења (у нашем примеру кориснички захтев се декомпонује на следеће захтеве: *креирање новог рачуна, претраживање рачуна, сторнирање рачуна и промена рачуна.*



б) *Декомпозиција код пројектовања софтвера*
У фази пројектовања прави се архитектура софтверског система која се обично декомпонује у три модула (компоненте): *кориснички интерфејс*, *апликациона логика* и *складиште података*. Након тога се ти модули даље декомпонују (нпр. код Ларманове методе *апликациона логика* се декомпонује на *контролер корисничког интерфејса*, *пословну логику* и *брокер базе података*). Сваки од модула који је добијен декомпозицијом може се независно пројектовати и имплементирати у односу на друге модуле.

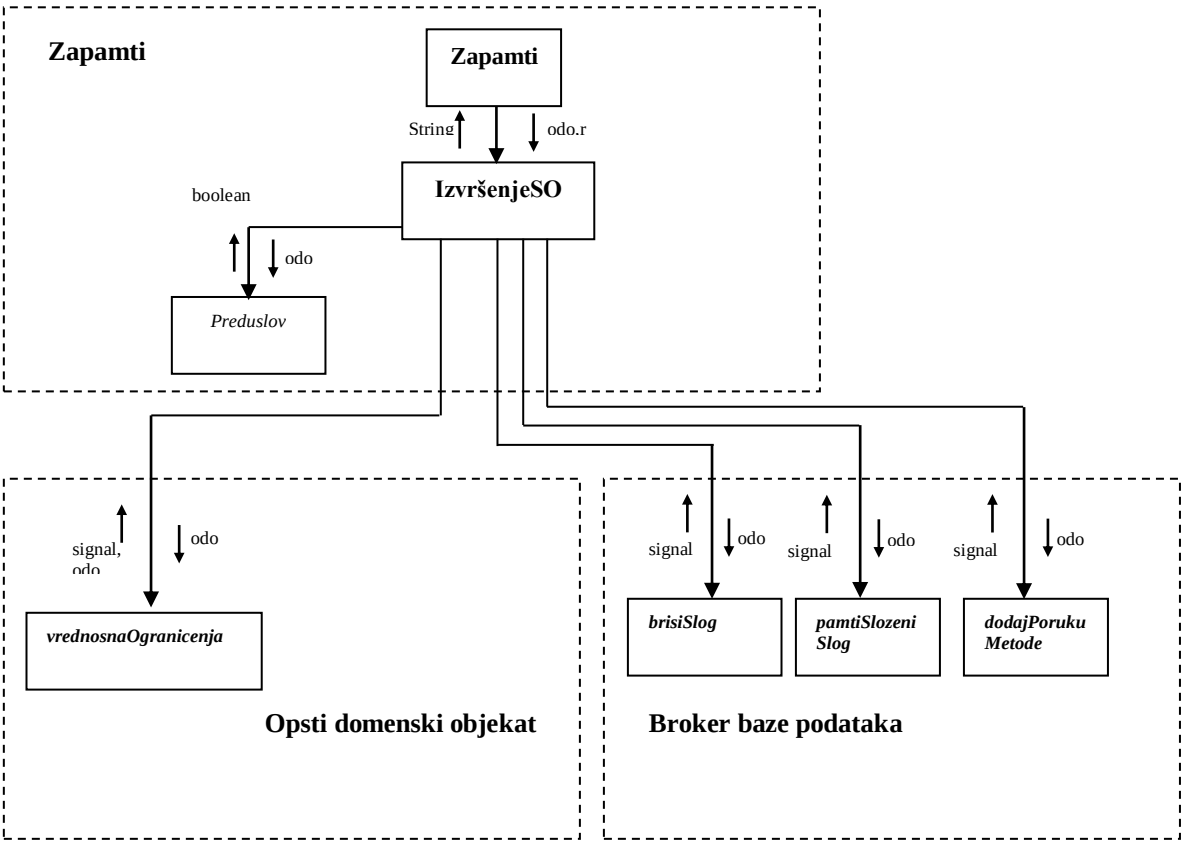
Постоји неколико принципа који требају да буду задовољени када се софтверски систем декомпонује у модуле:

1. Модули требају да имају јаку кохезију (*high cohesion*).
2. Модули треба да буду што је могуће слабије повезани (*weak coupling*).
3. Сваки модул треба да чува своје интерне информације *(information hiding*).

[25] У процесу декомпозиције користи се концепт апстракције.

Можемо да закључимо да је резултат процеса декомпозиције софтверског система дељење софтверског система у модуле. У том смислу кажемо да је **модуларизација** резултат процеса декомпозиције.

ц) Декомпозиција функција (метода)
Функција (метода) неке класе уколико је сложена може се декомпоновати у више подфункција. Ове подфункције се затим независно решавају, да би се на крају све подфункције интегрисале у једну целину како би се реализовала почетна фукција (нпр. пројектовање методе Zapamti*()*).



2.3.4.1.4 Учаурење (encapsulation)/ Сакривање информација (information hiding)

Учаурење је процес у коме се раздвајају: а) особине модула (нпр. класе) које су јавне за друге модуле од б) особина које су сакривене од других модула система. Особине које су јавне указују на суштинска одређења модула док особине које нису јавне указују на детаље модула[26]. Особине модула које су јавне могу користити други модули. Особине модула које нису јавне не могу користити други модули (те особине модула су сакривене од других модула). **Сакривање информација** настаје као резултат процеса учаурења, где се сакривају особине модула које нису јавне за корисника модула.

На пример уколико имамо класу X,

```
class X
{ …
  public      // особине класе које су јавне и које могу користити корисници.
  X(){…}
  void m1() {m2();m3();…}

  private    // особине класе које нису јавне и  које се сакривају од корисника.
  void m2(){a=5;}
  void m3() {b='x'}
  …

  int a;
  char b;
}

class Y
{
  public static void main(String arg[])
    { X x = new X();
      x.m1();
     }
}
```

њој се може приступити преко конструктор методе X() и методе m1 које су јавне док су методе m2 и m3 као и атрибути a и b приватни, односно сакривени од корисника те класе.
Корисник (класа Y) класе X зна како се иницијализује објекат[27] преко конструктора и зна да објекту (x) класе X може приступити преко методе m1. Осталим методама и атрибутима класе X корисник не може директно да приступи.

Корисник модула треба да зна само суштинске детаље модула а не мора да зна детаље о структури (атрибути a и b) и имплементацији модула (методе m2 и m3). Модули добијени као резултат процеса учаурења изгледају као "црна кутија[28]" за друге модуле у систему.

---

[26] Можемо да закључимо да је учаурење у суштини **апстракција**, јер сходно дефиницији апстракције: *Апстракција је **процес** који **издваја** из неке појаве **нешто** што је суштинско и општије од **нечега** што је детаљно и конкретније,* учаурење је процес који издваја из неког модула оно што је суштинско (јавне особине модула) од онога што представља детаље модула (особине модула које нису јавне).

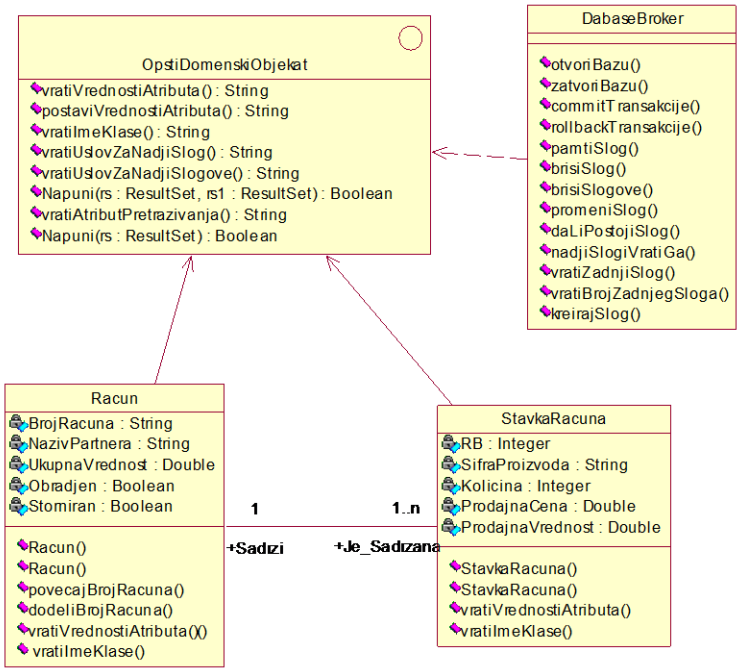
[27] Објекат који се користи је као "капсула" која има сервисе који се могу користити (при чему се не види како су ти сервиси имплементирани).

[28] Корисник зна шта "црна кутија" ради, али не зна како "црна кутија" ради. Корисник зна екстерно понашање "црне кутије", док је интерно понашање сакривено од корисника.

2.3.4.1.5 Одвајање интерфејса и имплементације (Separation of interface and implementation)

Интерфејс се одваја од имплементације и излаже се кориснику. Корисник не зна како су операције интерфејса имплементиране. У наведеном примеру корисник је DatabaseBroker, док је OpstiDomenskiObjekat интерфејс који излаже операције кориснику. DatabaseBroker не зна како су операције доменских класа (Racun, StavkaRacuna) имплементиране. Наведене доменске класе реализују интерфејс OpstiDomenskiObjekat.

Пример:



2.3.4.1.6 Довољност (sufficiency), комплетност и једноставност (primitiveness)

Довољност, комплетност и једноставност указују на особине софтверске компоненте која је **једноставна** за одржавање и надоградњу и **довољна** и **комплетна** да обезбеди жељену функционалност.

2.3.4.2 Стратегије пројектовања софтвера

Наводимо неколико стратегија пројектовања софтвера:

1. Подели и победи
2. С врха на доле
3. Одоздо нагоре
4. Итеративно – инкрементални приступ

POPO: Подели и победи (divide and conquer)

Стратегија *подели и победи* је заснована на принципу *декомпозиције* која почетни проблем дели у скуп подпроблема, који се независно решавају. На тај начин се лакше решава почетни проблем.

Примери:

1. *Прикупљање захтева* – Захтеви се описују преко скупа независних случајева коришћења.
2. *Анализа* – Структура се описује преко скупа концепата који чине концептулани модел. Понашање се описује преко скупа независних системским операција.
3. *Пројектовање* – Архитектура система се дели у три нивоа: кориснички интерфејс, апликациона логика и складиште података. Кориснички интерфејс се дели на 2 дела: екранске форме и контролер КИ. Апликациона логика се дели у три дела: Контролер апликационе логике, пословна логика и брокер базе података.

TODO: С врха на доле (top down)

Стратегија *с врха на доле* је заснована на принципу *декомпозиције функција* која почетну функцију декомпонује у више подфункција. Ове подфункције се затим независно решавају, да би се на крају све подфункције интегрисале у једну целину како би се реализовала почетна фукција (нпр. пројектовање методе *opsteIzvrsenjeSO()* апстрактне класе *opstaSO*.

Пример:

```
abstract class OpstaSO
{
  ...
  synchronized static String opsteIzvrsenjeSO(OpstiDomenskiObjekat rac, OpstaSO os)
   { if (!os.otvoriBazu()) return os.vratiPorukuMetode();

      if (!os.izvrsenjeSO(rac) && transakcija)
          { signal = os.rollbackTransakcije();
             return os.vratiPorukuMetode();
           }

     if (transakcija) os.commitTransakcije();
       return os.vratiPorukuMetode();
   }

   abstract boolean izvrsenjeSO(OpstiDomenskiObjekat rac);


   boolean otvoriBazu()
   { ... }

   boolean commitTransakcije()
    { ... }

   boolean rollbackTransakcije()
    { ... }

   String vratiPorukuMetode()
    { ... }
}
```

BOUP: Одоздо-на горе (bottom-up)
Стратегија *одоздо нагоре* је заснована на принципу *генерализације,* који у некој функцији, која је сложена, уочава једну или више логичких целина које проглашава за функције. На тај начин је једна сложена функција декомпонована у више независних функција. Композиција тих функција обезбеђује исту функционалност као и сложена функција.

Пример пре примене стратегије одоздо-на горе, где је метода *opsteIzvrsenjeSO()* била мало сложенија :

```
abstract class OpstaSO
{
 ...
 synchronized static String opsteIzvrsenjeSO(OpstiDomenskiObjekat rac, OpstaSO os)
  {
    if (BazaOtvorena == false)
        { BBP = new BrokerBazePodataka();
          BBP.isprazniPoruku();
         signal = BBP.otvoriBazu("RACUN");
         if (!signal) return os.vratiPorukuMetode();
        }
     BBP.isprazniPoruku();
     BazaOtvorena = true;

     if (!os.izvrsenjeSO(rac) && transakcija)
        { signal = os.rollbackTransakcije();
           return os.vratiPorukuMetode();
         }

    if (transakcija) os.commitTransakcije();
       return os.vratiPorukuMetode();
 }

 abstract boolean izvrsenjeSO(OpstiDomenskiObjekat rac);

  boolean commitTransakcije()
   { return BBP.commitTransakcije();}

  boolean rollbackTransakcije()
   { return BBP.rollbackTransakcije();}

  String vratiPorukuMetode()
   { System.out.println(BBP.vratiPorukuMetode());
     return BBP.vratiPorukuMetode();
   }

}
```

Пример после примене стратегије одоздо-на горе, где је метода *opsteIzvrsenjeSO()* поједностављена увођењем методе *OtvoriBazu()*:

```
abstract class OpstaSO
{
 ...
 synchronized static String opsteIzvrsenjeSO(OpstiDomenskiObjekat rac, OpstaSO os)
  { if (!os.otvoriBazu()) return os.vratiPorukuMetode();

    if (!os.izvrsenjeSO(rac) && transakcija)
        { signal = os.rollbackTransakcije();
           return os.vratiPorukuMetode();
         }

   if (transakcija) os.commitTransakcije();
      return os.vratiPorukuMetode();
 }

  abstract boolean izvrsenjeSO(OpstiDomenskiObjekat rac);


  boolean otvoriBazu()
  { if (BazaOtvorena == false)
```

```
        { BBP = new BrokerBazePodataka();
          BBP.ispraznPoruku();
         signal = BBP.otvoriBazu("RACUN");
         if (!signal) return false;
        }
    BBP.ispraznPoruku();
    BazaOtvorena = true;
    return true;
  }

  boolean commitTransakcije()
   { return BBP.commitTransakcije();}

  boolean rollbackTransakcije()
   { return BBP.rollbackTransakcije();}

  String vratiPorukuMetode()
   { System.out.println(BBP.vratiPorukuMetode());
     return BBP.vratiPorukuMetode();
   }
}
```

ITIN: Итеративно-инкрементални приступ (iterative and incremental approach)

Систем се дели у више мини пројеката који се независно развијају и пролазе кроз све фазе развоја софтвера. Мини пројекат може да буде везан за случај коришћења или за сложену системску операцију.
Сваки мини пројекат пролази кроз више **итерација**. Као резултат итерације добија се интерно издање (release) или build који представља **икремент** за систем. На крају се имплементирани мини пројекти интегришу у софтверски систем.

Пример:
У нашем примеру смо имали неколико независних случаја коришћења из којих смо добили неколико независних системских операција. Развој појединачне системске операције није зависила од развоја било које друге системске операције. То значи да нека системска операција може да буде имплементирана и интегрисана у систем, док друга системска операција може тек да буде идентификована у фази анализе.

2.3.3: Методе пројектовања софтвера

Наводимо неке од најважнијих метода пројектовања софтвера:
1. Функционо оријентисано пројектовање
2. Објектно оријентисано пројектовање
3. Пројектовање засновано на структури података
4. Пројектовање засновано на компонентама

Функционо оријентисано пројектовање је засновано на функцијама. Проблем се посматра из перспективе његовог понашања односно функционалности. Прво се уочавају функције системе, затим се одређују структуре података над којима се извршавају те функције.

Објектно оријентисано пројектовање је засновано на објектима (класама). Објекти могу да репрезентују и структуру (атрибуте) и понашање (методе) софтверског система. Код објектно оријентисаног пројектовања паралелно се развијају и струкура и понашање.

Пројектовање засновано на структури података проблем посматра из перспективе структуре. Прво се уочава структура система, затим се одређују функције које се извршавају над том структуром.

Пројектовање засновано на компонентама проблем посматра из перспективе постојећих компоненти које се могу (поново) користити у решавању проблема. Прво се уочавају делови проблема који се могу реализовати преко постојећих компоненти. Након тога се имплементирају делови проблема за које није постојало решење[29].

[29] То је у нашем примеру генерички *Брокер базе података* који се може користи у реализацији различитих софтверских система.

OOPR: Објектно оријентисано пројектовање (Object-oriented design)

Објектно оријентисано пројектовање је засновано на објектима (класама). Објекти могу да репрезентују и структуру (атрибуте) и понашање (методе) софтверског система. Код објектно оријентисаног пројектовања паралелно се развијају и струкура и понашање.

Пример:
У нашем примеру као резултат анализе се добија логичка структура и понашање софтверског система. Структура је описана преко концептуалног модела, док је понашање описано преко секвенцних дијаграма. Концептуални модел се састоји од скупа међусобно повезаних концепата (класа,односно њихових појављивања објеката), док секвенцни дијаграм описује интеракцију између објеката (актора и софтверског система). То значи да су у основи логичке структуре и понашања софтверског система објекти. Зато се оваква анализа назива **објектно-оријентисана анализа**. Касније се у фази пројектовања и имплементације одређују класе које ће да реализују струкуру (доменске класе) и понашање (класе које реализују системске операције). То значи да су у основи физичке структуре и понашања софтверског система такође налазе објекти, односно класе. Зато се такво пројектовање назива **објектно-оријентисано пројектовање**.

2.3.4.2 Принципи објектно оријентисаног пројектовања класа (Principles of Object Oriented Class Design)

Постоје следећи принципи код објектно-оријентисаног пројектовања класа:
1. Принцип Отворено-Затворено
2. Принцип замене Барбаре Лисков
3. Принцип инверзије зависности
4. Принцип уметања зависности
5. Принцип издвајања интерфејса

2.3.4.2.1 Принцип Отворено-Затворено (Open-Closed principle)

***Принцип Отворено – Затворено**: Модул треба да буде отворен за проширење али и затворен за модификацију.*

У наведеном примеру класа (модул) *OpstaSO* је **затворена** за промену њеног понашања али је у исто време и **отворена** за промену њеног понашања у класама које су из ње изведена као што је нпр. класа *KreirajNovi*:

```
abstract class OpstaSO
{
  ...
  synchronized static String opsteIzvrsenjeSO(OpstiDomenskiObjekat rac, OpstaSO os)
   { if (!os.otvoriBazu()) return os.vratiPorukuMetode();

      if (!os.izvrsenjeSO(rac) && transakcija)
          { signal = os.rollbackTransakcije();
            return os.vratiPorukuMetode();
           }

     if (transakcija) os.commitTransakcije();
       return os.vratiPorukuMetode();
   }

   abstract boolean izvrsenjeSO(OpstiDomenskiObjekat rac);

   boolean otvoriBazu()
   { ... }

   boolean commitTransakcije()
    { ... }

   boolean rollbackTransakcije()
    { ... }

   String vratiPorukuMetode()
    { ... }
}

class KreirajNovi extends OpstaSO
{
  public static String kreirajNovi(OpstiDomenskiObjekat odo)
    { KreirajNovi kn = new KreirajNovi();
      OpstaSO.transakcija = true;
      return OpstaSO.opsteIzvrsenjeSO(odo,kn);
    }

  // Prekrivanje metode klase OpstaSO
  boolean izvrsenjeSO(OpstiDomenskiObjekat odo)
   {  if (!BBP.kreirajSlog(odo))
        { PorukaIzvrsenjeSO("Sistem ne moze da kreira " + odo.vratiNazivNovogObjekta() + ".");
                return false;
        }
      PorukaIzvrsenjeSO("Sistem je kreirao " + odo.vratiNazivNovogObjekta() + ".");
      return true;
    }

}
```

2.3.4.2.2 Принцип замене Барбаре Лисков (The Liskov Substitution Principle (LSP))

Принцип замене Барбаре Лисков: *Подкласе треба да буду замењиве са њиховим надкласама.*

У наведеном примеру подкласе *Racun* и *StavkaRacuna* су замењиве са њиховом надкласом *OpstiDomenskiObjekat.*

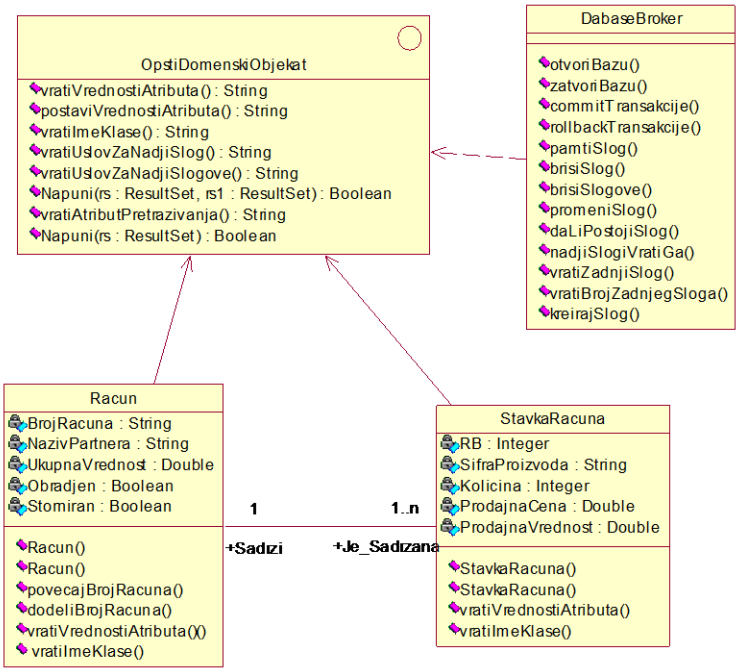
```
   public int pamtiSlog(OpstiDomenskiObjekat odo)
     { String upit;

        try{
               st = con.createStatement();
               upit ="INSERT INTO " + odo.vratiImeKlase()  +
             " VALUES (" + odo.vratiVrednostiAtributa() + ")";
      System.out.println("Upis unosa stavke pre izvesenja:" + upit);
      st.executeUpdate(upit);
      st.close();
           }   catch(SQLException esql)
                 {  System.out.println("Nije uspesno zapamcen slog u bazi: " + esql);
                                 return 32;
                  }
       return 31;
     }


...

Racun rac = new Racun();
pamtiSlog(rac);

StavkaRacuna srac = new StavkaRacuna();
pamtiSlog(srac);
```

2.3.4.2.3 Принцип инверзије зависности (The Dependency Inversion Principle)

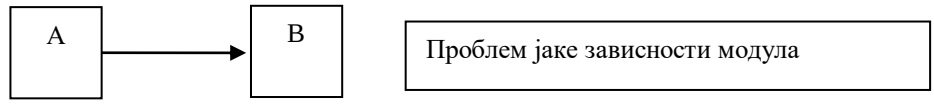
Принцип инверзије зависности: *Зависи од аптракције а не од конкретизације.*

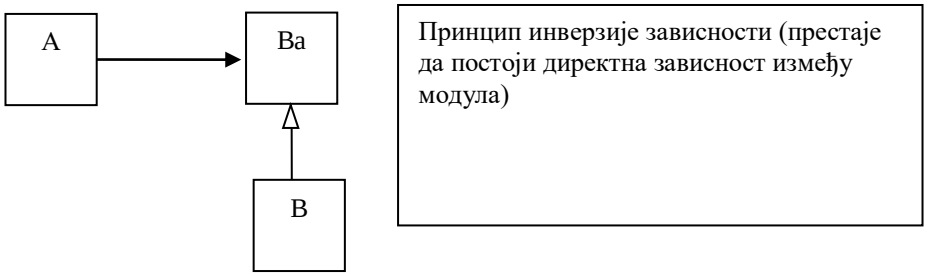
Модули вишег нивоа не треба да зависе од модула нижег нивоа. Оба треба да зависе од апстракције. Абстракције не треба да зависе од детаља. Детаљи треба да зависе од апстракције. (Robert Martin)

У наведеном примеру се избегава директна зависност између брокера базе података (модул вишег нивоа) и доменских класа (модули нижег нивоа) већ се брокер повезује са интерфејсом OpstiDomenskiObjekat кога реализују наведене доменске класе. На тај начин брокер базе података је посредно повезан преко интерфејса OpstiDomenskiObjekat са сваком класом која реализује тај интерфејс.



Општи случај:

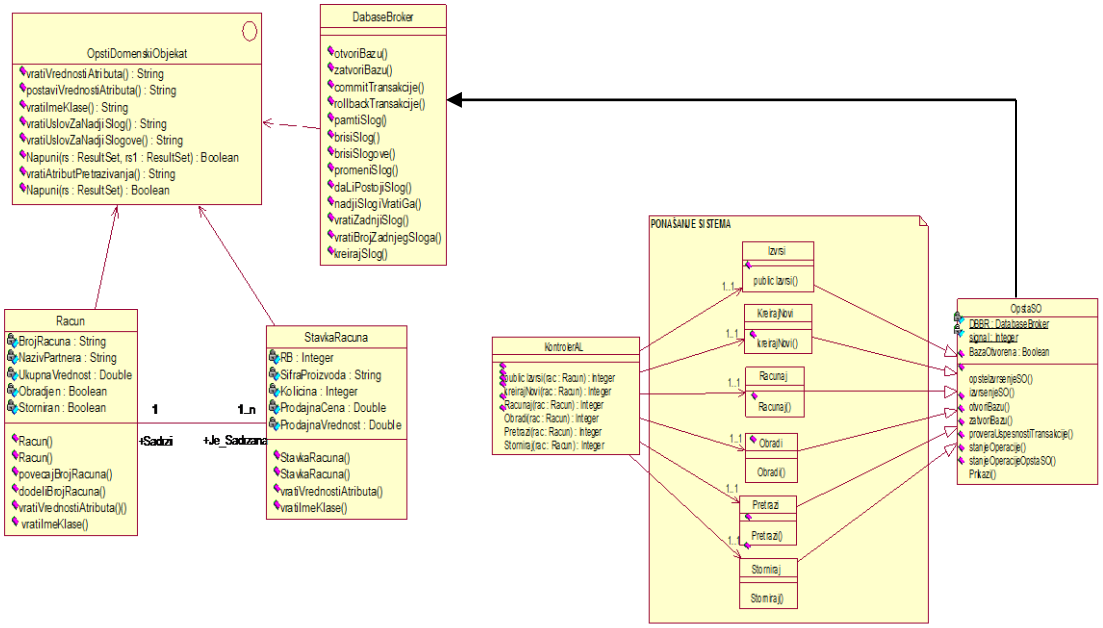




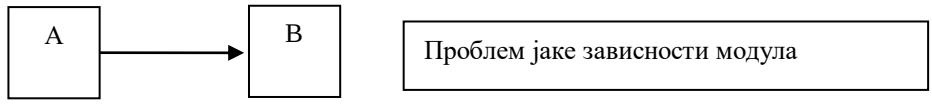
2.3.4.2.4 Принцип уметања зависности (The Dependency Injection Principle)

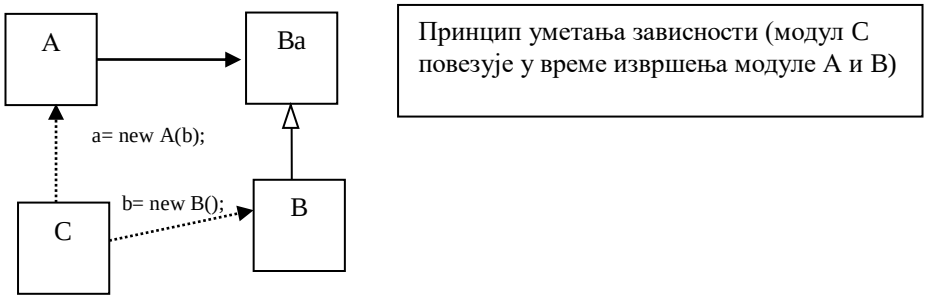
Принцип уметања зависности: *Зависности између 2 компоненте програма се успостављају у време извршења програма преко неке треће компоненте.*

У наведеном примеру зависност између брокера базе података и конкретне доменске класе се успоставља у време извршења програма преко класе *OpstaSO*.



Општи случај:





Детаљно објашњење принципа инверзије зависности, инверзије контроле и уметања зависности

Принцип инверзије зависности (Dependency Inversion Principle)

Принцип инверзије зависности је један од принципа објектно-оријентисаног пројектовања класа који омогућава да се пројектују класе које су слабо између себе повезане (купловане). Принцип инверзије зависности заснован је на следећа два правила:

1. Модули вишег нивоа не треба да зависе од модула нижег нивоа. Оба треба да зависе од апстракције.
2. Абстракције не треба да зависе од детаља. Детаљи треба да зависе од апстракције. (Robert Martin)

У следећем примеру модул високог нивоа (ModulVisokogNivoa) непосредно зависи од модула ниског нивоа (ModulNiskogNivoa1).

```
class ModulNiskogNivoa1
{
   public void Prikazi(String poruka)
   {
      System.out.println("Poruka modula niskog nivoa: " + poruka);
   }
}

class ModulVisokogNivoa
{

   ModulNiskogNivoa1 mnn = null; // зависност модула високог и ниског нивоа.

   public void Obrada(String poruka)
   {
      if (mnn == null)
      {
         mnn = new ModulNiskogNivoa1();
      }
      mnn.Prikazi(poruka);
   }
}


class Glavna1
{

public static void main(String[] str)
  {
     ModulVisokogNivoa mvn = new ModulVisokogNivoa();
     mvn.Obrada("danas je lep dan.");
  }

}
```

На тај начин је нарушен принцип инверзије зависности, што представља проблем код даље надоградње наведеног програма. Уколико би желели да додамо још један модул ниског нивоа (ModulNiskogNivoa2), који би на другачији начин приказао поруку од постојећег модула ниског нивоа (ModulNiskogNivoa1), морали би да мењамо модул високог нивоа.

Инверзија контроле (Inversion of Control)

Зависност из наведеног примера између модула високог и ниског нивоа треба да се прекине, како би се остварио принцип инверзије зависности. То се постиже помоћу механизма који се назива **инверзија контроле** (inversion of control) који успоставља зависност између модула високог нивоа и апстракције (апстрактног модула), и модула ниског нивоа и апстракције, уместо зависности између модула високог нивоа и модула ниског нивоа.
Наведени пример се проширује са интерфејсом ApstraktniModul са којим се повезују модул високог нивоа (ModulVisokogNivoa).

```
interface ApstraktniModul
{  public void Prikazi(String poruka); }

class ModulVisokogNivoa
{ ApstraktniModul am = null;
   public void Obrada(String poruka)
   {
      if (am == null)
          { am = new ModulNiskogNivoa1(); }
      am.Prikazi(poruka);
   }
}
```

Модул ниског нивоа(ModulNiskogNivoa1) такође успоставља везу са апстрактним модулом тако што га имплементира.

```
class ModulNiskogNivoa1 implements ApstraktniModul
{
   public void Prikazi(String poruka)
   { System.out.println("Poruka modula niskog nivoa 1: " + poruka); }
}
```

Уметање зависности (Dependency Injection)

Успостављање зависност, у време извршења програма, између модула високог и ниског нивоа преко апстрактног модула се остварује на три начина:

1. Уметање зависности преко конструктора (Constructor injection)
2. Уметање зависности преко методе (Method injection)
3. Уметање зависности преко поља (Field injection)

Уметање зависности преко конструктора (Constructor injection)

У конструктору модула високог нивоа (ModulVisokogNivoa) као параметар се поставља апстрактни модул. Тиме се омогућава да било који конкретни модул ниског нивоа (нпр. ModulNiskogNivoa1) који је изведен из апстрактног модула, може бити прихваћен као параметар преко конструктор методе модула високог нивоа.

```
interface ApstraktniModul
{ public void Prikazi(String poruka); }

class ModulNiskogNivoa1 implements ApstraktniModul
{
   public void Prikazi(String poruka)
   {  System.out.println("Poruka modula niskog nivoa 1: " + poruka);  }
}

class ModulVisokogNivoa
{   ApstraktniModul am = null;
   // уметање зависности преко конструктора
   ModulVisokogNivoa(ApstraktniModul am1) {am=am1;}
   public void Obrada(String poruka)
   { am.Prikazi(poruka);
   }
}

class Glavna2
{
   public static void main(String[] str)
  {  ModulNiskogNivoa1 mnn1= new ModulNiskogNivoa1();
      ModulVisokogNivoa mvn = new ModulVisokogNivoa(mnn1);
      mvn.Obrada("danas je lep dan.");
  }

}
```

Уколико би написали нови модул ниског нивоа:

```
class ModulNiskogNivoa2 implements ApstraktniModul
{
   public void Prikazi(String poruka)
   {  System.out.println("Poruka modula niskog nivoa 2: " + poruka);  }
}
```

и променили главни програм:

```
class Glavna21
{
  public static void main(String[] str)
  {  ModulNiskogNivoa2 mnn2 = new ModulNiskogNivoa1();
     ModulVisokogNivoa mvn = new ModulVisokogNivoa(mnn2);
     mvn.Obrada("danas je lep dan.");
  }
}
```

модул високог нивоа (ModulVisokogNivoa) се не би мењао, јер он може да прихвати преко параметра конструктор методе било који модул (ModulNiskogNivoa**N**) који је изведен из апстрактног модула:

```
class ModulNiskogNivoaN implements ApstraktniModul
{
   public void Prikazi(String poruka)
   {  System.out.println("Poruka modula niskog nivoa n: " + poruka);  }
}
```

Уметање зависности преко методе (Method injection)

Код неке од метода модула високог нивоа (ModulVisokogNivoa) као параметар се поставља апстрактни модул. Тиме се омогућава да било који конкретни модул ниског нивоа (нпр. ModulNiskogNivoa1, ModulNiskogNivoa2) који је изведен из апстрактног модула, може бити прихваћен као параметар преко неке од метода (нпр. Obrada()) модула високог нивоа.

```
interface ApstraktniModul
{ public void Prikazi(String poruka); }

class ModulNiskogNivoa1 implements ApstraktniModul
{
   public void Prikazi(String poruka)
   {  System.out.println("Poruka modula niskog nivoa 1: " + poruka);  }
}

class ModulNiskogNivoa2 implements ApstraktniModul
{
   public void Prikazi(String poruka)
   {  System.out.println("Poruka modula niskog nivoa 2: " + poruka);  }
}

class ModulVisokogNivoa
{   ApstraktniModul am = null;
   // уметање зависности преко методе
   public void Obrada(ApstraktniModul am1, String poruka)
   { this.am = am1;
     am.Prikazi(poruka);
   }
}

class Glavna22
{
  public static void main(String[] str)
  {  ModulNiskogNivoa1 mnn1= new ModulNiskogNivoa1();
     ModulVisokogNivoa mvn = new ModulVisokogNivoa();
     mvn.Obrada(mnn1,"danas je lep dan.");
  }
}
```

Уметање зависности преко поља (Field injection)

Апстрактни модул који је јавни атрибут (поље) модула високог нивоа (ModulVisokogNivoa) повезује се са неким од модула ниског нивоа изван модула високог нивоа. У нашем примеру апстрактни модул је повезан са модулом ниског нивоа у главном програму (Glavna23).

```
interface ApstraktniModul
{ public void Prikazi(String poruka); }

class ModulNiskogNivoa1 implements ApstraktniModul
{
   public void Prikazi(String poruka)
   {  System.out.println("Poruka modula niskog nivoa 1: " + poruka);  }
}

class ModulNiskogNivoa2 implements ApstraktniModul
{
   public void Prikazi(String poruka)
   {  System.out.println("Poruka modula niskog nivoa 2: " + poruka);  }
}


class ModulVisokogNivoa
{   public ApstraktniModul am = null;

   public void Obrada(String poruka)
   {  am.Prikazi(poruka);
   }
}

class Glavna23
{
  public static void main(String[] str)
  {  ModulNiskogNivoa1 mnn1= new ModulNiskogNivoa1();
     ModulVisokogNivoa mvn = new ModulVisokogNivoa();
     // уметање зависности преко поља
     mvn.am = mnn1;
     mvn.Obrada("danas je lep dan.");
  }
}
```

2.3.4.2.5 Принцип издвајања интерфејса (The Interface Segregation Principle)

Принцип издвајања интерфејса: *Боље је да постоји више специфичних клијентских интерфејса него један генерални интерфејс опште намене.*

Боље је имати више специфичних интерррфејса него један генерални интерфејс.

Пример:

Проблем:

```
class Servis
{  void m1(){…}
   void m2() {…}
   void m3() {…}
}

class Klijent1
{   Servis s;
    m1() {  s = new Servis();
              s.m1();
            }
}

class Klijent2
{   Servis s;
    m2() {  s = new Servis();
              s.m2();
            }
}

class Klijent3
{   Servis s;
    m3() {  s = new Servis();
              s.m3();
            }
}
```

Решење:

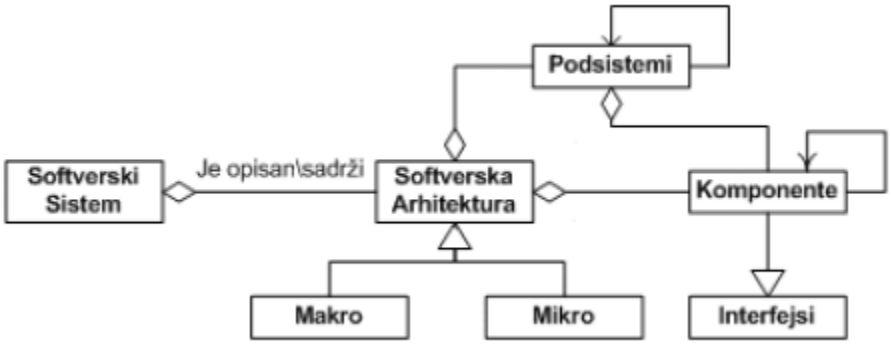
```
interface M1
 { void m1();
 }

interface M2
 { void m2();
 }

interface M3
 { void m3();
 }

class Servis implements M1,M2,M3
{  void m1(){…}
   void m2() {…}
   void m3() {…}
}

class Klijent1
{   M1 s;
    Klijent1(M1 s1) {s=s1;}
    m1(M1 s1) {  s.m1();
            }
}
class Klijent2
{   M2 s;
    Klijent2(M2 s1) {s=s1;}
    m2() {  s.m2();   }
}
class Klijent3
{   M3 s;
    Klijent3(M1 s1) {s=s1;}
    m3() { s.m3(); }
}
```

Закључак: *Уколико се промени класа сервис (дода се нека нова операција), мењаће се само интерфејс клијента који користи ту операције. Интерфејси осталих клијената се не мењају.*

2.3.5 Софтверске структуре (архитектуре)

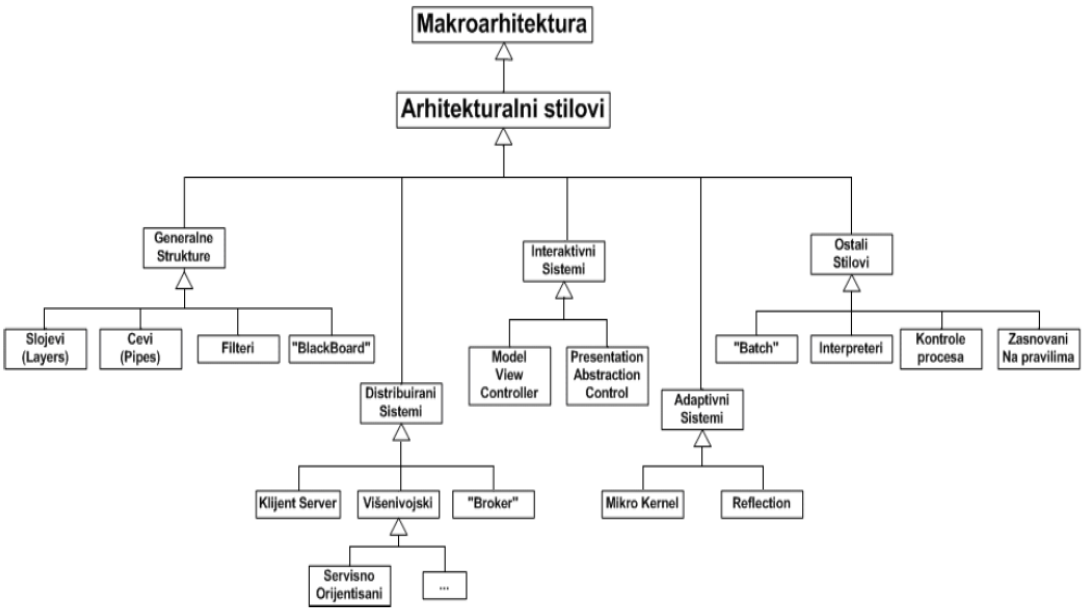
Софтверска архитектура

Софтверска архитектура се састоји од подсистема, компоненти и интерфејса компоненти софтверског система и веза између њих. Софтверска архитектура може бити макро и микро. Макро архитектура описује структуру и организацију софтверског система на највишем нивоу и она може бити реализована преко нпр. преко ECF (Enterprise Component Framework) или MVC (Model-View-Controler) патерна, док је микро архитектура реализована преко патерна пројектовања и то: креационих, структурних и патерна понашања.

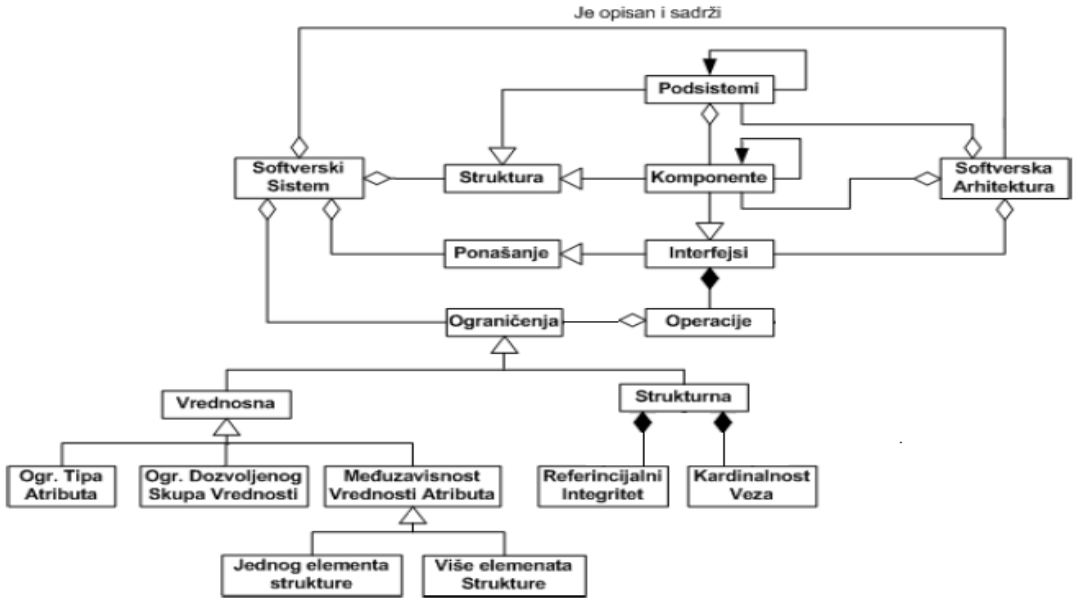


2.3.5.1 Макро архитектура

Макро архитектура описује структуру и организацију софтверског система на највишем нивоу. Постоје разни архитектурни стилови којим се описује архитектура софтверског система. Постоје генералне структуре (слојеви, цеви, филтери, ...), дистрибуирани системи (клијент-сервер, вишенивојски, ...), интерактивни системи (Model View Controler, Presentation Abstraction Control, ...), адаптивни системи (Micro Kernel, Reflection,...) и остали стилови (Batch, Interpreteri,...).

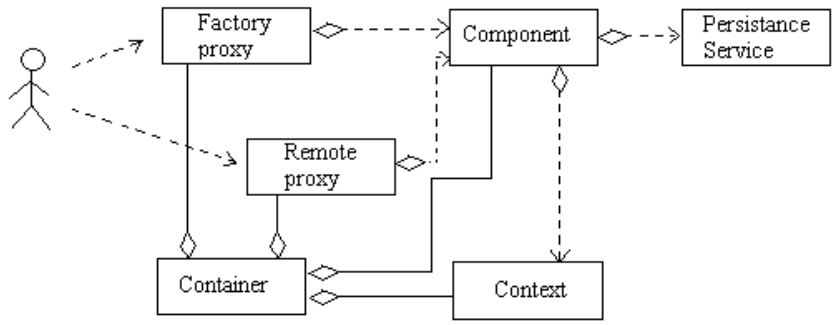


Као што је речено софтверска архитектура се састоји од подсистема, компоненти и интерфејса компоненти софтверског система и веза између њих. Наведени елементи софтверског система описују понашање, структуру и ограничења софтверског система. Структура се описује преко подсистема и компоненти. Понашање се описује преко интерфејса који се састоји из скупа операција које је изводе над компонентама под дефинисаним ограничењима. Ограничења могу да буду вредносна и структурна. Вредносна ограничења су везана за вредности и типове атрибута компоненти и међузависности између атрибута компоненти.Структурна ограничења се односе на пресликавања која постоје између компоненти софтверског система.



У даљем тексту објаснићу ECF и MVC макро-архитектурне патерне.

ECF (Enterprise Component Framework) патерн
ECF је макроархитектурни патерн за развој сложених (Enterprise) дистрибуираних апликација које су засноване на софтверским компонентама (Component), које се могу поново користити у новим проблемским ситуацијама.
ECF (Слика ECF) садржи следеће елементе: Client, FactoryProxy, RemoteProxy, Context, Component, Container i PersistenceService.
Наведени елементи имају следеће улоге:
Client поставља захтев за извршење операције над Component елементом.
Proxy елемент, пресреће клијентски захтев и и извршава операцију над Component елементом. FactoryProxy је задужен да обезбеди следеће операсије: create(), find() и remove(), док је RemoteProxy задужен да обезбеди остале операције које позива клијент.
Context елемент постоји за сваки component елемент и он чува његов контекст, као што је: стање трансакције, перзистентност, сигурност,…
Container елемент обухвата (агрегира) све елементе ECF узора осим Client и PersistenceService елемената. Он обезбеђује run-time окружење на коме се извршавају разни сервиси дистрибуиране обраде апликација, као што су: међупроцесна комуникација, сигурност, перзистентност и трансакције. Component елемент, када жели да обезбеди своју перзистентност позива PersistanceService елемент који је за то задужен.



Slika ECF: ECF патерн

Из ECF патерна су изведене EJB (Enterprise JavaBeans) и COM+ архитектуре.

2.3.5.1.1 MVC (Model-View-Controller) патерн

MVC (Слика MVC) је макроархитектурни патерн, који дели софтверски систем у три дела:
a) **view** - обезбеђује кориснику интерфејс (екранску форму) помоћу које ће корисник да уноси податке и позива одговарајуће операције које треба да се изврше над model-ом. View приказује кориснику стање модела.
b) **controller** – ослушкује и прихвата захтев од клијента за извршење операције. Након тога позива операцију која је дефинисана у моделу. Уколико model промени стање controller обавештава view да је промењено стање.
c) **model** – представља стање система. Стање могу мењати неке од операција model-a.

Правило 1: Контролер прати догађаје који се извршавају над view и на одговарајући начин реагује на њих.

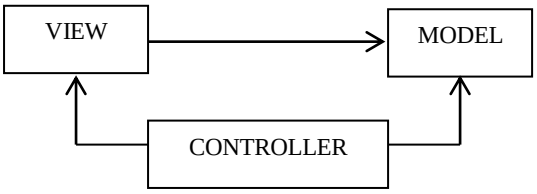
Правило 2: Контролер је диспечер који прима захтев од view-a и преусмерава га до model-a.

Правило 3: View мора да рефлектује стање модела. Сваки пут када се промени стање model-a, view треба да буде обавештен о томе. Нпр. код Observer патерна ConcreteSubject елемент је **Model** док је ConcreteObserver елемент **View**. Када се промени стање од ConcreteSubject елемента тада Subject (који је **Controller)** обавештава ConcreteObserver да је промењено стање и тада ConcreteObserver чита ново стање од ConcreteSubject-a.

Правило 4: Model не мора да зна ко је View и Controller.

Connelly Barnes је рекао: " Најлакши начин да разумете MVC је: model је податак, view је екранска форма, controller је лепак између modela и view-a.

У књизи **Design Patterns** MVC се објашњава на следећи начин: Model је апликациони објекат, View је екранска презентација а Controller дефинише како кориснички интерфејс реагује на корисничке улазе. MVC је објашњен у контексту прављења корисничког интерфејса код SmallTalk програмског језика.



Slika MVC: MVC патерн

Пример MVC1:

/* Кориснички захтев: Направити екранску форму која ће имати: а) поље за прихват бројева и б) поље за приказ збира свих до тада унетих бројева. Збир се рачуна кликом на дугме. */

```
// View.java
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class View extends JFrame
{ Model mod;
  // Labele koja sadrzi naziv ekranske forme koja se otvara.
  private JLabel LNazivForme;

  // Polja preko kojih se obradjuju podaci.
  private JFormattedTextField PBroj;
  private JFormattedTextField PZbir;

  // Labele koje opisuju polja za obradu podataka.
  private JLabel LBroj;
  private JLabel LZbir;

  // Dugme preko koga se poziva sistemska operacija.
  public JButton BZbir;

// 1. Konstruktor ekranske forme
public View (Model mod1)
  { KreirajKomponenteEkranskeForme();         // 1.1
    PokreniMenadzeraRasporedaKomponeti(); // 1.2
    PostaviImeForme();                          // 1.3
    PostaviPoljeZaPrihvatBrojeva();          // 1.4
    PostaviPoljeZaZbir();                       // 1.5
    PostaviLabeluZaPrihvatBrojeva();        //  1.6
    PostaviLabeluZaZbir();                      // 1.7
    PostaviDugmeZbir();                          // 1.8
    mod = mod1;
    postaviVrednost();
    pack();
    show();
    }

// 1.1 Kreiranje i inicijalizacija komponenti ekranske forme
void KreirajKomponenteEkranskeForme()
  { LNazivForme = new JLabel();
    PBroj = new JFormattedTextField();
    PZbir = new JFormattedTextField();
    LBroj= new JLabel();
    LZbir = new JLabel();
    BZbir = new JButton();
  }
```

```
// 1.2 Kreiranje menadjera rasporeda komponenti i njegovo dodeljivanje do kontejnera okvira(JFrame komponente).
void PokreniMenadzeraRasporedaKomponeti()
  { getContentPane().setLayout(new AbsoluteLayout());}

// 1.3 Odredivanje naslovnog teksta i njegovo dodeljivanje do kontejnera okvira.
void PostaviImeForme()
{  LNazivForme.setFont(new Font("Times New Roman", 1, 12));
   LNazivForme.setText("SABIRANJE NIZA BROJEVA");
   getContentPane().add(LNazivForme, new AbsoluteConstraints(20, 10, -1, -1));
}

// 1.4
void PostaviPoljeZaPrihvatBrojeva()
 { // Dodeljivanje pocetne vrednosti i formata polja.
   PBroj.setValue(new String("0"));
   // Polje se dodaje kontejneru okvira (JFrame).
    getContentPane().add(PBroj, new AbsoluteConstraints(50, 70, 70, -1));
}

// 1.5
void PostaviPoljeZaZbir()
 {  // Dodeljivanje pocetne vrednosti i formata polja.
    PZbir.setValue(new String("0"));
    // Vrednost polja ne moze da se menja.
   PZbir.setEditable(false);
    // Polje se dodaje kontejneru okvira (JFrame)
    getContentPane().add(PZbir, new AbsoluteConstraints(50, 100, 90, -1));
 }

// 1.6
void PostaviLabeluZaPrihvatBrojeva()
{ LBroj.setText("Broj");
  getContentPane().add(LBroj, new AbsoluteConstraints(20, 70, -1, -1));}
// 1.7
void  PostaviLabeluZaZbir()
 { LZbir.setText("Zbir");
   getContentPane().add(LZbir, new AbsoluteConstraints(20, 100, -1, -1));
 }

//*****************************************************************************
// 1.8
void PostaviDugmeZbir()
 { BZbir.setText("Zbir");
   getContentPane().add(BZbir, new AbsoluteConstraints(160, 60, -1, -1));
}

public int uzmiVrednost(){
      return Integer.parseInt((String)PBroj.getValue()); }

public void postaviVrednost()  { PZbir.setValue(String.valueOf(mod.uzmiBroj())); }

}
```

// Controller.java

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class Controller
{  Model mod;
   View view;
   Controller(View view1,Model mod1)    {mod = mod1; view = view1; OsluskujDugmeZbir(); }

    void OsluskujDugmeZbir()
     { view.BZbir.addActionListener(new mojOsluskivac(this));}

}
```

```
class mojOsluskivac implements ActionListener
 {  Controller c;

    mojOsluskivac (Controller c1)  {c=c1;}

    public void actionPerformed(ActionEvent evt)
                {  int pom = c.view.uzmiVrednost();
                   c.mod.sistemskaOperacija(pom);
                   c.view.postaviVrednost();
                }
}
```

// Model.java

```
public class Model
{ int broj;
  Model() {broj = 0;}
public void sistemskaOperacija(int broj1)  {  broj = broj + broj1; }
public int uzmiBroj () { return broj;}
}
```

// MVC1.java – glavni program

```
public class MVC1
{
public static void main(String args[])
 {  Model mod = new Model();
    View view = new View(mod);
   Controller con = new Controller(view, mod);
   }
}
```

Пример MVC2: /* Кориснички захтев: Направити три екранске форме истог типа које ће имати: а) поље за прихват бројева и б) поље за приказ збира свих до тада унетих бројева. Збир се рачуна кликом на дугме форме. Сваки пут када се промени збир унетих бројева та промена треба да се види на свакој од наведених форми. То значи да када се промени стање модела контролер треба да јави екранским формама да је промењено стање.*/

// View.java

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class View extends JFrame
{  // Isto kao u MVC1
...
// 1. Konstruktor ekranske forme
public View (Model mod1, String Naziv)
  { KreirajKomponenteEkranskeForme(Naziv);     // 1.1
    PokreniMenadzeraRasporedaKomponeti();     // 1.2
    PostaviImeForme();                         // 1.3
    PostaviPoljeZaPrihvatBrojeva();            // 1.4
    PostaviPoljeZaZbir();                      // 1.5
    PostaviLabeluZaPrihvatBrojeva();           // 1.6
    PostaviLabeluZaZbir();                     // 1.7
    PostaviDugmeZbir();                        // 1.8
    mod = mod1;
    postaviVrednost();
    pack();
    show();
    }

// 1.1 Kreiranje i inicijalizacija komponenti ekranske forme
void KreirajKomponenteEkranskeForme(String Naziv)
  { LNazivForme = new JLabel();
    PBroj = new JFormattedTextField();
    PZbir = new JFormattedTextField();
    LBroj= new JLabel();
    LZbir = new JLabel();
    BZbir = new JButton();
    Container con = getContentPane();
    con.setName(Naziv);
 } // Isto kao u MVC1
...}
```

```
// Controller.java
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;


public class Controller
{ Model mod;
  View view[];
   Controller(View[] view1,Model mod1)
      { mod = mod1; view = view1;
        for(int i=0;i<view.length;i++ ) OsluskujDugmeZbir(i); }

   void OsluskujDugmeZbir(int i)
    {   view[i].BZbir.addActionListener(new mojOsluskivac(this)); }
}




class mojOsluskivac implements ActionListener
{    Controller c;

     mojOsluskivac (Controller c1)
     {c=c1;}
     public void actionPerformed(ActionEvent evt)
        {   int pom = 0;
            Object ob = evt.getSource();
            JButton b = (JButton) ob;
            Container con = b.getParent(); // vraca ime kontejnera forme na kome se nalazi dugme koje je kliknuto
            for(int i1=0;i1<c.view.length;i1++ ) // trazi na kojoj je formi kliknuto dugne
                {  if (con.getName().equals(c.view[i1].getContentPane().getName()))
                        pom = c.view[i1].uzmiVrednost();
                }

            c.mod.sistemskaOperacija(pom);

            for(int i1=0;i1<c.view.length;i1++ )
               {  c.view[i1].postaviVrednost();
               }
        }
}


// Model.java
public class Model
{ int broj;
  Model() {broj = 0;}
  public void sistemskaOperacija(int broj1)   {  broj = broj + broj1; }
  public int uzmiBroj () { return broj;}
}


// MVC2.java
public class MVC2
{
// Glavni program
public static void main(String args[])
 {  Model mod = new Model();
    View[] view = new View[3];
    view[0] = new View(mod,"1");
    view[1] = new View(mod,"2");
    view[2] = new View(mod,"3");
    Controller con = new Controller(view, mod);
 }
}
```

Пример MVC3: /* Кориснички захтев: Направити три екранске форме истог типа које ће имати: а) поље за прихват бројева и б) поље за приказ збира свих до тада унетих бројева. Збир се рачуна кликом на дугме форме. Сваки пут када се промени збир унетих бројева та промена треба да се види на свакој од наведених форми. Овај задатак треба урадити преко аспекта. То значи да када се промени стање модела, аспект треба да обавести контролер да јави екранским формама да је промењено стање. */

```
// View.java
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class View extends JFrame
{  // Isto kao u primeru MVC2

}


// Controller.java
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;


public class Controller
{ Model mod;
  View view[];

   Controller(View[] view1,Model mod1)
                {mod = mod1; view = view1;
                 for(int i=0;i<view.length;i++ ) OsluskujDugmeZbir(i); }

    void OsluskujDugmeZbir(int i)
     {   view[i].BZbir.addActionListener(new mojOsluskivac(this));
     }

    void procitajStanje()
    {  for(int i1=0;i1<view.length;i1++ )
            {   view[i1].postaviVrednost();   }
    }

 }


class mojOsluskivac implements ActionListener
{     Controller c;
       mojOsluskivac (Controller c1)     {c=c1;}
       public void actionPerformed(ActionEvent evt)
         {      int pom = 0;
                Object ob = evt.getSource();
                JButton b = (JButton) ob;
                Container con = b.getParent();
                for(int i1=0;i1<c.view.length;i1++ )
                   {  if (con.getName().equals(c.view[i1].getContentPane().getName()))
                            pom = c.view[i1].uzmiVrednost();
                   }
                c.mod.sistemskaOperacija(pom);
          }
}
```

// Model.java
```
public class Model
{ int broj;
  Model() {broj = 0;}
  public void sistemskaOperacija(int broj1)   {  broj = broj + broj1; }
   public int uzmiBroj () { return broj;}
}
```

// MVC3.java
```
public class MVC3
{  static Controller con;
   static View[] view;
   static Model mod;
// Glavni program
public static void main(String args[])
   {   mod = new Model();
       view = new View[3];
       view[0] = new View(mod,"1");
       view[1] = new View(mod,"2");
       view[2] = new View(mod,"3");
       con = new Controller(view, mod);
   }
}
```

// Asp1.aj
```
public aspect Asp1
 {
     pointcut SO() : execution(public void Model.sistemskaOperacija(int ));

     after() returning : SO() { MVC3.con.procitajStanje();}

}
```

2.3.2.2 Микро архитектура (патерни)

ПА1: Основе о патернима

Сврха патерна и њихово место у процесу развоја софтвера

Патерни или узори (како смо их превели) имају за циљ да нам помогну у одржавању и надоградњи софтверског система. У основи сваког софтверског система налази се нека архитектура. Архитектура се у најопштијем смислу састоји од компоненти које су између себе повезане преко њихових интерфејса. Постоји макро и микро архитектура. **Макро архитектуру** је реализована нпр. преко **ECF** (Enterprise Component Framework) или **MVC** (Model-View-Controler) патерна, док је **микро архитектура** реализована преко узора пројектовања и то: **креационих**, **структурних** и **патерна понашања**. Поред наведених патерна који покривају пре свега фазу пројектовања у развоју софтверског система, постоје и други патерни који покривају и друге фазе у развоју софтвера, као што су патерни захтева, патерни анализе и имплементациони патерни (идиоми) који су везани за конкретне технологије, као што су нпр. Јава или C#.
Патерни пројектовања су независни од технологије, у којој ће бити имплементирани, тако да су они погодни да преко њих схватимо патерне у општем смислу.

Шта је патерн

Када говоримо о патернима и узорима тада о њима у најопштијем смислу можемо да кажемо да они представљају *решења неког проблема, у неком контексту, који се може поново искористити за решавање неких нових проблема.*

То значи да три елемента дефинишу патерн: проблем, решење и контекст. Контекст у суштини дефинише ограничења која морају бити задовољена када се решава неки проблем.

Поновна употребљивост патерна

Једно од основних својстава патерна јесте његова могућност да се може применити у решавању различитих проблема. Како се долази до наведеног својства, или је можда још прецизније питање, *како треба пројектовати софтверску компоненту да се она може применити за различите проблеме*?
Када се направи нека компонента, која решава неки проблем, потребно је направити допунски напор, како би се дошло до општијег или универзалнијег решења које неће бити примењиво само за један проблем, већ ће бити примењиво за један скуп или једну класу проблема.
Шта је оно, чему ми у суштини тежимо када развијамо софтверски систем?
Идеја је у томе, да ми направимо такве софтверске системе или софтверске производе, који ће се лако прилагодити сваком новом корисничком захтеву. То значи да ћемо без велике промене постојеће структуре и понашања софтверског система, моћи да додамо нову или променимо постојећу функционалност софтверског система сходно новим корисничким захтевима.
Идеално би било да имамо параметризован софтверски систем, који се може прилагодити (кастомизовати) различитим доменима проблема. Домен проблема се описује са неким скупом вредности, којима се параметри софтверског система иницијализују, пре покретања софтвера. Треба нагласити да промена вредности наведених параметара не тражи промену програмског кода софтверског система.

Књига Design Patterns

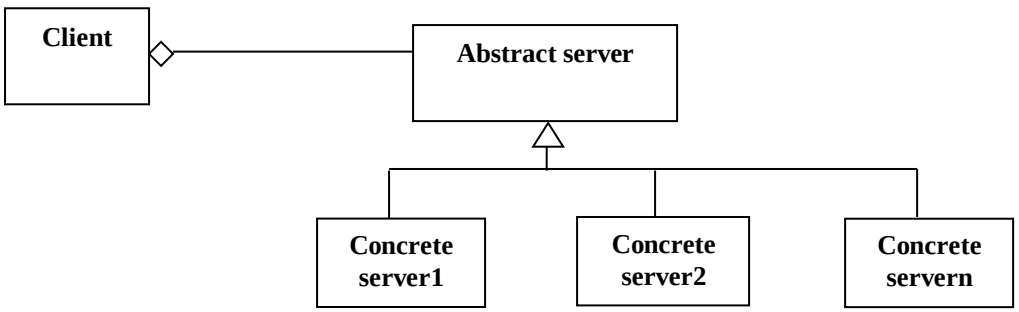
Патерне у општем смислу ћемо објаснити, коришћењем патерна пројектовања. Патерни пројектовања су своју афирмацију доживели са књигом Design Patterns, коју су написали Gamma и група аутора. Ова књига је једна од десет најцитиранијих књига у области софтверског инжењерства и она је поставила темеље схватању улоге патерна у фази пројектовања софтверског система. Ова књига представља аксиом или полазиште приче о патернима.
Књига Design Patterns је направила, по мом скромном мишљењу, кључни искорак у правцу прављења одрживих софтверских система. Када кажем одрживи софтверски системи, онда мислим на софтверске системе који се могу лако одржавати и надограђивати. Због свега тога, топло препоручујем да набавите и прочитате наведену књигу. Ја ћу се потрудити на предавањима, да вам из наведене књиге, објасним све оно, што мислим да је битно, како би схватили кључне идеје и концепте који се налазе у основи патерна пројектовања.

Примењивост патерна у различитим сферама живота

Ако посматрате живот, он се састоји од много процеса. Уколико не желимо да нам живот буде хаотичан, ми треба да управљамо са свим тим процесима. Патерни помажу да се лакше управља различитим животним процесима. Коришћењем патерна се лакше решавају, прате и управљају разни животни процеси. *Шта то значи?*
Када се деси неки проблем, тада се обично нађе неко решење. Поставља се следеће питање: *Шта треба урадити са решењем?* Треба одвојити мало времена и направити од решења, које је обично неко специфично решење, генеричко решење. Тиме добијате могућност да то генеричко решење примените за различите проблеме који се могу десити у животу. Генеричко решење, за разлику од специфичног решења, је непроменљиво у току развоја програма.
Минимално што треба урадити јесте да се опише решење, макар оно било и специфично. Када се у будућности деси неки нови сличан проблем, ви ћете препознати проблем који сте већ решили и потражићете постојеће решење. Оно не мора у потпуности да реши ваш проблем, али сигурно се из тог решења може пронаћи доста делова који се могу користити и код неког новог проблема. Уколико се не забележи решење, тада ћете више пута у животу решавати један те исти, или сличан проблем, сваки пут изнова. На тај начин ћете потрошити пуно више времена и енергије, него да сте запамтили решење и користили га као помоћ у решавању неког новог проблема. Десиће се после годину, две неки *déjà vu* (дежа ви), неки веома познат проблем, који знате да сте решавали, али нећете моћи да се сетите како сте га решили. Или ћете се присећати решења, што је прилично непоуздано и непрецизно. Тако ћете опет изгубити много сати, можда и дана у решавању нечега што сте већ решили. Решења треба по могућству јасно и прецизно написати како би она могла поново да се користе. Нама је циљ да коришћењем патерна са што мање утрошеног времена и енергије, у дужем временском периоду, постигнемо што је могуће већи ефекат.
Прављење генеричких решења дуже траје него што је то случај код специфичних решења. Међутим ефекат тога јесте да ће се нови проблеми решавати све лакше и лакше. Да нагласим, патерни у себи садрже потенцијал да могу да реше скуп проблема или класу проблема а не само један специфичан проблем.

ПА2: Општи облик патерна

У књизи Design Patterns постоје 23 GOF (Gang of Four) патерна пројектовања. Они су подељени у три основне групе: креационе патерне, патерне структуре и патерне понашања. Када се посматра 20 од тих 23 патерна, може да се примети једна структура[30] која постоји код сваког од тих патерна. Та структура у потпуности описује патерн или неки његов део. Наведена структура је **кључни механизам или својство** патерна пројектовања. Управо та структура омогућава лако одржавање и надоградњу програма. Та структура изгледа овако:

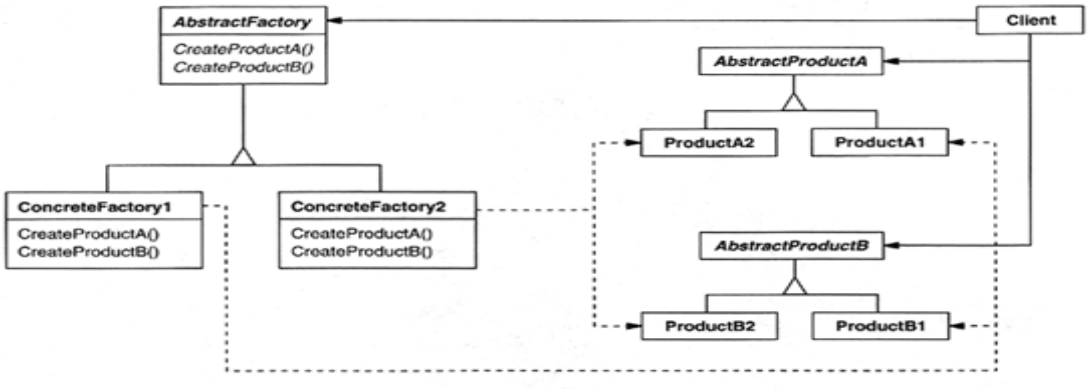


Клијент је везан за апстрактни сервер, док су из апстрактног сервера изведени различити конкретни сервери. У време компајлирања програма клијент се везује за апстрактни сервер[31]. То значи да ће се тек у време извршења програма разрешити који конкретни сервер ће да реализује

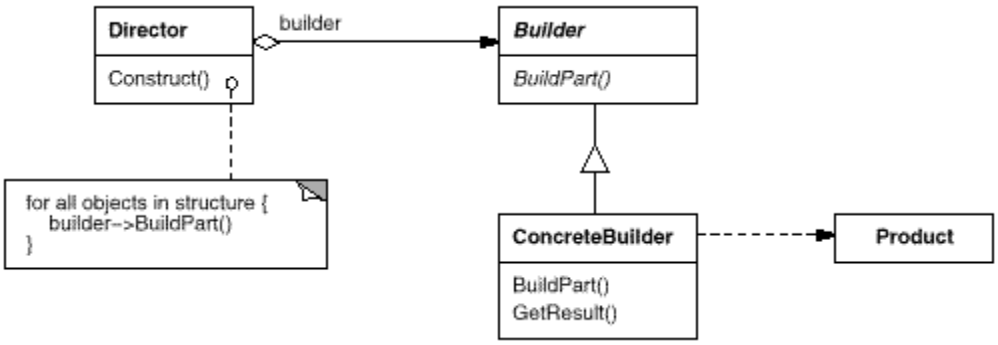
[30] Наведена структура омогућава да софтверски систем буде одржив. Због тога ћемо наведену структуру назвати: **одржива структура (viable structure)**
[31] Апстрактни сервер, уколико програмски посматрамо, може бити интерфејс (interface) или апстрактна класа (abstract class).

захтев клијента. Управо ово повезивање клијента са конкретним сервером у време извршења програма, даје самом програму флексибилност, да један клијентски захтев може бити реализован на различите начине, преко различитих сервера. Такође додавање новог конкретног сервера неће променити клијента.

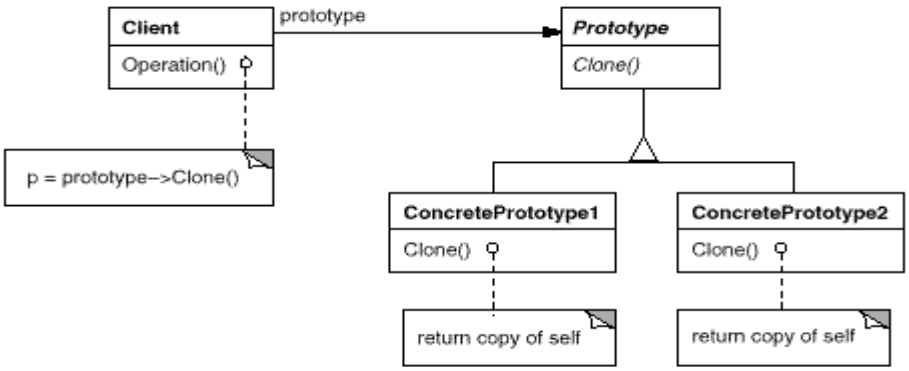
Навешћу неке од патерна чија структура садржи у потпуности или неким делом наведену *одрживу структуру:*
Abstract factory патерн има следећу структуру:



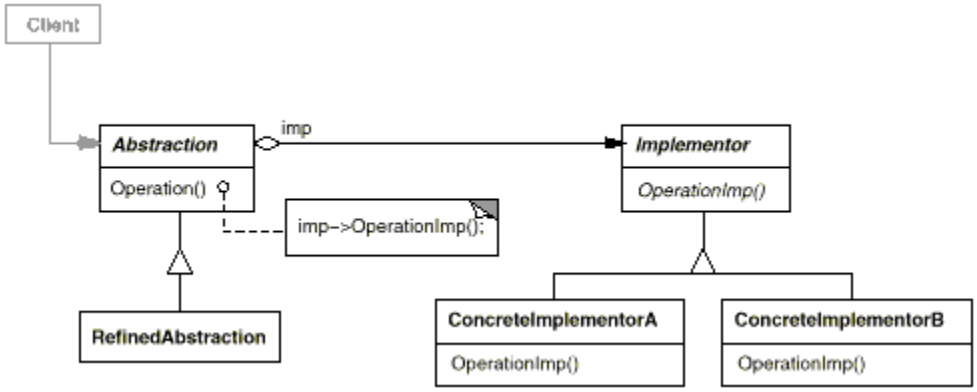
Builder патерн има следећу структуру:



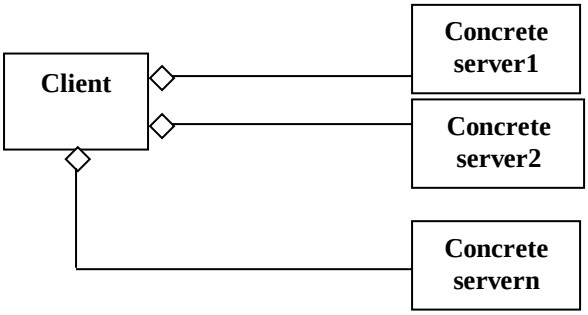
Prototype patern има следећу структуру:



Bridge патерн има следећу структуру:



Наведена одржива структура решава проблем структуре[32] код које је клијент директно повезан са конкретним серверима:



Наведена структура је тешка за одржавање јер се при додавању новог конкретног сервера мења клијент. Такође, клијент се у време компајлирања програма везује за конкретни сервер, што онемогућава флексибилност програма у току његовог извршавања.

Да би смо схватили патерн у општем смислу потребно је да извршимо малу анализу постојећих дефиниција патерна.

Christopher Alexander је рекао[33]:"*Сваки патерн описује **проблем** који се јавља изнова (непрестано) у нашем окружењу, а затим описује суштину **решења** тог проблема, на такав начин да ви можете користити ово решење милион пута, а да никада то не урадите два пута на исти начин.*"

Из наведене дефиниције се може приметити да патерн има два важна дела: проблем и решење. Такође се може видети да патерни имају особину поновљивости, што значи да се решење неког проблема може поновити више пута код другиһ, различитих проблема.

Александар је такође рекао[34]: "*Патерн је, у најкраћем, у исто време **ствар** која се дешава у свету и **правило** које нам говори како се креира та ствар и када се креира та ствар, то је у исто време и **процес** и ствар; описује се ствар која је жива (активна) и описује се процес који генерише ту ствар*."

---

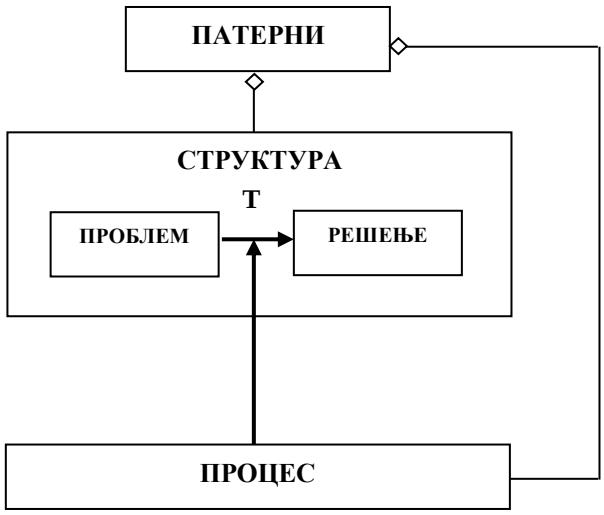
[32] Наведену структуру ћу назвати: **неодржива структура (unviable structure)**

[33] *"Each pattern describes a **problem** which occurs over and over again in our environment, and then describes the core of the **solution** to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.*[AIS]"

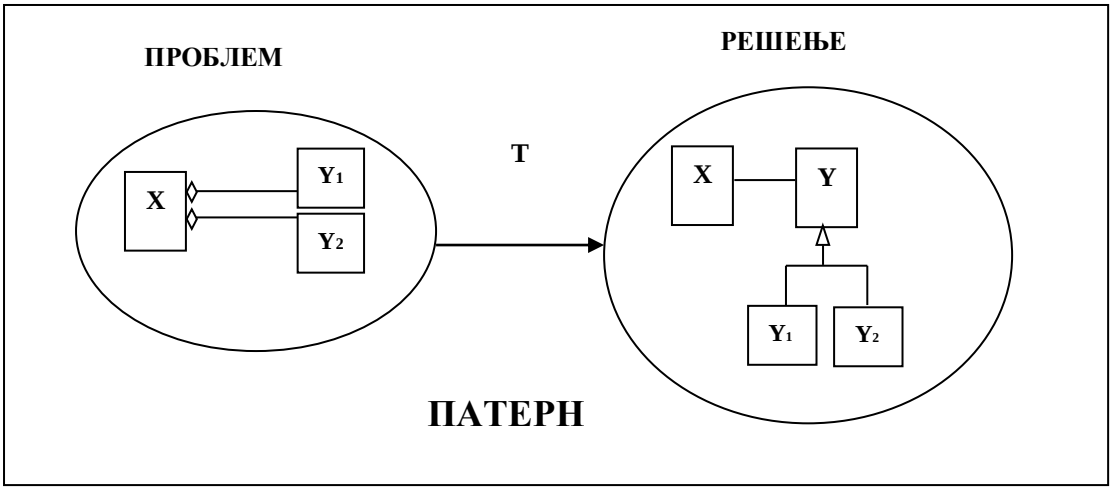
[34] *The pattern is, in short, at the same time a **thing**, which happens in the world, and the **rule** which tells us **how** to create that thing, and **when** we must create it. It is both a **process** and a **thing**; both a description of a thing which is alive, and a description of the process which will generate that thing*

Jim Coplien [Cop1], је такође објашњавајући патерне рекао[35]: "… *то је правило за израду ствари, али је такође, са много поштовања, и сама ствар* ."

Наведени експерти за патерне нам говоре да је патерн у исто време и структура (thing) и процес. Патерн има структуру проблема и решења. Патерн је процес који објашњава **када** и **како** се креира структура решења из структуре проблема.



Патерн би могао да се представи у најопштијем смислу на следећи начин:



Као што смо рекли, проблем и решење имају неку структуру. За структуру проблема смо рекли да је "неодржива структура", док је структура решења "одржива структура" .

Процес описује како се структура проблема трансформише (Т) у структуру решења:

Т
проблем --------> решење

То у суштини значи да је *патерн процес трансформације структуре проблема у структуру решења*. Такође можемо рећи да је *патерн процес који трансформише "неодрживу структуру" у "одрживу структуру"*.

Поставља се питање, када се дешава тај процес?

[35] *"…it is the rule for making the thing, but it is also,in many respect, the thing itself."*

Проблем је, ако би говорили из перспективе развоја софтвера, неки програм који има структуру која је тешка за одржавање и надоградњу.
Када се деси да је структура програма (проблем код патерна) тешка за одржавање и надоградњу, тада се прави нова структура програма (решење код патерна) која је лакша за одржавање и надоградњу.
То значи да се наведени процес трансформације структуре проблема у структуру решења дешава када је структура проблема тешка за одржавање и надоградњу (то је услов који одређује када се дешава трансформација). Ми смо покушали у нашој теорији да формализујемо наведену трансформацију проблема у решење како би схватили шта је суштина патерна[36].

Где треба поставити патерне код софтверских система
Када се нађе решење неког проблема треба препознати шта је у решењу опште а шта специфично, односно шта је непроменљиво а шта је променљиво. Наведена констатација се директно односи на писање програмског кода у току имплементације софтверског система. Уколико се препознају она места у софтверском систему, која су променљива и која се непрекидно мењају са новим корисничким захтевима, ту треба поставити патерне како би се зауставио потенцијални хаос који та променљива места у програму могу да направе. Та места, или те тачке, су по аналогији једнаке бифуркационим тачкама у теорији хаоса. Бифуркационе тачке воде систем у потпуни хаос, јер се у њима нагомилавају различитости. Нешто слично се може десити код развоја софтверских система. Уколико се правовремено не препознају "бифуркационе тачке" софтверског система, систем може да почне ортогонално да развија своју сложеност у односу на постојећи софтверски систем. Та ортогонална сложеност може да потпуно обори постојећи софтверски систем. Патерни се постављају у наведеним тачкама како се не би дозволило да софтверски систем оде у хаос. Ко развија софтвер, треба да има знање, искуство и осећај, да препозна те бифуркационе тачке софтверског система. Ко то препозна, он ће пре свој систем довести у ред и омогући ће да се систем лако одржава и надограђује.
"Шпагети код", појам познат из софтверског инжењерства, где имате више грана у оквиру једне од метода (више if услова), представља пример "бифуркационе тачке" код софтверских система. Поставља се питање, *зашто се дешава "шпагети код"*?
Сваки нови проблем се решава новог граном у оквиру постојеће методе. То доводи то тога да имамо методе које су велике и које се тешко могу одржавати.
Када приметите да се нешто овако дешава у вашем програму (кога ћемо преставити једном линијом), ви у суштини препознајете, тачке хаоса, које теже да сруше ваш програм. У једном тренутку, те тачке хаоса, са њиховом тенденцијом развоја сложености, се ортогонално развијају у односу на ваш програм. У тим тачкама се дешавају непрекидне промене. Те тачке се непрекидно "кувају" и "расту". Ако у тим тачкама поставите патерне, ви ћете зауставити хаос у процесу развоја вашег софтверског система. Уколико не зауставите хаос, вероватно ће, у неком тренутку да вам буде јефтиније да развијате нови софтверски систем, него да одржавате постојећи софтверских систем.
Како препознати тачке које воде софтверски систем у хаос?
Прате се кориснички затеви и њихов утицај на програмски код. Ако приметите да се неки делови програмског кода непропорционално развијају у односу на друге делове програма, то може да буде показатељ да су то потенцијалне тачке хаоса у које треба поставити патерн.
Више грана у програму не значи аутоматски да ту треба поставити патерн. Ако се број грана не мења у току развоја и одржавања програма, тада не треба уводити патерн, јер патерни повећавају сложеност софтверског система.

Раздвајање генералних од специфичних делова програмског кода
Како долази до тога да патерн представља опште решење за класу проблема? Када правимо неки софтверски систем или у ужем смислу неку софтверску компоненту, то радимо на основу неког корисничког захтева. Софтверски систем настаје као резултат процеса развоја софтвера, који пролази кроз све фазе развоја софтвера, која започиње дефинисањем корисничких захтева а завршава се имплементацијом софтверског система. Софтверски систем који је направљен, односно прва верзија софтвера, обично у себи садржи измешане генералне и специфичне делове програмског кода. Генерални делови се могу користити не само за решавање текућег проблема,

[36] На конференцији ASC2011 на Криту у јуну 2011 год. смо изложили рад под насловом: *The Explanation of the Design Patterns by the Symmetry Concepts,* где смо формално коришћењен симетријских концепата објаснили општи облик патерна пројектовања.

већ и за решавање неких других проблема. Специфични делови програмског кода су везани за текући проблем и они се не могу користити за неке друге проблеме. Можемо да кажемо да се генерални (***gen***) и специфични (***spec***) делови програмског кода налазе у једном модулу (***ModulA***), односно у једној логичкој целини програма:

ModulA = (gen + spec)

Наведена измешаност генералних и специфичних делова програма, који се налазе у једном модулу, у суштини представља проблем уколико би покушали да се наведени програм користи у решавању неких других проблема. Због тога се временом, наведени генерални и специфични програмски код раздвајају и постављају се у различите модуле (***ModulB*** и ***ModulC***):

ModulB = (gen)
ModulC = (spec)

Наведени процес, у слободном облику, би могли да представимо на следећи начин:

Lim modulA = modulB + modulC
t-->∞

односно,

Lim (gen + spec) = (gen) + (spec)
t-->∞

На основу наведеног можемо да закључимо да је крајњи циљ или сврха процеса развоја софтвера, изградња софтверског система код кога ће бити одвојени генерални од специфичних делова програмског кода[37]. На тај начин је омогућено да се генерални делови кода могу користити за класу проблема, при чему се специфични делови кода прилагођавају различитим проблемским ситуацијама.

...

Када развијате софтвер, ви ћете временом, свесно или несвесно ићи у смеру раздвајања генералног од специфичног кода. То је тенденција сваког искусног програмера који иза себе има много развијених софтверских система. Ова предавања и концепт патерна имају за циљ да схватите који је то механизам, који сваки програмер временом схвати, али је велико питање које је то време (које може ићи од неколико месеци до неколико година) за које ће он то схватити. Ми желимо да вам помогнемо, да пре него што постанете искусни програмери, знате концепт патерна, који ће вам пуно помоћи да схватите суштину одрживих софтверских система.

Да поновимо шта је суштина наведене приче: - У почетку имамо програм код кога су измешани генерални и специфични делови програмског кода, који се налазе у једном модулу. То значи да имамо програм код кога су измешани различити нивои апстракције[38].Како пролази време и како се појављују нови кориснички захтеви, генерални и специфични делови програма се раздвајају и смештају у различите модуле. Идеално би било да софтверски систем има само генералне делове програмског кода, док би специфични делови програмског кода требало да буду, "потиснути" из програма и смештени у датотеке односно табеле базе података. Тако би дошли до параметризираног софтверског система, који се прилагођава (кастомизује) различитим проблемима, променом датотеке или табеле (у којој се налазе параметри) који се иницијализује са вредностима којима се дефинише специфичан проблем[39]. То значи да уколико се деси нови проблем, улази се у наведене датотеке и табеле, и оне се пуне са вредностима који дефинишу тај

[37] Код *Јединственог процеса развоја софтвера (Unified Software Development Process)* која представља методу развоја софтвера јасно су раздвојени генерални од специфичних слојева код архитектуре софтверског система.

[38] Ако непрекидно мењате нивое апстракције код излагања, ваше излагање је нејасно, јер обично губите основни ток мисли, непрекидно понирете у детаље, остајете на њима и заборављате шта сте почели да објашњавате. Излаз из наведеног проблема јесте држања једног нивоа апстракције код објашњавања и избегавање да се превише улази у детаље. Ако се улази у детаље то треба урадити тако да се не изгуби и занемари основни ток (нит) размишљања.

[39] Пример за то је локализација неког софтверског система, којом се врши прилагођавање софтверског система различитим светским језицима.

проблем. Наглашавам оно што је генеричко, то је непроменљиво и на тај део програмског кода не утичу нови проблеми.
Шта добијамо као резултат коришћења патерна?
Патерни омогућавају да генерални и специфични делови софтверског система буду раздвојени.

Недостаци софтверског система при увођењу патерна
Увођењем патерна, повећава се сложеност софтверског система и смањује се брзина извршења програма јер се уводе нови слојеви[40] и нивои[41] у архитектури софтверског система[42]. Поред тога, увођење нивоа и слојева у архитектури отежава тестирање и контролу извршења програма(debug програма). Овај проблем је посебно наглашен када су различити нивои и слојеви архитектуре реализовани различитим технологијама. Као што се види, патерни обарају неке перформансе софтверског система, али са друге стране повећавају лакоћу одржавања и надоградње софтверског система. Наведени проблеми су пре свега технички проблеми и сматрам да ће током времена брзина рачунара бити довољно велика да се неће приметити значајна разлика између апликација које имају различит број нивоа и слојева.

Објашњење механизма одрживе структуре програма

Када смо правили математички формализам за опис патерна пажњу смо усмерили на то да формално опишемо патерне, односно да направимо језик којим ћемо описати патерне. У основи тог формализма су се налазили ***симетријски концепти*** и то симетријска трансформација и симетријска група. Они су нам у суштини помогли да схватимо који је то механизам у патернима који омогућава да структура програма буде одржива. Математички смо показали да сваки пут када се појави несиметријска група коју образују клијент и конкретни сервери), што представља проблем код патерна, уводе се две релације које представљају решење код патерна:
а) симетријска трансформација између клијента и апстрактног сервера.
б) симетријска група коју образују апстрактни сервер и конкретни сервери.
То значи, концептуално гледајући, да *одржива структура настаје тако што се несиметријска структура трансформише у симетријску структуру*. ***Патерни, у суштини представљају процес трансформације несиметријске у симетријску структуру.*** Понављам, симетријска структура је одрживија у односу на несиметријску структуру, јер се таква структура лакше одржава и надограђује.

Дефиниција **одрживости**: *способност нечега да расте и да се развија; способност нечега да живи (да траје).*
Ако бисмо дефиницију одрживости применили на софтвер онда би могли рећи да је одржив софтвер онај софтвер који може да расте и да се развија. Да би софтвер могао да се развија он мора да се:
а) **одржава**, да би обезбедио или променио постојећу функционалност и да се
б) **надограђује**, како би обезбедио допунске функционалности.
Патерни обезбеђују структуру која је одржива, односно структуру која може лако да се одржава и надограђује. Патерни обезбеђују механизам који ће структуре које су тешке за развој (неодрживе структуре) трансформисати у структуре које се могу лако развијати (одрживе структуре). Неодрживе структуре имају краћи век трајања од одрживих структура и веома се брзо деле и доводе систем у хаотично стање. У њих мора да се улаже велика спољна енергија како би систем могао да функционише и да се развија. Код одрживих система се улаже мања спољашња енергија код одржавања и надоградње система.
Код одрживих система додавање или промена неког елемента структуре неће утицати на остале елементе структуре. Код неодрживих структура, додавање или промена неког елемента структуре утиче на промену других елемената структуре. што чини да је дата структура непостојана и нестабилна. Међузависност елемената система је превише велика и такви системи су тешки за одржавање.

[40] Слојеви су хијерархијски организовани, при чему се на врху хијерархије налази најапстрактнији слој, док се на дну хијерархије налази најконкретнији слој.
[41] Макро патерни као што је нпр. MVC или микро патерн facade уводе допунске нивое у архитектури софтверског система.
[42] *Butler Lampson* је рекао: "Сви проблеми у рачунарству могу бити решени другим нивоом индирекције (All problems in computer science can be solved by another level of indirection) ". Наведена констатација је речена у ироничном смислу, али она посредно говори да свако побољшање неке перформансе софтверског система (у овом случају одржавање система) са једне стране, обара неке друге перформансе (у овом случају повећава се сложеност и брзина извршења софтверског система.)

Патерни, ред и хаос

У сваком систему различитости имају тенденцију да се остваре. Уколико две или више различитости не пронађу заједнички именитељ (заједничке особине) који ће их окупити, оне ће имати тенденцију да се даље деле (унутар њих самих) што води ка хаосу и нереду. То значи да ће се **апсолутни хаос** десити уколико се све различитости (до најситнијих различитости) у некој појави остваре. Патерни имају механизам који не дозвољава да систем уђе у апсолутни хаос. Патерни уводе одрживе структуре на местима које могу систем да уведу у хаос. Систем ће ући у хаос ако се дозволи да различитости (различите вредности) доминирају у односу на заједништво (заједничке вредности). Различитост има тенденцију да наруши заједништво. Заједништво има тенденцију да неутралише различитост. Систем ће постати тоталитаран ако се дозволи да заједништво неутралише различитости, односно да ред постане апсолутан, јер би тада имали тоталитарни систем који не прихвата различитости. Не треба заустављати почетак настанка неке различитости[43] јер се тиме спречава и успорава развој система. Хаос и ред се непрекидно смењују и то је нормалан процес у развоју било ког система. *Патерни држе хаос и ред у непрекидној равнотежи и не дозвољавају да било ко од њих постане апсолутан.*
...

Патерни описују процес у коме систем никада неће отићи у апослутни хаос или апсолутни ред. Патерни омогућавају да систем из реда пређе у "ограничени хаос" како би се десиле различитости које прилагођавају систем његовом окружењу. Такође патерни омогућавају да систем из хаоса пређе у "ограничени ред" како би се десило заједништво које јача систем изнутра. Хаос са једне стране слаби систем али га са друге стране чини прилагодљивијим окружењу. Ред са једне стране јача систем али га са друге стране чини крутим и мање прилагодљивим окружењу.

Из наведеног изводим следећу хипотезу: *Патерни омогућавају да производ реда и хаоса буде увек нека константа rh:*

rh = Red * Haos

ПАЗ: GOF узори пројектовања

GOF узори се користе у фази пројектовања софтверског производа. Они помажу у именовању и опису **генеричких решења**, која могу бити примењена у различитим проблемским ситуацијама.

Код развоја софтвера, уопштено гледајући, прво треба да се схвати и разуме разматрани проблем (систем). Затим се врши његова анализа, да би се на крају вршило његово пројектовање и имплементација. У току пројектовања уочавају се класе пројектовање. Уколико се жели да наведене класе буду **флексибилне** (у смислу њиховог једноставног одржавања и надоградње), њих треба организовати помоћу узора пројектовања. Узоре треба користити и онда када се жели **динамичка измена функционалности** програма у току његовог извршавања.
Узори пројектовања омогућавају флексибилност класа и динимичку измену функционалности али истовремено они повећавају **сложеност** система.

Узори су подељени у 3 групе:

- **Креациони** узори помажу да се изгради систем независно од тога како су објекти, креирани, компоновани и репрезентовани.
- **Структурни** узори описују сложене структуре међусобно повезаних класа и објеката.
- **Узори понашања** описују начин на који класе или објекти сарађују и распоређују одговорности.

[43] Право на различитост не подразумева наметање те различитости другима. То се посебно односи на оне који не подржавају ту различитост. Треба разликовати подржавање права на различитост од подржавања различитости. Неко може да подржи право некога да буде различит и у исто време да лично не подржи ту различитост. Неко може да подржи нечије право да се бори за различитост а у исто време да не подржи ту различитост. Право на различитост није право на наметање различитости. Ако неко има право на различитост, он нема право да намеће другима ту различитост. Нпр. Свако има право да слуша музику у своме стану, али нема право да појача ту музику и да је намеће другима који не воле (не подржавају) ту музику.

ПА3.1: Узори за креирање објеката (Creational patterns)

Узори за креирање објеката апстракују процес инстанцијације (instantiation process), односно процес креирања објеката. Они дају велику прилагодљивост (flexibility) у томе шта (what) ће бити креирано, ко (who) ће то креирати, како (how) ће то бити креирано и када (when).

Постоје следећи узори за креирање објеката:

1. ***Abstract Factory*** - Обезбеђује интерфејс за креирање фамилије повезаних или зависних објеката без навођења (специфицирања) њихових конкретних класа.

2. ***Builder*** - Дели конструкцију сложеног (комплексног) објекта од његове репрезентације, тако да исти конструкциони процес може да креира различите репрезентације.

3. ***Factory Method*** - Дефинише интерфејс за креирање објеката, али омогућава подкласама да донесу одлуку коју класу ће истанцирати. Фацтору метход преноси надлежност инстанцирања објеката са класе на подкласе.

4. ***Prototype*** - Одређује (специфицира) врсте објеката које ће бити креиране коришћењем прототипског појављивања и креира нове објекте копирањем тог прототипа.

5. ***Singleton*** - Обезбеђује класи само једно појављивање и глобални приступ до ње.

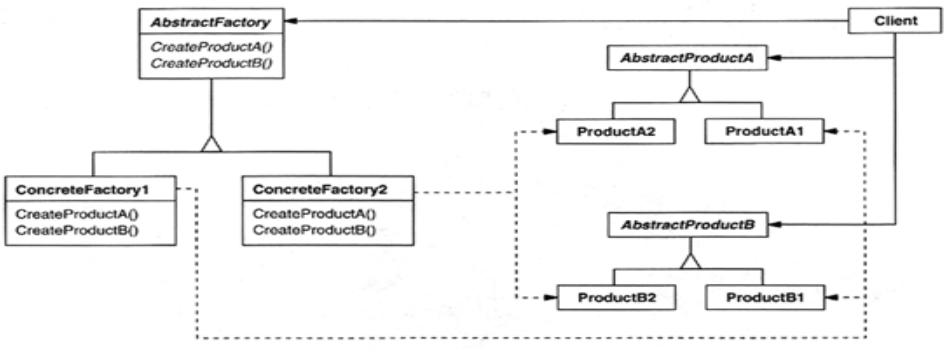
У ниже наведеном тексту даћемо детаље везане за неке од патерна.

ПК1: Abstract Factory патерн

Дефиниција:
Обезбеђује интерфејс за креирање фамилије повезаних или зависних објеката без навођења (специфицирања) њихових конкретних класа.

Појашњење дефиниције:
Обезбеђује интерфејс (AbstractFactory) за креирање (CreateProductA(), CreateProductB()) фамилије повезаних или зависних објеката (AbstractProductA, AbstractProductB) без навођења (специфицирања) њихових конкретних класа (ProductA1, ProductA2,ProductB1,ProductB2).

Структура[44] factory патерна:



Учесници:

- **AbstractFactory**
  Декларише интерфејс за операције које креирају <<производе>>.
- **ConcreteFactory**
  Имплементира операције <<AbstractFactory>> интерфејса, којима се креирају конкретни <<производи>>.
- **AbstractProduct**
  Декларише интерфејс за <<производе>>.

[44] Када је писана књига Design Patterns коришћена је Booch-ова нотација код описа дијаграма класа. Наведену нотацију ми смо преузели код спецификације GOF узора.

- **ConcreteProduct**
  - Дефинише <<производе>> који ће бити креирани преко конкретних <<faktory-a>>.
  - Имплементира операције <<AbstractProduct>> интерфејса.
- **Client**
  Користи <<AbstractFactory>> и <<AbstractProduct>> класе (интерфејсе) за креирање <<финалног производа>>.

Пример AbstractFactory патерна:

Кориснички захтев PAF1*: Управа Факултета је послала захтев Лабораторији за софтверско инжењерство да пошаље елементе понуде за израду софтверског система последипломских студија ФОН-а:*
а) Програмски језик у коме ће се развијати програм.
б) Систем за управљање базом података у коме ће се чувати подаци.
Након прихватања елемената понуде Управа Факултета ће направити (саставити) понуду у целини[45].

Уколико посматрамо кориснички захтев можемо приметити да Управа тражи од Лабораторије за С.И. да креира елементе понуде, односно програмски језик и СУБП у којима ће се реализовати софтверски систем. Лабораторију за СИ (SILAB) можемо представити на следећи начин:

```
// Улога: Декларише интерфејс за операције које креирају <<елементе понуде >>.
interface SILAB // AbstractFactory
{ ProgramskiJezik kreirajProgramskiJezik();
  SUBP kreirajSUBP();
}
```

Интерфејс *SILAB* је одговоран за дефинисање операција *kreirajProgramskiJezik* и *kreirajSUBP* помоћу којих се креирају елементи понуде[46].
У наведеном интерфејсу се може приметити да операције *kreirajProgramskiJezik* и *kreirajSUBP* враћају програмски језик и СУБП, који ће такође бити представљени преко интерфејса:

```
/*Улога: Декларише интерфејс за <<елементе понуде>>. */
interface ProgramskiJezik  // AbstractProductA
{String vratiProgramskiJezik();}

interface SUBP // AbstractProductB
{String vratiSUBP();}
```

Сви елементи захтева за понудом (понуда, програмски језик, СУБП) представљени су преко интерфејса[47].

Лабораторија за софтверско инжењерство је направила два тима. Један је оријентисан ка Јави, док је други оријентисан ка VB-у. Оба тима треба да одреде конкретан програмски језик и СУБП који ће дати у понуди[48].

```
/* Улога: Имплементира операције <<SILAB>> интерфејса, којима се креирају конкретни <<елементи понуде>>.*/
class JavaTimPonuda implements SILAB // ConcreteFactory1
{ public ProgramskiJezik kreirajProgramskiJezik(){return new Java();}
  public SUBP kreirajSUBP() {return new MySQL();}
}
```

---

[45] **Шта** ће бити креирано: **Понуда**
Ко ће креирати понуду: **Управа**
Како ће се креирати понуда: видети методу *Kreiraj()*.
Када ће се креирати понуда: видети методу *main()*.

[46] AbstractFactory интерфејс (*SILAB*) преноси одговорност за креирање објеката до његових ConcreteFactory подкласа (*JavaTimPonuda*, *VBTimPonuda*).

[47] Интерфејс је концепт у ОО програмирању који дефинише шта треба да се ради, без улажења у имплементацију, како ће то да се уради. Класе које имплементирају интерфејс су одговорне за имплементацију операција интерфејса.

[48] Сваки конкретан faktory има посебну (специфичну) имплементацију код креирања производа.

```
class VBTimPonuda implements SILAB // ConcreteFactory1
{ public ProgramskiJezik kreirajProgramskiJezik(){return new VB();}
  public SUBP kreirajSUBP() {return new MSAccess();}
}
```

У методама *kreirajProgramskiJezik()* и *kreirajSUBP()* су креирани конкретни програмски језици: *Java* и *VB:*

```
/* Улоге: а) Дефинише <<елементе понуде (Java,VB,MySQL и MSAccess)>> који ће бити креирани
              преко конкретних <<тимова за понуде(JavaTimPonuda и VBTimPonuda)>>.
          б) Имплементира операције <<ProgramskiJezik i SUBP>> интерфејса. */
class Java implements ProgramskiJezik // Product A1
{ public String vratiProgramskiJezik(){return "Java";}}

class VB implements ProgramskiJezik // Product A2
{ public String vratiProgramskiJezik(){return "VB";}}
```

као и конкретни СУБП: *MySQL* и *MSAccess*:

```
class MySQL implements SUBP // Product B1
{ public String vratiSUBP(){return "MySQL";}}

class MSAccess implements SUBP // Product B2
{ public String vratiSUBP(){return "MS Access";}}
```

На крају управа Факултета (Client[49]) приказује обе понуде преко *main* методе класе *UpravaFakulteta.* У методи *kreiraj()* се прави конкретна понуда, прво за Јава тим а после тога и за VB тим.

```
// Улога: Користи <<Ponuda>> и <<ProgramskiJezik и SUBP>> интерфејсе за креирање <<понуде>>.
class UpravaFakulteta  // Client
{
  SILAB sil; // Abstract Factory
  UpravaFakulteta(SILAB sil1){sil = sil1;}

   public static void main(String args[])
            {  UpravaFakulteta uf;
               JavaTimPonuda jat = new JavaTimPonuda();  // ConcreteFactory1
               uf = new UpravaFakulteta(jat);
               System.out.println("Ponuda java tima: " + uf.Kreiraj());

                VBTimPonuda vbt = new VBTimPonuda(); // ConcreteFactory2
                uf = new UpravaFakulteta(vbt);
                 System.out.println("Ponuda VB tima: " + uf.Kreiraj());
            }

      String Kreiraj()[50]
            {   ProgramskiJezik pj = sil.kreirajProgramskiJezik();
                SUBP subp = sil.kreirajSUBP();
                return "Programski jezik-" + pj.vratiProgramskiJezik() + "  SUBP-" + subp.vratiSUBP();
            }
}
```

Метода *kreiraj()* је генерички урађена јер је креирање програмског језика и СУБП-а везано за операције интерфејса. Управа факултета (Client) креира понуду преко апстрактног интерфејса (*SILAB*) и она не види како конкретне класе креирају објекте. Ово је добар пример једног од главних принципа поновног коришћења програмског кода у ОО пројектовању:

Програмирати према интерфејсу а не према имплементацији (Program to an interface, not an implementation [GOF, стр.18])

Предности AbstractFactory узора

Предности AbstractFactory узора се огледају у томе што додавање нове ConcreteFactory класе (у нашем случају то је нови тим који даје понуду, нпр: CTimPonuda) не захтева промену у постојећим класама и интерфејсима.

---

[49] Klijent treba da koristi različite konkretne faktore kada želi da kreira različite proizvode.

[50] 1. Управа је одговорна за контролу креирања понуде.
2. Тимови су одговорни за креирање елемената понуде.
3. Управа је одговорна за састављање понуде у целини.

```
class CTimPonuda implements SILAB // novi ConcreteFactory
{ public ProgramskiJezik kreirajProgramskiJezik(){return new C();}
  public SUBP kreirajSUBP() {return new Oracle();}
}
```

Недостаци AbstractFactory узора
Тешко се додају нове врсте производа до AbstractFactory узора. То је због тога што AbstractFactory (Ponuda) интерфејс има фиксан скуп производа који може да креира (Programski jezik и SUBP). Увођење нове врсте производа (нпр. OperativniSistem) захтева проширење интерфејса AbstractFactory класе и промену свих њених подкласа.

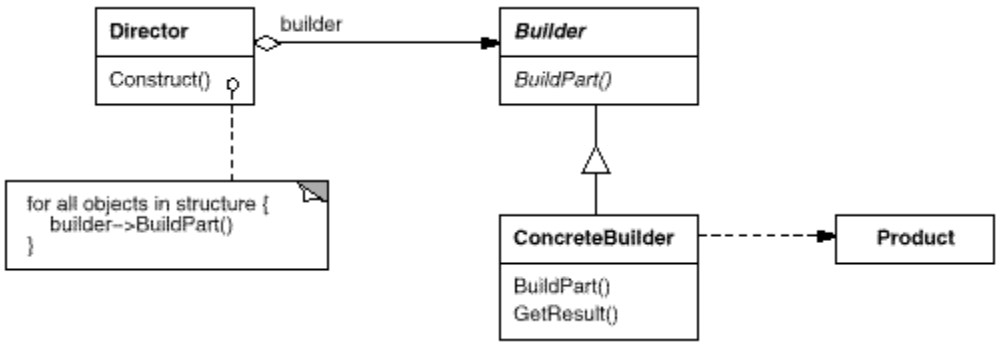
ПК2: Builder патерн

***Дефиниција*:**
Дели конструкцију сложеног (комплексног) објекта од његове репрезентације, тако да исти конструкциони процес може да креира различите репрезентације.

Појашњење дефиниције:
Дели одговорност за контролу конструкције (*Director*) сложеног (комплексног) објекта од одговорности за реализацију његове репрезентације-конкретне конструкције (*Builder*), тако да исти конструкциони процес (*Direktor.Construct()*) може да креира различите репрезентације конкретне конструкције (*ConcreteBuilder.BuildPart()*) у зависности од ConcreteBuilder-а.

Напомена: Код GOF дефиниције појам репрезентација указује на конкретну конструкцију (производ) који ће се добити као резултат контрукционог процеса.

Структура Builder патерна:



Учесници:

- Builder
  Специфицира апстрактни интерфејс за креирање <<делова производа[51]>>.
- ConcreteBuilder
  Конструише и групише <<делове производа>> имплементирајући <<Builder>> интерфејс.
  Обезбеђује интерфејс за узимање <<производа>>.
- Director
  Конструише објекат (<<производ>>) коришћењем <<Builder>> интерфејса.
  Контролише конструкцију <<производа>> коришћењем <<Builder>> интерфејса.
- Product
  - Репрезентује сложен (комплексан) објекат (<<производ>>) који се конструише.
  - Укључује класе (интерфејсе) који дефинишу <<делове производа>>, укључујући интерфејсе за груписање делова у финални резултат (<<производ>>).

[51] На слици то је Product.

ПА3.2: Структурни патерни

Структурни патерни описују сложене структуре међусобно повезаних класа и објеката.

Постоје следећи структурни патерни:
1. Adapter патерн - Адаптер патерн конвертује интерфејс неке класе у други интерфејс који клијент очекује. Другим речима, он прилагођава некомпатибилне интерфејсе.
2. Bridge патерн - Одваја (декуплује) апстракцију од њене имплементације тако да се оне могу мењати независно.
3. Composite патерн - Објекти се састављају (компонују) у структуру стабла како би представили хијерархију целине и делова. Composite узор омогућава да се једноставни и сложени (компоновани) објекти третирају јединствено.
4. Decorator патерн - Придружује одговорност до објекта динамички. Декоратор проширује функционалност објекта динамичким додавањем функционалности других објеката.
5. Facade патерн - Обезбеђује јединствен интерфејс за скуп интерфејса неког подсистема.
Facade узор дефинише интерфејс високог нивоа који омогућава да се подсистем лакше користи.
6. Flyweight патерн - Користи дељење да ефикасно подржи велики број објеката.
7. Proxy патерн - Обезбеђује посредника за приступање другом објекту како би се омогућио контролисани приступ до њега.

СП1: Адаптер патерн

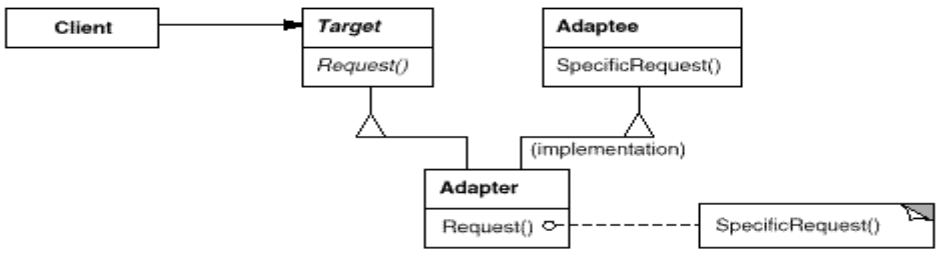
Дефиниција

Адаптер патерн конвертује интерфејс неке класе у други интерфејс који клијент очекује. Другим речима, он прилагођава некомпатибилне интерфејсе.

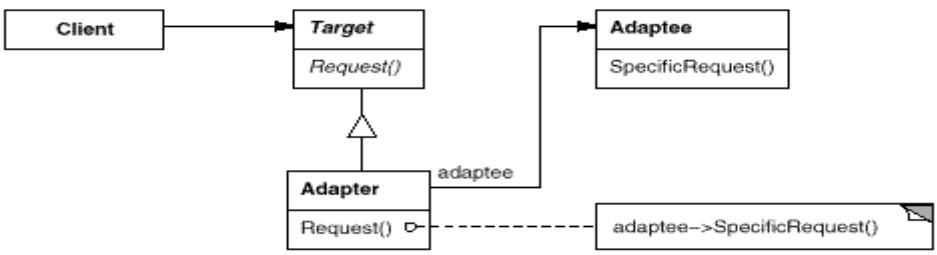
Појашњење ГОФ дефиниције:

Адаптер патерн конвертује интерфејс неке класе (Adaptee) у други интерфејс који клијент (Client) очекује (Target). Другим речима, он прилагођава некомпатибилне интерфејсе(Adaptee, Target) помоћу класе Adapter.

Наводимо **структуру** adapter узора, која се може јавити у два облика:
а) Класа Адаптер користи вишеструко наслеђивање код прилагођавања некомпатибилних интерфејса.



б) Класа Адаптер користи композицију код прилагођавања некомпатибилних интерфејса.



Учесници:

- **Target**

Дефинише доменски <<специфичан интерфејс>> који Client користи.

- **Client**

Позива операције преко жељеног интерфејса класе <<Target>>.

- **Adaptee**

Дефинише <<постојећи интерфејс>> који треба адаптирати.

- **Adapter**

Адаптира интерфејс <<Adaptee-а>> према <<Target>> интерфејсу.

Пример Adapter paterna:

*// **Кориснички захтев PAD1:** Управа Факултета је послала захтев Лабораторији за софтверско инжењерство да промени свој интерфејс, тако што ће имена операција kreirajProgramskiJezik(), kreirajSUBP(), kreirajPonudu() и vratiPonudu() променити у KrProgramskiJezik(),KrSUBP(),KrPonudu() и VrPonudu(). Тимови који праве понуде треба да прилагоде (адаптирају) стари интерфејс (SILAB) новом интерфејсу (SILABTarget), који очекује Управа Факултета без промене, старог интерфејса.*

```
//Улога: Дефинише доменски интерфејс <<SILABTarget>> који Client користи.
interface SILABTarget   // Target
{ void KrProgramskiJezik();
  void KrSUBP();
  void KrPonudu();
  String VrPonudu();
}

//Улога: Адаптира интерфејс <<SILAB>> према <<SILABTarget>> интерфејсу.*/
class Adapter implements SILABTarget   // Adapter
{ SILAB sil;
  Adapter(SILAB sil1) {sil=sil1; }
  public void KrProgramskiJezik(){sil.kreirajProgramskiJezik();}
  public void KrSUBP(){sil.kreirajSUBP();}
  public void KrPonudu() {sil.kreirajPonudu();}
  public String VrPonudu(){return sil.vratiPonudu();}
}

// Улога: Позива операције преко жељеног интерфејса << SILABTarget >>.
class UpravaFakulteta   // Client
{
  SILABTarget silta;

  UpravaFakulteta(SILABTarget silta1){silta = silta1;}

  // Kontroliše konstrukciju <<ponude>> korišcenjem interfejsa SILABTarget.
  void Konstruisi()
   { silta.KrProgramskiJezik();
     silta.KrSUBP();
     silta.KrPonudu();
   }

  public static void main(String args[])
            {  UpravaFakulteta uf;
               SILABTarget silta;
               JavaTimPonuda jat = new JavaTimPonuda();
               silta = new Adapter(jat);
               uf = new UpravaFakulteta(silta);
               uf.Konstruisi();
               System.out.println("Ponuda java tima: " + jat.vratiPonudu());

               VBTimPonuda vbt = new VBTimPonuda();
               silta = new Adapter(vbt);
               uf = new UpravaFakulteta(silta);
               uf.Konstruisi();
               System.out.println("Ponuda VB tima: " + vbt.vratiPonudu());
  }
}



// ***********************************************************
// Улога: Дефинише интерфејс <<Ponuda>> који треба адаптирати.
interface SILAB // Adaptee
{ void kreirajProgramskiJezik();
  void kreirajSUBP();
  void kreirajPonudu();
  String vratiPonudu();
```

```
}

/*Улоге: а) Репрезентује сложену  <<понуду>>  која се конструише.
         б) Укључује класе (интерфејсе) које дефинишу  <<елементе понуде>> */
class PonudaS { ProgramskiJezik pj;  SUBP subp;}

/* Улоге: а) Конструише и групише  <<елементе понуде>> имплементирајући
            <<Ponuda>>интерфејс.
         б) Обезбеђује интерфејс за узимање <<понуде>>. */
class JavaTimPonuda implements SILAB // ConcreteBuilder1
{  // Обезбеђује интерфејс за узимање <<понуде>>.
  PonudaS elpon; // elementi ponude
  String ponuda;
  JavaTimPonuda() {elpon = new PonudaS();}
  // Конструише и групише делове <<понуде>>.
  public void kreirajProgramskiJezik(){elpon.pj = new Java();}
  public void kreirajSUBP() {elpon.subp = new MySQL();}
  public void kreirajPonudu() { ponuda = "Programski jezik-" + elpon.pj.vratiProgramskiJezik() + "
                                SUBP-" + elpon.subp.vratiSUBP();}
  public String vratiPonudu(){return ponuda;}
}

class VBTimPonuda implements SILAB // ConcreteBuilder2
{ PonudaS elpon;
  String ponuda;
  VBTimPonuda(){elpon = new PonudaS();}
  public void kreirajProgramskiJezik(){elpon.pj = new VB();}
  public void kreirajSUBP() {elpon.subp = new MSAccess();}
  public void kreirajPonudu() { ponuda = "Programski jezik-" + elpon.pj.vratiProgramskiJezik() + "
                                     SUBP-" + elpon.subp.vratiSUBP();
  public String vratiPonudu(){return ponuda;}
}

// Наведени интерфејси и класе су преузети из примера за Abstract Factory узор.
// ***********************************************************
interface ProgramskiJezik
{String vratiProgramskiJezik();}

class Java implements ProgramskiJezik
{ public String vratiProgramskiJezik(){return "Java";}}

class VB implements ProgramskiJezik
{ public String vratiProgramskiJezik(){return "VB";}}

// ***********************************************************

interface SUBP // AbstractProductB
{String vratiSUBP();}

class MySQL implements SUBP
{ public String vratiSUBP(){return "MySQL";}}

class MSAccess implements SUBP
{ public String vratiSUBP(){return "MS Access";}}

// ***********************************************************
```

Коментар примера:

- **Разлика између креационих, структурних и узора понашања се огледа у њиховој** одговорности**. Креациони узори су одговорни за процес креирања објеката. Структурни узори су одговорни за рад са композицијом класа или објеката. Узори понашања су одговорни за сарадњу између класа или објеката.**
- Напомена: Обично узоре почињемо да примењујемо у току одржавања и надоградње програма. У случају Адаптер патерна постојећи интерфејс (SILAB) се прилагођава новом интерфејсу (SILABTarget).
- Управа факултета је била одговорна код Builder узора за процес прављења понуде. Она је позивала методу:

```
void Konstruisi()
  { sil.kreirajProgramskiJezik();
    sil.kreirajSUBP();
    sil.kreirajPonudu();
  }
```

- Управа Факултета тражи од Лабораторије за софтверско инжењерство да прилагоди интерфејс **SILAB:**

```
interface SILAB // Adaptee
{ void kreirajProgramskiJezik();
  void kreirajSUBP();
  void kreirajPonudu();
  String vratiPonudu();
}
```

sa интерфејсом **SILABTarger**:

```
interface SILABTarget    // Target
{ void KrProgramskiJezik();
  void KrSUBP();
  void KrPonudu();
  String VrPonudu();
}
```

- То се постиже помоћу Адаптера:

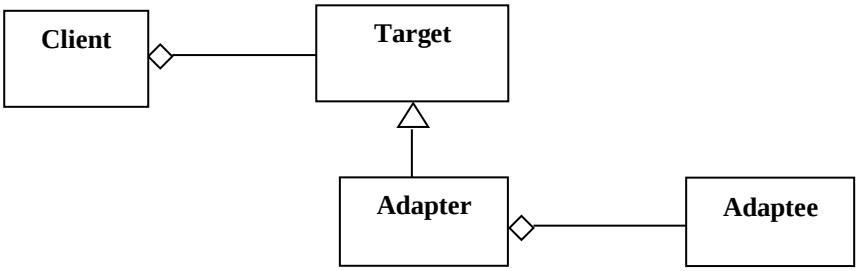
```
class Adapter implements SILABTarget   //Adapter
{ SILAB sil;
  Adapter(SILAB sil1) {sil=sil1; }
  public void KrProgramskiJezik(){sil.kreirajProgramskiJezik();}
  public void KrSUBP(){sil.kreirajSUBP();}
  public void KrPonudu() {sil.kreirajPonudu();}
  public String VrPonudu(){return sil.vratiPonudu();}
}
```

Адаптер прилагођава два различита интерфејса (SILABTarget и SILAB).

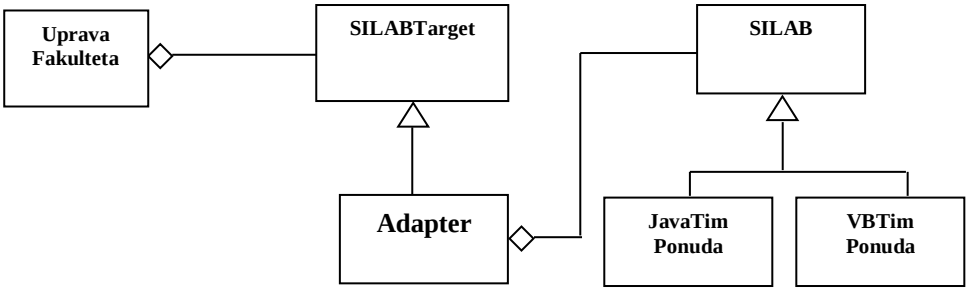
- Управа Факултета ће сада позивати *Konstruisi* методу новог интерфејса.

```
void Konstruisi()
   { silta.KrProgramskiJezik();
     silta.KrSUBP();
     silta.KrPonudu();
   }
```

- Адаптер узор има следећу структуру:



- У конкретном случају Адаптер узор има следећу структуру:



- Клијент је поставио захтев да се промени интерфејс. Треба да се постигну два циља помоћу Адаптер узора:
  а) Да се прилагоди постојећи интерфејс клијентовом.
  б) Не треба да мењамо наш постојећи интерфејс и класе које реализују интерфејс ако они могу да обезбеде жељену функционалност.

- Било би веома лоше када би смо на сваки захтев клијента мењали наш интерфејс. Интерфејс има смисла да се мења ако му се додаје нова функционалност.

- Пример Адаптер узора можемо представити у случају односа између:
  а) Наредби Јаве (Target) и Оперативног система (Adaptee) који се прилагођавају помоћу JVM (Adapter). Наредбе Јаве се пресликавају у наредбе конкретног Оперативног система.
  b) Наредби Јаве (Target) и SUBP (Adaptee) који се прилагођавају помоћу драјвера (Adapter). Наредбе Јаве се пресликавају у наредбе конкретног СУБП.

- Анализа програма:

```
class UpravaFakulteta   // Client
{
  SILABTarget silta;

  UpravaFakulteta(SILABTarget silta1){silta = silta1;}

  public static void main(String args[])
            {  UpravaFakulteta uf;
               SILABTarget silta;
               JavaTimPonuda jat = new JavaTimPonuda();
               silta = new Adapter(jat);
               uf = new UpravaFakulteta(silta);
               uf.Konstruisi();
               System.out.println("Ponuda java tima: " +
                jat.vratiPonudu());
               ...

         }
```

```
  void Konstruisi()
   {
   silta.KrProgramskiJezik();
     silta.KrSUBP();
     silta.KrPonudu();
   }

}
```

```
class JavaTimPonuda implements SILAB //
ConcreteBuilder1
{
  PonudaS elpon; // elementi ponude
  String ponuda;
  JavaTimPonuda() {elpon = new PonudaS();}
   public void kreirajProgramskiJezik()
         {elpon.pj = new Java();}
  public void kreirajSUBP()
         {elpon.subp = new MySQL();}
  public void kreirajPonudu()
         { ponuda = "Programski jezik-" +
           elpon.pj.vratiProgramskiJezik() + "
           SUBP-" + elpon.subp.vratiSUBP();}
  public String vratiPonudu(){return ponuda;}
}
```

```
class Adapter implements SILABTarget   // Adapter
{ SILAB sil;
  Adapter(SILAB sil1) {sil=sil1; }
  public void
  KrProgramskiJezik(){sil.kreirajProgramskiJezik();}
  public void KrSUBP(){sil.kreirajSUBP();}
  public void KrPonudu() {sil.kreirajPonudu();}
  public String VrPonudu(){return sil.vratiPonudu();}
}
```

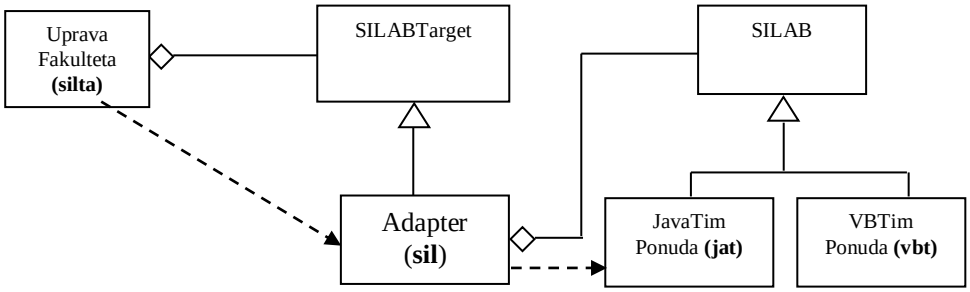
Када се повезују објекти узора то се ради на следећи начин:

a) Прво се креира објекат који од никога не зависе (**jat**).

b) Затим се креира адаптер објекат (**sil**), који се преко конструктора повезује са конкретним објектом који се адаптира (**jat**).

c) Затим се креира клијентски објекат (**uf**), који се преко конструктора повезује са конкретним адаптер објектом (**sil**).

- Креирају се објекти из десне у леву страну.

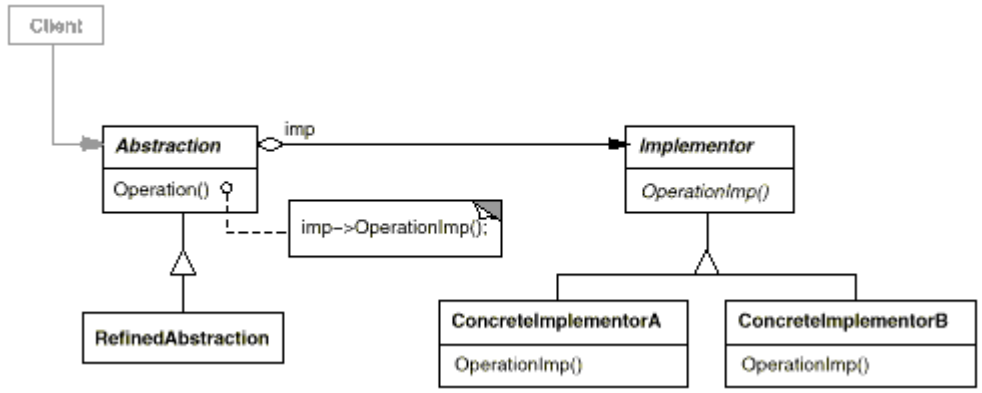


СП2: Bridge патерн

Дефиниција
Одваја (декуплује) апстракцију од њене имплементације тако да се оне могу мењати независно.

Појашњење ГОФ дефиниције:
Одваја (декуплује) апстракцију (Abstraction) од њене имплементације (Implementor) тако да се оне могу мењати независно.

Наводимо структуру bridge узора:



Учесници:

- **Abstraction**

Дефинише интерфејс <<апстракције>>. Чува референцу на објекат типа <<Implementor>>.

- **RefinedAbstraction**

Проширује интерфејс дефинисан класом <<Abstraction>>.

- **Implementor**

Дефинише интерфејс за имплементационе класе (<<ConcreteImplementorA>>, <<ConcreteImplementorB>>). Овај интерфејс не мора да одговара интерфејсу класе <<Abstraction>>. Обично <<Implementor>> интерфејс обезбеђује само примитивне операције а класа <<Abstraction>> дефинише операције високог нивоа које су засноване на наведеним примитивним операцијама.

- **ConcreteImplementor**

Имплементира интерфејс класе <<Implementor >>.

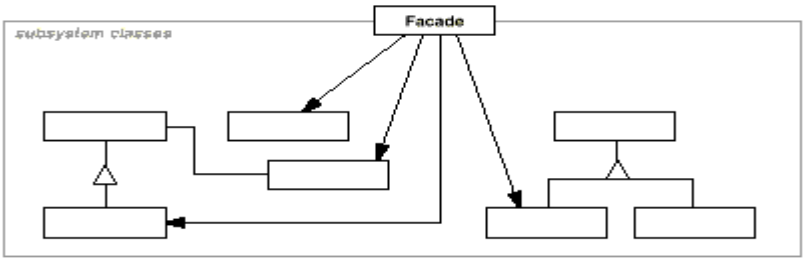
Напомена: *У примеру са почетка скрипте класа* ***DatabaseBroker*** *је реализована комбинацијом TemplateMethod и Bridge патерна.*

СП3: Facade патерн

Дефиниција
Обезбеђује јединствен интерфејс за скуп интерфејса неког подсистема. Facade узор дефинише интерфејс високог нивоа који омогућава да се подсистем лакше користи.

Појашњење ГОФ дефиниције:
Обезбеђује јединствен интерфејс (Facade) за скуп интерфејса (Sybsystem classes) неког подсистема. Facade узор дефинише интерфејс високог нивоа који омогућава да се подсистем лакше користи.

Наводимо **структуру** Facade узора:



Као и **учеснике** у узору:

- **Facade**
  Зна које <<подсистемске класе>> су одговорне за сваки од захтева који му се прослеђује.
  Преноси одговорност за извршење клијентских захтева до одговарајућих <<подсистемских објеката >>.
- **Sybsystem classes**
  Имплементирају <<подсистемске функционалности>>.
  Не знају ко је facade класа.

Напомена: *У примеру са почетка скрипте* ***контролер апликационе логије*** *је реализован преко Facade патерна.*

П3.3 : Патерни понашања

Патерни понашања описују начин на који класе или објекти сарађују и распоређују одговорности.

Постоје следећи патерни понашања:

1. Chain of responsibility - Избегава чврсто повезивање између пошиљаоца захтева и његовог примаоца, обезбеђујући ланац повезаних објеката, који ће да обрађују захтев све док се он не обради.

2. Command - Захтев се учаурује као објекат, што омогућава клијентима да параметризују различите захтеве. Наведеним приступом подржава се извршење повратних(undoable) операција као и опоравак података услед насилног прекида програма.

3. Interpreter - За дати језик, дефинише се репрезентација граматике језика заједно са интерпретером који користи ту репрезентацију да интерпретира реченице у језику.

4. Iterator – Обезбеђује начин да приступи елементима агрегатног објекта секвенцијално без излагања његове унутрашње репрезентације.

5. **Mediator -** Дефинише објекат који садржи скуп објеката који су у међусобној интеракцији. Интеракција између објеката може се независно мењати у односу на друге интеракције. Помоћу медијатора се успоставља слаба веза између објеката.

6. Memento - Без нарушавања учаурења memento патерн чува интерно стање објекта тако да објекат може бити враћен у то стање касније.

7. Observer - Дефинише један-више зависност између објеката, тако да промена стања неког објекта утиче аутоматски на промену стања свих других објеката који су повезани са њим.

8. State – Допушта објекту да промени понашање када се мења његова интерна структура.

9. Strategy - Дефинише фамилију алгоритама и обезбеђује њихову међузависнот. Стратегу узор омогућава промену алгоритма независно од клијента који га користи.

10. Template method – Дефинише скелет алгоритма у операцији, препуштајући извршење неких корака операција подкласама. Template method омогућава подкласама да редефинишу неке од корака алгоритама без промене алгоритамске структуре.

11. Visitor – Представља операцију која се извршава на елементима објектне структуре. Visitor омогућава да се дефинише нова операција без промене класа или елемената над којима она (операција) оперише.

ПП1. Template method патерн

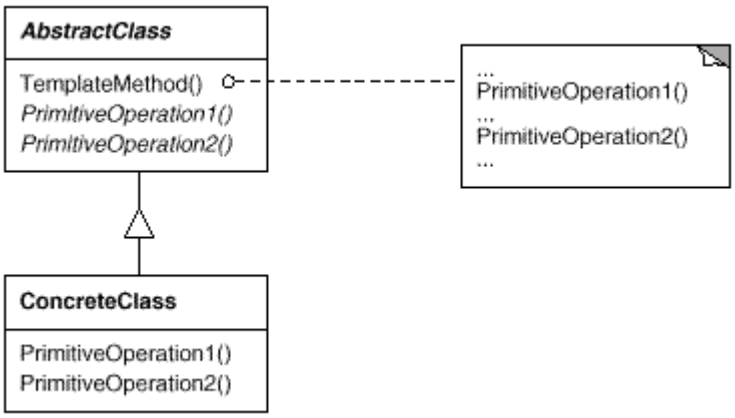
Дефиниција

Дефинише скелет алгоритма у операцији, препуштајући извршење неких корака операција подкласама. Template method омогућава подкласама да редефинишу неке од корака алгоритама без промене алгоритамске структуре.

Појашњење ГОФ дефиниције:

Дефинише скелет алгоритма у операцији (TemplateMethod), препуштајући извршење неких корака операција подкласама (ConcreteClass). Template method омогућава подкласама да редефинишу неке од корака алгоритама (PrimitiveOperation1, PrimitiveOperation2) без промене алгоритамске структуре (TemplateMethod).

*Наводимо структуру **Template method** патерна:*



Као и учеснике у **Template method** патерну:

- **AbstractClass**

  Дефинише апстрактне примитивне операције које конкретна подкласа имплементира.

  Имплементира алгоритам template method-а. Template method позива примитивне операције.

- **ConcreteClass**

  Имплементира примитивне операције које описују специфична понашања подкласа.

Пример патерна:

Кориснички захтев: *Управа Факултета захтева од Лабораторије за софтверско инжењерство да*
предложи најважније кораке у развоју софтверског система. Java и VB тим ће предложити конкретне кораке који ће се извшити у развоју софтверског система. Управа Факултета одређује тим који ће развијати софтверски систем.

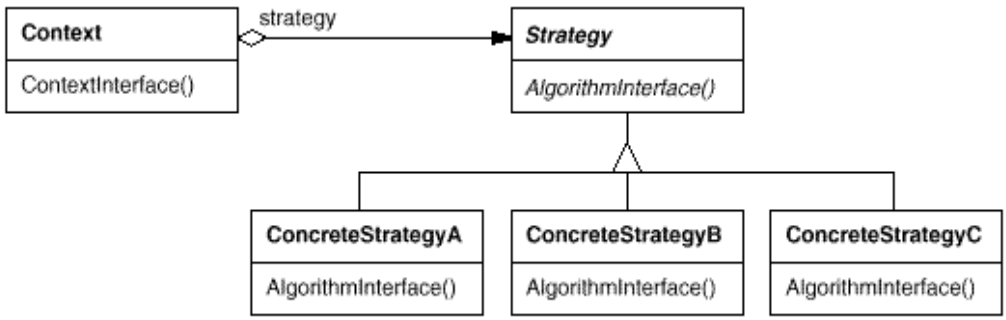
```
class UpravaFakulteta
{
 static SILAB sil; // AbstractClass
 public static void main(String args[])
            {  sil = new  JavaTimPonuda();
               System.out.println(sil.koraciRazvoja());
            }
}
```

```
abstract class SILAB //  AbstractClass
{ String koraciRazvoja ()
      { String pom = IzborStrategije();
        pom = pom + IzborMetode();
        pom = pom + IzborModela();
        return pom;
       }
   abstract String IzborStrategije();
   abstract String IzborMetode();
   abstract String IzborModela();

}

class JavaTimPonuda extends SILAB // ConcreteClass1
{  String IzborStrategije() {return "Use case driven strategija.";}
   String IzborMetode() {return " Larmanova metoda.";}
   String IzborModela() { return " Iterativno-inkrementalni model.";}
}

class VBTimPonuda extends SILAB // ConcreteClass2
{  String IzborStrategije() {return "Test driven strategija.";}
   String IzborMetode() {return " Extreme programming . ";}
   String IzborModela() { return " Iterativno-inkrementalni model.";}
}
```

Напомена: *У примеру са почетка скрипте класа* ***OpstaSO*** *је реализована преко TemplateMethod патерна.*

ПП2. Strategy патерн
Дефиниција
Дефинише фамилију алгоритама, учаурује сваки од њих и обезбеђује да они могу бити замењиви. Strategy патерн омогућава промену алгоритма независно од клијента који га користи.
Појашњење GOF дефиниције:
Дефинише фамилију алгоритама (ConcreteStrategyA, ConcreteStrategyB, ...), учаурује сваки од њих и обезбеђује да они могу бити замењиви (Strategy). Strategy патерн омогућава промену алгоритма (ConcreteStrategyA, ConcreteStrategyB, ...) независно од клијента (Context) који га користи.

Наводимо структуру ***Strategy*** *патерна:*



Као и учеснике у **Strategy** патерну:

- **Strategy**
  - Декларише interfejs koji je zajednički za sve algoritme koji ga podržavaju. Context koristi ovaj interfejs da pozove algoritam definisan sa ConcreteStrategy klasom.
- **ConcreteStrategy**
  - Implementira algoritam koji koristi Strategy interfejs.

- **Context**

- Context објекат је конфигурисана са ConcretyStrategy објектом.
- Он садржи референцу на Strategy објекат.
- Може да дефинише интерфејс, који омогућава Strategy објекту приступ, до његових података.

Пример патерна:

Кориснички захтев: *Управа Факултета захтева од Java и VB тима да предложе најважније кораке у развоју софтверског система. Управа Факултета одређује тим који ће развијати софтверски систем.*

```
class UpravaFakulteta   // Context
{
  static SILAB sil; // State

  public static void main(String args[])
              {   sil = new  JavaTimPonuda();
                  System.out.println(sil.strategijaRazvoja());
              }
}
abstract class SILAB// Strategy
{ abstract String strategijaRazvoja ();
}

class JavaTimPonuda extends SILAB // ConcreteStrategy1
{ String strategijaRazvoja() { String pom = "Use case driven strategija.";
                               pom = pom +   " Larmanova metoda.";
                               pom = pom +  " Iterativno-inkrementalni model.";
                               return pom;}
}

class VBTimPonuda extends SILAB // ConcreteStrategy 2
{ String strategijaRazvoja() { String pom =  "Test driven strategija.";
                               pom = pom +  " Extreme programming .";
                               pom = pom +  " Iterativno-inkrementalni model.";
                               return pom;}
}
```

2.3.5.3 Софтверски оквири

Софтверски оквир је софтверски систем који се може користити у развоју различитих доменских апликација. Софтверски оквир садржи софтверску библиотеку која има генеричку функционалност која је представљена преко интерфејса за програмирање апликације (application programming interface(API)). Софтверски оквир обезбеђује окружење које омогућава подршку за развој сложених софтверских апликација (пројеката) које се са састоје из више софтверских компоненети. Кључне особине оквира које га разликују од обичних софтверских библиотека су:

1. **инверзија контрола (inversion of control)** – у оквирима, насупрот библиотекама или нормалним корисничким апликацијама, током извршења програма се управља преко оквира.

2. **подразумевано понашање (default behavior)** – Оквир има подразумевано понашање.

3. **проширивост (extensibility)** – понашање оквира може бити проширено и промењено преко корисника прекривањем делова понашања оквира.

4. **непромењивост програмског кода оквира (non-modifiable framework code) –** Програмски код оквира не може да се мења.

Софтверски оквир има за циљ да олакша развој софтверске апликације омогућавајући пројектантима и програмерима да више времена посвете домену проблема (пословној логици) него да се баве програмирањем ниског нивоа детаљности. На тај начин се смањује укупно време развоја софтверске апликације. Оквир покрива аспекте апликације који су везани за трансакције, управљање стањима (state management), обраду захтева (request handling),..., итд.

2.4. Имплементација

2.4.1. Имплементационе технологије

2.4.1.1 Нити

Едсгер Дијкстра је рекао да се "конкурентност дешава када постоје два или више извршних токова (процеса) који су способни да се извршавају симултано (истовремено)". Процеси могу истовремено да користе дељене ресурсе (*shared resources*) што може да резултује непредвиђеним понашањем система. Увођење **међусобног искључења** (***mutual exclusion***) може да спречи непредвиђено понашање код коришћења дељених ресурса али може да доведе до појава **мртвог закључавање (*deadlock*)** и **гладовања (*starvation*).**

Deadlock је мултитаскинг проблем[52] који се дешава када два процеса заузму ресурсе и медусобно се цекају да ослободе те ресурсе. ***Starvation*** је мултитаскинг проблем који се дешава када неки процес заузме неки ресурс и не дозвољава другим процесима да га користе. То доводи до тога да други процеси не могу да се до краја изврше.

Уколико више процеса међусобно сарађују у извршењу неког задатка јавља се проблем њихове међусобне комуникације и размене података јер сваки процес заузима посебан меморијски простор. Тај проблем је решен појавом **нити (*threads*)** које деле исти меморијски простор.

Јава је један од програмских језика који подржава вишенитно програмирање. То значи да један програм (процес) може да обавља више нити истовремено (конкурентно). **Нит представља део програма који може истовремено да се извршава са другим нитима истог програма.**

Нити, као што је речено, деле исти адресни простор у оквиру једног процеса и комуникација између њих је доста једноставнија у односу на комуникацију између процеса. Радом са више процеса управља оперативни систем, док радом са више нити управља Јава окружење.

НИ1: Главна програмска нит
Када програм у Јави почне да се извршава, он аутоматски креира и извршава једну нит која се зове главна програмска нит.

```
/*
   Primer NT1: Napisati program koji ce da ukaze na glavnu nit koju treba
   uspavati 5 sekundi.
 */

/**
 * @author Sinisa Vlajic
 * SILAB - Labartorija za Softversko Inzenjerstvo
 * FON - Beograd
 *http://silab.fon.bg.ac.yu
 */

class NT1 {

    public static void main(String args[]) {
        Thread gnit = Thread.currentThread();
        System.out.println("Glavna nit:" + gnit + '\n' + "Pauza od 5 sekundi");
        try {
            gnit.sleep(5000); // 5 sekundi pauze
        } catch (InterruptedException e) {
            System.out.println("Prekid niti");
        }
        gnit.setName("Glavna"); // Promena naziva niti
        System.out.println("Glavna      nit:"+gnit+'\n'+"Naziv      glavne      niti:"+
gnit.getName());
```

[52] *Multitasking* је метода помоћу које више задатака (*tasks*), односно процеса, деле заједничке ресурсе као што је *CPU* (*Central Processing Unit*). У случају компјутера са једним *CPU*, у неком тренутку времена само један задатак може да се извршава, што практично значи да *CPU* активно извршава само инструкције тог задатка. *Multitasking* омогућава да се у неком периоду времена више процеса извршава, тако што прави распоред (*scheduling*) који задатак ће се извршавати у ком тренутку времена а који задаци ће чекати док на њих не дође ред. Додељивање *CPU* од једног до другог задатка се назива *контекстна додела* (*context switch)*. Када се *context switch* дешава учестало ствара се илузија истовремености извршења више задатака. *Mulititasking* омогућава да се изврши више задатака на једном *CPU* у односу на компјутере са више *CPU* (мултипроцесорски компјутери) који не користе *multitasking*.

```
    }
}
/*
 Rezultat:
 Glavna nit:Thread[main,5,main]
 Pauza od 5 sekundi.
 Glavna nit:Thread[Glavna,5,main]
 Naziv glavne niti: Glavna
*/
```

НИ2: Прављење нити

У Јави се нит може направити на 2 начина:

- Реализацијом интерфејса `Runnable`
- Проширењем класе `Thread`

НИ2.1: Прављење нити реализацијом класе Runnable

Када се нит прави реализацијом интерфејса `Runnable`, тада је једино потребно се имплементира (реализује) метода `run()` интерфејса `Runnable`[53].

```
/* Primer NT2: Napraviti nit pomocu interfejsa Runnable.*/
/**
 * @author Sinisa Vlajic
 * SILAB - Labartorija za Softversko Inzenjerstvo
 * FON - Beograd
 *http://silab.fon.bg.ac.yu
 */

class NT2 implements Runnable {
    NT2() {
        //nova nit ce pozvati run metodu od this objekta
        nit = new Thread(this, "Nova nit");
        nit.start();
    }
    public void run() {
        System.out.println("Nit:" + nit);
    }

    public static void main(String args[]) {
        NT2 nn = new NT2();
        Thread gnit = Thread.currentThread();
        gnit.setName("Glavna nit");
        System.out.println("Nit:" + gnit);
    }
    Thread nit;
}
/*
 Rezultat:
 Nit:Thread[Glavna nit,5,main]
 Nit:Thread[Nova nit,5,main]
*/
```

НИ2.2: Прављење нити проширењем класе Thread

Када се нит прави проширењем класе `Thread`, тада је потребно се прекрије метода `run()` класе `Thread`. Класа `Thread` поред методе `run()` садржи и друге методе које могу да се прекрију, за разлику од интерфејса `Runnable` који има само методу `run()`.

```
/*Primer NT3: Napraviti nit pomocu klase Thread.*/

/**
 * @author Sinisa Vlajic
 * SILAB - Labartorija za Softversko Inzenjerstvo
 * FON - Beograd
 *http://silab.fon.bg.ac.yu
```

[53] `main()` метода се за главну нит у суштини понаша исто као и `run()` метода за нити које су креиране у `main()` методи.

```
 */

class NT3 extends Thread {
    NT3() {
        super("Nova nit");
        start();
    }
    public void run() {
        System.out.println("Nit:" + currentThread());
    }
    public static void main(String args[]) {
        NT3 nn1 = new NT3();
        Thread gnit = Thread.currentThread();
        gnit.setName("Glavna nit");
        System.out.println("Nit:" + gnit);

    }
}
/*
 Rezultat:
 Nit: Thread[Glavna nit,5,main]
 Nit: Thread[Nova nit,5,main]
*/
```

НИ3: Стања нити
Нит може бити у једном од 4 стања:

- ново (*new*)
- извршно (*runnable*)
- блокирано (*blocked*)
- свршено (*dead*)

Нова нит
У почетку нит је декларисана: `Thread nit;`
Када се нит креира са `new` оператором: `nit = new Thread(this,"Nova nit");`
нит прелази у стање "**ново**". У том стању, нит се још не извршава. Након тога, позива се метода `start()`.

Извршавање нити
Након позива методе `start()`, нит прелази у стање "**извршно**" . Извршно стање не значи аутоматски да се нит "**извршава**" (*running*). Нит се извршава када контрола програма пређе на тело методе `run()`, коју позива метода `start()`.

```
// NapraviNit.java
/**
 * @author Sinisa Vlajic
 * SILAB - Labartorija za Softversko Inzenjerstvo
 * FON - Beograd
 * http://silab.fon.bg.ac.yu
 */

class NT3 implements Runnable{

    NT3() {
        nit = new Thread(this, "Nova nit"); // Nit je u stanju ”novo”
        nit.start(); // Nit je u stanju ”izvrsno”
    }

    public void run() {
        System.out.println("Nit:" + nit);
    } // Nit je u stanju “izvrsava”

    // Nakon izvrsenja tela metode run() nit prelazi u stanje ”svrseno”
    public static void main(String args[]) {
        NT3 nn = new NT3();
        Thread gnit = Thread.currentThread();
```

```
        gnit.setName("Glavna nit");
        System.out.println("Nit:"+ gnit);
}

Thread nit;
}
/*
 Rezultat:
 Nit: Thread[Glavna nit,5,main]
 Nit: Thread[Nova nit,5,main]
*/
```

НИ4: Прекид нити

Нит се завршава када се тело `run()` методе изврши:

```
public void run{
        while(signal)   {
            izvrsenje niti…
        }
}
```

Докле год `signal` има вредност `true`, нит се извршава. Када `signal` добије вредност `false`, позивом нпр. методе `promeni()` нит престаје да се извршава.

```
public void promeni(){
        signal = false;
}
```

Међутим у случају када је нит у блокираном стању, прекид извршења нити не може остварити на предходни начин. Тада се користи `interrupt()` метода која позива нит која је блокирана. Она ће прекинути блокаду нити генерисањем изузетка `InterruptedException`. Блокада може престати ако је иста настала на основу `sleep()` или `wait()` методе.

Наведена `interrupt()` метода прекида блокаду нити, али не прекида извршење нити. Уколико се жели да прекид нити преко `interrupt()` методе прекине извршење нити, тада се ухваћени изузетак `InterruptedException` обрађује на одговарајући начин.

```
public void run (){
  try {…
          while(signal){
                  izvrsenje niti…
                  }
      } catch(InterruptedException e) {
      // Prekinuta blokada niti koja je nastala na osnovu sleep() ili wait() metode.
      }
…
// izlaz iz run() metode i prekid izvršenja niti.
}
```

Метода `interrupt()` статус нити поставља на `true`.

У случају када `interrupt()` метода позове нит која није блокирана, тада се неће десити изузетак `InterruptedException`. Због тога се код логичке контроле извршења нити позива метода `interrupted()`, која проверава да ли је текућа нит "претрпела" `interrupt()` методу (да ли је статус нити `true`).

Уколико је статус нити `true` треба да се прекине извршење нити:

```
          while( !interrupted() && signal){
                        izvrsenje niti…
          }
```

или

Метода `interrupted()` има спољни ефекат (*side effect*), јер стање нити поставља на `false`.

Поред наведених постоји и метода `isInterrupted()` која враћа стање прекида нити али нема спољни ефекат:

```
          while( !isInterrupted() && signal){
              izvrsenje niti…
          }
```

На основу наведеног може се закључити да постоје неколико сценарија покушаја прекида нити у зависности од тога: а) да ли је нит блокирана или не и б) како се покушава прекинути нит (`interrupt` методом или логичком контролом).

НИ5: Сихронизација

Уколико се јави потреба да две или више нити деле заједнички ресурс, при чему нити не могу истовремено да користе заједнички ресурс, мора се обезбедити механизам које ће омогућити искључиви (ексклузивни) приступ једне нити до заједничког ресурса. Тек након завршетка обраде заједничког ресурса од стране једне нити, могуће је да друга нит, такође искључиво, приступи до њега.

Монитор обезбеђује механизам за искључиви приступ нити до заједничког ресурса. Када нека нит X уђе у монитор, ниједна друга нит не може у њега ући, све док нит X не изађе из њега. Поступак којим монитор обезбеђује наведени механизам назива се синхронизација.

Потреба истовременог приступа до дељеног објекта доводи до тзв. *race condition*[54] ситуација.

НИ5.1: Комуникација нити без синхронизације

Онемогућавање истовременог приступа нити до дељених објеката се назива синхронизација приступа. Уколико нема синхронизације приступа може се јавити следећа ситуација:

Уколико постоји банка са 10 рачуна (за сваки рачун се прави посебна нит), између којих се врши пренос новца на случајно изабран начин, може се десити да износ истог рачуна истовремено повећава 2 или више нити.

Нпр. `racun[5]` се истовремено повећава помоћу две нити.

Стање рачуна пре промене: `racun[5] = 500;`

Прва нит : `racun[5] +=100;`

Друга нит : `racun[5] +=50;`

Проблем се јавља на нивоу атомских операција наредби:

Прва нит: пуни се меморијски регистар `MR1` са износом од `racun[5]` преко наредбе `load()`.

```
MR1 = 500
racun[5] = 500
```

Након тога се повећава износ регистра за 100 преко наредбе `add()`.

```
MR1 = 600
racun[5] = 500
```

Уколико тада почне да се извршава друга нит, тада се пуни меморијски регистар `MR2` са износом од `racun[5]` са наредбом `load()`.

```
MR2 = 500
racun[5] = 500
```

Након тога се повећава износ регистра за 50 преко наредбе `add()`.

```
MR2 = 550
racun[5] = 500
```

Ако тада `racun[5]` добије вредност регистра `MR2` са наредбом `store()`.

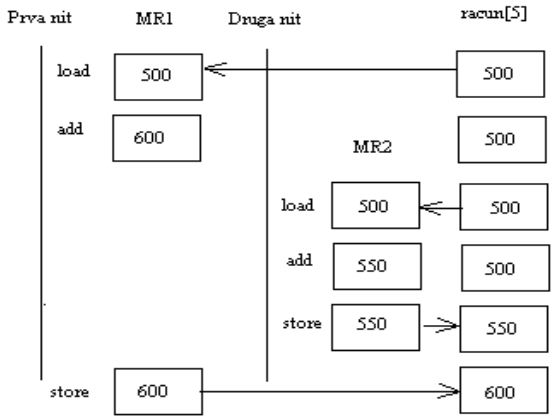
```
racun[5] = 550
```

На крају ће `racun[5]` добити вредност регистра `MR1` са наредбом `store()`.

```
racun[5] = 600
```

Добијени износ је погрешан. Валидна вредност коју би `racun[5]` требао да има је 650**.**

[54] Када на тркама 2 такмичара, скоро у исто време пролазе кроз циљ, јавља се проблем прецизног одређивања ко је први прошао кроз циљ.



НИ5.2: Закључавање објеката
Када нит позове методу објекта који треба да се синхронизује, објекат постаје закључан. Прецизније речено при позиву синхронизоване методе: "Нит налази једини кључ од објекта испред врата, убацује кључ у браву од објекта, откључава га, улази у објекат и браву са друге стране врата закључава. Брава остаје закључана све док нит не изађе из објекта, односно док нит не извади кључ из браве и стави га испред врата". Нити које чекају на приступ синхонизованим методама дељеног објекта, могу приступити до несинхронизованих метода дељеног објекта.

НИ5.3: Коришћење синхронизованих метода
У Јави сваки објекат има свој монитор. Да би се монитор покренуо потребно је да се објекту приступи преко једне од његових синхронизованих метода. Испред метода које треба синхронизовати ставља се кључна реч ***synchronized.*** Треба нагласити да извршавање једне од синхронизованих метода неког објекта онемогућава приступ до било које друге синхронизоване методе истог објекта. Тек након завршетка извршења синхронизоване методе неког објекта могуће је позвати неку другу синхронизовану методу истог објекта. За несинхронизоване методе не важи наведено ограничење.

НИ5.4: Коришћење синхронизованих објеката
У случају да преко метода не може да се синхронизује приступ објекту, могуће је експлицитно синхронизовати сам објекат. Општи облик експлицитне синхронизације објекта је:

```
synhronized (objekat) {
        blok naredbi koje će biti sinhronizovane
}
```

Када се синхронизација објекта врши преко методе, тада се пре извршења синхронизоване методе закључава објекат. У том статусу објекат остаје све док се не изврши синхронизована метода.
Када се синхронизација објекта врши експлицитно, тада се пре извршења блока наредби које треба синхронизовати закључава објекат. У том статусу објекат остаје све док се не изврши наведени блок наредби.

У нашем примеру нити користимо код контролера апликационе логике:

```
/*
 * KontrolerAL.java
 *
 * 02.05.2011
 *
 * @autor  Dr Sinisa Vlajic
 *
 * Katedra za softversko inzenjerstvo
 *
 * Laboratorija za softversko inzenjerstvo
 *
 * Fakultet organizacionih nauka - Beograd
 *
 */

import java.net.*;
import java.io.*;


class KontrolerAL // Kontroler aplikacione logike
 {   static ServerSocket ss;
             static Klijent kl[];

     public static void main(String[] args) throws Exception
               {   kl = new Klijent[10];
                   ss = new ServerSocket(8189);

                   System.out.println("Podignut je serverski program:");
                   for (int brojKlijenta = 0;brojKlijenta < 10;brojKlijenta ++)
                         { Socket soketS = ss.accept();
                           System.out.println("Klijent " + (brojKlijenta+1));
                           // pravi se nova nit za svakog klijenta koji se poveže sa serverskim programom
                           kl[brojKlijenta] = new Klijent(soketS,brojKlijenta+1);
                         }
               }
}
```

class Klijent extends Thread

```
          { public Klijent(Socket soketS1,int brojKlijenta1)
            { soketS = soketS1; brojKlijenta=brojKlijenta1;
              System.out.println("Konstruktor");
              start();
            }

            public void run()
            { try { String signal = "";
                    out = new ObjectOutputStream(soketS.getOutputStream());
                    in = new ObjectInputStream(soketS.getInputStream());

                    System.out.println("run");
                        while (true)
                           { // Citanje naziva operacije i racuna
                             String NazivSO = (String) in.readObject();
                             OpstiDomenskiObjekat rac = (OpstiDomenskiObjekat) in.readObject();
                              if (NazivSO.equals("kreirajNovi")==true)
                                   signal = KreirajNovi.kreirajNovi(rac);
                              if (NazivSO.equals("Pretrazi")==true)
                                  signal = Pretrazi.Pretrazi(rac);
                                ...
                            // Slanje promenjenog racuna i signala o uspesnosti operacije
                              out.writeObject(rac);
                              out.writeObject(new String(signal));
                           }
                } catch (Exception e)  { System.out.println(e);  }
            }
            private
            Socket soketS;
            int brojKlijenta;
            ObjectOutputStream out;
            ObjectInputStream in;
}
```

Код класе OpstaSO се користи синхронизација методе *opsteIzvrsenjeSO*:

```
/*
 * OpstaSO.java
 *
 * 02.05.2011
 *
 * @autor  Dr Sinisa Vlajic
 *
 * Katedra za softversko inzenjerstvo
 *
 * Laboratorija za softversko inzenjerstvo
 *
 * Fakultet organizacionih nauka - Beograd
 *
 */
abstract class OpstaSO
{
  static BrokerBazePodataka BBP;
  static boolean signal;
  static boolean BazaOtvorena = false;
  static boolean transakcija = false;

// Сихронизација static методе закључава  static атрибуте класе. Сихронизација не static метода закључава
// објекат за који је позвана та метода.
synchronized static String opsteIzvrsenjeSO(OpstiDomenskiObjekat rac, OpstaSO os)
   { if (!os.otvoriBazu()) return os.vratiPorukuMetode();
     if (!os.izvrsenjeSO(rac) && transakcija)
         { signal = os.rollbackTransakcije();
           return os.vratiPorukuMetode();
         }

     if (transakcija) os.commitTransakcije();
        return os.vratiPorukuMetode();
   }

   abstract boolean izvrsenjeSO(OpstiDomenskiObjekat rac);


   boolean otvoriBazu()
   { if (BazaOtvorena == false)
        { BBP = new BrokerBazePodataka();
          BBP.isprazniPoruku();
          signal = BBP.otvoriBazu("RACUN");
          if (!signal) return false;
        }
     BBP.isprazniPoruku();
     BazaOtvorena = true;
     return true;
   }

   boolean commitTransakcije()
     { return BBP.commitTransakcije();}

   boolean rollbackTransakcije()
     { return BBP.rollbackTransakcije();}

   String vratiPorukuMetode()
     { System.out.println(BBP.vratiPorukuMetode());
                return BBP.vratiPorukuMetode();
     }
}
```

2.4.1.2 Мрежа - Сокети

MP1: Адреса рачунара

Сваки рачунар на глобалној мрежи (Интернету) има своју адресу (***Internet address***) која се састоји од 4 троцифрена броја, при чему бројеви могу узимати вредност из опсега од `0 – 255.` На пример: 132.163.135.130 представља адресу сервера Националног института за стандарде у Колораду.
Повезивање рачунара у мрежу и пренос података између њих се остварује најчешће помоћу `TCP/IP` (***Transimion Control Protocol/Internet Protocol***) протокола. `IP` је одговоран да пронађе интернет адресу циљног рачунара и да усмери податке ка њему од изворног рачунара. TCP је задужен за успостављање и раскидање везе између рачунара, као и за одређене контролне функције. Често се за интернет адресу каже да је то **`IP`** адреса.
Класе које омогућавају рад у мрежи у Јави налазе се у пакету `java.net.`

Уколико се жели показати интернет адреса локалне машине користи се метода `getLocalHost().` Класа која садржи информације о интернет адреси је `InetAddress`.

```
/* Primer  Mreza1: Prikazati Internet adresu tekuce masine. */

/*
@author Sinisa Vlajic
* SILAB – Laboratorija za Softversko Inzenjerstvo
* FON – Beograd
* http://silab.fon.bg.ac.yu
*/
import java.net.*;
class Mreza1 {
    public static void main(String args[]) throws UnknownHostException {
        InetAddress tekucaAdresa = InetAddress.getLocalHost();
        System.out.println(tekucaAdresa);
    }
}
/*
 Rezultat:
 gsi/147.91.128.109
*/
```

Поред нумеричке адресе рачунара постоји и симболичка адреса рачунара, која се представља преко скупа симбола (знакова). Пример симболичке адресе је: `gsi.fon.bg.ac.yu.` Једна симболичка адреса може бити везана за више нумеричких адреса рачунара.
Симболичка адреса састоји се из два дела:
матичног имена (***host name***) рачунара и
имена домена (***domain name***) коме припада матични рачунар.
На пример симболичка адреса `gsi.fon.bg.ac.yu` састоји се из матичног имена: `gsi` и имена домена: `fon.bg.ac.yu.`
Сервис за именовање домена (***Domain Naming Service - DNS***), повезује симболичке адресе са интернет адресама. *DNS* сервиси се извршавају на *DNS* серверима.

Уколико се жели видети интернет адреса симболичке адресе, користи се метода `getByName().`

```
/*  Primer    Mreza2:  Prikazati  IP  adresu  masine  koja  ima  simbolicku  adresu
"java.fon.bg.ac.yu" . */
/*
@author Sinisa Vlajic
* SILAB – Laboratorija za Softversko Inzenjerstvo
* FON – Beograd
* http://silab.fon.bg.ac.yu
*/
import java.net.*;
class Mreza2 {
    public static void main(String args[]) throws UnknownHostException {
        InetAddress tekucaAdresa = InetAddress.getByName("java.fon.bg.ac.yu");
        System.out.println(tekucaAdresa); // Prikazuje simbolicku i IP adresu.
        System.out.println(tekucaAdresa.getHostAddress());  //  Prikazuje  simbolicku
adresu.
```

```
        System.out.println(tekucaAdresa.getHostName()); // Prikazuje IP adresu.
    }
}
/*
 Rezultat:
 java.fon.bg.ac.yu/147.91.128.18
 147.91.128.18
 java.fon.bg.ac.yu
*/
```

MP2: URL адреса

Интернет и симболичке адресе рачунара омогућавају да се преко њих приступи до жељених рачунара у мрежи. Уколико се жели приступити до одређених сервиса и датотека на рачунару користи се *URL* (***Uniform Resource Locator***) адреса.
URL адреса састоји се из четири дела:
Протокол који се користи (***http, ftp, gopher*** *или* ***file***), који је одвојен : од остатка адресе. У последње време углавном се ради преко http протокола.
Адреса рачунара (интернет или симболичка адреса). Испред адресе се ставља '//' а на крај адресе се ставља '/' уколико не постоји порт, односно ':' ако постоји.
Број порта (прикључка). Иза порта се ставља '/'. Овај део није обавезан.
Путања до датотеке, укључујући и име датотеке.

Следећи пример показује све делове задате *URL* адресе.

```
/* Programski zahtev Mreza4: Prikazati svaki od delova URL adrese. */
/*
@author Sinisa Vlajic
* SILAB – Laboratorija za Softversko Inzenjerstvo
* FON – Beograd
* http://silab.fon.bg.ac.yu
*/

import java.net.*;

class Mreza4 {
    public static void main(String args[]) throws MalformedURLException {
        URL hp = new URL("http://java.fon.bg.ac.yu:80/index.htm");
        System.out.println("Protokol:" + hp.getProtocol());
        System.out.println("Port:" + hp.getPort());
        System.out.println("Racunar:" + hp.getHost());
        System.out.println("Datoteka:" + hp.getFile());
        System.out.println("Zajedno:" + hp.toExternalForm());
    }
}
/*
 Rezultat:
 Protokol: http
 Port: 80
 Racunar: java.fon.bg.ac.yu
 Datoteka: /index.html
 Zajedno: http://java.fon.bg.ac.yu:80/index.htm
*/
```

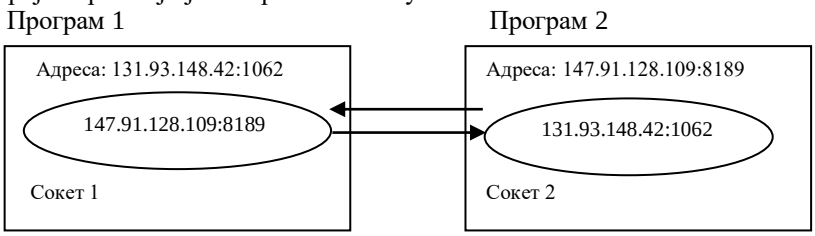
MP3: Сокети

Сокет, у ширем смислу, је механизам који омогућава комуникацију између програма који се извршавају на различитим рачунарима у мрежи. Он је пројектован тако да подржи имплементацију клијент/серверских апликација. Сокети користе *TCP/IP* протокол при повезивању, контроли и преносу података између два или више програма

При повезивању два програма преко сокета, по један сокет се генерише за сваки програм. Сваки од сокета садржи референцу на други сокет. То практично значи да први сокет садржи референцу на други сокет, док други сокет садржи референцу на први сокет.

Адреса сокета састоји се из два дела:

- адресе рачунара на коме се налази програм који је генерисао сокет
- броја порта који је генерисан помоћу сокета



Конекција између два програма је остварена када се успостави веза између њихових сокета. Сокети, у ужем смислу, представљају објекте помоћу којих се шаљу/прихватају подаци ка/од других сокета.
Сокет је по својој природи улазно-излазни ток и он се понаша на сличан начин као:

- системски објекат `System.in`, помоћу кога се подаци прихватају са стандардног улаза (тастатуре), док се код сокета подаци прихватају са спољашњег улаза (са мреже) од другог сокета.
- системски објекат `System.out`, помоћу кога се подаци шаљу ка стандардном излазу (екрану), док се код сокета подаци шаљу ка спољашњем излазу (ка мрежи) до другог сокета.

Сокети се повезују са улазно-излазним токовима на сличан начин као што је то случај са `System.in` и `System.out` објектима, када се жели извршити обрада података коју сокети размењују.

Типичан сценарио **развоја клијент/серверских апликација** помоћу сокета је следећи:

Покреће се програм на серверском рачунару. Интернет адреса сервер рачунара је `147.91.128.109`.
Коришћењем наредбе:

```
ServerSocket ss = new ServerSocket(8189);
```

прави се сервер сокет који се повезује[55] нпр. са портом `8189`[56].
Адреса сервер сокета `ss` је `147.91.128.109:8189`.
Након тога извршава се наредба

```
Socket soketS = ss.accept();
```

којом сервер сокет долази у стање чекања на клијенте. Он ће бити у том стању све док се клијент не повеже са њим.

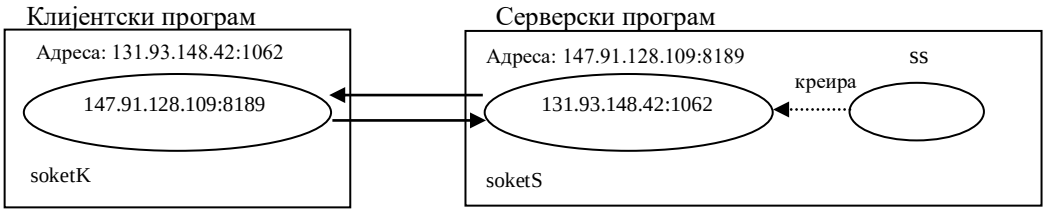
На клијентској страни се извршава наредба:

```
Socket soketK = new Socket("147.91.128.109",8189);
```

којом се врши повезивање клијента са сервером[57]. Сокет који је направљен на клијентској страни прослеђује до серверског сокета своју адресу: `131.93.148.42:1062`.
У току успостављања везе (конекције) између серверског сокета и клијентског сокета дешавају се следеће активности:
Клијентски сокет `soketK` добија референцу (`147.91.128.109",8189`) на серверски сокет.
Серверски сокет `cc` генерише нови сокет `soketS`. Адреса новог сокета ће бити иста као и адреса серверског сокета.

[55] За сво време трајања програма сокет ослушкује (прати) рад наведеног порта.
[56] Порт 8189 не користи ни један од стандарних сервиса.
[57] Сокет на клијентској страни се повезује са сервер сокетом на серверској страни.

Нови сокет `soketS` добија референцу (`131.93.148.42:1062`) на клијентски сокет.



Као резултат наведених активности добијени су сокети `soketS` и `soketK` који показују један на другог. Пошто се жели извршити размена података између сокета, тако да исти могу да се обраде преко стандардних улазно-излазних уређаја, сокети се повезују са улазно-излазним објектима на следећи начин:

```
BufferedReader in  = new BufferedReader(new InputStreamReader(X.getInputStream()));
PrintWriter out = new PrintWriter(X.getOutputStream(),true);
```

`X ∈ (soketS, soketK)`

Након тога је могуће да клијент пошаље поруку до сервера и обрнуто са наредбом:

```
out.println(" Y je spreman za rad\n");
```

`Y ∈ (KLIJENT, SERVER)`

Клијент, односно сервер прихвата податке на следећи начин:

```
String line =in.readLine();
```

На крају клијент, односно сервер на стандардном излазу приказују поруку коју је примио:

```
System.out.println(" Z  je primio poruku od Z1:" + line);
```

`Z ≠ Z1`

`Z , Z1∈ (KLIJENT, SERVER)`

Из наведеног се може закључити да постоји симетрија метода код размене података између сокета. То значи да место креирања сокета (клијентски или серверски програм), након успостављања конекције, не утиче ни на који начин на размену података. Сокети су равноправни у комуникацији.

Серверски и клијентски програм за наведени пример имају следећи изглед:

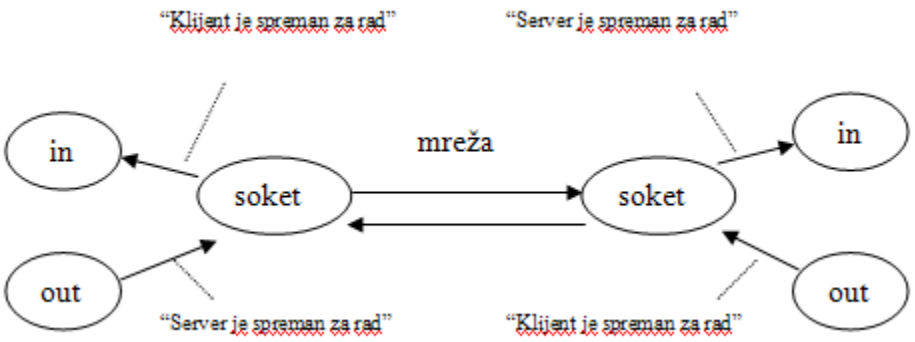
Серверски програм

```
/*  Primer MR6S: Napisati program koji ce kreirati serverski soket na portu 8189.
    Nakon toga se povezati sa klijentskim soketom. Na kraju poslati poruku klijentskom
    soketu.
*/

/*
@author Sinisa Vlajic
* SILAB - Laboratorija za Softversko Inzenjerstvo
* FON - Beograd
* http://silab.fon.bg.ac.yu
*/
import java.io.*;
import java.net.*;
public class ServerSoket {
    public static void main(String[] args) {
     try {
       ServerSocket ss = new ServerSocket(8189);
       Socket soketS = ss.accept();
      BufferedReader in = new BufferedReader(new
                          InputStreamReader(soketS.getInputStream()));
       PrintWriter out = new PrintWriter(soketS.getOutputStream(), true);
       out.println(" SERVER je spreman za rad\n");
       String line = in.readLine();
       System.out.println(" SERVER je primio poruku od klijenta:" + line);
    }catch (Exception e) {
            System.out.println(e);
      }
    }
}
```

Клијентски програм

```
/*
  Primer MR6K: Napisati program koji ce kreirati klijentski soket, koji ce se
  povezati sa serverskim soketom koji je podignut na racunaru cija je adresa
  147.91.128.109 na portu 8189.Poslati poruku serverskom racunaru.
*/
/*
@author Sinisa Vlajic
* SILAB – Laboratorija za Softversko Inzenjerstvo
* FON – Beograd
* http://silab.fon.bg.ac.yu
*/
import java.io.*;
import java.net.*;
public class SoketKlijent {
    public static void main(String[] args) {
     try {
       Socket soketK = new Socket("147.91.128.109", 8189);
       BufferedReader          in          =          new          BufferedReader(new
InputStreamReader(soketK.getInputStream()));
       PrintWriter out = new PrintWriter(soketK.getOutputStream(), true);
       out.println(" KLIJENT je spreman za rad\n");
       String line = in.readLine();
       System.out.println(" KLIJENT je primio poruku od servera:" + line);
     } catch (Exception e) {
         System.out.println(e);
     }
    }
}
```

-



Уколико се жели видети:

- порт сокета на који показује други сокет, користи се метода `getPort()`.
- `IP` адреса сокета на који показује други сокет, користи се метода `getInetAddress()`.
- локални порт на коме је подигнут сокет користи се метода `getLocalPort()`. То значи да се пуна адреса сокета на који показује други сокет добија као: `getInetAddress() + getPort()` што се може видети у следећем примеру:

Повезивање сервера са више клијената

Сокети омогућавају да се више клијентских програма (клијент сокета) повеже на један серверски програм (серверски сокет). За сваки од клијентских сокета прави се по једна нит, тако да се у оквиру серверског програма конкурентно извршава више нити. Наведене нити могу да приступе заједничким ресурсима сервера.

```
/*
Primer MR10S: Napisati program koji ce kreirati serverski soket na portu 8189. Serverski soket moze da se  poveze sa najvise 10 klijenata (klijentskih soketa). Za svakog klijenta napraviti posebnu nit koja ce se nezavisno izvrsavati u odnosu na druge niti. U okviru svake niti ce se vrsiti obrada kolicine robe (prodaja i nabavka).
Kolicina robe ce biti zajednicki atribut svih klijenata. Kada roba padne na koliciju jednaku 0 server treba da o tome obavesti sve klijente.
*/ author Sinisa Vlajic
*  SILAB – Laboratorija za Softversko Inzenjerstvo
*  FON – Beograd
*  http://silab.fon.bg.ac.yu
*/
import java.io.*;
import java.net.*;

public class ObradaRobe1 {
   public static void main(String[] args)
     {
       try {
            KreiranjeNiti kn = new KreiranjeNiti();
            kn.Kreiranje();
           } catch (Exception e) { System.out.println(e); }
   }
}
class KreiranjeNiti {
   int kolicina;
   ObradaNiti on[];
   ServerSocket ss;
   int brojKlijenta;

   KreiranjeNiti()
    {  on = new ObradaNiti[10]; }

   public void Kreiranje()
    {
     try { ss = new ServerSocket(8189);
           kolicina = 10;
           for (brojKlijenta = 0; brojKlijenta < 10; brojKlijenta++)
              { Socket soketS = ss.accept();
                System.out.println("Klijent " + brojKlijenta);
                on[brojKlijenta] = new ObradaNiti(soketS, brojKlijenta, this);
                on[brojKlijenta].start();
               }
         } catch (Exception e) { System.out.println(e + " greska!"); }
   }

  public void Prodaja(float p)
     {
       kolicina -= p;
       if (kolicina <= 0) {Azuriranje();}
     }
   public void Nabavka(float n)
     { kolicina += n; }

   public void Azuriranje()
    {
     NitObavestenje obavestenje = new NitObavestenje(this, "Magacin se upravo izpraznio !");
     obavestenje.start();
     System.out.println("kreirao nit obavestenje");
    }
}
```

```
class ObradaNiti extends Thread
  {
   public ObradaNiti(Socket soketS1, int c, KreiranjeNiti kn1)
     {
      soketS = soketS1;
      brojKlijenta = c + 1;
      kn = kn1;
     }
  public void run()
   {
     try {  in = new BufferedReader(new InputStreamReader(soketS.getInputStream()));
            out = new PrintWriter(soketS.getOutputStream(), true);
            boolean done = false;
            while (!done) {   out.println("Izaberite jednu od sledecih opcija:\n");
                              out.println("1.PRODAJA. 2.NABAVKA 3. IZLAZ");
                              out.println("  ");
                              String line = in.readLine();
                               if (line.equals(""))
                                  { out.println("Zavrsetak rada klijenta");
                                    out.println("999");
                                    done = true;
                                  }
                                else
                                  { switch (line.charAt(0))
                                      { case '1': out.println("Echo: (" + brojKlijenta + "): IZABRANA  PRODAJA");
                                        if (kn.kolicina - 4 < 0)
                                          {
                                            out.println("Nema dovoljno robe na zalihama da bi se izvršila prodaja!");
                                          }
                                        else
                                          {  kn.Prodaja(4);
                                          }
                                        break;
                                        case '2': out.println("Echo: (" + brojKlijenta + "): IZABRANA NABAVKA");
                                                  kn.Nabavka(2);
                                                  break;
                                        case '3':  out.println("Zavrsetak rada klijenta");
                                                   out.println("999");
                                                   done = true;
                                        }
                                  }
                               out.println("Ukupno je ostalo komada:" + kn.kolicina);
                           }
          soketS.close();
          System.out.println("Zatvorio klijenta " + brojKlijenta);
        } catch (Exception e) { System.out.println(e);  }
   }
   private Socket soketS;
   public int brojKlijenta;
   private KreiranjeNiti kn;
   BufferedReader in;
   PrintWriter out;
}


class NitObavestenje extends Thread {
   public NitObavestenje(KreiranjeNiti k, String o)
     {
       kn = k;
       obavestenje = o;
     }
   private KreiranjeNiti kn;
   private String obavestenje;

   public void run()
     {
        for (int i = 0; i < kn.brojKlijenta; i++)
          {
            try {
                  kn.on[i].out.println("Echo(" + kn.on[i].brojKlijenta + ") " + obavestenje);
                  System.out.println("Obavestio klijenta " + (i + 1));
                } catch (Exception e) { System.out.println("Nema klijenta " + (i + 1)); }
          }
     }
}
```

Клијент има две нити, главну која ће да шаље податке до сервера, док ће друга нит да прихвата обавештење од сервера.

```
/*  Primer MR10K: Napisati program koji ce kreirati klijentski soket koji ce se  povezati sa serverskim soketom koji je podignut na racunatu cija je IP adresa  127.0.0.1 na portu 8189. Omoguciti da klijent moze u svakom trenutku da primi  poruku od servera.
*/

/* author Sinisa Vlajic
* SILAB – Laboratorija za Softversko Inzenjerstvo
* FON – Beograd
* http://silab.fon.bg.ac.yu
*/
import java.io.*;
import java.net.*;

public class SoketKlijent1
  {
      public static void main(String[] args)
        {   try {  String s;
                   Socket soketK = new Socket("127.0.0.1", 8189);
                   BufferedReader in = new BufferedReader(new InputStreamReader(soketK.getInputStream()));
                   PrintWriter out = new PrintWriter(soketK.getOutputStream(), true);
                   BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
                   boolean signal = true;
                   NitKlijent nk = new NitKlijent(in);
                   nk.start();
                    while (true) { // Prihvata preko tastature podatke
                                   s = br.readLine();
                                   // Salje podatke do servera
                                    out.println(s);
                                  }
                  } catch (Exception e) {   System.out.println(e);   }
        }
}

class NitKlijent extends Thread
   {
     NitKlijent(BufferedReader in1)
       {  in = in1;
          signal = true;
       }

    public void run()
     {
       try {   while (signal)
                  {  // Prihvata podatke od servera
                     String line = in.readLine();
                     if (line.equals("999"))
                       { break; }
                     // Prikazuje podatke na monitoru
                     System.out.println(line);
                   }
            } catch (Exception e) { System.out.println("Lose primljena poruka od servera!"); }
     }

    void Prekini()
     {  signal = false;  }
   boolean signal = true;
   BufferedReader in;
}
```

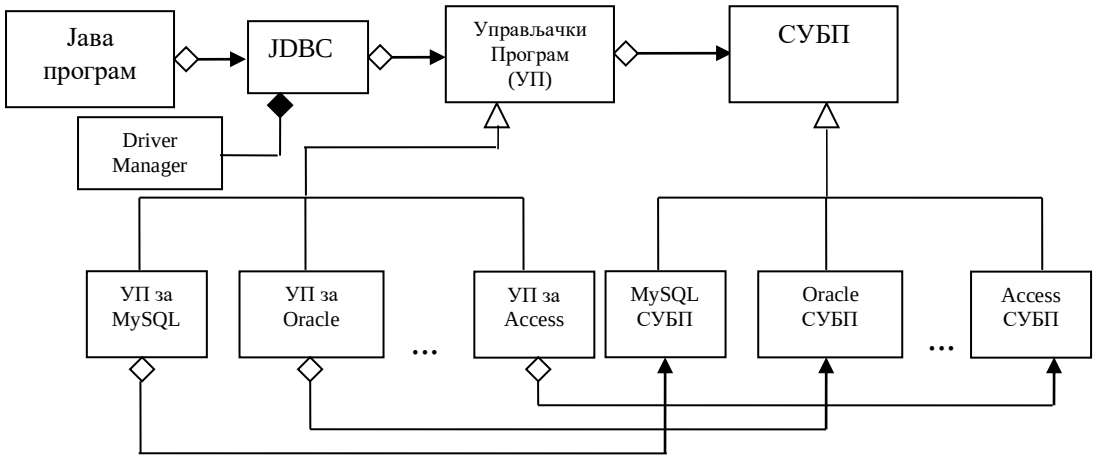
У нашем примеру сервер преко конролера апликационе логике се повезује са више клијената. Клијенти се повезују са сервером преко контролера корисничког интерфејса:

```
/*
 * KontrolerAL.java
 *
 * 02.05.2011
 *
 * @autor  Dr Sinisa Vlajic
 *
 * Katedra za softversko inzenjerstvo
 *
 * Laboratorija za softversko inzenjerstvo
 *
 * Fakultet organizacionih nauka - Beograd
 *
 */

import java.net.*;
import java.io.*;


class KontrolerAL // Kontroler aplikacione logike
 {   static ServerSocket ss;
     static Klijent kl[];

    public static void main(String[] args) throws Exception
              {   kl = new Klijent[10];
                  ss = new ServerSocket(8189);

                  System.out.println("Podignut je serverski program:");
                  for (int brojKlijenta = 0;brojKlijenta < 10;brojKlijenta ++)
                        { Socket soketS = ss.accept();
                          System.out.println("Klijent " + (brojKlijenta+1));
                          // pravi se nova nit za svakog klijenta koji se povezе sa serverskim programom
                          kl[brojKlijenta] = new Klijent(soketS,brojKlijenta+1);
                        }
              }
}


class Klijent extends Thread
            { public Klijent(Socket soketS1,int brojKlijenta1)
              { soketS = soketS1; brojKlijenta=brojKlijenta1;
                System.out.println("Konstruktor");
                start();
              }

              public void run()
              { try { String signal = "";
                      out = new ObjectOutputStream(soketS.getOutputStream());
                      in = new ObjectInputStream(soketS.getInputStream());

                      System.out.println("run");
                          while (true)
                             { // Citanje naziva operacije i racuna
                                String NazivSO = (String) in.readObject();
                                OpstiDomenskiObjekat rac = (OpstiDomenskiObjekat) in.readObject();
                                 if (NazivSO.equals("kreirajNovi")==true)
                                      signal = KreirajNovi.kreirajNovi(rac);
                                 if (NazivSO.equals("Pretrazi")==true)
                                     signal = Pretrazi.Pretrazi(rac);
                                  ...
                              // Slanje promenjenog racuna i signala o uspesnosti operacije
                                 out.writeObject(rac);
                                 out.writeObject(new String(signal));
                              }
                    } catch (Exception e)  { System.out.println(e);  }
              }
             private
             Socket soketS;
             int brojKlijenta;
             ObjectOutputStream out;
             ObjectInputStream in;
}
```

```
/*
 * KontrolerKI.java
 *
*
 * @autor  Dr Sinisa Vlajic
 *
 * Katedra za softversko inzenjerstvo
 *
 * Laboratorija za softversko inzenjerstvo
 *
 * Fakultet organizacionih nauka - Beograd
 *
 */

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.table.*;
import java.net.*;
import java.io.*;

abstract class OpstiKontrolerKI
  { Socket soketK;
    ObjectOutputStream out;
    ObjectInputStream in;
    String signal;
    OpstiDomenskiObjekat odo;
    OpstaEkranskaForma oef;

    OpstiKontrolerKI() throws IOException
      { soketK = new Socket("127.0.0.1",8189);
        out = new ObjectOutputStream(soketK.getOutputStream());
         in = new ObjectInputStream(soketK.getInputStream());
      }

   public String SOPretrazi()
    {   odo = oef.kreirajObjekat();
        KonvertujGrafickiObjekatUDomenskiObjekat();
        /******** POZIVA SE KONTROLER APL. LOGIKE DA IZVRSI SISTEMSKU OPERACIJU *********/
                signal = pozivSO("Pretrazi");
        /*************************************************************************************/
        KonvertujDomenskiObjekatUGrafickiObjekat();
        return signal;
     }
    ...

String pozivSO(String nazivSO)
  {
      try { out.writeObject(nazivSO);
            out.writeObject(odo);
          } catch (IOException io) {return "Neuspesno slanje objekata ka serveru.";}

      try { odo = (OpstiDomenskiObjekat) in.readObject();
            signal = (String) in.readObject();
          } catch (Exception e) {return "Neuspesno citanje objekata sa servera";}
     return signal;
  }

... }
```

2.4.1.3 Базе података - JDBC

Повезивање неког прогама који је написан Јави и неког од Система за управљање базом података (*MySQL, Oracle, SQL Server,..., MS Access*) ради се преко: а) Јавиног *JDBC* (*Java Database Conectivity*) *API*-а и b) управљачког програма (драјвера) који се прави посебно за сваки СУБП (Слика СлБП1).



СлБП1: Веза Јава програма са СУБП

Једном написан Јава програм који се извршава над неким Системом за управљање базом података (СУБП) може се извршавати непромењен и над другим СУБП[58]. Једини предуслов извршења неког Јава програма над неким СУБП јесте постојање управљачког програма за тај СУБП.

БП1: Поступак повезивања Јава програма и СУБП-а

Поступак повезивања Јава програма и базе података изабраног СУБП се изводи у следећим корацима:

- укључивање у Јава програм *JDBC API*-а
- учитавање управљачког програма у Јава програм
- успостављање конекције (везе) између Јава програма и базе података изабраног СУБП

У даљем тексту ће наведени кораци бити детаљно објашњени.

а) укључивање у Јава програм *JDBC API*-а

Укључивање *JDBC API*-а у Јава програм се ради преко следеће наредбе[59]:

```
import java.sql.*;
```

б) учитавање управљачког програма у Јава програм

Учитавање управљачког програма у Јава програм се ради преко следеће наредбе:

```
Class.forName("com.mysql.jdbc.Driver");
```

[58] У суштини се Јава програм извршава над неком базом података која се налази унутар изабраног СУБП. Нпр. Јава програм се повезује са базом података *Student* која се налази унутар *MySQL* СУБП.
[59] У пакету *java.sql* налазе се класе које се користе у раду са базама података СУБП.

```
/* Primer BP1: Pokazati na jednom primeru kako se vrši ucitavanje drajvera za MySQL SUBP i za MS Access SUBP*/

class BP1 {
  public static void main(String[] args) {
    try {
      // Ucitavanje drajvera za MySQL bazu
      Class.forName("com.mysql.jdbc.Driver");
      // Ucitavanje drajvera za MS Access bazu
      Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
      System.out.println("Upravljacki programi  su ucitani!");
    } catch (ClassNotFoundException cnfe) {
      System.out.println("Nije ucitan upravljacki program: " + cnfe);
    }
  }
}
```

У даљем тексту покушаћемо да детаљније објаснимо шта је све потребно урадити у окружењу Јава програма како би се успешно извршила наредба:

```
Class.forName("com.mysql.jdbc.Driver");
```

Управљачки програм је у суштини Јавина *class* датотека која се обично налази у некој *jar* датотеци. Тако се на пример један од *MySQL* управљачких програма (***Driver.class***) налази у датотеци:

mysql-connector-java-3.1.12-bin.jar

Датотека *Driver.class* налази се унутар *mysql-connector-java-3.1.12-bin.jar*[60] датотеке у пакету com.mysql.jdbc[61]
На основу наведеног може се закључити да:

1. параметар forName методе класе Class:

```
"com.mysql.jdbc.Driver"
```

у суштини представља путању (com.mysql.jdbc) унутар *mysql-connector-java-3.1.12-bin.jar* датотеке до *Driver.class* датотеке, која је, наглашавамо, **управљачки програм**.

2. уколико желимо да учитамо управљачки програм у Јава програм потребно је повезати Јава програм са *mysql-connector-java-3.1.12-bin.jar* датотеком.
Повезивање Јава програма са *mysql-connector-java-3.1.12-bin.jar* датотеком се ради на стандардан начин[62], као и са било којом другом jar датотеком.

ц) успостављање конекције (везе) између Јава програма и базе података изабраног СУБП.
Успостављање конекције се ради помоћу *JDBC* DriverManager класе. Као резултат успостављања конекције са базом података преко *JDBC* DriverManager класе добија се објекат класе Connection који чува конекцију ка бази података.

[60] Драјвер за *MS Access* се налази у *rt.jar* датотеци.
[61] Када би се датотека *mysql-connector-java-3.1.12-bin.jar* распаковала она би креирала фолдер *jdbc* који се налази испод фолдера *com/mysql.* Прецизније речено добија се следећа хијерархија фолдера: *com/mysql/jdbc.* У фолдеру *jdbc* се налази управљачки програм (*Driver.class*) за *MySQL* СУБП.
[62] Уколико се ради са неким од једноставнијих Јавиних развојних окружења (нпр. *TextPad*) тада је потребно у системској променљивој *CLASSPATH* да се се наведе пут до наведене jar датотеке:
C:\Install\MySQL5.0\mysql-connector-java-3.1.12\mysql-connector-java-3.1.12\mysql-connector-java-3.1.12-bin.jar;
Уколико се ради у неком од сложенијих Јавиних развојних окружења (нпр. *NetBeans)* могуће је у Јава програм на једноставан начин (*properties/AddJar*) укључити наведену *jar* датотеку .

```
/*Primer BP21: Pokazati kako se uspostavlja veza (konekcija) sa MySQL bazom podataka.
*/
import java.sql.*;

class BP21 {
  public static void main(String[] args) {
    try {
      String dbUrl = "jdbc:mysql://127.0.0.1:3306/student";
      String user = "root";
      String pass = "root";
      Class.forName("com.mysql.jdbc.Driver");
      Connection naredba = DriverManager.getConnection(dbUrl, user, pass);
      System.out.println("Uspostavljena je konekcija izmedju driver manager-a i baze");
    } catch (ClassNotFoundException cnfe) {
      System.out.println("Nije ucitan upravljacki program:  " + cnfe);
    } catch (SQLException sqle) {
      System.out.println("Greska: " + sqle);
    }
  }
}
```

Објашњење најважнијих делова програма ВР21:

Када се учита *MySql* управљачки програм у Јава програм преко наредбе:

```
Class.forName("com.mysql.jdbc.Driver");
```

тада управљачки програм добија симболички назив преко кога се он касније позива[63]. У случају наведене наредбе управљачки програм добија симболички назив[64]: jdbc:mysql

У примеру ВР21 се налази наредба:

```
String dbUrl="jdbc:mysql://127.0.0.1:3306/student";
```

у којој променљива dbUrl добија следећу *url* адресу:

```
"jdbc:mysql://127.0.0.1:3306/student"
```

Наведена адреса се може декомпоновати на следеће делове:
Назив управљачког програма (jdbc:mysql) који је учитан преко наредбе:

```
Class.forName("com.mysql.jdbc.Driver");
```

у Јава програм.

IP адреса (*127.0.0.1*) машине на којој се налази *MySQL* СУБП.
Порт (*3306*) на коме је подигнут *MySQL* СУБП.
Име базе података (*student*) са којом Јава програм успоставља конекцију преко наредбе:

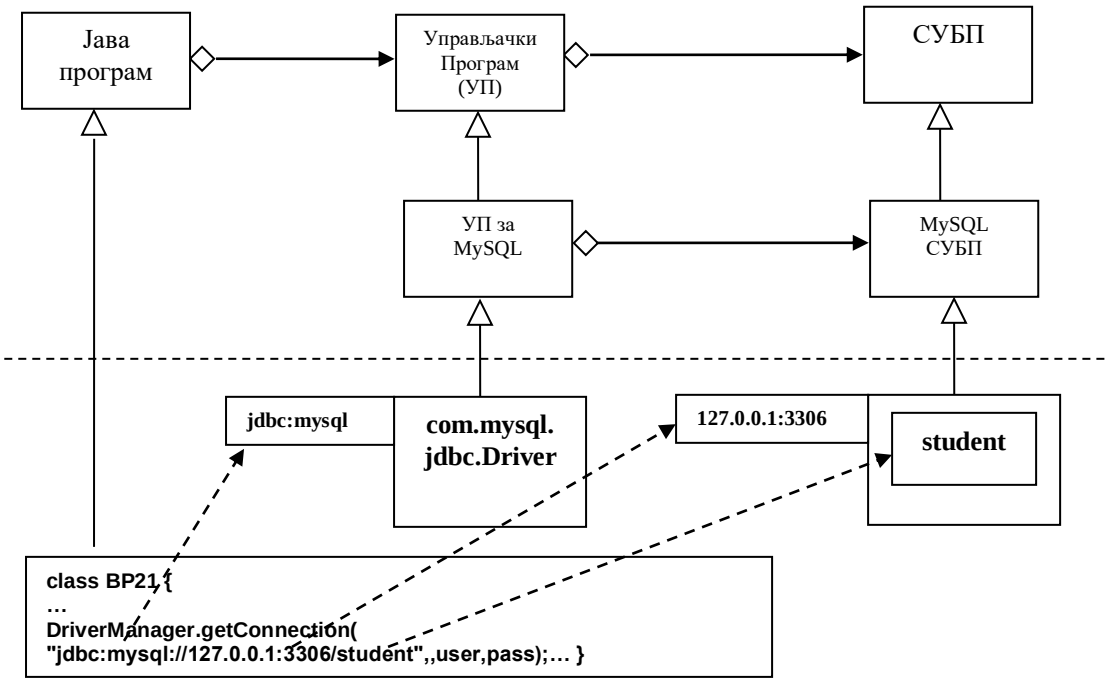
```
Connection CONECTION=DriverManager.getConnection(dbUrl,user,pass);
```

На основу наведеног се може закључити да наредба:

```
String dbUrl="jdbc:mysql://127.0.0.1:3306/student";
```

[63] Тај симболички назив се чува у *Driver Manager*-у. Када се *Driver Manager* напуни са симболичким називом неког драјвера, за такав драјвер кажемо да је регистрован.
[64] Када се учита *MS Access* управљачки програм у Јава програм преко наредбе:
`Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");`
тада управљачки програм добија симболички назив:
`jdbc:odbc`

даје адресу до базе података **student** која се налази на *MySQL* серверу који се налази на адреси: **127.0.0.1:3306** преко управљачког програма чији је симболички назив: jdbc:mysql. Наведени однос између Јава програма B21, управљачког програма се види на слици СлБП2.



СлБП2: Однос између Јава програма, управљачког програма и базе података

На крају се помоћу наредбе:

```
Connection CONECTION=DriverManager.getConnection(dbUrl,user,pass);
```

успоставља конекција са базом података између Јава програма и изабране базе података (*student*). Када се позове метода getConnection(), она итеративно пролази кроз драјвере регистроване у *DriverManager*-у и испитује да ли постоји драјвер са задатим називом. Уколико се пронађе драјвер, класа DriverManager прави објекат Connection, који успоставља везу са базом података, помоћу пронађеног драјвера.

БП2: Поступак извршења операција над базом података СУБП
Поступак извршења операција над СУБП се изводи у следећим корацима:
а) прављење објекта класе `Statement`

Помоћу конекције која је успостављена преко наредбе:

```
Connection konekcija =DriverManager.getConnection(dbUrl,user,pass);
```

се прави објекат наредба класе `Statement` преко:

```
Statement naredba=konekcija.createStatement();
```

Помоћу објекта наредба се изводе операције над базом података преко:

```
naredba.executeQuery(upit);
```

У примеру ВР31 ће се видети како се изводи операција *SELECT* над базом података.

```
/* Primer BP31: Napisati program  kojim se prikazuje trenutni sadrzaj tabele Student.
Baza podataka, u kojoj se nalazi tabela Student, je realizovana preko MySQL i MS Acess
SUBP[65]*/
import java.sql.*;

class BP31 {
  public static void main(String[] args) {
    try {
      String dbUrl = new String();
      String user = "root";
      String pass = "root";

      if (args[0].equals("1")) {
        dbUrl = "jdbc:odbc:student";
        Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
      }

      if (args[0].equals("2")) {
        dbUrl = "jdbc:mysql://127.0.0.1:3306/student";
        Class.forName("com.mysql.jdbc.Driver");
      }

      Connection konekcija = DriverManager.getConnection(dbUrl, user, pass);
      Statement naredba = konekcija.createStatement();
      String upit = "SELECT brind,ime,prezime FROM Student";
      ResultSet rs = null;
      try {
        rs = naredba.executeQuery(upit);
      } catch (SQLException sqle) {
        System.out.println("Greska u izvr. upita: " + sqle);
      }

      System.out.println("Trenutan izgled tabele studenata!");
      while (rs.next()) {
        System.out.println(rs.getString("brind") + " " + rs.getString("ime")
            + " " + rs.getString("prezime"));
      }

      naredba.close();
      konekcija.close();
    } catch (ClassNotFoundException cnfe) {
      System.out.println("Nije ucitan upravljacki program:  " + cnfe);
    } catch (SecurityException se) {
      System.out.println("Nedozvoljena operacija: " + se);
    } catch (SQLException sqle) {
      System.out.println("Greska konekcije: " + sqle);
    }
  }
}
```

.

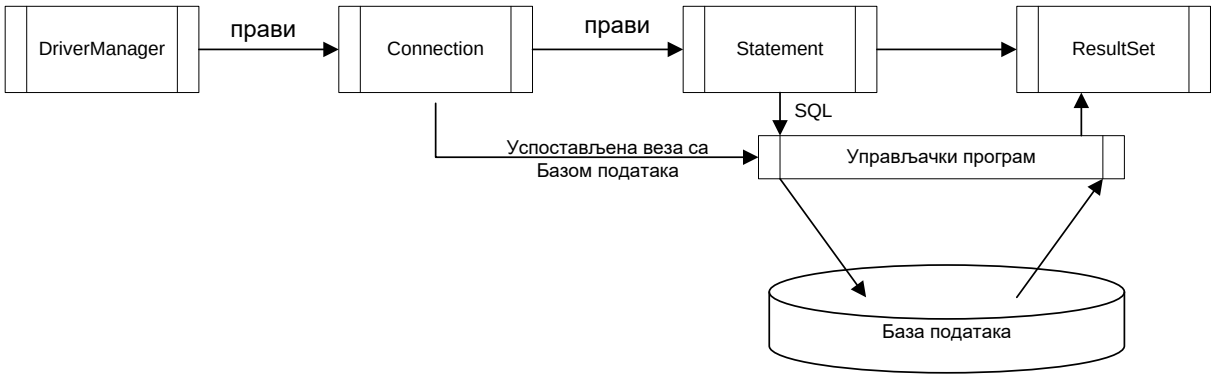
У наведеном примеру се упит:

"SELECT brind,ime,prezime FROM Student"

шаље као параметар методе `executeQuery` која се извршава над објектом наредба. Резултат те наредбе се чува у `ResultSet`-у `rs`. У `ResultSet`-у се чувају сви слогови табеле *Student.* Кроз наведени `ResultSet` се пролази и приказује се његов садржај.

На слици СЛБПЗ се виде одговорности класа које учествују у поступку извршења операције над базом података..

[65] Уколико се жели из Јава програма приступити некој *MS Access* бази података која се налази на локалном рачунару, мора се приступити регистрацији базе података. Регистрација базе се на *Windows* оперативним системима обавља у *ODBC Data Source Administrator*-у који се налази на путањи: ***Start -> Control Panel -> Administrative Tools -> Data Sources.***



СлБП3: Поступак извршења операције над базом података

```
/* Primer BP4: Napisati program koji u tabelu Student  sa atributima broj indeksa, ime
i prezime unosi novog studenta sa brojem indeksa 01/06 ,cije je ime Pera , a   prezime
Peric.*/
import java.sql.*;

class BP4 {
  public static void main(String[] args) {
    try {
      String dbUrl = "jdbc:odbc:student";
      String user = "";
      String pass = "";
      Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
      Connection CONECTION = DriverManager.getConnection(dbUrl, user, pass);
      String upit = "INSERT INTO Student(brind,ime,prezime) VALUES (?,?,?)";
      PreparedStatement PSTATEMENT = CONECTION.prepareStatement(upit);
      PSTATEMENT.setString(1, new String("01/06"));
      PSTATEMENT.setString(2, new String("Pera"));
      PSTATEMENT.setString(3, new String("Peric"));
      try {
        PSTATEMENT.executeUpdate();
        System.out.println("Novi student je zapamcen u bazi");
      } catch (SQLException e) {
        System.out.println("Izuzetak: " + e);
      }
      PSTATEMENT.close();
      CONECTION.close();
    } catch (ClassNotFoundException cnfe) {
      System.out.println("Nije ucitan upravljacki program:   " + cnfe);
    } catch (SQLException sqle) {
      System.out.println("Greska kod konekcije: " + sqle);
    }
  }
}
```

Литература

Beck, K., Beedle, M., van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., … Thomas, D. (2001). *Manifesto for Agile Software Development*. https://agilemanifesto.org

Boehm, B., & Turner, R. (2004). *Balancing Agility and Discipline: A Guide for the Perplexed*. Addison-Wesley.

Highsmith, J. (2002). *Agile Software Development Ecosystems*. Addison-Wesley.

Larman, C. (2004). *Agile and Iterative Development: A Manager's Guide*. Addison-Wesley.

Pressman, R., & Maxim, B. (2014). *Software Engineering: A Practitioner's Approach* (8th ed.). McGraw-Hill.

Sommerville, I. (2016). *Software Engineering* (10th ed.). Pearson.

Schmidt, D. C. (2006). *Model-driven engineering*. *IEEE Computer*, 39(2), 25–31. https://doi.org/10.1109/MC.2006.54

Larman, C. (1998). *Applying UML and patterns: An introduction to object-oriented analysis and design*. Prentice Hall, New Jersy.

Петровић, Б. (1998). Теорија система, ФОН, Београд.

Jacobson, I., Booch, G., & Rumbaugh, J. (1999). *The unified software development process*. Addison-Wesley.

Endres, A., & Rombach, H. D. (2003). *A handbook of software and systems engineering: Empirical observations, laws and theories*. Addison-Wesley.

Tsichritzis, D. C., & Klug, A. (1978). *The ANSI/X3/SPARC DBMS framework: Report of the Study Group on Database Management Systems*. Information Systems, 3(4), 173-191.

Schulte, R. (1995). *Three-Tier Computing Architectures and Beyond*, Published Report Note R-401-134, Garther Group.

Budgen, D. (2003). *Software Design* (2nd ed.). Addison-Wesley.

Booch, G., Maksimchuk, R. A., Engle, M. W., Young, B. J., Conallen, J., & Houston, K. A. (2007). *Object-Oriented Analysis and Design with Applications* (3rd ed.). Addison-Wesley Professional.