Софтверски патерни (Software patterns)

Book · January 2014

CITATIONS
0

READS
7,770

1 author:

Siniša Vlajić
Faculty of organisational sciences - University of Belgrade
44 PUBLICATIONS **68** CITATIONS

SEE PROFILE

УНИВЕРЗИТЕТ У БЕОГРАДУ

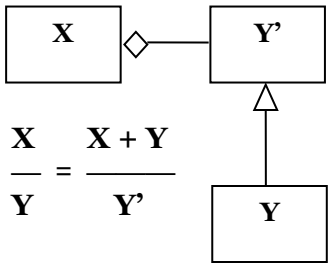
ФАКУЛТЕТ ОРГАНИЗАЦИОНИХ НАУКА

Катедра за софтверско инжењерство

СОФТВЕРСКИ ПАТЕРНИ

Аутор:

Др Синиша Влајић ред.проф.



Београд - 2014.

Аутор
Др Синиша Влајић, ред.проф.

Наслов
СОФТВЕРСКИ ПАТЕРНИ

Прво издање

Издавач
Агенција за пружање интелектуалних услуга, издаваштво и трговину
“ЗЛАТНИ ПРЕСЕК”
Илије Ђуричића 56, Београд
Тел/факс: (011) 2 502-872

Главни уредник
Сузана Влајић

Рецензент
Др Владан Девеџић, ред. проф. ФОН-а, Универзитет у Београду
Мр Бранислав Селић, научни сарадник (Adjunct Professor) за рачунарске науке, Универзитет у Торонту

Припрема и дизајн
Аутор

Коректура
Аутор

CIP - Каталогизација у публикацији
Народна библиотека Србије, Београд

004.424.6(075.8)

ВЛАЈИЋ, Синиша, 1968-
Софтверски патерни / Синиша Влајић. - 1. изд. - Београд : Златни пресек, 2014 (Београд : PC Systems). - 145 стр. : илустр. ; 30 cm. - (#Библиотека #Софтверско инжењерство)

На врху насл. стр.: Универзитет у Београду, Факултет организационих наука, Катедра за софтверско инжењерство. - Тираж 200. - Напомене уз текст. - Библиографија: стр. 144-145.

ISBN 978-86-86887-30-6

а) Програмирање - Препознавање образаца
COBISS.SR-ID 210069260

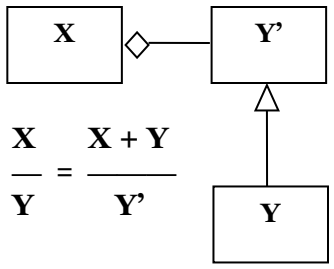
Напомена: Електронска верзија књиге Софтверски патерни је бесплатна. Уколико сте у могућности донирајте Универзитетску дечију клинику, Тиршова 10, Београд.
Веб адреса клинике, на којој се налазе информације везано за донацију су:
http://tirsova.rs/lat/donacije-na-racun/

БИБЛИОТЕКА

СОФТВЕРСКО ИНЖЕЊЕРСТВО

УРЕДНИК

СИНИША ВЛАЈИЋ



ЗЛАТНИ ПРЕСЕК

Предговор

У књизи Софтверски патерни аутор је на веома инспиративан начин објаснио кључне концепте који су потребни како би се патерни пројектовања у целини схватили. На почетку књиге аутор даје кратку историју настанка софтверских патерна, са нагласком на особе које су поставиле темеље развоја патерна, као једне од веома битних дисциплина у софтверском инжењерству. Дат је преглед основних дефиниција патерна и њихова класификација. У наставку аутор објашњава шта треба урадити како би пројектовани софтверски систем био применљив као решење скупа сличних проблема који покривају неку класу проблема. Размишљање аутора о патернима као механизму који држи "хаос" и "ред" у непрекидној равнотежи, указује на веома битно својство патерна које се може применити за различите системе, не само на софтверске. Проблемом формализације патерна аутор се бави од 2000. године, када је почео да ради докторску дисертацију, коју је одбранио 2003. године.
.Аутор је у књизи навео најважније формализме (језике патерна) који се данас користе. Полазиште његовог истраживања су радови Едена и Коплина, који се односе на формализацију патерна. Аутор критички сагледава суштину постојећих језика патерна пројектовања, сматрајући да они нису дали прецизну формалну дефиницију патерна пројектовања у општем смислу, како би објаснили како и када треба користити патерне. Аутор покушава формално, преко симетријских концепата, да успостави везу између елемената проблема и решења код патерна пројектовања како би направио формалну основу за прављење стабилних и одрживих софтверских система, који ће моћи у исто време и да се мењају и да буду имуни на промене. Његова истраживања формализације патерна се могу свести на дефиницију до које је дошао: *"Патерн је процес који трансформише несиметријску структуру (проблем) у симетријску структуру (решење)"*.
Током истраживања патерна аутор је указао на кључни механизам или својство GOF патерна пројектовања, који му је помогао да дефинише општи облик GOF патерна пројектовања. Урађена је анализа 23 GOF патерна пројектовања и доказано је да се општи облик GOF патерна пројектовања може применити за њих 20. Аутор је веома детаљно образложио симетријске концепте и њихову везу са UML-овим дијаграмима класа и објектним дијаграмима, са посебним акцентом на схватање концепта симетријске трансформације и симетријске групе. Помоћу наведених концепата аутор је дао прецизну дефиницију GOF патерна помоћу симетрије. Посебно је интересантно размишљање аутора о могућој вези између софтверских и физичких система помоћу концепата симетрије.
Након теоријског разматрања патерна, аутор даје преглед основних објектно-оријентисаних концепата потребних да би се схватили патерни пројектовања. Ту је посебан нагласак стављен на наслеђивање класа и касно повезивање објекта са методом. Затим је дат један скуп примера (програма) где се може видети практична примена општег облика GOF патерна пројектовања. Детаљно је објашњено како се праве генеричке методе и дати су примери за рефлексију и коришћење Јавиног generic концепта.
..
У другом делу књиге аутор веома детаљно објашњава сваки од GOF патерна пројектовања. За сваки патерн је дат конкректан пример са одговарајућим дијаграмом класа и објектним дијаграмом. Оно што је посебно интересантно, што омогућава лакше праћење примера, јесте исти домен проблема који се користи у сваком примеру. Онај ко буде читао ову књигу и ко буде схватио првих неколико патерна пројектовања, моћи ће веома брзо и лако да схвати и остале патерне пројектовања. Аутор у књизи објашњава шта су ECF и MVC патерни кроз неколико примера. Поред тога, аутор објашњава и имплементационе патерне, односно идиоме.
У последњем делу књиге аутор објашњава различите концепте, као што су фазе прикупљања захтева и анализе у развоју софтвера, неке од софтверских технологија и објектно-оријентисаних принципа пројектовања софтвера и њихову везу са општим обликом GOF патерна. Посебно је интересантно ауторово виђење неколико мудрих прича из перспективе патерна. На крају књиге аутор на филозофско-песнички начин описује патерне и њихову суштину. На тих неколико последњих страна аутор, заправо, открива лично филозофско виђење патерна као нечега што у великој мери превазилази оквире пројектовања софтвера и затвара круг започет на почетку књиге где говори о патернима као механизму који држи "хаос" и "ред" у равнотежи.
Сматрам да је књига одлично урађена и написана и са задовољством је предлажем за објављивање.

Рецензент: др Владан Девецић, ред. проф.

По својој функционалности и обиму, савремени софтверски програми, неспорно спадају у најсложеније техничке уређаје које је човечанство икад остварило. Штавише, умрежењем путем интернета, ови системи досежу и глобалне размере. С обзиром на далекосежност и значај ових система, сасвим је разумљиво очекивати да иза софтвера стоји солидна научна основа, која гарантује њихову исправност и поузданост.

Нажалост, за разлику од других много зрелијих инжењерских струка, софтверско инжењерство још увек пати од недостатка чврстих научних темеља, са доказаним и опште прихваћеним теоријским начелима и приступима. То се стање одсликава и у области такозваних софтверских образаца или "патерна" (од енглеске речи "pattern"), који су настали као резултат нагомиланог, у суштини занатског приступа, развоју софтвера. У питању су нека уопштена техничка решења која појединачно покривају целе категорије хомоморфних конкретних софтверских проблема. Иако су доступне многобројне књиге које описују и чак класификују патерне, још увек се осећа недостатак како теоријских тако и методолошких основа, који би служили као водиље у њиховој практичној примени. Шта, заправо, сачињава софтверски патерн? Како да га препознамо, како да га дефинишемо, и како да знамо где и на који начин да га применимо?

У овој књизи нам професор Влајић систематично одговара на ова и слична питања, користећи се између осталог и формалним математичким језиком – успостављајући тиме потребну научну основу за патерне – као и бројним практичним примерима, који су непоходни да читалац стекне интуитивни осећај о томе шта стоји иза теоријских поставки. Колега Влајић то постиже тиме што открива патерне у самим патернима (то јест, мета-патерне), повезујући их са дубљим филозофским размишљањима о нужној динамичкој равнотежи између симетрије и њеног нарушавања. Ово последње је извор креативности која, за узврат, води новим патернима и новој равнотежи. Са те тачке гледишта, Влајић открива да су софтверски патерни само специфична технолошка манифестација неких универзалних решења од којих нека потичу чак из праисторије. На крају, наспрам прецизне техничке природе већег дела текста, у завршном кратком поглављу, аутор нас, такође, подсећа да је пројектовање и реализација софтвера у суштини стваралачки чин, који као и стваралаштво у уметности, има и врло опојну, захвалну компоненту.

Сасвим је извесно да су нам преко потребни овакви уџбеници, јер софтвер, као једна од кључних технологија двадесет и првог века, мора врло брзо да превазиђе своју првобитну занатску фазу и да се преобрази у праву инжењерску дисциплину. Овом књигом, професор Влајић нам показује један пут који нас води том циљу.

Рецензент: Бранислав Селић (Париз, 2014)

Садржај

1. Увод	1
2. Основе о патернима	3
2.1 Основне дефиниције патерна	3
2.2 Класификација софтверских патерна	4
2.3 Улога патерна у развоја софтвера	4
2.4 Поновна употребљивост патерна	4
2.5 Када треба користити софтверске патерне	6
2.6 Патерни, ред и хаос	6
2.7 Примењивост патерна у живота	7
3. Анализа патерна пројектовања	8
3.1 Формализација патерна	8
3.2 Општи облик GOF патерна пројектовања	10
3.3 Патерни пројектовања и симетрија	14
3.3.1 Симетријска трансформација и симетријска група	14
3.3.2 Трансформације код дијаграма класа и објектних дијаграма	15
3.3.3 Симетријска трансформације код дијаграма класа и објектних дијаграма	17
3.3.4 Симетријска група код дијаграма класа	17
3.3.5 Дефиниција GOF патерна помоћу симетрије	22
4. Програмски концепти патерна пројектовања	24
4.1 Објектно-оријентисани концепти потребни за схватање патерна пројектовања	24
4.1.1 Класе и објекти	24
4.1.2 Наслеђивање и динамички полиморфизам	25
4.1.3 Касно повезивање објекта са методом	26
4.1.4 Апстрактне класе и интерфејси	27
4.2 Имплементација општег облика GOF патерна пројектовања	28
4.3 Прављење генеричких метода	33
4.4 Рефлексија	39
4.5 Generic механизам	41
5. Патерни пројектовања и имплементације	43
5.1: GOF патерни пројектовања	43
ПК: Патерни за креирање објеката	43
ПК1 – Abstract Factory	44
ПК2 – Builder	48
ПК3 – Factory method	51
ПК4 – Prototype	55
ПК5 – Singletone	58
СП: Структурни патерни	61
СП1: Adapter	62
СП2: Bridge	66
СП3: Composite	69
СП4: Decorator	75
СП5: Facade	79
СП6: Flyweight	81
СП7: Proxy	85
ПП: Патерни понашања	88
ПП1: Chain of responsibility	89
ПП2: Command	92
ПП3: Interpreter	95
ПП4: Iterator	100
ПП5: Mediator	103
ПП6: Memento	105
ПП7: Observer	108

ППП8: State	111
ППП9: Strategy	113
ППП10: Template method	115
ППП11: Visitor	117
5.2 ECF и MVC патерни	121
5.3 Имплементациони патерни - идиоми	128
6. Веза општег облика GOF патерна пројектовања и других концепата	130
6.1 Фаза прикупљања захтева и патерни	130
6.2 Фаза анализе и патерни	133
6.3 Софтверске технологије и патерни	135
6.4 Принципи објектно-оријентисаног пројектовања и патерни	137
6.5 Мудре приче и патерни	140
7. Филозофија патерна – субјективни осврт	143
8. Литература	144

СОФТВЕРСКИ ПАТЕРНИ

Посвећено мојој породици Александри, Драги, Ђорђу и Сузи ♥

1. Увод

Једног лепог дана, давне 2000-те године, мој тадашњи шеф проф. др Видојко Ћирић, донео ми је из Канаде књигу ***Design Patterns: Elements of Reusable Object-Oriented Software*** и рекао ми је: "*Ова књига је број један данас на западу, погледај је, можда ћеш у њој наћи нешто интересантно за твој докторат!* ". Узео сам књигу, не схватајући да узимам нешто што ће у значајној мери да утиче на моје стручне и животне погледе и ставове. Тада сам се први пут упознао са појмом патерна и једноставно они су постали део неке моје животне приче. Искрен да будем, када сам први пут прочитао ту књигу, био сам у некој врсти "полу-кошмара". Осећао сам да се ту дешава нешто веома битно али нисам могао да схватим шта је то. После другог, трећег, не знам ни сам ког читања, коначно су почели "*да ми се отварају видици*". Схватио сам шта је суштина патерна, и следеће три године радио сам докторат у коме сам се бавио математичком формализацијом патерна. Идеја доктората се односила на схватање патерна у општем смислу и његова математичка формализација.

...

У то време једино се Еден (Amnon Eden) озбиљно, у светским оквирима, бавио формализацијом патерна. Еден је тада направио патерн језик (***LePUS***) којим је желео да формализује патерне пројектовања. Почео сам да проучавам његове радове, посебно његов докторат, који су ми помогли да дубље схватим патерне пројектовања из песпективе математичког формализма који је он развио.

Тај приступ, без обзира на његов значај у афирмацији и развоју патерна, није ми се генерално свидео, јер нисам схватао шта суштински добијам тиме ако патерн, опишем математичким формализмом а не програмским кодом. Еден је све то радио како би (упрошћено речено) у неком програмском коду препознао да ли постоји неки од постојећих GOF патерна пројектовања. Тада сам се питао, као што се и сада питам: "Шта се тиме добија?". Можда грубо звучи, али рекао бих ништа специјално. Шта мени значи ако констатујем да се у програму налази нпр. State или Visitor патерн. Мене интересује да ли у програму постоје места где се требају уградити патерни, како би програм могао лако да се одржава и надограђује. Која је сврха појединачних патерна пројектовања ако се не схвате особине које има сваки патерн.

...

Путоказ у даљем истраживању патерна, дали су ми радови Коплина (Jim Coplien), који је указао на значај симетрије у формализацији патерна. Докторат сам завршио 2003. године и тада сам развио формалну теорију патерна, која је на оригиналан начин описала општи облик GOF патерна пројектовања коришћењем симетријских концепата. Ментор докторске дисертације је био проф. др. Видојко Ћирић, а чланови Комисије, проф. др. Бранислав Лазаревић, проф. др. Душан Велашевић, проф. др. Братислав Петровић и проф. др. Божидар Раденковић.

...

Након тога сам ступио у непосредну комуникацију са Коплином, који ми је доста помогао да још боље схватим патерне и да критички сагледам све дотадашње резултате до којих сам дошао у истраживању. Коплин је 2004. године (први пут) гостовао на ФОН-у, када је одржао изваредан курс за наше нставнике и студенте под називом: ***Pattern theory and practice – Toward a general design theory.***

...

После 14 година проучавања патерна и њихове примене у развоју одрживих софтверских система, написао сам књигу, у којој сам покушао да систематизујем и објасним теме које се односе на патерне пројектовања. Књига се састоји из осам поглавља. Након увода објаснио сам основе патерна (друго поглавље), у којима сам дао преглед најважнијих дефиниција патерна, њихову класификацију и улогу коју они имају у развоју софтвера. Посебан акценат је стављен на поновну употребљивост патерна у развоју нових софтверских система и ситуацијама када треба користити патерне како би се увео "ред" у програм и спречио потенцијални "хаос" током развоја и одржавања програма. На крају првог поглавља сам објаснио, из моје субјективне перспективе, примену патерна у свакодневном животу.

У трећем поглављу сам извршио анализу патерна пројектовања, тако што сам дао преглед неких од познатих језика патерна, који се користе код формализације патерна пројектовања. Након тога сам описао општи облик GOF патерна пројектовања, кога сам формализовао помоћу симетријских концепата.

У четвртом поглављу објашњени су програмски концепти који су потребни како би се схватили GOF патерни пројектовања. Дат је преглед објектно-оријентисаних концепата (објекат, класа,

апстрактна класа, интерфејс и наслеђивање). Затим је наведено неколико примера (програма) у којима се примењује општи облик GOF патерна, када се праве генеричке методе. На крају поглавља су дати примери рефлексије и generic механизма у Јави.
Пето поглавље се бави патернима пројектовања и имплементације. Објашњени су сви GOF патерни пројектовања, тако што је дата њихова дефиниција, структура, учесници у патерну и одговарајући пример. За сваки пример је дат програм као и дијаграм класа и објектни дијаграм за тај програм. Примери су развијани кроз исти домен проблема, тако да схватање почетних патерна, односно њихових примера доста олакшава схватање примера осталих патерна. Затим су представљени, преко примера, ECF и MVC патерни. На крају поглавља је објашњено неколико имплементационих патерна – идиома.
У шестом поглављу је дата веза општег облика GOF патерна пројектовања и других концепата. Описана је веза општег облика патерна са фазама прикупљања захтева и анализе у развоју софтвера. Размотрено је неколико софтверских технологија и принципа објектно-оријентисаног пројектовања помоћу општег облика патерна. На крају поглавља је дато неколико мудрих прича које су објашњене преко патерна.
У седмом поглављу сам написао кратко субјективно филозофско размишљање о патернима. На крају књиге је дата литература.
...
Узгред, док сам "брусио" ову књигу, у последњих месец дана интезивног рада, непрекидно сам слушао песме Џенис Џоплин, које су ме веома надахнуле и инспирисале, да са пуно воље и елана радим и приведем крају овај велики посао.
...
У ужем смислу, ова књига је намењена за студенте IV године **Факултета организационих наука, Универзитета у Београду**, за предмет **Софтверски патерни**, који се слуша као изборни предмет на основним студијама, на студијском програму **Информациони системи и технологија.**
У ширем смислу ова књига је намењена свима онима који користе или намеравају да користе патерне у развоју и одржавању објектно-оријентисаних софтверских система.

Аутор

2. Основе о патернима

2.1 Основне дефиниције патерна

Корени патерна се налазе у радовима Кристофера Александера (*Christopher Alexander*), који се односе на пројектовање архитектуре у грађевинарству *[AC1-AC3]*. Многи термини, који се користе код софтверских патерна, као што су силе (forces), патерни (patterns), језици патерна (pattern languages), и други долазе од Александера.

Александер је дао следећу дефиницију патерна *[AC3]*: "*Сваки патерн је троделно правило, које успоставља релацију између неког проблема, његовог решења и њиховог контекста. Патерн је у исто време и ствар, која се дешава у стварности, и правило које говори када и како се креира наведена ствар". (Деф1)*

Када говори о патернима *[AC2]* Александер каже да сваки патерн описује неки **проблем** који се више пута понавља на различите начине и **решење** тог проблема. Решење се код патерна може применити за скуп сличних проблема који покривају неку класу проблема. Александер је уочио патерне на основу структура градова и њихових грађевина. Он је сматрао да документовање ових патерна може помоћи људима у свакодневном животу. Патерн треба да прати одговарајућа документација, како би се олакшало налажење решење у сличној проблемској ситуацији. Александер је покушао да објасни патерне и из перспективе људког мишљења *[AC3]*: *Патерни у нашем мишљењу су мање више, менталне слике патерна у стварности. Они су апстрактна репрезентација морфолошких правила која дефинишу патерне у стварности. У нашем мишљењу патерни су динамички јер имају неку силу. Поред тога патерни имају особину генералности.*

Историјски гледајући, Бек *(Kent Back)* и Канингхам започели су 1987. године експерименте са идејом примене патерна у програмирању и презентовали су њихове резултате на OOPSLA[1] конференцији *[UPLOOP]*. Након тога и други значајни аутори у овој области почињу да истражују и развијају софтверске патерне (*Erich Gamma, Jim Coplien, Ward Cunningham, Richard Helm, Bruce Anderson, Ralph Johnson, John Vlissides, Desmond De Souza, Richard Gabriel,...).* Патерни доживљавају пуну афирмацију 1994. године са појавом књиге ***Design Patterns: Elements of Reusable Object-Oriented Software,*** коју су написали *Erich Gamma, Richard Helm, Ralph Johnson* и *John Vlissides*[2] *[GOF].*

Габриел (*Richard Gabriel*) је дао дефиницију софтверских патерна *[GABR1]* која је заснована на дефиницији *Деф1* коју је дао Александер: *"Сваки патерн је троделно правило, које успоставља релацију између неког* ***контекста****, неког система сила који се понављају у том контексту* ***(проблем)*** *и софтверске конфигурације која омогућава тим силама да успоставе одговарајуће односе* ***(решење)****".* Термин *контекст* указује на неки софтверски систем са дефинисаним ограничењима, док се термин *сила* користи да укаже на елементе тог софтверског системе и њихове међусобне односе. Термин *софтверска конфигурација* указује на структуру софтверског система у којој су односи између елемената софтверског система такви да омогућавају поновну употребу те структуре у различитим проблемским ситуацијама. На могућност поновне употребе патерна указују и Рихл (*Dirk Riehle*) и Цулиговен (*Heinz Zullighoven*) који кажу *[Dirk1]* да *патерн представља апстракцију конкретног облика која се може поново користити у специфичним контекстима.*

Коплин *(Jim Coplien)* за патерн каже *[Cop1]*: *Патерн је правило за грађење ствари, али је оно истовремено и сама ствар.* Он наглашава да патерни не представљају методу пројектовања. Они обухватају праксу која се развијала из постојећих метода развоја софтвера. У том смислу он каже да патерни нису CASE алат који по аутоматизму може извршити неку трансформацију. Патерни су пре свега резултат људске активности и управо они праве разлику између људске и компјутерске интелигенције. Добро решење код патерна има довољно детаља који говоре шта патерн ради, али то решење је истовремено и довољно генерално да укаже на широк контекст могућих проблема које оно задовољава. Канингхам *(Ward Cunningham)*, када у метафори објашњава патерне, каже да је патерн нешто што више личи на рецепт него на план *[Cop1]*: "*Ја желим да направим разлику између рецепта и плана. План се може добити, инверзним поступком, из постојеће грађевине, али рецепт се не може добити, инверзним поступком".*

[1] OOPSLA - Object-Oriented Programming, Systems, Languages & Applications

[2] Наведени аутори су познати као "четворочлана банда" (GOF - Gang of Four).

2.2 Класификација софтверских патерна

Софтверски патерни се могу класификовати на следећи начин:

а) Тронивојски патерни – Постоје три нивоа апстракције којима може бити придружен неки патерн *[Cop1]*. За најнижи ниво апстракције се везују **идиоми**(*idioms*). За средњи ниво апстракције се везују **узори пројектовања** (*design paterns*). Док се за највиши ниво апстракције везују **оквири** (*framework*).

- **Идиоми** су патерни најнижег нивоа који зависе од специфичне имплементационе технологије, као што је нпр. програмски језик. Коплин је у том смислу дао посебан допринос *[Cop3]*.
- **Патерни пројектовања** су патерни који су независни од конкренте имплементационе технологије. Они су микроархитектура: они садрже структуру која може бити сложена али не и довољно велика да би се за њу рекло да је подсистем[3]. Главни допринос у схаватању патерна пројектовања дао је Гама (*Erich Gamma*) и други у књизи *Design Patterns[GOF]*.
- **Оквири** су патерни системског нивоа. Они су пројектовани тако да садрже комплетан програмски код једне од основних функција система или целог система, који може бити проширен за конкретну апликацију. То значи да оквир обезбеђује опште решење а различите апликације га користе и проширују својим специфичностима. Један оквир може да садржи друге патерне различитих нивоа апстракције (*идиоме, узоре пројектовања и оквире*).

б) Анти-патерни –Патерни који су се у пракси показали као лоши, који не раде или делимично раде називају се анти-патерни. На анти-патерне указали су у независним истраживањима Sam Adams и Andrew Koenig *[Koe1]*. У тим истраживањима наведени аутори су објаснили последице лоших решења код анти-патерна.

ц) Мета патерни – У истраживањима Приа *(Wolfgang Pree)* се први пут уводи појам мета-патерни *[Pree]*. Он је генерализовао кључне структуре многих GOF патерна. Наведене структуре су представљале блокове (мета-патерне) из којих се могу направити други патерни.

2.3 Улога патерна у развоја софтвера

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 (**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. Тако имамо патерне захтева, патерне анализе и имплементационе патерне (идиоме) који су везани за конкретне технологије, као што су нпр. Јава или C#. *Патерни који се користе у развоја софтверских система, имају улогу да : а) олакшају развој нових софтверских система и б) помогну у одржавању и надоградњу постојећих софтверских система.*

2.4 Поновна употребљивост патерна

Када говоримо о патернима, у најопштијем смислу, можемо да кажемо да они представљају *решења неког проблема, у неком контексту, који се може поново искористити за решавање нових проблема.* То значи да три кључна елемента дефинишу патерн: **проблем**, **решење** и **контекст**. **Контекст** у суштини дефинише софтверски систем и његова ограничења која морају бити задовољена када се даје **решење** неког **проблема**.

Једно од основних својстава патерна јесте његова могућност да се може применити у решавању **скупа различитих проблема**. Како се долази до наведеног својства, или је можда још прецизније питање, *шта треба урадити како би пројектовани софтверски систем био примењив као решење скупа различитих проблема*?

Када правимо неки софтверски систем или у ужем смислу када правимо 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. Развој софтвера 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, већ и за решавање неких других проблема. Специфични

[3] Када кажемо подсистем мислимо на неку од основних функција које чине један софтверски систем.

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. Ако би се временом јављали нови кориснички захтеви, који су слични захтевима за која су већ пројектоване софтверске компоненте, јавила би се тенденција у пројектовању компоненти да се генерални и специфични програмски код раздвоји и постави 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

bkz-->∞

односно,

lim (gen + spec) = (gen) + (spec)

bkz-->∞

Уколико би се број корисничких захтева ***(bkz)*** непрекидно повећавао, пројектант програм би са сваким следећим захтевом имао све већу тенденцију да у потпуности раздвоји генералне од специфичних делова програма.

На основу наведеног можемо да закључимо да је крајњи циљ или сврха процеса развоја софтвера, (који ће моћи лако да се одржавају и надограђују) изградња софтверског система код кога ће бити одвојени генерални од специфичних делова програмског кода[4]. Уколико се генерални делови програмског кода налазе у једном модулу они се могу применити као решење скупа сличних проблема, односно класе проблема.

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.

Да још једном поновимо суштину пројектовања софтверског система, који се може применити као решење за класу проблема. На почетку имамо програм код кога су помешани генерални и специфични делови програмског кода, који се налазе у једном модулу. То значи да имамо програм код кога су помешани различити нивои апстракција[5]. Временом се појављују нови кориснички захтеви, који воде ка томе да се генерални и специфични делови програма раздвоје и сместе у различите модуле. Идеално би било да софтверски систем има само генералне делове програмског кода, док би специфични делови програмског кода били "измештени" из програма и смештени у датотеке или 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 софтверског система. Параметри софтверског система се иницијализује са вредностима којима се дефинише нови специфичан проблем[6]. *Оно што је генеричко, то је непроменљиво и на тај део програмског кода не утичу нови проблеми.*

Треба правити такве софтверске системе или софтверске производе, који ће се лако прилагодити сваком новом корисничком захтеву. Такви софтверски системи ће моћи да додају нове или мењају постојеће функционалност без велике промене њихове постојеће структуре и понашања.

Патерни омогућавају да генерални и специфични делови софтверског система буду раздвојени, и да пројектовано решење буде примењиво за неки скуп сличних проблема, односно класу проблема.

---

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

[5] Наведени став, по аналогији са држањем предавања, може да се представи на следећи начин. Ако непрекидно мењате нивое апстракције код предавања, ваше предавање 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.

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

2.5 Када треба користити софтверске патерне

Када се нађе решење неког проблема треба препознати шта је у решењу опште а шта специфично, односно шта је непромењиво а шта је промењиво. Наведена констатација се директно односи на писање програмског кода у току имплементације софтверског система. Уколико се препознају она места у софтверском систему, која се непрекидно мењају са појавом нових корисничких захтева, ту треба поставити патерне како би се зауставио потенцијални "хаос" која та промењива 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 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 потпуно да "обори" постојећи софтверски систем. Патерни се постављају у тим 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, што ће омогућити лакше коришћење и одржавање система. "Шпагети код", појам познат из софтверског инжењерства, који указује на сложени и компликовани алгоритам неке методе, који се непрекидно повећава са новим корисничким захтевима, представља потенцијални случај "бифуркационе тачке" код софтверских система. Такве методе су велике и тешко се могу одржавати.

Када приметите да се нешто овако дешава у вашем програму, ви у суштини препознајете потенцијалне тачке хаоса, које теже да "сруше" ваш програм. У неком тренутку, те тачке хаоса, са њиховом тенденцијом развоја сложености, ортогонално се 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.

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 неком делу програма не значи аутоматски да ту треба поставити патерн. Ако се број грана не мења у току развоја програма, тада не треба уводити патерне, јер они повећавају сложеност софтверског система. Сложеност се огледа у смањењу брзине извршења програма, јер се уводе нови слојеви[7] и нивои[8] у архитектури софтверског система[9]. Поред тога, увођење нивоа и слојева у архитектури отежава тестирање и контролу извршења 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.

2.6 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, тако да се деси **"апсолутни ред"**, јер би тада имали "тоталитарни"

[7] Слојеви су хијерархијски организовани, при чему се на врху хијерархије налази најапстрактнији слој, док се на дну хијерархије налази најконкретнији слој.

[8] Макро патерни као што је нпр. MVC или микро патерн facade уводе допунске нивое у архитектури софтверског система.

[9] *Butler Lampson* је рекао: "Сви проблеми у рачунарству могу бити решени другим нивоом индирекције. 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.)

систем који не прихвата различитости. Не треба заустављати почетак настанка неке различитости[10] јер се тиме спречава и успорава развој система. Хаос и ред се непрекидно смењују и то је нормалан процес у развоју било ког софтверског 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[11]:***

rh = Red * Haos

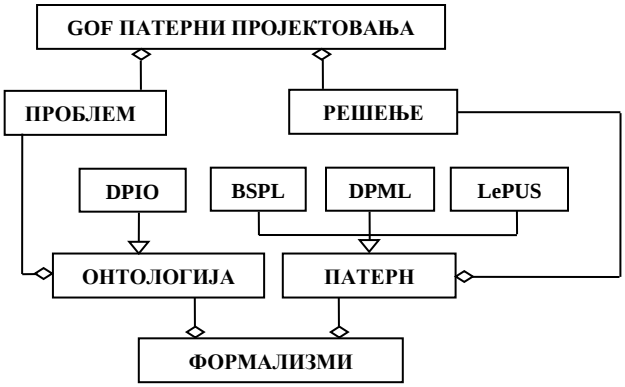
2.7 Примењивост патерна у живота
Ако посматрате живот, он се састоји од много процеса. Уколико не желимо да нам живот буде хаотичан, ми треба да управљамо са свим тим процесима. Патерни помажу да се лакше управља различитим животним процесима. Коришћењем патерна се лакше решавају, прате и управљају разни 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.
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 нека "већ виђена" (*déjà vu)* ситуација, неки "веома" познат проблем, који знате да сте некада решавали, али нећете моћи да се сетите решења. Тако ћете **опет** изгубити много сати, можда и дана у решавању нечега што сте некада 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 лакше и брже.

[10] 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.
[11] Наведена хипотеза неће у овој књизи бити предмет даљег разматрања.

3. Анализа патерна пројектовања

3.1 Формализација патерна

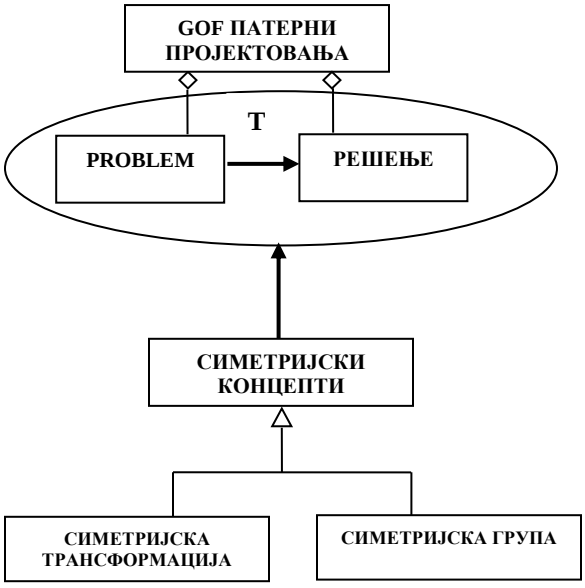
Развој савремених објектно-оријентисаних софтверских система подразумева коришћење патерна пројектовања. Патерни пројектовања су један од најважнијих механизама у развоју *одрживих* софтверских система, јер обезбеђују велику флексибилност током одржавања и надоградње софтверских система. Они се посматрају као генеричка решења, која се могу применити више пута, за различите али сличне проблеме. Наведене позитивне особине патерна (могућност поновног коришћења и лакоћа одржавања софтверских система) су водили ка формализацији патерна пројектовања *[DPFT],* што је довело до појаве бројних језика за опис патерна пројектовања *(Слика 1)*, као што су: *Balanced pattern specification language (**BPSL**), Design Pattern Modeling Language (**DPML**),…, **LePUS**: A Formal Language for Modeling Design Patterns.*



Слика 1: Формализми за опис патерна пројектовања

Наведени језици патерна су помогли да се *[TOUF]*: а) боље схвате патерни и њихова композиција б) прецизно опишу патерни, како би се избегло њихово дуплирање и поједноставио процес управљања складиштем патерна и ц) омогућио развој алата који подржавају патерне у процесу развоја софтверског система. ***Међутим, наведени језици нису дали прецизну формалну дефиницију патерна пројектовања у општем смислу, нити су објаснили када и како се патерни дешавају.*** Ако ми схватамо патерне као решење проблема у неком контексту *[FOW1],* тада ми можемо да кажемо су наведени језици патерна усмерени на прецизну дефиницију и спецификацију решења, док се мање пажње обраћа на проблеме који треба да буде решени. Формализација патерна,из перспективе проблема, је описана у раду *[KAM1],* у коме су патерни пројектовања формализовани и класификовани сходно проблему који решавају помоћу ***DPIO*** *(Design Patterns Intent Ontology).* Наведени формализми на концептуални начин објашњавају везу између проблема и решења код патерна, без улажења у дубину и разумевање релације између елемената проблема и решења. *У нашим истраживањима [DRSV, SV1-SV5] ми смо покушали да успоставимо везу између елемената проблема и решења код патера пројектовања и да **формално објаснимо** **(Слика 2)** **процес трансформације (T) проблема у решење помоћу симетријских концепата (симетријске трансформације и симетријске групе).***

Полазиште нашег истраживања били су радови Едена (*Amnon Eden*) *[Eden1-6],* творца *LePUS* језика патерна и радови Коплина *[Cop1-Cop5].* ***Комутативни аксиом*** Едена, који је он користио у формализацији патерна, представља основу од које смо кренули у развоју наше формалне теорије патерна пројектовања. Коплин је указао на значај Едена у прецизној формализације патерна, али је сматрао да је Еден направио крупну грешку, код формализације патерна, јер није узео у обзир **концепте симетрије** (симетријска група, симетријска трансформација и прекид симетрије), који су по његовом мишљењу кључни концепти код формализације патерна.



Слика 2: Процес трансформације проблема у решење описан помоћу симетријских концепата

Наведени став Коплина је усмерио наше истраживање на концепте симетрије и њихову везу са патернима. Књига *Symmetry rules – how science and nature are founded on symmetry [Ros1],* од Росена (*Joe Rosen*) нам је доста помогла да схватимо формализацију симетријских концепата. Сходно дефиницији симетрије коју је дао Росен: *"Симетрија је имунитет на могућу промену"* ми смо дефинисали један од основних циљева нашега истраживања, *израдњу формалне основе за прављење стабилних и одрживих софтверских система, који ће бити засновани на концептима симетрије*[12]*, који ће омогућити да се софтверски систем мења али и да буде имун на промену*[13]*.*

На основу Еденовог комутативног аксиома и концепата симетрије о којима је говорио Коплин, а које је одлично формализовао Росен ми смо направили нашу формалну теорију патерна*[14]*.

Током извођења наше теорије патерна ми смо извршили анализу GOF патерна пројектовања код којих смо уочили структуру решења која је заједничка за већину GOF патерна. То нас је навело на хипотезу да наведена структура представља кључни механизам код патерна и да она представља "језгро", "биће", односно "суштину" патерна. На основу наведене структуре решења дефинисали смо структуру проблема и објаснили како се врши трансформација структуре проблема у структуру решења. Наведеним приступом ми логички повезујемо (преко симетријских концепата) структуре проблема и решење код патерна на основу чега можемо, у општем смислу, објаснити ***шта су патерни, када настају и како настају патерни***. У том смислу дајемо једну од наших дефиниција патерна: ***Патерн је процес који трансформише несиметријску структуру (проблем) у симетријску структуру (решење).*** Наведена дефиниција говори, по нашем мишљењу, о главном својству патерна, да он у суштини описује процес симетризације, који преводи несиметријску у симетријску структуру. Зашто је то важно? Зато што можемо да докажемо да је симетријска структура, за разлику од несиметријске структуре, лакша за одржавање и надоградњу.

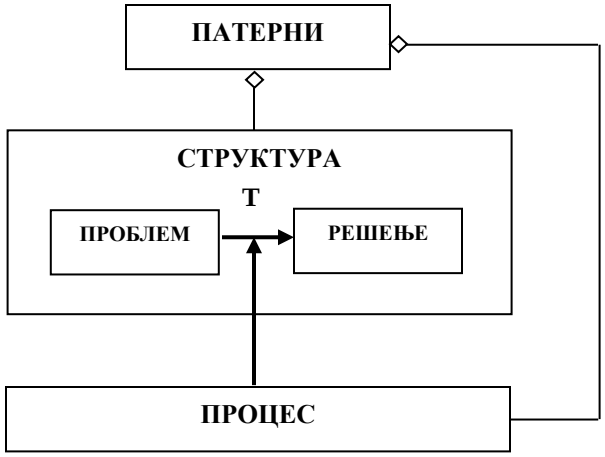
[12] Концепти симетрије се налазе у основи закона одржања у природи (закон одржања енергије, закон одржања импулса, закон одржања момента импулса).

[13] Наведени циљ је у складу са **Open-Closed** принципом, који је један од основних принципа објектно-оријентисаног пројектовања.

[14] Коплин (James Copliena) је 2004 године одржао курс на ФОН-у под називом: ***Pattern theory and practice – Toward a general design theory.*** Тада смо Коплина упознали детаљно са нашим истраживањем *[DRSV]* нашта је он дао један веома позитиван коментар о нашој патерн теорији у чланку **East of West and West of East: Engaging Other Academic Cultures** (http://www.artima.com/weblogs/viewpost.jsp?thread=72425) : " What had most attracted me to the university in the first place was Sinisa Vlajic's works. His Ph.D. thesis, completed in July 2003, **presents an interesting formalization of patterns based on symmetry and symmetry breaking** (*Formalizacija jedinstvenog procesa razvoja softvera pomocu uzora*, doctorska disertacija, Beograd, 2003). "

3.2 Општи облик GOF патерна пројектовања

Да би смо схватили општи облик патерна и његов кључни механизам потребно је да извршимо малу анализу постојећих дефиниција патерна. Александер *(Christopher Alexander)* је рекао [*AC2*]:"*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 *поновне употребљивости*, што значи да се решење неког проблема може поновити више пута код других, различитих проблема, који припадају некој класи проблема. Александар је такође рекао: "*Патерн је у исто време **ствар** која се дешава у стварности и **правило** које говори **када** и **како** се креира та ствар*". Jim Coplien *[Cop1]*, је такође објашњавајући патерне рекао: "*Патерн је правило за грађење ствари, али је оно истовремено и сама ствар*". На основу наведеног може се закључити да је патерн у исто време и **структура** (ствар) и **процес** *(Слика 3)*.



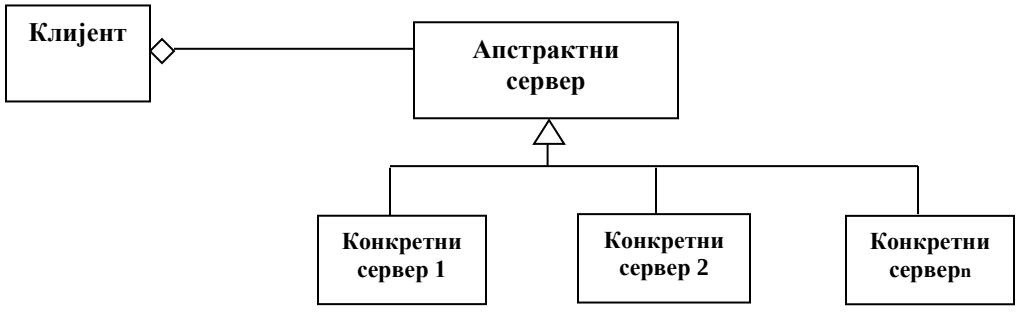
Слика 3: Патерн као структура и процес

Патерн садржи *структуру проблема и структуру решења*. Патерн је такође и *процес* који објашњава **када** и **како** се трансформацијом ***(Т)*** креира структура решења из структуре проблема:

Т

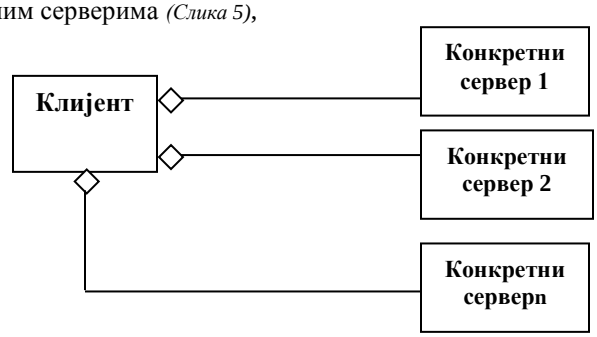
структура проблема --------> структура решења

У књизи Design Patterns постоје 23 GOF патерна пројектовања. Они су подељени у три основне групе: креациони патерни, патерни структуре и патерни понашања. Уочили смо да се код 20, од тих 23 патерна, може приметити једна структура која постоји код сваког од тих патерна. Та структура у потпуности описује патерн или неки његов део. Наведена структура је **кључни механизам или својство** GOF патерна пројектовања, која се јавља у **структури решења GOF патерна пројектовања**. Та структура изгледа овако *(Слика 4)*:



Слика 4: Структура решења GOF патерна пројектовања

Структура решења GOF патерна пројектовања[15] је уређена тројка ***(Клијент, Апстрактни сервер, Конкретни сервер),*** где може бити произвољан број (***n***) конкретних сервера[16]. Дајемо дефиниције елемената структуре решења патерна:
Клијент је елемент структуре патерна који *користи функционалности* апстрактног и конкретног сервера како би могао да обави сопствену функционалност.
Апстрактни сервер је елемент структуре патерна који клијенту *даје (обезбеђује) апстрактну функционалност*, која се може реализовати са више конкретних функционалности.
Конкретни сервер је елемент структуре патерна који клијенту *даје конкретну функционалност*, која представља реализацију апстрактне функционалности.
Између клијента и апстрактног сервера (уколико користимо UML нотацију) може бити веза **агрегације** (aggregation) или **зависности** (dependency). Између апстрактног сервера и конкретног сервера је веза **наслеђивања** (inheritance) или **реализације** (realization). Између клијента и конкретног сервера **не постоји** директна веза, већ је она **индиректна** преко апстрактног сервера. Када се код структуре решења патерна додаје нови конкретни сервер не мора да се мења клијент.
Уколико посматрамо структуру решења патерна из перспективе објекно-оријентисаног програма можемо рећи да се клијент у време компајлирања везује за апстрактни сервер, док се у време извршења програма везује за конкретни сервер. То значи да ће се тек у време извршења програма разрешити који конкретни сервер ће да реализује захтев клијента. Управо ово повезивање клијента са конкретним сервером у време извршења програма, даје самом програму флексибилност (код одржавања и надоградње програма), јер се један клијентски захтев може реализовати на више различитих начина, преко различитих конкретних сервера.
СРП се добија из структуре проблема GOF патерна пројектовања, код које је клијент директно повезан са конкретним серверима *(Слика 5)*,



Слика 5: Структура проблема GOF патерна пројектовања

Структура проблема GOF патерна пројектовања[17] је уређена двојка ***(Клијент, Конкретни сервер)*** где може бити произвољан број (***n***) конкретних сервера. Дајемо дефиниције елемената структуре проблема патерна:
Клијент је елемент структуре патерна који *користи функционалности* конкретног сервера како би могао да обави сопствену функционалност.
Конкретни сервер је елемент структуре патерна који клијенту *даје конкретну функционалност*.
Између клијента и конкретног сервера (уколико користимо UML нотацију) може бити веза **агрегације** (aggregation) или **зависности** (dependency).
Наведена структура је тешка за одржавање јер се при додавању новог конкретног сервера мења клијент. Из перспективе објектно-оријентисаног програма, клијент се у време компајлирања програма везује за један конкретни сервер, што онемогућава флексибилност програма у току његовог извршавања.

[15] У даљем тексту структуру решења код GOF патерна пројектовања, звaћемо структура решења патерна, са скраћеницом **СРП.**
[16] Из перспективе објектно-оријентисаног програмиња клијент може бити обична или апстрактна класа, апстрактни сервер може бити интерфејс, апстрактна класа или обична класа, конкретни сервер може бити обична класа.
[17] У даљем тексту структуру проблема код GOF патерна пројектовања, звaћемо структура проблема патерна, са скраћеницом **СОП.**

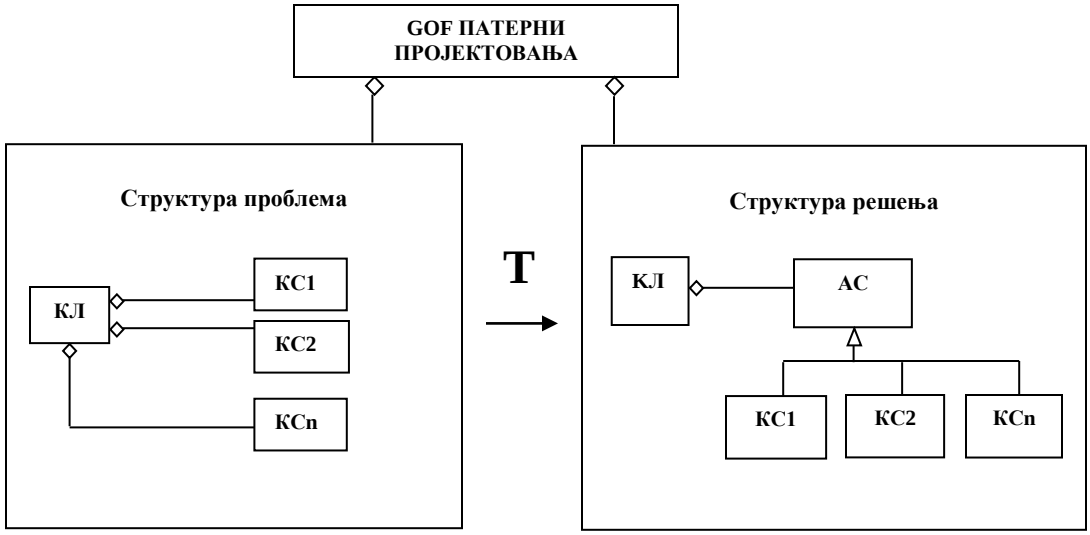
Постоји 5 категорија присуства СРП у GOF патернима пројектовања:
а) **Целином-једна СРП** - Једна СРП је у целини присутна у неком од GOF патерна (**6** GOF патерна)
б) **Целином-више СРП** - Више СРП су у целини присутну у неком од GOF патерна(**4** GOF патерна).
ц) **Делом-једна СРП** - Једна СРП је делом присутна у неком од GOF патерна (**7** GOF патерна).
д) **Делом-више СРП** - Више СРП је делом присутно у неком од GOF патерна (**3** GOF патерна).
е) **Не постоји СРП** - Не постоји СРП у GOF патерну (укупно **3** GOF патерна).

У табели 1 биће приказано присуство СРП у GOF патернима пројектовања. Код *FactoryMethod* и *TemplateMethod* патерна клијенти се не могу непосредно видети мада они практично постоје када се патерн користи. Још једном наглашавамо, СРП се јавља код ***20 GOF патерна***.

GOF патерни пројектовања	Клијент	Апстрактни сервер	Конкретни сервер	Категорија присуства СРП
1. Abstract Factory	Client	AbstractFactory	ConcreteFactory	Целином-више СРП
	Client	AbstractProductA	ProductA	
	Client	AbstractProductB	ProductB	
2. Builder	Director	Builder	ConcreteBuilder	Целином-једна СРП
3. Factory Method	Creator	Product	ConcreteProduct	Делом-једна СРП
4. Prototype	Client	Prototype	ConcretePrototype	Целином-једна СРП
5. Singleton	Singleton			Не постоји СРП
6. Adapter	Client	Target	Adapter	Делом-једна СРП
7. Bridge	Client	Abstraction	RefinedAbstarction	Целином-више СРП
	Abstraction	Implementor	ConcreteImplementor	
8. Composite	Client	Component	Leaf или Composite	Целином-више СРП
	Composite	Component	Leaf или Composite	
9. Decorator	Decorator	Component	ConcreteComponent или Decorator	Делом-једна СРП
	Decorator	Component	Decorator	
10. Façade	*main*	Façade		Не постоји СРП
11. Flyweight	FlyweightFactory	Flyweight	ConcreteFlyweighti или UnsharedConcreteFlyweight	Делом-једна СРП
12. Proxy	Client	Subject	Proxy или RealSubject	Целином -једна СРП
13. Chain of Responsibility	Client	Handler	ConcreteHandler	Целином-више СРП
	Handler	Handler	ConcreteHandler	
14. Command	Invoker	Command	ConcreteCommand	Делом-једна СРП
15. Interpreter	Client	AbstractExpression	TerminalExpression или NonterminalExpression	Делом-више СРП
	NonterminalExpression	AbstractExpression	TerminalExpression или NonterminalExpression	
16. Iterator	Client	Aggregate	ConcreteAggregate	Делом-више СРП
	Client	Iterator	ConcreteIterator	
17. Mediator	Colleague	Mediator	ConcreteMediator	Делом-једна СРП
18. Memento	CareTaker	Memento		Не постоји СРП
19.Observer	Subject	Observer	ConcreteObserver	Делом-једна СРП
20. State	Context	State	ConcreteState	Целином-једна СРП
21. Strategy	Context	Strategy	ConcreteStrategy	Целином-једна СРП
22. Template Method	Client	AbstractClass	ConcreteClass	Целином-једна СРП
23. Visitor	Client	Visitor	ConcreteVisitor	Делом-више СРП
	ObjectStructure	Element	ConcreteElementi	

Табела 1: Присуство СРП код GOF патерна пројектовања

Општи облик GOF патерна пројектовања би могао да се представи следећи начин *(Слика 6)*:



Слика 6: Општи облик GOF патерна пројектовања

Општи облик GOF патерна садржи:
а) структуру проблема,
б) структуру решења и
ц) трансформацију *(T)* структуре проблема у структуру решења, која се може представити као:

$$\textit{структура проблема} \xrightarrow{T} \textit{структура решења}$$

Трансформацијом ***T*** се структура проблема патерна, односно уређена двојка *(Клијент, Конкретни сервер)* трансформише у структуру решења патерна, односно уређену тројку *(Клијент, Апстрактни сервер, Конкретни сервер)*. *Трансформација T се дешава када* се структура проблема често мења (са појавом нових корисничких захтева), јер се тада повећава број конкретних сервера, који утичу на честу промену клијента. Програм који има структуру проблема, која се често мења, је тежак за одржавање и надоградњу. За такву структуру програма кажемо да је "неодржива" структура[18]. Са друге стране програм који има структуру решења, код кога број конкретних сервера не утиче на клијента, може лако да се одржава и надограђује. За такву структуру програма кажемо да је "одржива" структура[19]. То значи да патерни, трансформацијом *T* обезбеђује механизам, којим се неодрживе структуре трансформишу у одрживе структуре. Неодрживе структуре имају краћи век трајања од одрживих структура, јер оне програм могу релативно брзо да доведу у нестабилно и хаотично стање. У такве програме мора да се улаже велика људска енергија како би они могли да функционишу и да се развијају. На основу наведеног можемо да дамо нашу дефиницију патерна:
Дефиниција PT1: Патерн је процес који врши трансформацију структуре проблема у структуру решења.

[18] По дефиницији **одрживост** је *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.

[19] Код одрживих структура 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 таква структура непостојана и нестабилна. Међузависност елемената система је превише велика и такви системи су тешки за одржавање.

3.3 Патерни пројектовања и симетрија

3.3.1 Симетријска трансформација и симетријска група

У овом поглављу објаснићемо нашу формалну теорију о патернима пројектовања која је заснована на симетријским концептима. Да би се разумели симетријске концепте потребно је поћи од концепта система и стања система*[Ros1]*. Систем је било шта што има неко својство или особину[20]. Систем може бити у више *(n)* различитих стања. Након креирања, систем је у неком почетном стању. Током рада, систем трпи неке утицаје које врше трансформације његових стања. Трансформација (*T*) неког стања система *si* у стање *sj* се представља на следећи начин*(Слика 7)*:

$$si \xrightarrow{T} sj,\ si, sj \in State;\ i,j \in n$$ *, где је n број могућих стања система.*

После трансформације између стања *si* и *sj* настаје нека релација *R:*
si R sj

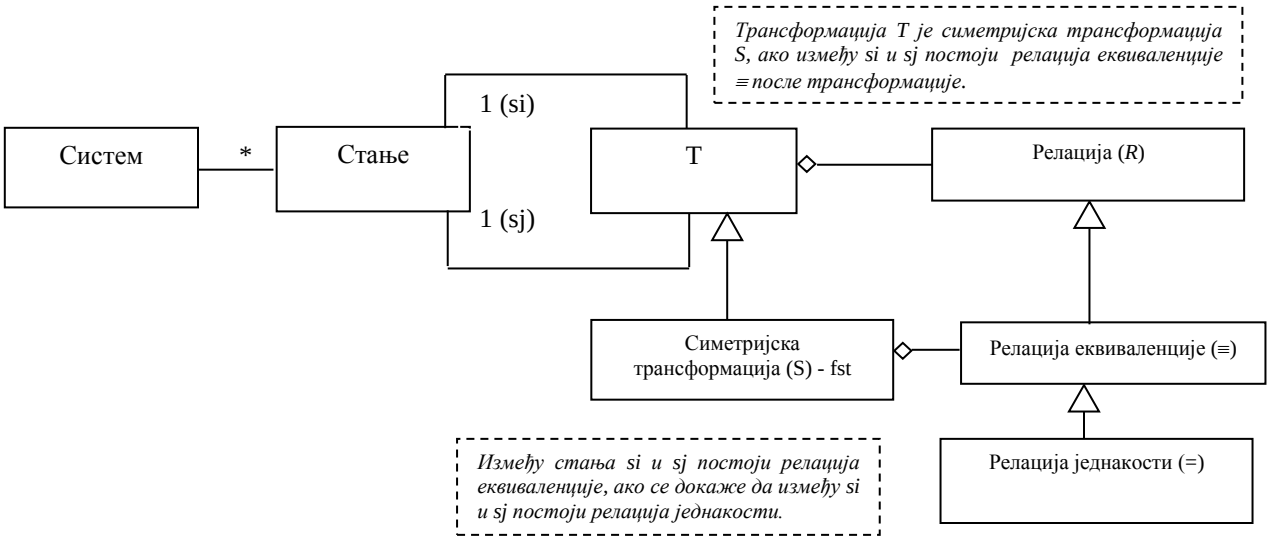
Дефиниција TR: Трансформација *(T) је симетријска трансформација S, ако између si и sj постоји релација еквиваленције после трансформације.*

$$si \xrightarrow{T} sj \equiv si$$

Релација еквиваленције између два стања настаје ако су задовољени услови: рефлексивности, симетричности и транзитивности.

1. *Рефклексивност*. $a \equiv a$ *за свако* a (сваки елемент скупа је еквивалентан њему самом).
2. *Симетричност*. $a \equiv b \Leftrightarrow b \equiv a$ *за свако* a, b.
3. *Транзитивност*. $a \equiv b, b \equiv c \Rightarrow a \equiv c$ за свако a, b, c.

Релација једнакости = између *si* и *sj* је једна од могућих реализација релације еквиваленције:
si = sj



Слика 7: Систем, стање и симетријска трансформација

Дефиниција SG: *Скуп свих инвертибилних симетријских трансформација простора стања система за релацију еквиваленције обликује групу, односно подгрупу трансформационе групе[21], која се назива* ***симетријска група*** *система за релацију еквиваленције.*

[20] Особина софтверских система је описана њеним атрибутима и системским операцијама. Атрибути описују структуру система, док системске операције описују понашање система. Атрибути представљају концепте реалног система који описују статичке карактеристике система (нпр.*Račun, 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.

[21] Трансформациона група обухвата све комбинације трансформација у простору стања система.

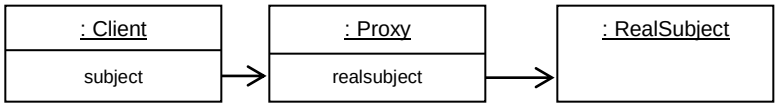
Дефиниција GR: *Група је непразан скуп G који задовољава 4 аксиома:*

1. ***Затвореност.*** *$a,b \in G$, скуп G је затворен за композицију $a*b$ тако да: $a*b, b*a \in G$*
2. ***Асоцијативност.*** *За свако $a,b,c \in G$, композиција је асоцијативна: $(a*b)*c = a*(b*c)$*
3. ***Постојање елемента идентитета.*** *За свако $a \in G$, постоји елемент $e \in G$ такав да $a*e = a = e*a$*
4. ***Постојање инверзије.*** *За свако $a \in G$, постоји a^{-1} тако да $a^{-1}*a = e = a*a^{-1}$*

Да би за неки скуп симетријских трансформација рекли да образује симетријску групе, морају да буду задовољене дефиниције SG и GR.

3.3.2 Трансформације код дијаграма класа и објектних дијаграма

Уколико концепт система и стања система применимо код UML-ових дијаграма класа и објектних дијаграма, добићемо да су **дијаграми класа** и **објектни дијаграми** репрезенти структуре *система,* док су **класе** и њихови **објекти** репрезенти *стања система*.
Када се објектно-оријентисани програм извршава он користи објекте различитих класа како би постигао жељену функционалност. То значи да програм током његовог извршавања, у неком унапред дефинисаном редоследу (сходно постојећим алгоритмима програма), користи објекте и њихове методе. Тренутно стање програма је одређено објектом програма који тренутно извршава неку од својих метода. На пример уколико имамо објектни дијаграм Proxy патерна *(Слика 8)*, програм може бити у три могућа стања, у зависности од тога да ли се извршава *Client, Proxy* или *RealSubject* објекат. Сваки од наведених објеката представља једно од стања програма. Програм је у почетку у стању "Client", након тога прелази у стање "Proxy", да би на крају прешао у стање "RealSubject". Прелазак програма са једног у друго стање представља **трансформацију** стања програма.



Слика 8: Објектни дијаграм Proxy патерна

Трансформацију *(T_1,T_2)* наведених објеката можемо да представимо на следећи начин:

$\overset{T_1}{\longrightarrow}$
:Client ---> :Proxy

$\overset{T_2}{\longrightarrow}$
:Proxy ---> :RealSubject

Наведени објектни дијаграм је заснован на дијаграму класа Proxy патерна *(Слика 9),* који дефинише могуће трансформације између објеката на основу веза које постоје између класа[22]:

1. $\overset{T_1}{\longrightarrow}$ *Client ---> Subject*

2. $\overset{T_2}{\longrightarrow}$ *Proxy ---> Subject*

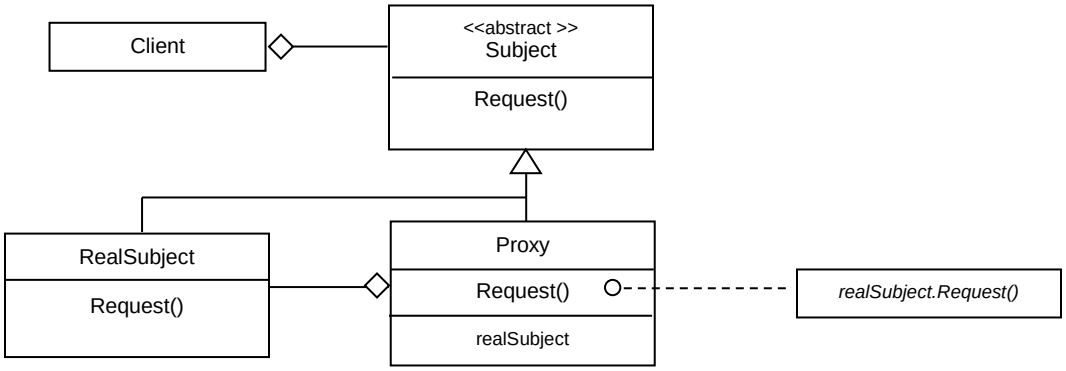
3. $\overset{T_3}{\longrightarrow}$ *RealSubject ---> Subject*

4. $\overset{T_1}{\longrightarrow}$ *Client ---> Subject,* $\overset{T_2}{\longrightarrow}$ *Proxy ---> Subject => Client* $\overset{T_4}{\longrightarrow}$ *Proxy*

5. $\overset{T_1}{\longrightarrow}$ *Client ---> Subject,* $\overset{T_3}{\longrightarrow}$ *RealSubject ---> Subject => Client* $\overset{T_5}{\longrightarrow}$ *RealSubject*

6. $\overset{T_6}{\longrightarrow}$ *Proxy ---> RealSubject*

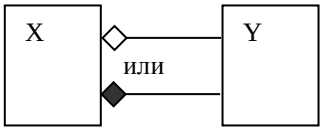
[22] Ако класе у току извршења програма, не мењају динамички своју структуру и понашање (не додају се нови атрибути и методе до класе) анализу релација између класа је могуће урадити пре извршења програма.



Слика 9: Дијаграм класа Proxy патерна

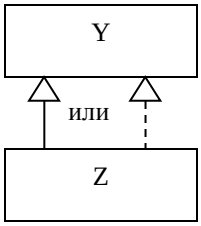
Трансформације између класа[23], у општем смислу, можемо да представимо на следећи начин:
а) Уколико између X и Y постоји веза *агрегације* или *композиције* тада се X може трансформисати у Y[24].

T1
X ---> Y ***(a)***



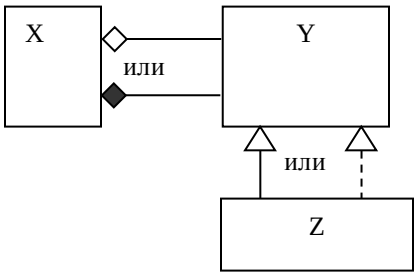
b) Уколико између Y (надкласа) и Z (подкласа) постоји веза *наслеђивања* или *реализације* тада се Z може трансформисати у Y[25].

T2
Z ---> Y ***(b)***



c) Уколико између X и Y постоји веза *агрегације* или *композиције*, а између Y (надкласа) и Z (подкласа) веза *наслеђивања* или *реализације* тада се X може трансформисати у Z.

T1 ***T2*** ***T3***
X ---> Y , Z ---> Y => X ---> Z ***(c)***



[23] Када кажемо класа, онда на њу мислимо као апстракцију концепата класе, апстрактне класе и интерфејса из перспективе објектно-оријентисаног програмирања.

[24] Из перспективе објектно-оријентисаног програмирања X може бити класа и апстрактна класа. Y може бити класа, апстрактна класа или интерфејс.

[25] Из перспективе објектно-оријентисаног програмирања Y може бити класа, апстрактна класа или интерфејс. Z такође може бити класа, апстрактна класа или интерфејс. Класа може да наследи класу или апстрактну класу. Класа може да реализује интерфејс. Апстрактна класа може да наследи класу или апстрактну класу. Апстрактна класа не може да реализује интерфејс.Интерфејс може да наследи интерфејс. Интерфејс не може да наследи класу или апстрактну класу. Интерфејс не може да реализује интерфејс.

3.3.3 Симетријска трансформације код дијаграма класа и објектних дијаграма

Уколико би желели да покажемо су трансформације T1,T2 између наведених класа (X,Y,Z) у случајевима *(a,b)* симетријске трансформације потребно је да одредимо да ли између класа X,Y,Z постоји релација еквиваленције, односно релација једнакости. Наводимо ригорозну дефиницију једнакости класа.

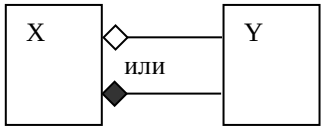
Дефиниција ЈК: Две класе су једнаке уколико имају исте чланице (атрибуте и/или методе).

За класе X и Y из (a) се може рећи да су једнаке (X=Y) јер класа X садржавајући класу Y садржи све чланице класе Y. За класе Yи Z из (b) се може рећи да су једнаке (Z=Y) јер класа Z наслеђујући класу Y наслеђује све чланице класе Y.

На основу наведеног можемо закључити да је трансформација T1 између X и Y симетријске трансформације ST1.

T_1

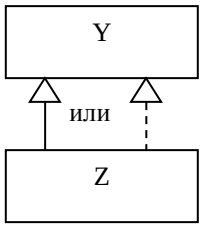
$X \dashrightarrow Y \equiv X$, јер је $Y = X$



Такође можемо закључити да је трансформација T2 између Z и Y симетријске трансформације ST2.

T_2

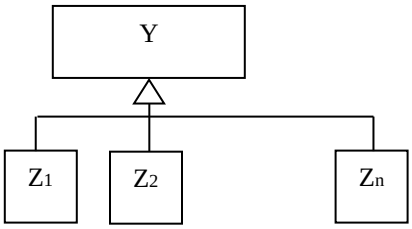
$Z \dashrightarrow Y \equiv Z$, јер је $Y = Z$



3.3.4 Симетријска група код дијаграма класа

Разматраћемо 5 случаја дијаграма класа како би утврдили да ли они образују или не симетријску групу.

SG1) Када класе $Z_1, Z_2, ..., Z_n$ наслеђују или реализују класу Y,



постоје следеће симетријске трансформације:

ST_1

$Z_1 \dashrightarrow Y \equiv Z_1$, јер је $Y = Z_1$

ST_2

$Z_2 \dashrightarrow Y \equiv Z_2$, јер је $Y = Z_2$

...

ST_n

$Z_n \dashrightarrow Y \equiv Z_n$, јер је $Y = Z_n$

За дати скуп симетријских трансформација могуће је дати општи облик симетријске трансформације, који образује класу симетријских трансформација:

$$Zi \xrightarrow{ST_i} Y \equiv Zi, \text{ јер је } Y = Zi, \text{ где је } i \in (1..n),\ n - \text{број симетријских трансформација} \quad (d)$$

Класа симетријских трансформација састоји се од скупа симетријских трансформација између којих постоји релација еквиваленције, јер су класе које учествују у трансформацијама једнаке између себе $Y = Z_1 = Z_2 = ... = Z_n$. То значи да можемо дефинисати још општију класу симетријских трансформација:

$$Zi \xrightarrow{S\,Tj} Y \equiv Zi, \text{ јер је } Y = Zi, \text{ где је } i,j \in (1..n),\ n - \text{број симетријских трансформација} \quad (e)$$

За општи облик симетријске трансформације (d) постоји њен инверзни облик који важи за сваку од симетријских трансформација:

$$Y \xrightarrow{S\,Ti} Zi \equiv Y \text{ јер је } Zi = Y$$

На основу наведеног можемо закључити да имамо *скуп инвертибилних симетријских трансформацију за релацију еквиваленције*. чиме је задовољен **први услов** да скуп симетријских трансформација образује симетријску групу.
Сада ћемо показати да ли наведени скуп симетријских трансформација образује групу, тако што ћемо проверити 4 услова која морају да буду задовољена:

*1.**Затвореност.** За свако $ST_1, ST_2 \in G$, скуп G је затворен за композицију $ST_1 * ST_2$ тако да: $ST_1 * ST_2, ST_2 * ST_1 \in G$,*
$G \in (ST_1, ST_2, ..., ST_n)$.

Симетријске трансформације ST_1, ST_2:

$$Z_1 \xrightarrow{ST_1} Y \equiv Z_1$$

$$Z_2 \xrightarrow{ST_2} Y \equiv Z_2$$

где ST_1, ST_2 могу бити било које симетријске трансформација из скупа G, док Z_1 и Z_2 могу бити било које класе из скупа $(Z_1, Z_2, .., Z_n)$, можемо представити као:

$$Y \xrightarrow{ST_1} Y, \text{ јер је } Y = Z_1$$

$$Y \xrightarrow{ST_2} Y, \text{ јер је } Y = Z_2$$

на тај начин резултат композиције $ST_1 * ST_2$:

$$ST_1 * ST_2 = Y \xrightarrow{ST_1} Y \xrightarrow{ST_2} Y$$

једнак је ST, где је ST једнако са било којом симетријском трансформацијом $ST_1, ST_2, ..., ST_n$. Исти резултат се добија када се направи композиција *$ST_2 * ST_1$:*

$$ST_2 * ST_1 = Y \xrightarrow{ST_2} Y \xrightarrow{ST_1} Y$$

За наведени скуп симетријских трансформација *задовољен је услов затворености*.

*2.**Асоцијативност.** За свако $ST_1, ST_2, ST_3 \in G$, композиција је асоцијативна: $(ST_1 * ST_2) * ST_3 = ST_1 * (ST_2 * ST_3)$, $G \in (ST_1, ST_2, ..., ST_n)$.*

Симетријске трансформације ST_1, ST_2, ST_3:

$$Z_1 \xrightarrow{ST_1} Y \equiv Z_1$$

$$Z_2 \xrightarrow{ST_2} Y \equiv Z_2$$

$$Z_3 \xrightarrow{ST_3} Y \equiv Z_3$$

где ST_1, ST_2, ST_3 могу бити било које симетријске трансформација из скупа G, док Z_1, Z_2 и Z_3 могу бити било које класе из скупа $(Z_1, Z_2, .., Z_n)$, можемо представити као:

$$Y \xrightarrow{ST_1} Y, \text{ јер је } Y = Z_1$$
$$Y \xrightarrow{ST_2} Y, \text{ јер је } Y = Z_2$$
$$Y \xrightarrow{ST_3} Y, \text{ јер је } Y = Z_3$$

на тај начин:

$$(ST_1 * ST_2) * ST_3 = (Y \xrightarrow{ST_1} Y \xrightarrow{ST_2} Y) \xrightarrow{ST_3} Y$$

$$ST_1 * (ST_2 * ST_3) = Y \xrightarrow{ST_1} (Y \xrightarrow{ST_2} Y \xrightarrow{ST_3} Y)$$

резултат композиције $(ST_1 * ST_2) * ST_3$ и $ST_1 * (ST_2 * ST_3)$ једнак је ST, где је ST једнако са било којом симетријском трансформацијом ST1,ST2,...,STn. За наведени скуп симетријских трансформација *задовољен је услов асоцијативности*.

3. Постојање елемента идентитета. *За свако* $ST \in G$, *постоји елемент* $e \in G$ *такав да* $ST*e = ST = e*ST$, $G \in (ST_1, ST_2, ..., ST_n)$.

Симетријске трансформације *ST и e:*

$$Z_1 \xrightarrow{ST} Y \equiv Z_1$$
$$Z_2 \xrightarrow{e} Y \equiv Z_2$$

где ST и e могу бити било која симетријска трансформација из скупа G, док Z_1 и Z_2 могу бити било које класе из скупа $(Z_1, Z_2, .., Z_n)$, можемо представити као:

$$Y \xrightarrow{ST} Y, \text{ јер је } Y = Z_1$$
$$Y \xrightarrow{e} Y, \text{ јер је } Y = Z_2$$

на тај начин добијамо:

$$ST*e = Y \xrightarrow{ST} Y \xrightarrow{e} Y$$

$$ST = Y \xrightarrow{ST} Y$$

$$e*ST = Y \xrightarrow{e} Y \xrightarrow{ST} Y$$

Доказали смо да је задовољен услов: $ST*e = ST = e*ST$. То значи да је за наведени скуп симетријских трансформација *задовољен услов постојања елемента идентитета*.

4. ***Постојање инверзије.*** *За свако* $ST \in G$, *постоји* ST^{-1} *тако да* $ST^{-1} * ST = e = ST * ST^{-1}$, $G \in (ST_1, ST_2, ..., ST_n)$.

Симетријске трансформације ST, ST^{-1} и *e:*

$$Z \xrightarrow{ST} Y \equiv Z$$
$$Y \xrightarrow{ST^{-1}} Z \equiv Y$$
$$Z \xrightarrow{e} Y \equiv Z$$

где ST и e могу бити било која симетријска трансформација из скупа G, док *Z* може бити било која класа из скупа $(Z_1, Z_2, .., Z_n)$, можемо представити као:

$\overset{ST}{Y \dashrightarrow Y}$, јер је $Y = Z$

$\overset{ST^{-1}}{Y \dashrightarrow Y}$, јер је $Y = Z$

$\overset{e}{Y \dashrightarrow Y}$, јер је $Y = Z$

На тај начин добијамо:

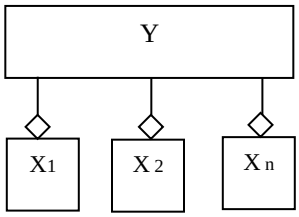
$ST^{-1} * ST = Y \overset{ST^{-1}}{\dashrightarrow} Y \overset{ST}{\dashrightarrow} Y$

$e = Y \overset{e}{\dashrightarrow} Y$

$ST * ST^{-1} = Y \overset{ST}{\dashrightarrow} Y \overset{ST^{-1}}{\dashrightarrow} Y$

Доказали смо да је задовољен услов: $ST^{-1}*ST = e = ST*ST^{-1}$. То значи да је за наведени скуп симетријских трансформација *задовољен услов постојања инверзије*.
На основу наведеног можемо закључити да *скуп инвертибилних симетријских трансформацију за релацију еквиваленције* задвољава и **други услов** да је група. ***Можемо да закључимо да класе $Z_1, Z_2, ..., Z_n$ које наслеђују или реализују класу Y, образују симетријску групу.***

SG2) Када класе $X_1, X_2, ..., X_n$, композицијом или агрегацијом, садрже класу Y,



постоје следеће симетријске трансформације:

$\overset{ST_1}{X_1 \dashrightarrow Y \equiv X_1}$, јер је $Y = X_1$

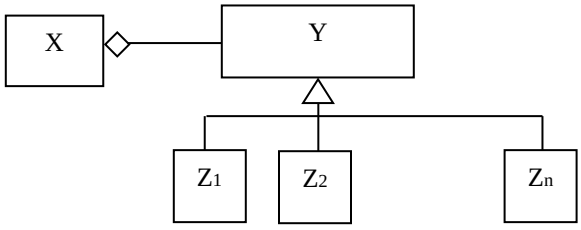
$\overset{ST_2}{X_2 \dashrightarrow Y \equiv X_2}$, јер је $Y = X_2$

...

$\overset{ST_n}{X_n \dashrightarrow Y \equiv X_n}$, јер је $Y = X_n$

На сличан начин као и у SG1 може се доказати да наведени скуп симетријских трансформација ***образује симетријску групу.***

SG3) Када класа X композицијом или агрегацијом, садржи класу Y и када класе $Z_1, Z_2, ..., Z_n$ наслеђују или реализују класу Y,

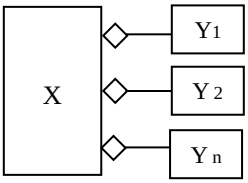


постоје следеће симетријске трансформације:

$$X \xrightarrow{ST_1} Y \equiv Z_1, \text{ jep je } Y = X$$
$$Z_1 \xrightarrow{ST_2} Y \equiv Z_1, \text{ jep je } Y = Z_1$$
$$Z_2 \xrightarrow{ST_3} Y \equiv Z_2, \text{ jep je } Y = Z_2$$
$$\dots$$
$$Z_n \xrightarrow{ST_n} Y \equiv Z_n, \text{ jep je } Y = Z_n$$

На сличан начин као и у SG1 може се доказати да наведени скуп симетријских трансформација ***образује симетријску групу.***

SG4) Када класа X композицијом или агрегацијом, садржи класе $Y_1, Y_2, \dots, Y_n$.

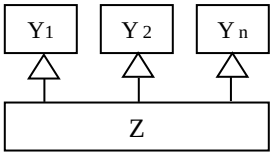


постоје следеће трансформације:

$$X \xrightarrow{T_1} Y_1, \; Y_1 \neq X, \text{ jer je } Y_1 \neq Y_1 + Y_2 + \dots + Y_n$$
$$X \xrightarrow{T_2} Y_2, \; Y_2 \neq X, \text{ jer je } Y_2 \neq Y_1 + Y_2 + \dots + Y_n$$
$$\dots$$
$$X \xrightarrow{T_n} Y_n, \; Y_n \neq X, \text{ jer je } Y_n \neq Y_1 + Y_2 + \dots + Y_n$$

Наведене трансформације $T_1, T_2, \dots, T_n$ нису симетријске трансформације и самим тим не могу образовати симетријску групу. За дати скуп трансформација $T_1, T_2, \dots, T_n$ кажемо да образује **несиметријску групу**.

SG5) Када класа Z наслеђује или реализује класе $Y_1, Y_2, \dots, Y_n$,



постоје следеће трансформације:

$$Z \xrightarrow{T_1} Y_1, \; Y_1 \neq Z, \text{ jer je } Y_1 \neq Y_1 + Y_2 + \dots + Y_n$$
$$Z \xrightarrow{T_2} Y_2, \; Y_2 \neq Z, \text{ jer je } Y_2 \neq Y_1 + Y_2 + \dots + Y_n$$
$$\dots$$
$$Z \xrightarrow{T_n} Y_n, \; Y_n \neq Z, \text{ jer je } Y_n \neq Y_1 + Y_2 + \dots + Y_n$$

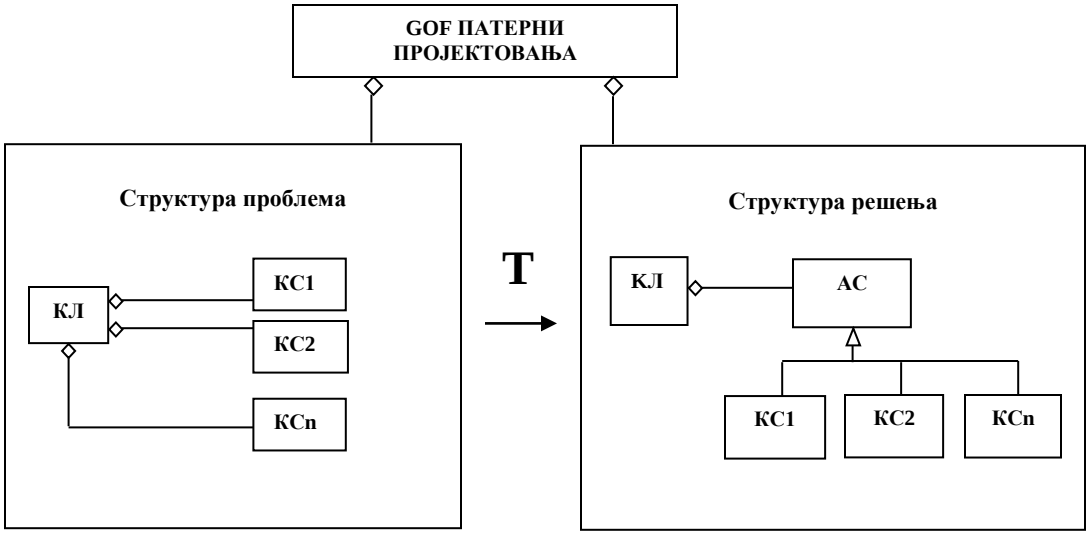
Наведене трансформације $T_1, T_2, \dots, T_n$ нису симетријске трансформације и самим тим не могу образовати симетријску групу. За дати скуп трансформација $T_1, T_2, \dots, T_n$ кажемо да образује **несиметријску групу**.

3.3.5 Дефиниција GOF патерна помоћу симетрије

Када смо говорили о општем облику GOF патерна пројектовања рекли само да се он састоји од:
а) структуре проблема,
б) структуре решења и
ц) трансформације *(T)* структуре проблема у структуру решења, која се може представити као:

T
структура проблема --------> структура решења

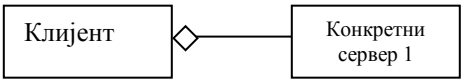
Трансформацијом ***T*** се структура проблема патерна, односно уређена двојка *(Клијент, Конкретни сервер)* трансформише у структуру решења патерна, односно уређену тројку *(Клијент, Апстрактни сервер, Конкретни сервер)*.



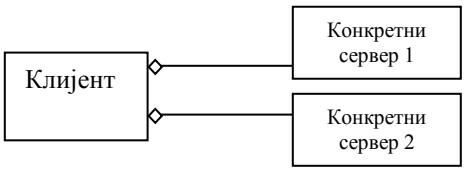
*Сходно дијаграмима класа **SG3** и **SG4,** можемо рећи да **структура проблема** код патерна образује **несиместрујску групу**, док **структура решења** код патерна образује **симетријску групу.*** На основу наведеног можемо да дамо дефиницију GOF патерна помоћу симетријских концепата:

Дефиниција PT2: Патерн је процес који врши трансформацију несиметријске групе (структуре проблема) у симетријску групу (структуру решења).

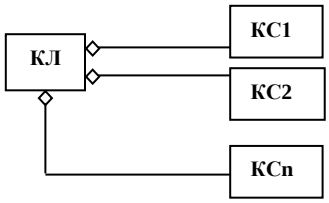
Мало ћемо образложити **када** се дешава процес трансформације структуре проблема у структуру решења код патерна из контекста симетријских концепата.
Предпоставимо да у почетку имамо клијента и један конкретни сервер између којих постоји веза агрегације или композиције. Између клијента и конркретног сервера постоји симетријска трансформација сходно дефиницији **JK**.



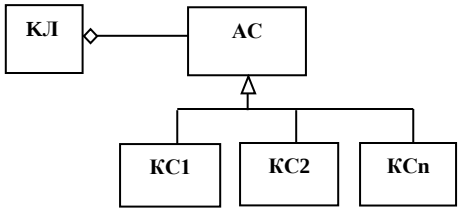
Докле год између клијента и конкретног сервера постоји симетријска трансформација нема потребе да се врши трансформација структуре проблема у структуру решења код патерна. Уколико се појави нови конкретни сервер, (КС2) који је везан за клијента тада настаје несиметријска група (структура проблема):



Трансформација Т се дешава ***када*** се структура проблема, односно број конкретних сервера непрекидно повећава (са појавом нових корисничких захтева), што утиче на честу промену клијента[26]. Тако долазимо до структуре (дијаграм класа **SG3)** која је и тешка за одржавање**,** код које је клијент везан за **n** конкретних сервера.



Након трансформације Т добија се структура решења (дијаграм класа **SG4**) која представља симетријску групу, код које је клијент агрегацијом или композицијом везан за апстрактни сервер, док конкретни сервери наслеђују или реализују апстрактни сервер.



На крају овог поглавља даћемо још једну дефиницију патерна:
Дефиниција РТ3: Патерн је процес који несиметријску групу, коју образује клијент и конкретни сервери (између којих је веза агрегације или композиције), трансформише у симетријску групу. Симетријску групу образују клијент, апстрактни сервер и конкретни сервери. Између клијента и апстрактног сервера је веза агрегације или композиције, док је између асптрактног сервера и конкретних сервера веза наслеђивања или реализације.

Сходно дефиницији симетрије коју је дао Росен *[Ros1]*: *"Симетрија је имунитет на могућу промену",* основни циљ овог истраживања се односи на дефинисање формалне основе за прављење стабилних и одрживих софтверских система, који су засновани на концептима симетрије, који имају особину да могу у исто време и да се мењају и да буду имуни на промене. Сматрамо да даљи развој софтверских система треба озбиљно да размотри концепте симетрије зато што они постоје у основи свих закона одржања у природи (одржање енергије, одржање импулса,...,итд.). У том смислу Нотерова теорема *(Noether's Theorem)* наглашава да постоји један-један пресликавање између закона одржања и симетрије. Добитник Нобелове награде Андерсон (*Philip Warren Anderson*), амерички физичар, је рекао (парафразирам), да је можда претерано рећи о физици да се она своди на студирање симетрије, али се у суштини симетрија значајно користи у објашњавању различитих концепата из физике. Верујемо да ће у времену које долази "**правила одржања софтверских система**" и истраживање симетрије у том контексту, повезати софтверске и физичке системе и омогућити коришћење бројних знања из физике у софтверском инжењерству.

[26] Уколико је број конкретних сервера који су везани за клијента коначан и не мења се са појавом нових корисничких захтева тада треба размислити да ли треба да се врши трансформација Т. Треба имати на уму да трансформација Т повећава сложеност софтверског система.

4. Програмски концепти патерна пројектовања

4.1 Објектно-оријентисани концепти потребни за схватање патерна пројектовања

Да би патерни пројектовања могли да се схвате потребно је разумети неке од основних концепата објектно-оријентисаног програмирања[27]:
а) Класе и објекте
б) Наслеђивање
ц) Динамички полиморфизам
ц) Апстрактне класе
д) Интерфејси

4.1.1 Класа и објекти

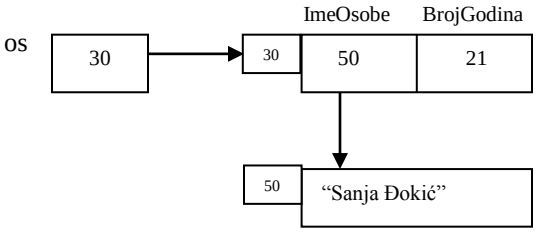
Као што је познато **класа** се састоји од скупа атрибута и метода. Атрибути описују **структуру** класе док методе описују њено **понашање**. На пример класа *Osoba*:

```
class Osoba
  { String ImeOsobe; int BrojGodina;
    void Postavi(String ImeOsobe1, int BrojGodina1) {ImeOsobe = ImeOsobe1; BrojGodina = BrojGodina1;}
    String Prikazi() { System.out.println("Ime osobe: " + ImeOsobe + " Broj godina:" + BrojGodina);}
    public static void main(String arg[])
          { Osoba os;
            os = new Osoba();
            os.Postavi("Sanja Đokić",21);

          }
  }
```

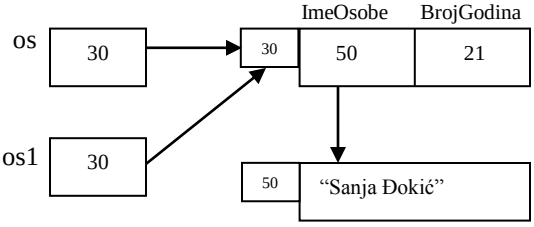
има атрибуте *ImeOsobe* и *BrojGodina* и методе *Postavi()* и *Prikazi()*. По дефиницији, ***класа је општи представник неког скупа објеката који имају исте особине (атрибуте и методе).*** Из тога следи да објекат представља једно појављивање (примерак, инстанцу) класе. У наведеном примеру објекат се креира са наредбом:

```
os = new Osoba();
```

Тада објектна променљива *os* добија адресу (нпр. 30) објекта који је креиран.



Наглашавам да се у слободном жаргону обично за *os* каже да је објекат. То није тачно јер је *os* **показивач на објекат**, односно **објектна променљива** чији је садржај адреса објекта на који она показује. Ово треба узети озбиљно у обзир јер ћемо у случају следеће наредбе,
Osoba os1 = os;
имати резултат:



Обе објектне променљиве *os* и *os1* показиваће на исти објекат на адреси 30.

[27] Наведене концепте ћемо објаснити у програмском језику **Јава**. Препоручујем да погледате скрипту ***Основни концепти Јаве*** у којој су детаљно објашњени концепти: *класе, објекти, методе, наслеђивање, апстрактне класе, интерфејси, рад са стринговима, пакети и изузеци*. Адреса где се налази скрипта:
http://silab.fon.rs/index.php?option=com_docman&task=doc_download&gid=706&&Itemid=56

4.1.2 Наслеђивање и динамички полиморфизам

Један од најважнијих концепата објектно-оријентисаног програмирања је концепт **наслеђивања.** Уколико имамо основну класу *Osoba,*

```
class Osoba
  { String ImeOsobe;
    int BrojGodina;
    Osoba(String ImeOsobe1, int BrojGodina1)  {ImeOsobe = ImeOsobe1; BrojGodina =  BrojGodina1;}
    void Prikazi() { System.out.println("Ime osobe: " + ImeOsobe + "Broj godina:" + BrojGodina);}
    void PrikaziOsobu() { System.out.println("Ime osobe: " + ImeOsobe);}
  }
```

коју наслеђује класа *Student,*

```
class Student extends Osoba
  { String BrojIndeksa;
    Student(String ImeOsobe1, int BrojGodina1,String BrojIn1){super(ImeOsobe1,BrojGodina1); BrojIndeksa = BrojIn1;}
    void Prikazi() { super.Prikazi(); System.out.println("Broj indeksa:" + BrojIndeksa);}
    void PrikaziStudenta() { super.PrikaziOsobu(); System.out.println("Broj indeksa:" + BrojIndeksa);}
  }
```

тада можемо да кажемо да класа *Student* наслеђује све атрибуте и јавне методе класе *Osoba*. Из концепта наслеђивања директно произилази концепт **компатибилности објектних типова** код кога *објектна променљива надкласе може да добије референцу на било који објекат који припада класи која наслеђује надкласу*. У нашем примеру објектна променљива класе *Osoba* може да добије референцу на објекат класе *Student* што можемо представити следећим примером:

```
Osoba os;
Student st = new Student();
os = st;
```

Компатибилност објектних типова је посебно важна код прављења **генеричких програма** када треба да се обезбеди могућност да *једна наредба програма има различито понашање у зависности од логике програма*. На пример, уколико дефинишемо главни програм који користи наведене класе *Osoba* и *Student*:

```
public static void main(String arg[])
  {   Osoba os;
      Osoba os1 = new Osoba("Sanja Djokic",21);
      Student st = new Student("Maja Stanilovic", 22,"12/09");
      if (arg[0].equals("1"))
         os = os1;
      else
         os = st;
      os.Prikazi();
          }
```

за наредбу

```
os.Prikazi();
```

можемо да кажемо да се различито понаша у зависности од логике извршења програма. Уколико је задовољен услов да је улазни аргумент arg[0] једнак 1 тада ће објектна променљива *os* да добије референцу на објекат класе *Osoba.* Код наредбе *os.Prikazi()* биће позвана метода *Prikazi()* класе *Osoba.* Уколико није задовољен наведени услов објектна променљива *os* ће добити референцу на објекат класе *Student.* У том случају наредба *os.Prikazi()* ће позвати методу *Prikazi()* класе *Student.*
Уколико једна наредба може да има различито понашање за њу се може рећу да је **полиморфна**[28]**.** На овај начин смо дошли до ткз. **динамичког**[29] **полиморфизма,** код кога објектна променљива у време извршења програма (run-time) добија адресу објекта који ће да изврши дефинисану методу. То значи да се тек у време извршења програма одређује (разрешава) понашање неке полиморфне наредбе.

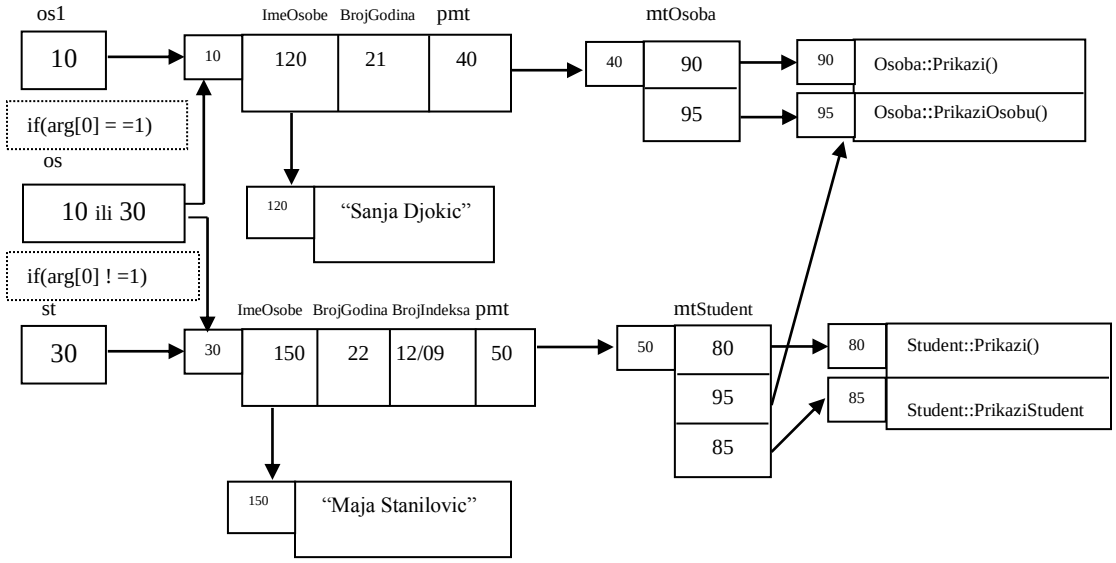
---

[28] Појам **полиморфизам** су први користили стари Грци када су желели да објасне да *нешто може да има више облика*.

[29] Појам "**динамички**" се односи на време извршења програма. Насупрот томе појам "**статички**" се односи на време компајлирања програма. Каже се да се објекат **динамички** повезује са методом у време извршења програма, док се објекат повезује **статички** са методом у време компајлирања програма.

4.1.3 Касно повезивање објекта са методом

Методе у Јава програмском језику се повезују са објектима у време извршења програма. Овакво повезивање се зове **касно повезивање** (late binding) или **динамичко повезивање** објекта са методом. За наведени пример дајем изглед оперативне меморије[30], након извршења главног програма, како би објаснили како се остварује касно повезивање објекта са методом:



Сваки објекат (на адреси 10 и 30) има невидљиви атрибут *pmt* који представља показивач на табелу метода класе (*mtOsoba*) тог објекта. У наведеном примеру објекат класе *Osoba* има атрибут *pmt* чији је садржај (40), што је адреса табеле метода класе *Osoba* (*mtOsoba*). Табела метода класе *Osoba* (*mtOsoba)* садрже низ показивача на адресе (90,95) где се налазе методе класе *Osoba*.
Објекат класе *Student* има атрибут *pmt* чији је садржај (50), што је адреса табеле метода класе *Student* (*mtStudent*). Табела метода класе *Student* (*mtStudent)* садрже низ показивача на адресе (80,95,85) где се налазе методе класе *Student*. Будући да класа *Student* наслеђује класу *Osoba* самим тим наслеђује и њене методе. Пошто је класа *Student* прекрила методу *Prikazi()* класе Osoba, метода *Prikazi()* класе *Student* налази се на посебној адреси (80), док се метода *Prikazi()* класе *Osoba* налази на адреси (90).
Методу *PrikaziOsobu* од класе *Osoba* класа *Student* није прекрила (није је имплементирала) тако да табеле метода класа *Osoba* и *Student* показују на исту адресу (95) где се налази метода *PrikaziOsobu* класе *Osoba*. Класа *Student* поред две наведене методе *Prikazi()* и *PrikaziOsobu()* има и додатну методу *PrikaziStudent()* која се налази на адреси 85. Табеле метода (*mtOsoba* и *mtStudent*) се пуне по редоследу наведених метода у класама.

У време компајлирања наредба:
os.Prikazi();

се преводи у следећи облик сличан наведеном:
os.pmt[0];

У време компајлирања се не може разрешити која метода *Prikazi()* ће бити позвана јер се не зна на који ће објекат *os* да добије референцу (адресу).

У време извршења програма наредба:
os = os1;

има следећи ефекат на наредбу:
os.pmt[0];

Преко садржаја од *os* се приступа садржају од *pmt* који приступа садржају [0] индекса од *mtOsoba*.
S(os).S(pmt).S([0]) ------> 10.40.90 – Приступа се до методе *Prikazi ()* класе *Osoba*[31].

[30] Адресе на којима се налазе промењиве и објекти су фиктивни.

[31] Нотација нпр. S(os) указује на садржај од *os*, што је у наведеном примеру адреса 10 где се налази објекат класе *Osoba*.

У време извршења програма објекат који се налази на адреси 10 (коме се приступа преко *os*) се повезује са методом која је на адреси 90.

У време извршења програма наредба
os = st;

има следећи ефекат на наредбу:
os.pmt[0];

Преко садржаја од *st* се приступа садржају од *pmt* који приступа садржају [0] индекса од *mtStudent.*

S(st).S(pmt).S([0]) ------> 30.50.80 – Приступа се до методе *Prikazi ()* класе *Student.*

У време извршења програма објекат који се налази на адреси 30 (коме се приступа преко *os*) се повезује са методом која је на адреси 80.

Уколико се стави кључна реч static испред имена методе, таква метода се повезује са објектом у време компајлирања програма. Такво повезивање се зове **рано повезивање** (early binding) или статичко повезивање објекта са методом. Код раног повезивање објекат је повезан са једном методом и не може се у току извршења програма повезати са неком другом методом.

4.1.4 Апстрактне класе и интерфејси

Уколико желимо да понашање неке класе издигнемо на општији ниво тада користимо **апстрактне класе** и **интерфејсе**. Апстрактне класе поред атрибута и обичних метода имају и **апстрактне методе**, док интерфејси имају само **операције**. И апстрактне методе и операције имају *само потпис без имплементације*. Класе које наслеђују апстрактне класе или имплементирају интерфејс морају да реализују апстрактне методе и операције. На пример, уколико имамо апстрактну класу *Osoba* која има апстрактну методу *Prikazi(),*

```
abstract class Osoba
  { String ImeOsobe; int BrojGodina;
    Osoba(String ImeOsobe1, int BrojGodina1)  {ImeOsobe = ImeOsobe1; BrojGodina =  BrojGodina1;}
    abstract void Prikazi();
    void PrikaziOsobu() { System.out.println("Ime osobe: " + ImeOsobe);}
  }
```

потребно је да изведена класа *Student* имплементира (реализује) наведену апстрактну методу:

```
class Student extends Osoba
  { String BrojIndeksa;
    Student(String ImeOsobe1, int BrojGodina1,String BrojIn1){ super(ImeOsobe1,BrojGodina1); BrojIndeksa = BrojIn1;}
    void Prikazi() { System.out.println("Ime osobe: " + ImeOsobe + "Broj godina:" + BrojGodina + " Broj indeksa:" + BrojIndeksa);}
    void PrikaziStudenta() { super.PrikaziOsobu(); System.out.println("Broj indeksa:" + BrojIndeksa);}
  }
```

На сличан начин уколико имамо интерфејс *Osoba,*

```
interface Osoba
  {  void Prikazi();
     void PrikaziOsobu();
  }
```

класа *Student* која имплементира интерфејс *Osoba* треба да реализује методе интерфејса:

```
class Student implements Osoba
  {  String ImeStudenta; int BrojGodina; String BrojIndeksa;
     Student(String ImeOs1, int BrojGod1,String BrojIn1){ ImeOsobe = ImeOs1; BrojGodina =  BrojGod1; BrojIndeksa = BrojIn1;}
     void Prikazi() { System.out.println("Ime osobe: " + ImeOsobe + "Broj godina:" + BrojGodina + " Broj indeksa:" + BrojIndeksa);}
     void PrikaziOsobu() { System.out.println("Ime osobe: " + ImeStudenta);}
     void PrikaziStudenta() { super.PrikaziOsobu(); System.out.println("Broj indeksa:" + BrojIndeksa);}
  }
```

Могуће је направити објектну променљиву типа апстрактне класе или интерфејса,
Osoba os;
али није могуће направити појављивање апстрактне класе или интерфејса:
Osoba os = new Osoba();
Наведена наредба није коректна.

За објектну променљиву типа апстрактне класе или интерфејса важи правило компатибилности објектних типова, што значи да она може да добије референцу на објекат класе која наслеђује апстрактну класу или имплементира интерфејс. Нпр:

```
Osoba os;
Student st = new Student();
os = st;
```

4.2 Имплементација општег облика GOF патерна пројектовања

Као што је речено патерн је процес трансформације структуре проблема у структуру решења. У наставку ћу дати имплементацију структуре проблема и структуре решења код општег облика GOF патерна пројектовања. Пре тога ћемо објаснити пример програма који је тежак за одржавање и надоградњу и објаснићемо главни разлог зашто уводимо патерне. Уколико имамо класу *BrokerBazePodataka* и њену методу *brisiSlog(),*

```
class BrokerBazePodataka
 {  Student st;
    Predmet pr;
    BrokerBazePodataka(Student st1, Predmet pr1)  {st = st1; pr = pr1;}

    public boolean brisiSlog(String ImeKlase)
           {  String upit;
              String UslovZaNadjiSlog;
              If ImeKlase.equals("Student")
                    UslovZaNadjiSlog = st.vratiUslovZaNadjiSlog();
              If ImeKlase.equals("Predmet")
                    UslovZaNadjiSlog = pr.vratiUslovZaNadjiSlog();
              try {  st = con.createStatement();
                     upit ="DELETE *  FROM " + imeKlase + " WHERE " + UslovZaNadjiSlog;
                     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;
           }

    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 transakcije.
                   } 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;
           }
}
```

која се позива из главног програма,

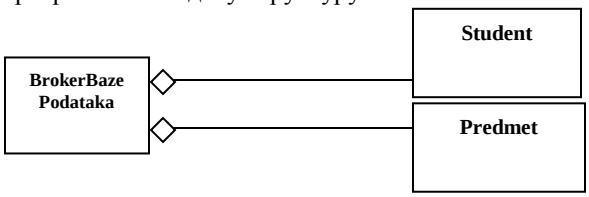
```
class Main
{  public static void main(String arg[])
    {  Student st = new Student();
       Predmet pr = new Predmet();
       BrokerBazePodataka bbp = new BrokerBazePodataka(st,pr);
       bbp.otvoriBazu("Fakultet");
       st.BrojIndeksa = "123-09";
       bbp.brisiSlog("Student");
       pr.SifraPredmeta = 11;
       bbp.brisiSlog("Predmet");
    }
}
```

и која треба да обришу слогове из табела *Student* и *Predmet*, на основу вредности објеката *st* и *pr* класа *Student* и *Predmet*.

```
class Student
{  String BrojIndeksa;
   String ImeStudenta;
   …
   public String vratiUslovZaNadjiSlog() { return " BrojIndeksa = '"+ BrojIndeksa + "'"; }
}
```

```
class Predmet
{ int SifraPredmeta;
  String NazivPredmeta;
  ...
  public String vratiUslovZaNadjiSlog() { return " SifraPredmeta = " + SifraPredmeta; }
}
```

Наведени програм има следећу структуру:

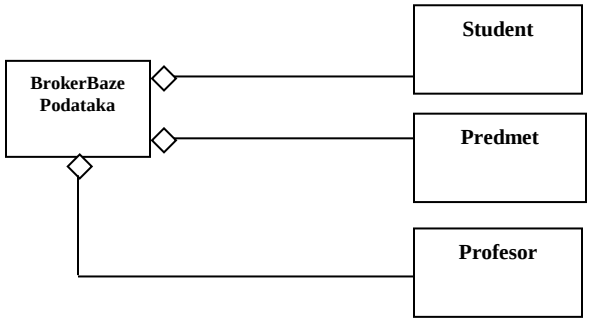


У наведеном примеру *BrokerBazePodataka* је клијент, док су *Student* и *Predmet* конкретни сервери. Наведена структура је тешка за одржавање и надоградњу јер свако додавање нове класе, нпр.класе *Profesor*

```
class Profesor
{ String ImeProfesora;
   …
  public String vratiUslovZaNadjiSlog() { return " ImeProfesora = '"+ ImeProfesora + "'"; }
 }
```

која би била повезана са брокером базе података захтева промену брокера (клијента):

```
class BrokerBazePodataka
 {   Student st;
     Predmet pr;
     Profesor prof;
     BrokerBazePodataka(Student st1, Predmet pr1)  {st = st1; pr = pr1;}
      ...
     public boolean brisiSlog(String ImeKlase)
           {  String upit;
              String UslovZaNadjiSlog;
               If ImeKlase.equals("Student")
                    UslovZaNadjiSlog = st.vratiUslovZaNadjiSlog();
               If ImeKlase.equals("Predmet")
                    UslovZaNadjiSlog = pr.vratiUslovZaNadjiSlog();
               If ImeKlase.equals("Profesor")
                    UslovZaNadjiSlog = prof.vratiUslovZaNadjiSlog();
                     ...
           }
  …
 }
```



Наведени проблем се решава тако што се уводи апстрактна класа *OpstaDomenskaKlasa*, која има апстрактну методу *vratiUslovZaNadjiSlog()*

```
abstract class OpstaDomenskaKlasa
 { public String vratiUslovZaNadjiSlog();
   public String vratiImeKlase();
}
```

```
class Student extends OpstaDomenskaKlasa
{  String BrojIndeksa;
   String ImeStudenta;
    …
   public String vratiUslovZaNadjiSlog() { return " BrojIndeksa = '"+ BrojIndeksa + "'"; }
   public String vratiImeKlase() {return "Student";}
}
class Predmet extends OpstaDomenskaKlasa
{ int SifraPredmeta;
   String NazivPredmeta;
   ...
   public String vratiUslovZaNadjiSlog() { return " SifraPredmeta = " + SifraPredmeta; }
   public String vratiImeKlase() {return "Predmet;}
}
class Profesor extends OpstaDomenskaKlasa
{
   String ImeProfesora;
    …
   public String vratiUslovZaNadjiSlog() { return " ImeProfesora = '"+ ImeProfesora + "'"; }
   public String vratiImeKlase() {return "Profesor;}
}
```

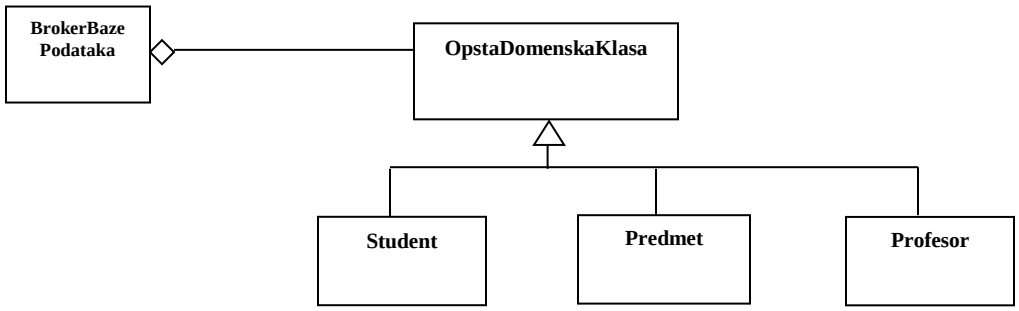
док ће брокер базе података да има следећи изглед:

```
class BrokerBazePodataka
 {
     OpstaDomenskaKlasa odk;
      ...
      public boolean brisiSlog()
            { String upit;
              String UslovZaNadjiSlog;
              String ImeKlase;
               ImeKlase = odk.vratiImeKlase();
               UslovZaNadjiSlog = odk.vratiUslovZaNadjiSlog();
                try {  st = con.createStatement();
                       upit ="DELETE *  FROM " + imeKlase + " WHERE " + UslovZaNadjiSlog;
                        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;
               }
       …
}
```

док ће главни програм имати следећи изглед:

```
class Main
{  public static void main(String arg[])
    {  Student st = new Student();
       Predmet pr = new Predmet();
       Profesor prof = new Profesor();
       BrokerBazePodataka bbp = new BrokerBazePodataka();
       bbp.otvoriBazu("Fakultet");
       st.BrojIndeksa = "123-09";
       bbp.odk = st;
       bbp.brisiSlog();
       pr.SifraPredmeta = 11;
       bbp.odk = pr;
       bbp.brisiSlog();
      pr.ImeProfesora = "Milan Petrovic";
       bbp.odk = prof;
       bbp.brisiSlog();
    }
}
```

Наведени програм има следећу структуру:



Ова структура је лакша за одржавање јер додавање нове класе не мења клијента, односно брокер базе података. Уколико се додаје нова класа, нпр. *Sala*, она треба само да имплементира апстрактне методе апстрактне класе *OpstaDomenskaKlasa*.

```
class Sala extends OpstaDomenskaKlasa
{ int SifraSale;
   …
  public String vratiUslovZaNadjiSlog() { return " SifraSale = "+ SifraSale; }
  public String vratiImeKlase() {return "Sala";}
}
```

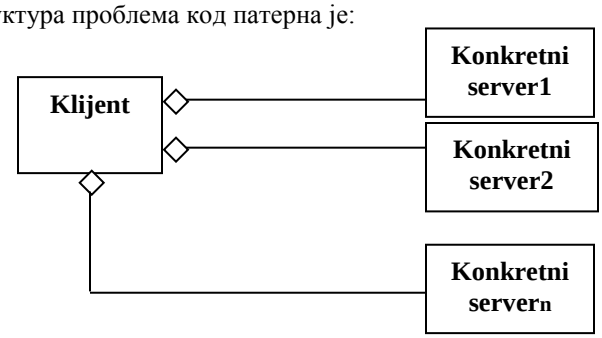
Можемо да приметимо да се метода *brisiSlog()* класе *BrokerBazePodataka* не мења.

На основу наведеног примера можемо да објаснимо општи облик GOF патерна пројектовања. Проблем код патерна можемо да представимо на следећи начин[32]:

```
class Klijent
{ KonkretniServer1 ks1; KonkretniServer2 ks2; … KonkretniServern ksn;
 Klijent(KonkretniServer1 ks11, KonkretniServer1 ks22,..., KonkretniServer1 ksn1){ ks1=ks11;ks2=ks21;…;ksn = ksn1;}
  void op(uslov)
     {  if (uslov1 = uslov)  ks1.op();
        if (uslov2 = uslov)  ks2.op();
        …
       if (uslovn = uslov)    ksn.op();
     }
 }
class KonkretniServer1
{ op() {…}  // имплементација методе op() класе KonkretniServer1
}
class KonkretniServer2
{ op() {…}  // имплементација методе op() класе KonkretniServer2
}
...
class KonkretniServern
{ op() {…}  // имплементација методе op() класе KonkretniServern
}
class Main
{ public static void main(String arg[])
    {  KonkretniServer1 ks1 = new KonkretniServer1();
       KonkretniServer2 ks2 = new KonkretniServer2();
        …
        KonkretniServern ksn = new KonkretniServern();
        Klijent kl = new Klijent(ks1,ks2,…,ksn);

       kl.op(uslov1);
       kl.op(uslov2);
       ...
       kl.op(uslovn);
    }
  }
```

[32] Објашњење и имплементацију ћемо урадити у објектном псеудокоду који је сличан програмском језику Јава.

Структура проблема код патерна је:



За наведену структуру је речено да је тешка за одржавање јер се при додавању новог конкретног сервера, нпр. **KonkretniServer$_{n+1}$**, мења клијент. Такође, клијент се у време компајлирања програма везује за конкретни сервер, што онемогућава флексибилност програма у току његовог извршавања. Наведени проблем код патерна се решава, нпр. увођењем апстрактног сервера:

```
abstract class ApstraktniServer
{abstract op();}
```

из кога су изведени конкретни сервери:

```
class KonkretniServer1 extends ApstraktniServer
{ op() {…}  // имплементација методе op() класе KonkretniServer1
}
class KonkretniServer2 extends ApstraktniServer
{ op() {…}  // имплементација методе op() класе KonkretniServer2
}
...
class KonkretniServerₙ extends ApstraktniServer
{ op() {…}  // имплементација методе op() класе KonkretniServerₙ
}
```

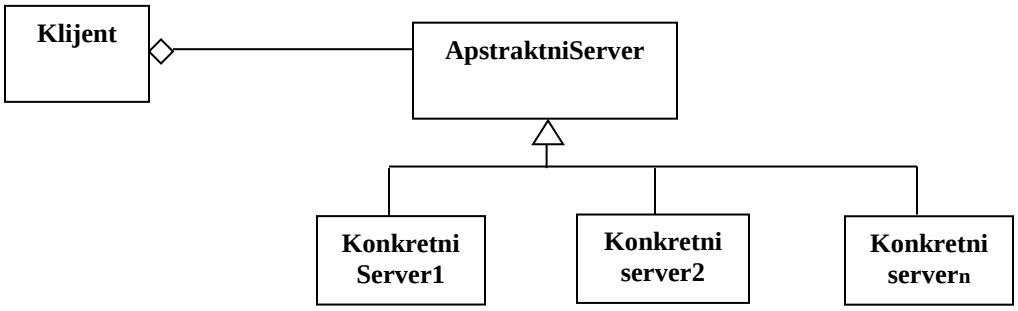
док клијент има следећи изглед:

```
class Klijent
{ ApstraktniServer as;
  void op(){ as.op();}
}
```

Главни програм је:

```
class Main
{ public static void main(String arg[])
    {   KonkretniServer1 ks1 = new KonkretniServer1();  KonkretniServer2 ks2 = new KonkretniServer2();   …
        KonkretniServerₙ ksₙ = new KonkretniServerₙ();
        Klijent kl = new Klijent();
        kl.as = ks1; kl.op();     kl.as = ks2; kl.op();     ...   kl.as = ksₙ; kl.op();
    }
 }
```

Структура решења код патерна је:



Додавање новог конкретног сервера *KonkretniServer$_{n+1}$* не мења клијента:

```
class KonkretniServerₙ₊₁ extends ApstraktniServer
{ op() {…}  // имплементација методе op() класе KonkretniServerₙ₊₁
}
```

4.3 Прављење генеричких метода

Као што је раније речено, када развијамо одржив програм потребно је да раздвојимо генеричке од специфичних делова програма. Генерички делови програма су представљени методама које се могу користити у различитим доменима проблема. Нпр. брокер базе података је пример класе која садржи скуп генеричких метода које омогућавају перзистентност доменских објеката при раду са базом података.

Када правимо генеричку методу потребно је да у методи препознамо места која се мењају са променом домена проблема и места која су непроменљива. На променљивим местима треба поставити механизам који ће омогућити непромењивост програмског кода. Променљива места у програму се обично реализују коришћењем операција интерфејса или апстрактних метода апстрактних класа.

На пример уколико имамо две методе које сортирају низ у растућем и опадајућем редоследу:

```
class Sortiranje
 {  static void sortRastuci(int n[])
       { int pom = 0;
         for(int i=0; i< n.length-1;i++)
               for(int j=0; j<n.length; j++)
                     { if(n[i]>n[j])
                           { pom = n[i];  // prvi jednako drugi
                             n[i] = n[j]; // drugi jednako treci
                             n[j] = pom;  // treci jednako prvi
                           }
                      }
     }
static void sortOpadajuci(int n[])
       { int pom = 0;
          for(int i=0; i< n.length-1;i++)
              for(int j=0; j<n.length; j++)
               { if(n[i]<n[j])
                    { pom = n[i];  // prvi jednako drugi
                      n[i] = n[j]; // drugi jednako treci
                      n[j] = pom;  // treci jednako prvi
                    }

               }
       }
static void Prikazi(int n[])
  {   System.out.println("Elementi niza su:");
      for(int i=0; i<n.length;i++) { System.out.println(n[i]);}
  }
public static void main(String arg[])
   { int n[] = {7,1,2,8,3};
     sortOpadajuci(n);   Prikazi(n);
     sortRastuci(n);       Prikazi(n);
   }
 }
```

можемо да приметимо да се наведене две методе разликују у наредби која одређује услов када се два елемента низа мењају место. Прецизније речено те две методе се разликују у оператору > односно <. Од наведене две методе треба направити једну генеричку методу:

```
static void sort (int n[])
       {  int pom = 0;
          for(int i=0; i< n.length-1;i++)
                for(int j=0; j<n.length; j++)
                   { if(n[i] © n[j])  // симбол © може бити > или <.  Уколико се изабере оператор >
                                      // низ ће бити сортиран у растућем редоследу. Уколико се изабере
                                      // оператор < низ ће бити сортиран у  опадајућем редоследу
                       { pom = n[i];
                         n[i] = n[j];
                         n[j] = pom;
                       }
                    }
       }
```

У наведеној методи симбол © у програму указује на оно што је променљиво у програму.
Наредба **if(n[i] © n[j])** је променљива. Она може да има два облика:

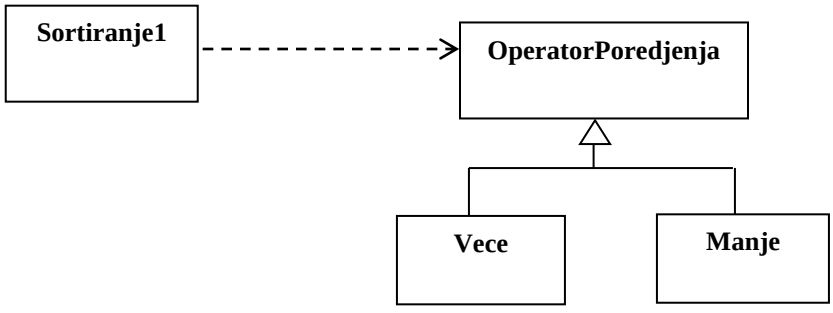
a) **if (n[i] > n[j])** – сортирање низа у растућем редоследу.
b) **if (n[i] < n[j])** – сортирање низа у опадајућем редоследу.

Наведени симбол ће бити имплементиран коришћењем операције ***Poredi()*** интерфејса ***OperatorPoredjenja.***

```
class Sortiranje1
{   static void sort(int n[], OperatorPoredjenja op)
      { int pom = 0;
        for(int i=0; i< n.length-1;i++)
            for(int j=0; j<n.length; j++)
              { if(op.poredi(n[i],n[j])) { pom = n[i];  n[i] = n[j];  n[j] = pom;   }
              }
      }
    static void Prikazi(int n[])
      {   System.out.println("Elementi niza su:");
          for(int i=0; i<n.length;i++) { System.out.println(n[i]); }
      }
     public static void main(String arg[])
        { int n[] = {7,1,2,8,3};
          sort(n,new Manje());
          Prikazi(n);
          sort(n,new Vece());
          Prikazi(n);
       }
}
interface OperatorPoredjenja
{boolean poredi(int a, int b);}

class Vece implements OperatorPoredjenja
{ public boolean poredi(int a, int b)
   {  if (a>b) return true;
      return false;
   }
}

class Manje implements OperatorPoredjenja
{ public boolean poredi(int a, int b)
   {  if (a<b) return true;
      return false;
   }
}
```

На основу наведеног можемо да изведемо правило:
Уколико желимо да метода постане генеричка, потребно је да се уоче промељиве наредбе у телу методе (if(n[i]>n[j]), if(n[i]<n[j])) и и да се замене са генеричким наредбама (op.poredi(n[i],n[j])). Оно што је специфично (промењљиве наредбе) у методи постаје параметар методе (OperatorPoredjenja op) који је обично типа апстрактне класе или интерфејса (interface OperatorPoredjenja).

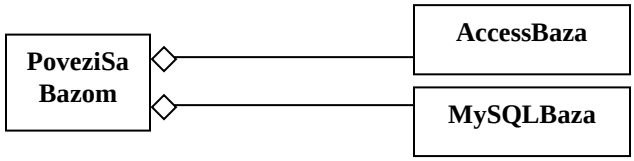
Можемо да приметимо да је у процесу прављења генеричких метода коришћен концепт патерна, прецизније речено структура решења код патерна:



У следећем примеру се Јава програм повезује са два различита система за управљање базом података помоћу две различите методе ***Povezi()*** класа ***AccessBaza*** и ***MySqlBaza***:

```
import java.sql.*;
class AccessBaza
 {public void povezi()
    {   try { String dbUrl="jdbc:odbc:student";
              String user= "root";
              String pass="root";
              Class.forName("sun.jdbc.odbc.JdbcOdbcDriver"); // C:\Program Files\Java\jdk1.5.0_06\jre\lib\rt.jar
              Connection c=DriverManager.getConnection(dbUrl,user,pass);
              System.out.println("Program je povezan sa MS Access SUBP!!!");
              c.close();
            } catch(ClassNotFoundException cnfe){ System.out.println("Nije ucitan upravljacki program: "+cnfe);    }
              catch(SQLException sqle){  System.out.println("Greska kod konekcije: "+sqle);  }
    }
}
class MySqlBaza
 { public void povezi()
     {  try{  String dbUrl = "jdbc:mysql://127.0.0.1:3306/student";
              String user= "root";
              String pass="root";
              Class.forName("com.mysql.jdbc.Driver"); // C:\Install\MySQL5.0\mysql-connector-java-3.1.12\mysql-connector-java-3.1.12.jar
              Connection c = DriverManager.getConnection(dbUrl,user,pass);
              System.out.println("Program je povezan sa MySQL SUBP!!!");
              c.close();
            } catch(ClassNotFoundException cnfe){  System.out.println("Nije ucitan upravljacki program: "+cnfe);    }
              catch(SQLException sqle){  System.out.println("Greska kod konekcije: "+sqle);  }
     }
}

class PoveziSaBazom
{ AccessBaza ab;
  MySqlBaza ma;
  public static void main(String arg[])
  { PoveziSaBazom psb;
    psb.ab = new AccessBaza();
    psb.ab.povezi();
    psb.ma = new MySqlBaza();
    psb.ma.povezi();
  }
}
```

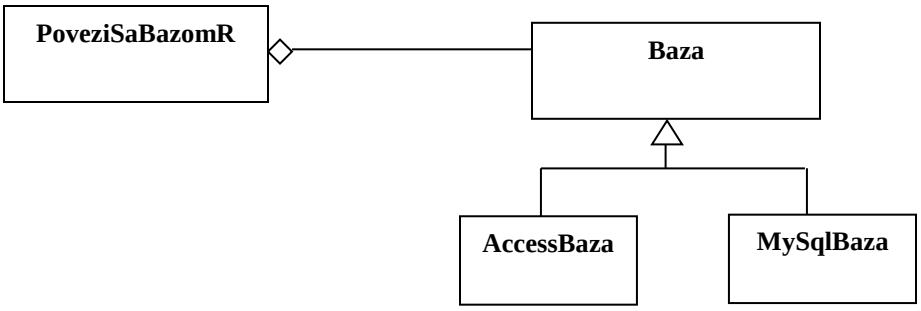
Структура наведеног програма је:



Уколико желимо да направимо генеричку методу помоћу које ће бити омогућено повезивање Јава програма са различитим системима за управљање базама података потребно је да препознамо променљива места у наведеним методама и да на њихово место поставимо генеричке наредбе (*vratiURL(), vratiDrajver(),…*). Наведене методе су апстрактне методе које се имплементирају у оквиру класа (*AccessBaza, MySqlBaza*) које се односе на конкретне системе за управљање базом података са којима ће Јава програм бити повезан.

```
import java.sql.*;
abstract class Baza
{ public void povezi()
      {  try {   String dbUrl = vratiUrl();
                 String user = vratiUserName();
                 String pass = vratiPassword();
                 Class.forName(vratiDrajver());
                 Connection c = DriverManager.getConnection(dbUrl,user,pass);
                  System.out.println(vratiPoruku());
                  c.close();
              } catch(ClassNotFoundException cnfe){ System.out.println("Nije ucitan upravljacki program: "+cnfe);    }
                catch(SQLException sqle){  System.out.println("Greska kod konekcije: "+sqle);  }
            }
   abstract String vratiUrl();
   abstract String vratiDrajver();
   abstract String vratiPoruku();
   abstract String vratiUserName();
   abstract String vratiPassword();
}

class AccessBaza extends Baza
 { String vratiUrl(){return "jdbc:odbc:student";}
   String vratiDrajver(){return "sun.jdbc.odbc.JdbcOdbcDriver";}
   String vratiPoruku(){return "Program je povezan sa MS Access SUBP!!!";}
   String vratiUserName(){return "";}
   String vratiPassword(){return "";}
}

class MySqlBaza extends Baza
 { String vratiUrl(){return "jdbc:mysql://127.0.0.1:3306/student";}
   String vratiDrajver(){return "com.mysql.jdbc.Driver";}
   String vratiPoruku(){return "Program je povezan sa MySQL SUBP!!!";}
   String vratiUserName(){return "root";}
   String vratiPassword(){return "root";}
  }
class PoveziSaBazomR
{ Baza ba;
  public static void main(String arg[])
  { PoveziSaBazomR psb;
    AccessBaza ab = new AccessBaza();
    psb.ba = ab;
    psb.ba.povezi();
    MySqlBaza ma = new MySqlBaza();
    psb.ba = ma;
    psb.ba.povezi();
  }
}
```

Структура наведеног решења је:



Из наведеног примера можемо да изведемо следеће правило:
Генеричка метода се имплементира у помоћу структуре решења код патерна.

У следећем примеру се приказују слогови из две различите табеле коришћењем методе *prikazi()* која није генеричка, јер се мења сваки пут када се додаје нова табела, односно када се желе приказати слогови нове табеле.

```
import java.sql.*;

class AccessBaza
 {  Connection c;
    public void povezi()
      {   try {   String dbUrl="jdbc:odbc:student";
                  Class.forName("sun.jdbc.odbc.JdbcOdbcDriver"); // C:\Program Files\Java\jdk1.5.0_06\jre\lib\rt.jar
                  c=DriverManager.getConnection(dbUrl,"","");
                   System.out.println("Program je povezan sa MS Access SUBP!!!");
              } catch(ClassNotFoundException cnfe){ System.out.println("Nije ucitan upravljacki program: "+cnfe);    }
                  catch(SQLException sqle){  System.out.println("Greska kod konekcije: "+sqle);  }
       }

void prikazi(String imeTabele)
  {   try {   Statement naredba =c.createStatement();
              String upit="SELECT * FROM " + imeTabele;
              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 " + imeTabele);
                    while(rs.next())
                        {  if (imeTabele.equals("Student"))
                             System.out.println(rs.getString("brind")+" " + rs.getString("ime")+" "+rs.getString("prezime"));
                           if (imeTabele.equals("Predmet"))
                             System.out.println(rs.getString("sifraPredmeta")+" " + rs.getString("nazivPredmeta"));
                        }
                    naredba.close();
                    c.close();
                   }  catch(SQLException se){ System.out.println("Nedozvoljena operacija: "+se);}
           }
}

class PrikaziSlogoveTabele
{ public static void main(String arg[])
   { AccessBaza ab = new AccessBaza();
     ab.povezi();
     ab.prikazi("Student");

     ab.povezi();
     ab.prikazi("Predmet");
   }
}
```

Уколико желимо да направимо генеричку методу потребно је да следећи програмски код методе *prikazi()* учинимо непромењивим:

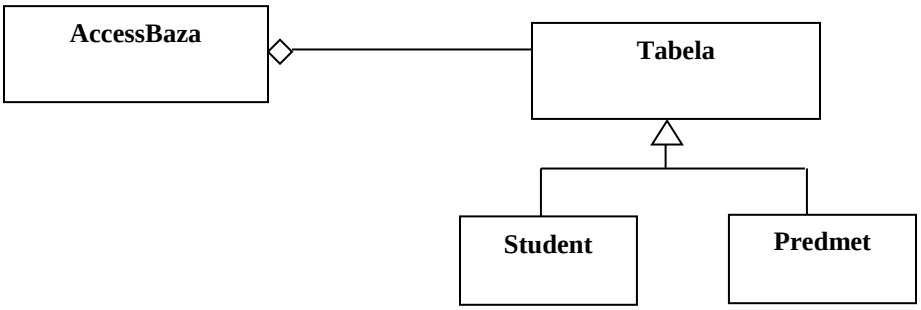
```
if (imeTabele.equals("Student"))
       System.out.println(rs.getString("brind")+" " + rs.getString("ime")+" "+rs.getString("prezime"));
if (imeTabele.equals("Predmet"))
    System.out.println(rs.getString("sifraPredmeta")+" " + rs.getString("nazivPredmeta"));
```

Наводим програм код кога је направљена метода *Prikazi()*.

```
import java.sql.*;

class AccessBaza
{
  Connection c;
  public void povezi(){
  try{   String dbUrl="jdbc:odbc:student";
         Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
         c=DriverManager.getConnection(dbUrl,"","");
         System.out.println("Program je povezan sa MS Access SUBP!!!");
    } catch(ClassNotFoundException cnfe){ System.out.println("Nije ucitan upravljacki program: "+cnfe);    }
      catch(SQLException sqle){  System.out.println("Greska kod konekcije: "+sqle);  }
}
```

```
void prikazi(Tabela ta)
  {  try {  Statement naredba =c.createStatement();
            String upit="SELECT * FROM " + ta.vratiImeTabele();
            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 " + ta.vratiImeTabele());
             while(rs.next()) {  System.out.println(ta.vratiSlog(rs)); }
             naredba.close();
             c.close();
       }   catch(SQLException se){ System.out.println("Nedozvoljena operacija: "+se);}
  }
}

class PrikaziSlogoveTabeleR
{  public static void main(String arg[])
    { AccessBaza ab = new AccessBaza();
      ab.povezi();
      ab.prikazi(new Student());
      ab.povezi();
      ab.prikazi(new Predmet());
    }
}
interface Tabela
{  String vratiImeTabele();
   String vratiSlog (ResultSet rs) throws SQLException;
}
class Student implements Tabela
{ public String vratiImeTabele(){return "Student";}
  public String vratiSlog (ResultSet rs) throws SQLException
           {return rs.getString("brind")+" " + rs.getString("ime")+" "+rs.getString("prezime");}
}
class Predmet implements Tabela
{ public String vratiImeTabele(){return "Predmet";}
  public String vratiSlog (ResultSet rs) throws SQLException
           {return rs.getString("sifraPredmeta")+" " + rs.getString("nazivPredmeta");}
}
```

Структура наведеног решења:



У наведеном примеру је такође генеричка метода имплементирана помоћу структуре решења код патерна.

4.4 Рефлексија

Рефлексија је механизам који омогућава добијање основних информација о класи и свим њеним чланицама. Ове информације називамо метаподацима, које се у случају објектно орјентисаног програмирања, чувају у мета-објектима

За дату класу Racun:

```
class Racun
{
 String BrojRacuna;
 String NazivPartnera;
 Double UkupnaVrednost;
 boolean Obradjen;
 boolean Storniran;

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

public String vratiVrednostiAtributa()
  { return """+ BrojRacuna + "', '" + NazivPartnera + "', " +  UkupnaVrednost.doubleValue() + ", " + Obradjen + ", " +
Storniran;}

public String postaviVrednostiAtributa()
  { return "BrojRacuna = '"+ BrojRacuna + "', NazivPartnera ='" + NazivPartnera + "', UkupnaVrednost = " +
         UkupnaVrednost.doubleValue() + ", Obradjen = " + Obradjen + ", Storniran = " + Storniran;
  }

public String vratiImeKlase() { return "Racun";}
public String vratiAtributPretrazivanja() { return "BrojRacuna";}
public void postaviNazivPartnera(String NazivPartnera1) { NazivPartnera = NazivPartnera1; }
}
```

Уколико желимо приказати све атрибуте класе *Racun*, то можемо урадити на следећи начин :

```
import java.lang.reflect.Field;

public class Ref1
{ public Ref1() {}
  public static void main(String[] args)
    { Class metaX = Racun.class;
      System.out.println("informacije o klasi Racun");
      System.out.println("************************");
      System.out.println("naziv klase:"+ metaX.getName());
      Field [] atributi = metaX.getDeclaredFields();
      System.out.println("************************");
      System.out.println("atributi klase:");
      for (int i = 0; i < atributi.length; i++)
          {  System.out.println(atributi[i]);
          }
      System.out.println("************************");
    }
}
```

Уколико желимо приказати све методе класе *Racun*:

```
import java.lang.reflect.Method;

public class Ref2
  { public Ref2() { }
    public static void main(String[] args)
     {
       Class metaX = Racun.class;
```

```
      System.out.println("informacije o klasi Racun");
      System.out.println("*************************");
      System.out.println("naziv klase:"+metaX.getName());
      Method [] metode = metaX.getDeclaredMethods();
      System.out.println("*************************");
      System.out.println("metode klase:");
      for (int i = 0; i < metode.length; i++) { System.out.println(metode[i]); }
      System.out.println("*************************");
   }
}
```

Уколико желимо приказати све атрибуте класе *Racun*, њихова имена и типове:

```
import java.lang.reflect.Field;
public class Ref3
 { public Ref3() {}
   public static void main(String[] args)
   {  Class metaX = Racun.class;
      System.out.println("informacije o klasi Racun");
      System.out.println("*************************");
      System.out.println("naziv klase:"+metaX.getName());
      Field [] atributi = metaX.getDeclaredFields();
      System.out.println("*************************");
      System.out.println("atributi klase:");
      for (int i = 0; i < atributi.length; i++) {System.out.println("atribut "+atributi[i].getName()+" tipa "+atributi[i].getType().getName());}
      System.out.println("*************************");
   }
}
```

Уколико желимо доделити атрибутима класе *Racun* неку вредност:

```
import java.lang.reflect.Field;
public class Ref4
 {   public static void main(String[] args)  throws IllegalAccessException
      {   Racun rac = new Racun();
          Class metaX = ((Object)rac).getClass();
          Field [] atributi = metaX.getDeclaredFields();
          for (int i = 0; i < atributi.length; i++)
              { atributi[i].set(rac,getValue(atributi[i].getType())); }
           for (int i = 0; i < atributi.length; i++)
              { System.out.println("Vrednost atributa: " + atributi[i].getName() + " je: " + atributi[i].get(rac));  }
      }
    static Object getValue(Class c) throws IllegalAccessException
     { if (c.getName().equals("java.lang.String")) return ((Object) new String("prazno"));
       if (c.getName().equals("java.lang.Double")) return ((Object) new Double(0.0));
       if (c.getName().equals("boolean")) return ((Object) new Boolean(true));
       return null;
     }
}
```

Наредба:

atributi[i].set(rac,getValue(atributi[i].getType()));

може да се схвати на следећи начин:

rac.atributi[i].set(getValue(atributi[i].getType()));

Наредба:

atributi[i].get(rac);

може да се схвати на следећи начин:

rac.atributi[i].get();

4.5 Generic механизам (Јава)

Generic механизам у Јави омогућава да се параметризују интерфејси и класе како би се омогућило писање генеричких метода чија логика не зависе од типова тих параметара.

```
// Задатак GENM1: Коришћењем generic механизма у Јави направити класу која ће имати методу која сортира
// низ  елемената произвољне класе у растућем редоследу.
abstract class Sortiranje<T>
 { void sort(T n[])
       { T pom;
         for(int i=0; i< n.length-1;i++)
                    for(int j=i+1; j<n.length; j++)
                       { if(poredi(n[i],n[j]))
                            { pom = n[i];
                              n[i] = n[j];
                              n[j] =pom;
                            }
                       }
       }
 void Prikazi(T n[]) { System.out.println("Elementi niza su:");  for(int i=0; i<n.length;i++) { System.out.println(n[i]);}}
 abstract boolean poredi(T a, T b);
}
class PorediCeleBrojeve extends Sortiranje <Integer>
 { public boolean poredi(Integer a, Integer b)
    {  if (a.compareTo(b)>0) { return true;}
       return false;
    }
 }
class PorediStringove extends Sortiranje <String>
 { public boolean poredi(String a, String b)
    {  if (a.compareTo(b)>0) return true;
       return false;
    }
 }
class Sortiranje1
 {   public static void main(String arg[])
         { Integer n[] = {7,1,2,8,3};
           PorediCeleBrojeve pcb = new PorediCeleBrojeve();
           pcb.Prikazi(n);
           pcb.sort(n);
           pcb.Prikazi(n);
           String n1[] = {"bata","ana","dejan","ceca"};
           PorediStringove ps = new PorediStringove();
           ps.Prikazi(n1);
           ps.sort(n1);
           ps.Prikazi(n1);
         }
}
```

Дајемо пример како изгледа програм са истим захтевом који није решен преко generic механизма већ преко генеричке Јавине класе *Object*:

```
abstract class Sortiranje
{  void sort(Object n[])
       { Object pom;
         for(int i=0; i< n.length-1;i++)
              for(int j=i+1; j<n.length; j++)
                 { if(poredi(n[i],n[j]))
                      { pom = n[i];
                        n[i] = n[j];
                        n[j] =pom;
                      }
                 }
       }

 void Prikazi(Object n[]) {  System.out.println("Elementi niza su:"); for(int i=0; i<n.length;i++) { System.out.println(n[i]);}}
abstract boolean poredi(Object a, Object b);
}
```

```
class PorediCeleBrojeve extends Sortiranje
 { boolean poredi(Object a, Object b)
    {  Integer a1 = (Integer)a;
       Integer b1 = (Integer)b;
       if (a1.compareTo(b1)>0) { return true;}
          return false;
    }
 }

class PorediStringove extends Sortiranje
 { boolean poredi(Object a, Object b)
    { String a1 = (String)a;
      String b1 = (String)b;
      if (a1.compareTo(b1)>0) { return true;}
         return false;
    }
 }

class Sortiranje2
 { public static void main(String arg[])
     { Integer n[] = {7,1,2,8,3};
       PorediCeleBrojeve pcb = new PorediCeleBrojeve();
       pcb.Prikazi(n);    pcb.sort(n);     pcb.Prikazi(n);
       String n1[] = {"bata","ana","dejan","ceca"};
       PorediStringove ps = new PorediStringove();
       ps.Prikazi(n1);    ps.sort(n1);    ps.Prikazi(n1);
        }
}
```

Дајемо пример како изгледа програм са истим захтевом који није решен преко *generic* механизма већ преко *рефлексије слањем методе као параметра* :

```
import java.lang.reflect.Method;

class Sortiranje
{ void sort(Object n[],Method met,Object ob) throws Exception
       { Object pom;
         for(int i=0; i< n.length-1;i++)
               for(int j=i+1; j<n.length; j++)
                  { if((boolean)met.invoke(ob,n[i],n[j]))
                       { pom = n[i];  n[i] = n[j];  n[j] =pom;
                       }
                  }
       }
 void Prikazi(Object n[])
       {  System.out.println("Elementi niza su:");
          for(int i=0; i<n.length;i++) { System.out.println(n[i]);}
        }
}

class PorediCeleBrojeve
    {   public boolean poredi(Integer a, Integer b)   { if (a.compareTo(b)>0)   return true;  return false;   }
    }

class PorediStringove
    {  public boolean poredi(String a, String b)  {  if (a.compareTo(b)>0)  return true; return false;  }
    }

class Sortiranje3
 {    public static void main(String arg[]) throws Exception
        {  Sortiranje s = new Sortiranje();
           Integer n[] = {7,1,2,8,3};
           PorediCeleBrojeve pcb = new PorediCeleBrojeve();
           Method m = pcb.getClass().getMethod("poredi",Integer.class,Integer.class);
           s.Prikazi(n);    s.sort(n,m,pcb);     s.Prikazi(n);
           PorediStringove ps = new PorediStringove();
           m = ps.getClass().getMethod("poredi",String.class,String.class);
           String n1[] = {"bata","ana","dejan","ceca"};
           s.Prikazi(n1);   s.sort(n1,m,ps);     s.Prikazi(n1);
        }
}
```

5. Патерни пројектовања и имплементације

5.1 GOF патерни пројектовања

Код развоја софтвера, уопштено гледајући, прво треба да се схвати и разуме разматрани проблем. Затим се врши његова анализа, да би се на крају вршило његово пројектовање и имплементација. У фази пројектовања софтверског система уочавају се класе пројектовања. Уколико се жели да наведене класе буду флексибилне (у смислу њиховог једноставног одржавања и надоградње), њих треба организовати помоћу **патерна пројектовања**. Патерни пројектовања помажу у именовању и опису **генеричких решења**, која могу бити примењена у различитим проблемским ситуацијама.

Ово поглавље је посвећено GOF патернима пројектовања*[GOF]*, који су поставили основу за даљи развој и схватање патерна пројектовања. **Патерни пројектовања** *[GOF] су описи комуникација између објеката, односно класа, који су прилагођени (кастомизовани) да реше генерални проблем у одређеном контексту.* Решење проблема треба да буде: а) **специфично,** како би се решио конкретан проблем и б) довољно **опште**, да буде решење или део решења будућих проблема и захтева. Патерни пројектовања омогућавају да се избегне или минимизира поновно пројектовање решења[33] за неку класу проблема.

Колико пута се десило да решење неког проблема код вас пробуди осећај "већ виђеног (déjà vu)", да сте већ раније решили неки проблем али не знате тачно где и како? Уколико се сећате детаља предходног проблема и начина како сте га решили тада ви можете да користите то *искуство* у решавању нових проблема уместо да их опет изнова истражујете. Сврха књиге **Design patterns** јесте да се *запамте искуства* у пројектовању објектно-оријентисаног софтвера и та искуства се називају **патерни пројектовања** (**design patterns**). *[GOF].*

Патерне пројектовања треба користити онда када се жели **динамичка измена функционалности** програма у току његовог извршавања.

Патерни су подељени у 3 групе:

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

ПК: Патерни за креирање објеката

*Патерни за креирање објеката апстракују процес **инстанцирања**, односно процес **креирања** објеката. Они дају велику **прилагодљивост** у томе **шта** ће бити креирано, **ко** ће то креирати и **када** и **како** ће то бити креирано.*

Постоје следећи патерни за креирање објеката:

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

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

3. **Factory Method** - Дефинише интерфејс за креирање објекта (производа), али преноси на подкласе одлуку коју ће класу истанцирати. Factory method преноси надлежност инстанцирања 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.

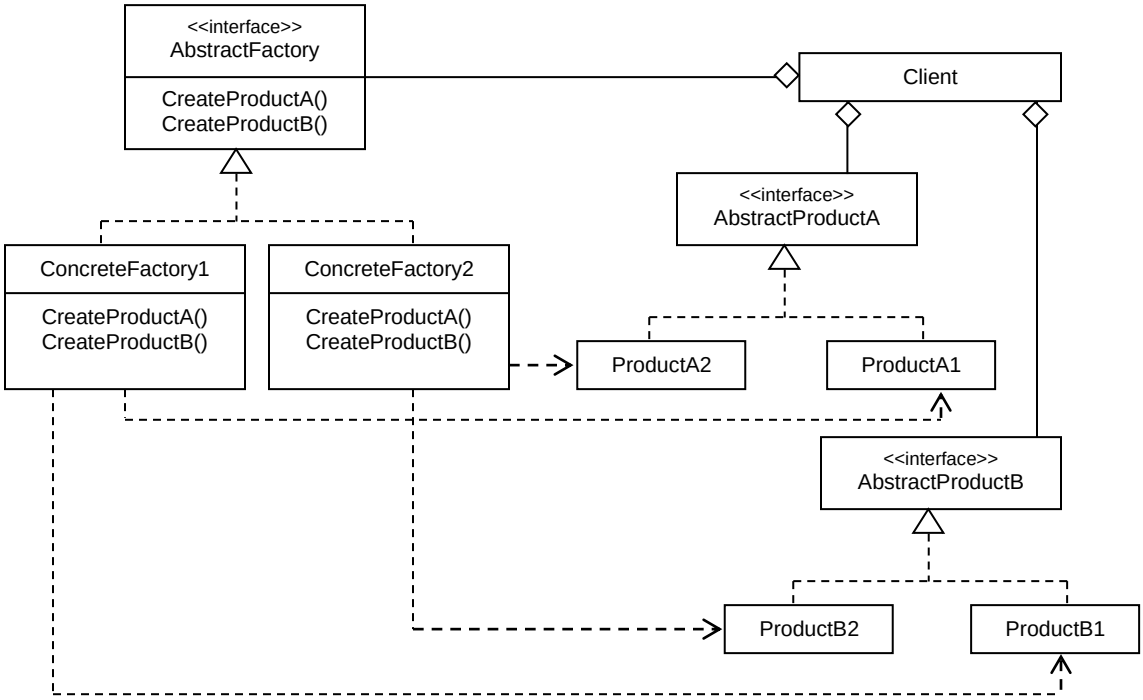
[33] Искуства пројектаната објектно-оријентисаног софтвера, говоре да поновно употребљиво (**reusable**) и прилагодљиво решење (**flexible design**) је тешко да се добије из првог пута. Пре него што се добије такво решење потребно је да се оно неколико пута користи (**reuse**) и модификује док се не добије генеричко решење које може да покрије класу проблема.

ПК1: Abstract Factory патерн

Дефиниција
Обезбеђује интерфејс[34] за креирање фамилије повезаних или зависних објеката (производа) без навођења њихових конкретних класа.

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

Структура Abstract Factory патерна



Учесници

- **Client** - Користи *AbstractFactory* и *AbstractProduct* интерфејсе и из њих изведене класе за креирање сложеног производа.
- **AbstractFactory** - Декларише интерфејс за операције *(CreateProductA(), CreateProductB())* које креирају производе [35].
- **ConcreteFactory** - Имплементира операције *(CreateProductA(), CreateProductB())* интерфејса *AbstractFactory*, којима се креирају производи *(ProductA1, ProductA2,ProductB1,ProductB2)*.
- **AbstractProduct** - Декларише интерфејс (*AbstractProductA, AbstractProductB*) за производе.
- **ConcreteProduct** - Дефинише производе *(ProductA1, ProductA2,ProductB1,ProductB2)* који ће бити креирани преко *ConcreteFactory* класа. Имплементира операције *AbstractProduct* интерфејса.

[34] Појам **интерфејс,** у ужем смислу, је концепт у објектно-оријентисаном 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. Интерфејс, у ширем смислу, подразумева нешто што садржи скуп операција које су видљиве крајњем кориснику, без улажења у то да ли су те операције имплементиране или нису у том интерфејсу. Интерфејс у ширем смислу, код објектно-оријентисаних програмских језика, може бити представљен преко класе (*class*), апстрактне класе (*abstract class*) или интерфејса (*interface*).

[35] Производи у суштини представљају делове сложеног производа који ће бити креиран преко Client класе. У каснијим примерима, док објашњавамо GOF патерне, када кажемо **производ** мислићемо на део сложеног производа. Када кажемо **сложени производ** подразумеваћемо да се он састоји од производа (његових делова).

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[36].

Уколико посматрамо кориснички захтев можемо приметити да Управа тражи од Лабораторије за Софтверско инжењерство (*SILAB*) да креира елементе понуде, односно програмски језик (*ProgramskiJezik*) и Систем за управљање базом података (*SUBP*). То се може 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[37].
У наведеном интерфејсу се може приметити да операције *kreirajProgramskiJezik* и *kreirajSUBP* враћају *ProgramskiJezik* и *SUBP*, који ће такође бити представљени преко интерфејса:

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

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

Лабораторија за софтверско инжењерство је направила два тима. Један је оријентисан ка Јави, док је други оријентисан ка Visual Basic-у (VB). Оба тима треба да одреде конкретан *ProgramskiJezik* и *SUBP* који ће дати у понуди[38].

```
// 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();}
}

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 *ProgramskiJezik (Java, VB)* и *SUBP (MySQL, MSAccess).*

---

[36] **Шта** ће бити креирано: ***Ponuda***
Ко ће креирати понуду: ***UpravaFakulteta***
Како ће се креирати понуда: видети методу ***Kreiraj().***
Када ће се креирати понуда: видети методу ***main().***

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

[38] Свака *ConcreteFactory* класа има посебну (специфичну) имплементацију код креирања производа (елементи понуде у овом примеру).

```
/* 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 интерфејса ProgramskiJezik и SUBP. */
class Java implements ProgramskiJezik // Product A1
{ public String vratiProgramskiJezik(){return "Java";}}

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

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[39]) 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 интерфејсе SILAB, ProgramskiJezik и SUBP и класе изведене из њих за креирање понуде.
class UpravaFakulteta // Client
{
  SILAB sil; // Abstract Factory
  ProgramskiJezik pj; // AbstractProductA
  SUBP subp; // AbstractProductB
  Ponuda pon;

  UpravaFakulteta(SILAB sil1){sil = sil1; pon = new Ponuda();}
  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()[40]
            {   pj = sil.kreirajProgramskiJezik();
               subp = sil.kreirajSUBP();
               pon.ponuda  = "Programski jezik-" + pj.vratiProgramskiJezik() + "  SUBP-" + subp.vratiSUBP();
               return pon.ponuda;
          }
}

// Улога: Дефинише понуду коју ће креирати класа UpravaFakulteta преко методе Kreiraj()l
class Ponuda {String ponuda;}
```

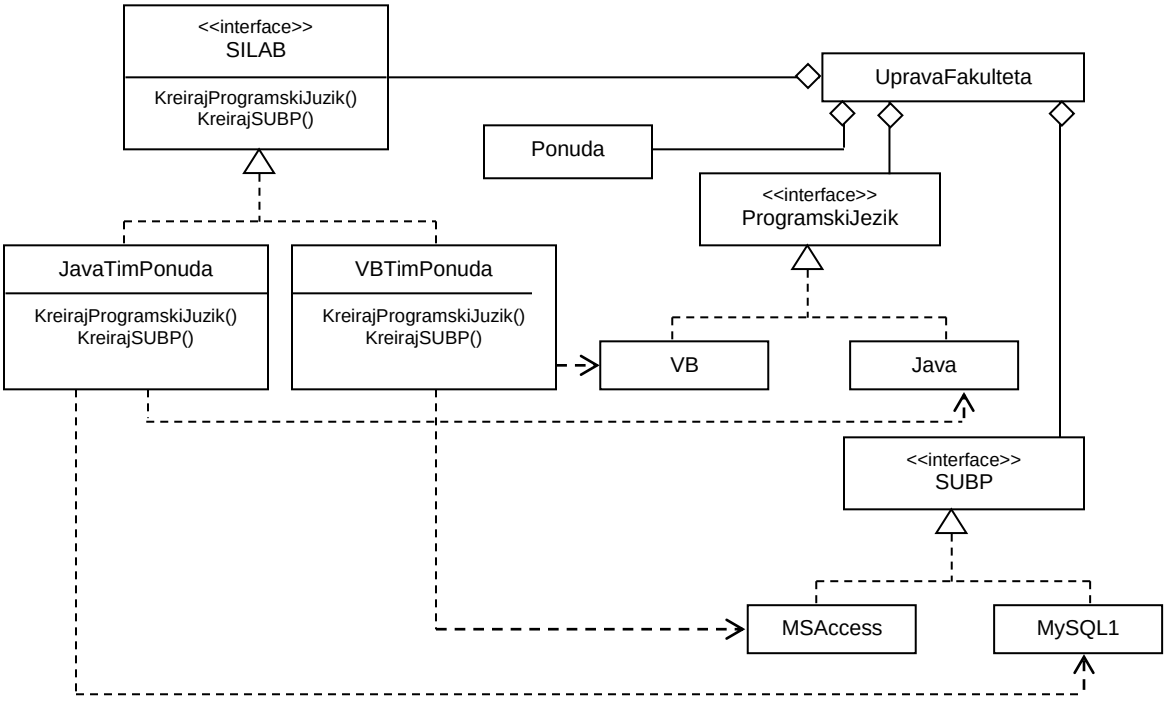
Пошто је класа *UpravaFakulteta* повезана са интерфејсима *SILAB, ProgramskiJezik* и *SUBP* може се рећи да је у наведеном примеру коришћен један од главних принципа поновног коришћења програмског кода у ОО пројектовању:

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

[39] Client класа користи различите *ConcreteFactory* класе када жели да креира различите сложене производе.

[40] Управа је одговорна за **контролу креирања** понуде. Управа је одговорна за **састављање понуде у целини**.Тимови су одговорни за **креирање елемената понуде**.

Дијаграм класа примера PAF1



Предности AbstractFactory патерна

Предности AbstractFactory патерна 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. Мења се једино *Client* класа (у нашем случају *UpravaFakulteta*) уколико се жели креирање елемената понуде преко нове *ConcreteFactory* класе.

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

Задатак ZAF1: *Извршити промене у програму PAF1 када се дода нови тим CTimPonuda.*

Недостаци AbstractFactory патерна

Тешко се додају нове врсте производа до *AbstractFactory* патерна. То је због тога што AbstractFactory 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, као и промену класе *Client.*

Задатак ZAF2: *Извршити промене у програму PAF1 када се дода нови елемент понуде OperativniSistem.*

Задатак ZAF3*: *Извршити промене у програму PAF1 тако да AbstractFactory патерн не зависи од броја производа који чине финални производ.*

Веза AbstractFactory патерна и општег облика патерна

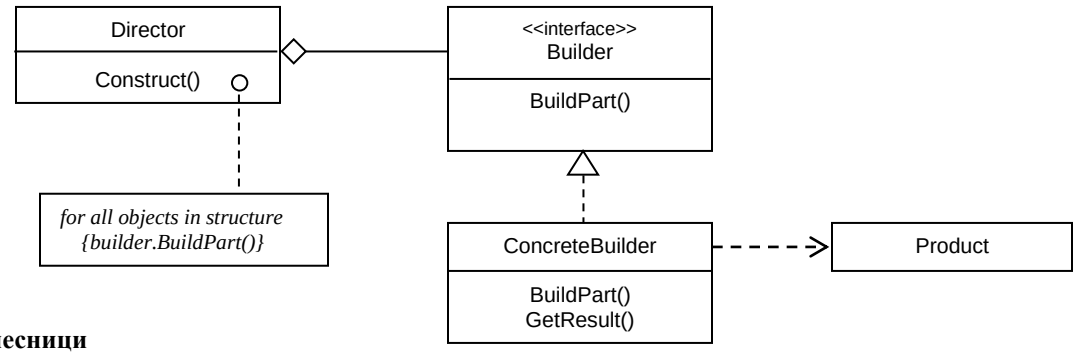
Број **СРП** код *AbstractFactory* патерна једнака је 1 (*Client, AbstractFactory, ConcreteFactory*) плус број производа (***n***) који чине сложени производ (*Client, AbstractProduct**i**, ConcreteProduct**ij***), где је i = (1..n), j = (1..m), где је **m** број *ConcreteProduct* класа i-тог *AbstractProduct* интерфејса.

ПК2: Builder патерн

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

Појашњење дефиниције
Дели одговорност за контролу конструкције (*Director*) сложеног производа од одговорности за реализацију његове репрезентације (*Builder*), тако да исти конструкциони процес (*Direktor.Construct()*) може да креира различите репрезентације (сложене производе).

Структура Builder патерна



Учесници

- **Director**
  Контролише процес конструкције сложеног производа коришћењем *Builder* интерфејса.
- **Builder**
  Специфицира интерфејс за креирање сложеног производа.
- **ConcreteBuilder**
  Конструише и групише производе у сложени производ имплементирајући Builder интерфејс.
- **Product**
  Репрезентује сложени производ који се конструише.

Пример Builder патерна

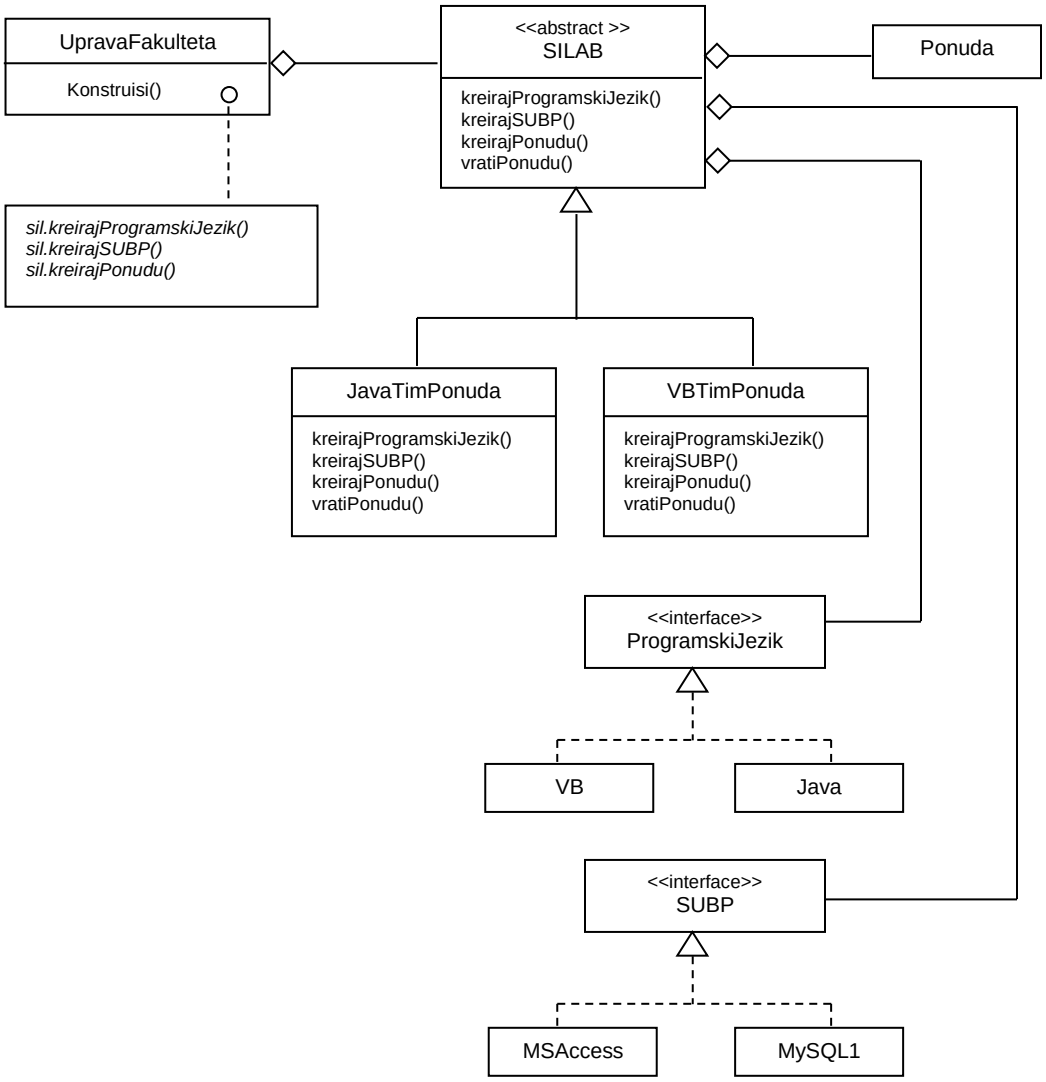
Кориснички захтев PBU1: *Управа Факултета је послала захтев Лабораторији за софтверско инзењерство да направи (састави) 2 понуде за израду софтверског система последипломских студија ФОН-а. У понуди треба се наведе:*
а) Програмски језик у коме ће се развијати програм.
б) Систем за управљање базом података у коме ће се чувати подаци.
Управа Факултета ће надзирати (контролисати) израду понуда.

```
// Улога: Контролише конструкцију понуде коришћењем Builder интерфејса.
class UpravaFakulteta   // Director
{
  SILAB sil; // Builder
  UpravaFakulteta(SILAB sil1){sil = sil1;}

  void Konstruisi()[41]
    { sil.kreirajProgramskiJezik();
      sil.kreirajSUBP();
      sil.kreirajPonudu();}
```

[41] Управа Факултета је одговорна за контролу креирања понуде. Тимови су одговорни за креирање елемената понуде и састављање понуде у целини.

```
    public static void main(String args[])
        {  UpravaFakulteta uf;
           JavaTimPonuda jat = new JavaTimPonuda();  // ConcreteBuilder1
           uf = new UpravaFakulteta(jat);
           uf.Konstruisi();
           System.out.println("Ponuda java tima: " + jat.vratiPonudu());
           VBTimPonuda vbt = new VBTimPonuda(); // ConcreteBuilder2
           uf = new UpravaFakulteta(vbt);
           uf.Konstruisi();
           System.out.println("Ponuda VB tima: " + vbt.vratiPonudu());
        }
}

// Улога: Специфицира апстрактну класу  за креирање понуде.
abstract class  SILAB // Builder
{  ProgramskiJezik  pj;
   SUBP subp;
   Ponuda pon;

  abstract void kreirajProgramskiJezik();
  abstract void kreirajSUBP();
  abstract void kreirajPonudu();
  abstract String vratiPonudu();
 }

// Улоге: а) Репрезентује понуду  која се конструише.
class Ponuda {String ponuda;} // Product

//  Улоге: а) Конструише и групише елементе понуде у понуду имплементирајући SILAB интерфејс.
class JavaTimPonuda extends SILAB // ConcreteBuilder1
{
  JavaTimPonuda() {pon = new Ponuda();}
  public void kreirajProgramskiJezik(){pj = new Java();}
  public void kreirajSUBP() { subp = new MySQL();}
  public void kreirajPonudu() { pon.ponuda = "Programski jezik-" + pj.vratiProgramskiJezik() + "
                                SUBP-" + subp.vratiSUBP();}
  public String vratiPonudu(){return pon.ponuda;}
}

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

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

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

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

interface SUBP {String vratiSUBP();}

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

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

Дијаграм класа примера PBU1



Веза Builder патерна и општег облика патерна

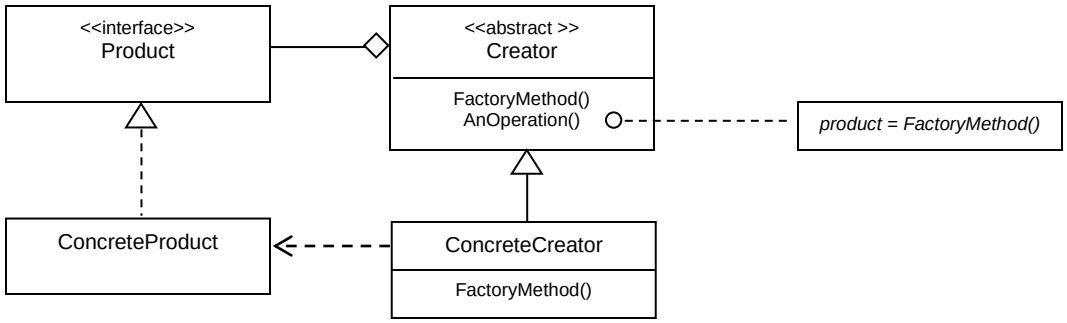
Код Builder патерна постоји једна **СРП**: (*Director, Builder, ConcreteBuilder*).

ПК3: Factory method патерн

Дефиниција
Дефинише интерфејс за креирање објекта (производа), али преноси на подкласе одлуку коју ће класу истанцирати[42]. Factory method преноси надлежност инстанцирања са класе на подкласе.

Појашњење дефиниције
Дефинише интерфејс *(Creator)* за креирање објекта (производа), али преноси на подкласе *(ConcreteCreator)* одлуку коју ће класу (*ConcreteProduct*) истанцирати. Factory method преноси надлежност инстанцирања са класе *(Creator)* на подкласе*(ConcreteCreator)*[43].

Структура Factory method патерна[44]



Учесници

- **Creator**
  Дефинише интерфејс[45] за креирање производа.
- **ConcreteCreator**
  Креира производ (*ConcreteProduct*) помоћу *FactoryMethod* методе.
- **Product**
  Дефинише интерфејс за производе који ће бити креирани.
- **ConcreteProduct**
  Дефинише производ који ће бити креиран и имплементира *Product* интерфејс.

Пример Factory method патерна

Кориснички захтев PFM1: *Управа Факултета је послала захтев Лабораторији за софтверско инжењерство да направи (састави) 2 понуде за израду софтверског система последипломских студија ФОН-а. У понуди треба се наведе:*
а) Програмски језик у коме ће се развијати програм.
б) Систем за управљање базом података у коме ће се чувати подаци.

---

[42] Инстанцирање је креирање објекта (инстанце) неке класе. Класа се инстанцира и као резултат инстанцирања се добија објекат (инстанца,појављивање) те класе.

[43] *ConcreteCreator* класа инстанцира класу *ConcreteProduct*, односно креира објекат класе *ConcreteProduct.* Када се каже да неко инстанцира класу, мисли се да неко креира објекат (инстанцу) те класе.

[44] У наведеној структури је уведена веза агрегације између интерфејса *Creator* и *Product* (која не постоји на оригиналној слици у књизи Design Patterns[GOF]), јер је више него очигледно да класа *Creator* као резултат операције *AnOperation()* добија објекат неке од класа (*ConcreteProduct*) које имплементирају интерфејс *Product.* Наведеном везом добија се структура Factory method патерна која нам говори да различите *ConcreteCreator* класе могу да инстанцирају различите *ConcreteProduct* класе.

[45] У наведеном случају, када се каже да *Creator* дефинише интерфејс за креирање производа, мисли се на интерфејс у ширем смислу, при чему је *Creator* представљен као апстрактна класа.

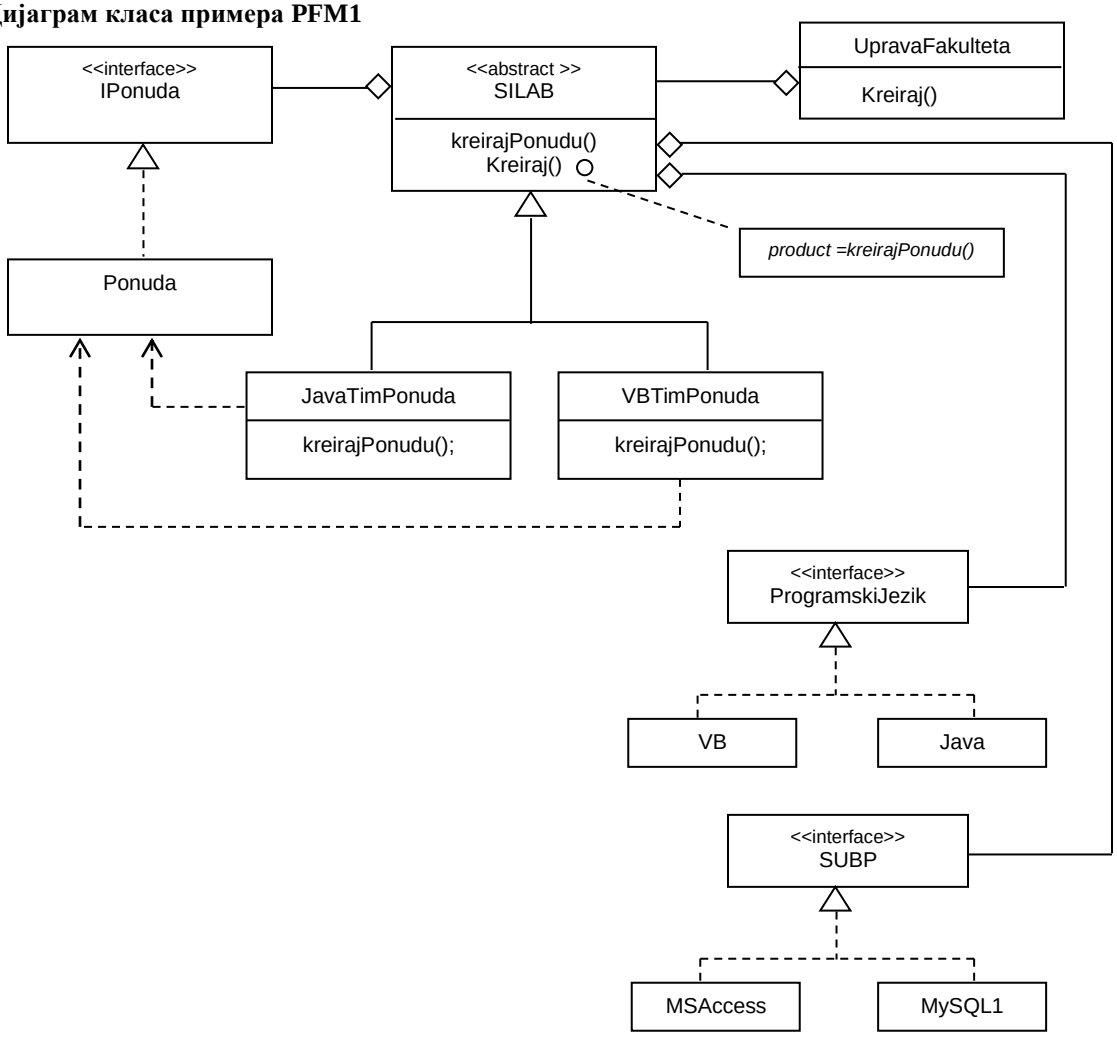
Управа Факултета неће надзирати (контролисати) израду понуда.Надзор израде понуде је пренет на Лабораторију за софтверско инзењерство.

```
class UpravaFakulteta   // Client
{
  SILAB sil; // Creator
  UpravaFakulteta(SILAB sil1){sil = sil1;}

   void  Kreiraj() { sil.Kreiraj();}[46]

   public static void main(String args[])
            {  UpravaFakulteta uf;

               JavaTimPonuda jat = new JavaTimPonuda();  // ConcreteCreator1
               uf = new UpravaFakulteta(jat);
               uf.kreiraj();
               System.out.println("Ponuda java tima: " + jat.vratiPonudu());

               VBTimPonuda vbt = new VBTimPonuda(); // ConcreteCreator2
               uf = new UpravaFakulteta(vbt);
               uf.kreiraj();
              System.out.println("Ponuda VB tima: " + vbt.vratiPonudu());
          }
}

/*Улога: Дефинише интерфејс за Ponuda објекте који ће бити креирани*/
interface IPonuda {String vratiPonudu();} // Product


/*Улога: Дефинише Ponuda објекат  који ће бити креиран. Имплементира IPonuda интерфејс. /*
class Ponuda implements IPonuda // ConcreteProduct
{ String ponuda;
  public String vratiPonudu() {return ponuda;}
}

/* Улоге: Декларише интерфејс за креирање Ponuda објекта. */
abstract class SILAB // Creator
{   ProgramskiJezik  pj;
    SUBP subp;
    IPonuda pon;
    void Kreiraj(){pon = kreirajPonudu();}
    abstract IPonuda kreirajPonudu();
    public String vratiPonudu(){ return pon.vratiPonudu();}
}

/* Улоге: Креира производ  помоћу  KreirajPonudu() методе.*/
class JavaTimPonuda extends SILAB // ConcreteCreator1
{
  public IPonuda kreirajPonudu()
        {  Ponuda pon = new Ponuda();
           pj = new Java();
           subp = new MySQL();
           pon.ponuda = "Programski jezik-" + pj.vratiProgramskiJezik() + "  SUBP-" + subp.vratiSUBP();
           return pon;
        }
}
class VBTimPonuda extends SILAB // ConcreteCreator2
{
  public IPonuda kreirajPonudu()
         {  Ponuda pon = new Ponuda();
         pj = new VB();
         subp = new MSAccess();
         pon.ponuda = "Programski jezik-" + pj.vratiProgramskiJezik() + "  SUBP-" +  subp.vratiSUBP();
         return pon;
        }
}
```

[46] 1. Тимови су одговорни за контролу креирања понуде. 2. Тимови су одговорни за креирање елемената понуде. 3. Тимови су одговорни за састављање понуде у целини.

```
// Наведени интерфејси и класе су преузети из примера за Abstract Factory патерн.
// ***********************************************************
interface ProgramskiJezik  {String vratiProgramskiJezik();}
class Java implements ProgramskiJezik { public String vratiProgramskiJezik(){return "Java";}}
class VB implements ProgramskiJezik { public String vratiProgramskiJezik(){return "VB";}}
interface SUBP {String vratiSUBP();}
class MySQL implements SUBP { public String vratiSUBP(){return "MySQL";}}
class MSAccess implements SUBP  { public String vratiSUBP(){return "MS Access";}}
```

Дијаграм класа примера PFM1



Веза Factory method патерна и општег облика патерна

Код Factory method патерна постоји једна **СРП**: (*Creator*, *Product*, *ConcreteProduct*).

Компаративна анализа AbstractFactory, Builder и FactoryMethod патерна

У примеру који је дат, Управа Факултета код:

- **AbstractFactory патерна** шаље захтев Лабораторији за софтверско инжењерство да направи и пошаље елементе понуде. Управа Факултета преузима на себе обавезу да контролише израду понуде и да састави понуду у целини.
- **Builder патерн** шаље захтев Лабораторији за софтверско инжењерство да направи елементе понуде и понуду у целини. Управа Факултета преузима на себе обавезу да надзире (контролише) израду понуда.
- **Factory method патерн** шаље захтев Лабораторији за софтверско инжењерство да направи елементе понуде и понуду у целини и да надзире процес прављења понуде. То значи да Управа Факултета у потпуности преноси одговорност надзора и прављења понуде на Лабораторију за софтверско инжењерство.

На основу наведене анализе може да се закључи да код:

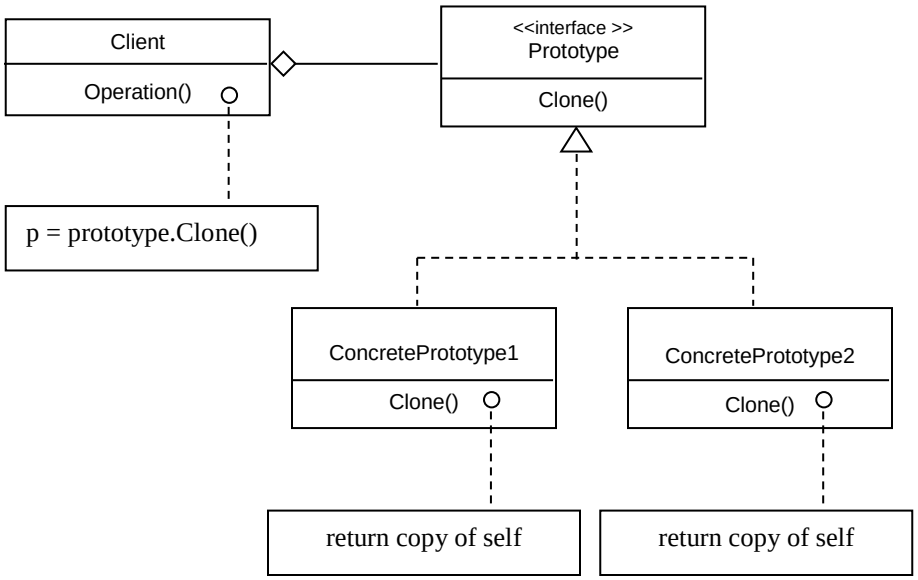
- **AbstractFactory патерна - Client** надзире процес прављења сложеног производа и израђује сложени производ. Израда производа (који су делови сложеног производа) се преносе на **ConcreteFactory** класе.
- **Builder патерн – Direktor** надзире процес прављења сложеног производа. Израда сложеног производа и његових делова се преноси на **ConcreteBuilder** класе.
- **Factory method патерн – ConcreteCreator** надзире процес прављења сложеног производа и израђује сложени производ и његове делове.

ПК4: Prototype патерна

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

Појашњење дефиниције
Одређује (специфицира) врсте објеката (*ConcretePrototype1*, *ConcretePrototype2)* које ће бити креиране коришћењем прототипског појављивања (*prototype*) и креира нове објекте (*p*) копирањем тог прототипа (*prototype.Clone()*).

Структура Prototype патерна



Учесници

- **Prototype**
  Декларише интерфејс за сопствено клонирање.
- **ConcretePrototype**
  Имплементира операцију за сопствено клонирање.
- **Client**
  Захтева од прототипа да се клонира.

Пример Prototype патерна[47]

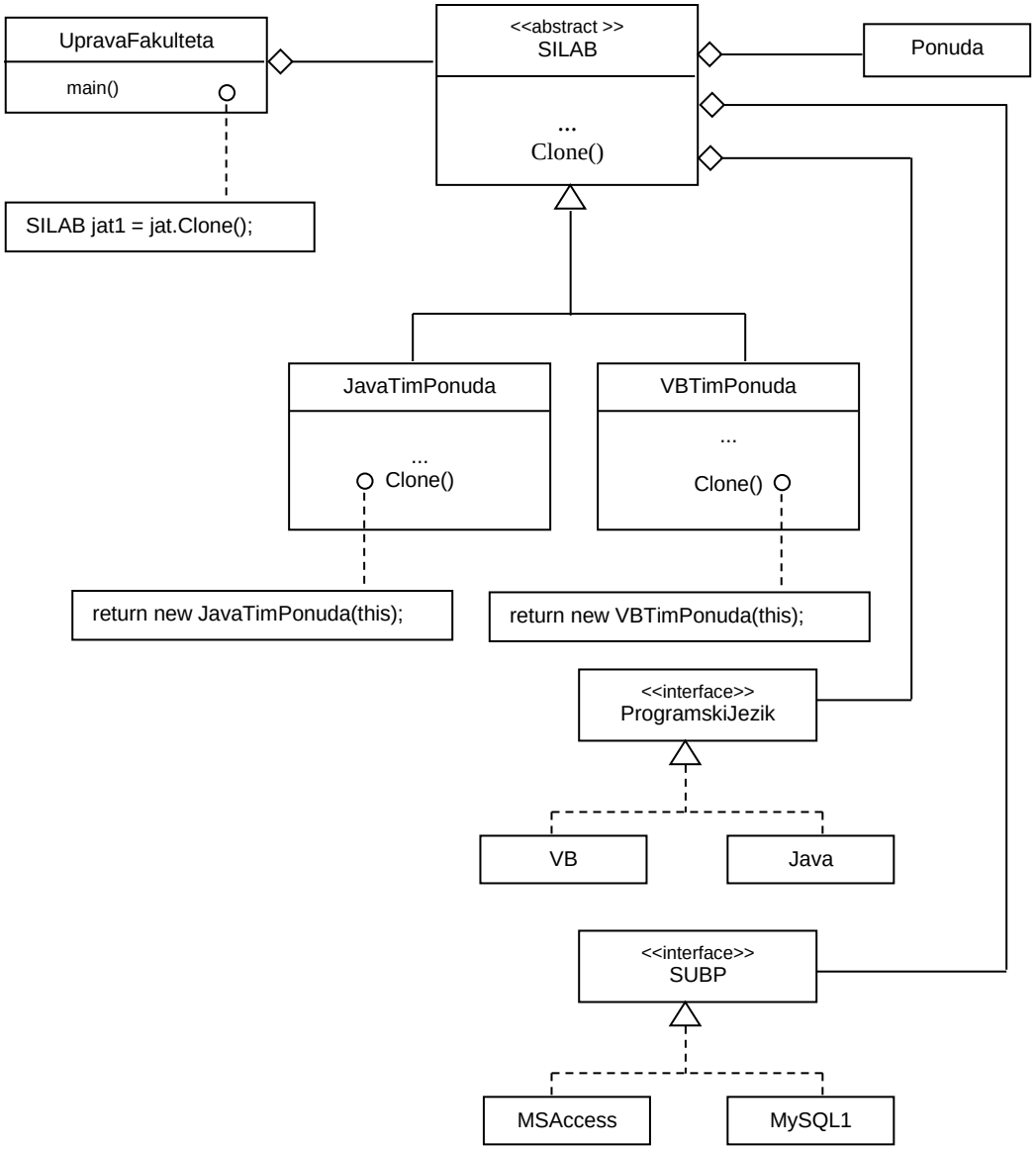
Кориснички захтев PPR1: *Управа Факултета након пријема понуда од Јава и VB тима треба да направи копију сваке од понуда.*

```
// Улога: Захтева од прототипа да се клонира.
class UpravaFakulteta   // Client
{ SILAB sil; // Prototype
  UpravaFakulteta(SILAB sil1){sil = sil1;}

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

[47] Наведени пример представља проширење примера који је урађен код Builder патерна. У овом примеру делови кода који су додати у односу на пример код Builder патерна су форматирани као italic и bold.

```
  public static void main(String args[])
            {  UpravaFakulteta uf;
               JavaTimPonuda jat = new JavaTimPonuda();  // ConcreteBuilder1
               uf = new UpravaFakulteta(jat);
               uf.Konstruisi();
               System.out.println("Ponuda java tima: " + jat.vratiPonudu());
               SILAB jat1 = jat.Clone(); // jat je prototip koji se klonira..
               System.out.println("Ponuda java tima-kopija: " + jat1.vratiPonudu());

               VBTimPonuda vbt = new VBTimPonuda(); // ConcreteBuilder2
                uf = new UpravaFakulteta(vbt);
                uf.Konstruisi();
                System.out.println("Ponuda VB tima: " + vbt.vratiPonudu());
                SILAB vbt1 = vbt.Clone(); //vbt je prototip koji se klonira.
                System.out.println("Ponuda VB tima-kopija: " + vbt1.vratiPonudu());
             }
}

// Улога: Декларише интерфејс (Clone методу) за сопствено клонирање.
abstract class  SILAB // Prototype
{  ProgramskiJezik  pj;
    SUBP subp;
    Ponuda pon;

   abstract void kreirajProgramskiJezik();
   abstract void kreirajSUBP();
   abstract void kreirajPonudu();
   abstract String vratiPonudu();
   abstract public SILAB Clone();
 }

class Ponuda {String ponuda;}

// Улога: Имплементира операцију Clone за сопствено клонирање.
class JavaTimPonuda extends SILAB // ConcretePrototype1
{ JavaTimPonuda() {pon = new Ponuda();}
  JavaTimPonuda(JavaTimPonuda jtp) {pon = new Ponuda(); pon.ponuda = new String(jtp.pon.ponuda);}
  public void kreirajProgramskiJezik(){pj = new Java();}
  public void kreirajSUBP() { subp = new MySQL();}
  public void kreirajPonudu() { pon.ponuda = "Programski jezik-" + pj.vratiProgramskiJezik() + " SUBP-" +
                                subp.vratiSUBP();}
  public String vratiPonudu(){return pon.ponuda;}
  public SILAB Clone(){return new JavaTimPonuda(this);}
}

class VBTimPonuda extends SILAB // ConcretePrototype1
{ VBTimPonuda(){pon = new Ponuda();}
  VBTimPonuda(VBTimPonuda vbtp) {pon = new Ponuda(); pon.ponuda = new String(vbtp.pon.ponuda);}
  public void kreirajProgramskiJezik(){pj = new VB();}
  public void kreirajSUBP() {subp = new MSAccess();}
  public void kreirajPonudu() { pon.ponuda = "Programski jezik-" + pj.vratiProgramskiJezik() + " SUBP-" +
                                subp.vratiSUBP();}
  public String vratiPonudu(){return pon.ponuda;}
  public SILAB Clone(){return new VBTimPonuda(this);}
}


interface ProgramskiJezik  {String vratiProgramskiJezik();}

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

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

interface SUBP {String vratiSUBP();}

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

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

Дијаграм класа примера PPR1



Веза Prototype патерна и општег облика патерна

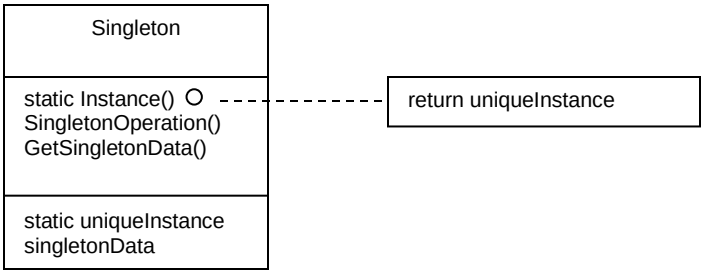
Код *Prototype* патерна постоји једна **СРП**: (*Client, Prototype, ConcretePrototype*).

ПК5: Singleton патерн

Дефиниција

Обезбеђује класи само једно појављивање и глобални приступ до ње.

Структура Singleton патерна



Учесници

- **Singleton** – дефинише **Instance()** операцију која омогућава клијентима приступ до њеног јединственог појављивања.

Пример Syngleton патерна[48]

Кориснички захтев PSI1*: Онемогућити да се добију две понуде од Јава тима.*

```
class UpravaFakulteta
{
 SILAB sil; // Builder
 UpravaFakulteta(SILAB sil1){sil = sil1;}

 void Konstruisi()
  { sil.kreirajProgramskiJezik();
    sil.kreirajSUBP();
    sil.kreirajPonudu();}
    public static void main(String args[])
      {  UpravaFakulteta uf;
        JavaTimPonuda jat = JavaTimPonudaSingleton.Instance();// Јавиће поруку да се  креира нова понуда.
         uf = new UpravaFakulteta(jat);
         uf.Konstruisi();
         System.out.println("Ponuda java tima: " + jat.vratiPonudu());

        JavaTimPonuda jat1 = JavaTimPonudaSingleton.Instance();  // Јавиће поруку да је понуда већ креирана.
     }
}

abstract class  SILAB
{  ProgramskiJezik  pj;
   SUBP subp;
   Ponuda pon;

  abstract void kreirajProgramskiJezik();
  abstract void kreirajSUBP();
  abstract void kreirajPonudu();
  abstract String vratiPonudu();
 }

class Ponuda {String ponuda;}
```

[48] Наведени пример представља мало измењен пример који је урађен код Builder патерна. У овом примеру делови кода који су промењени у односу на пример код Builder патерна су форматирани као italic и bold.

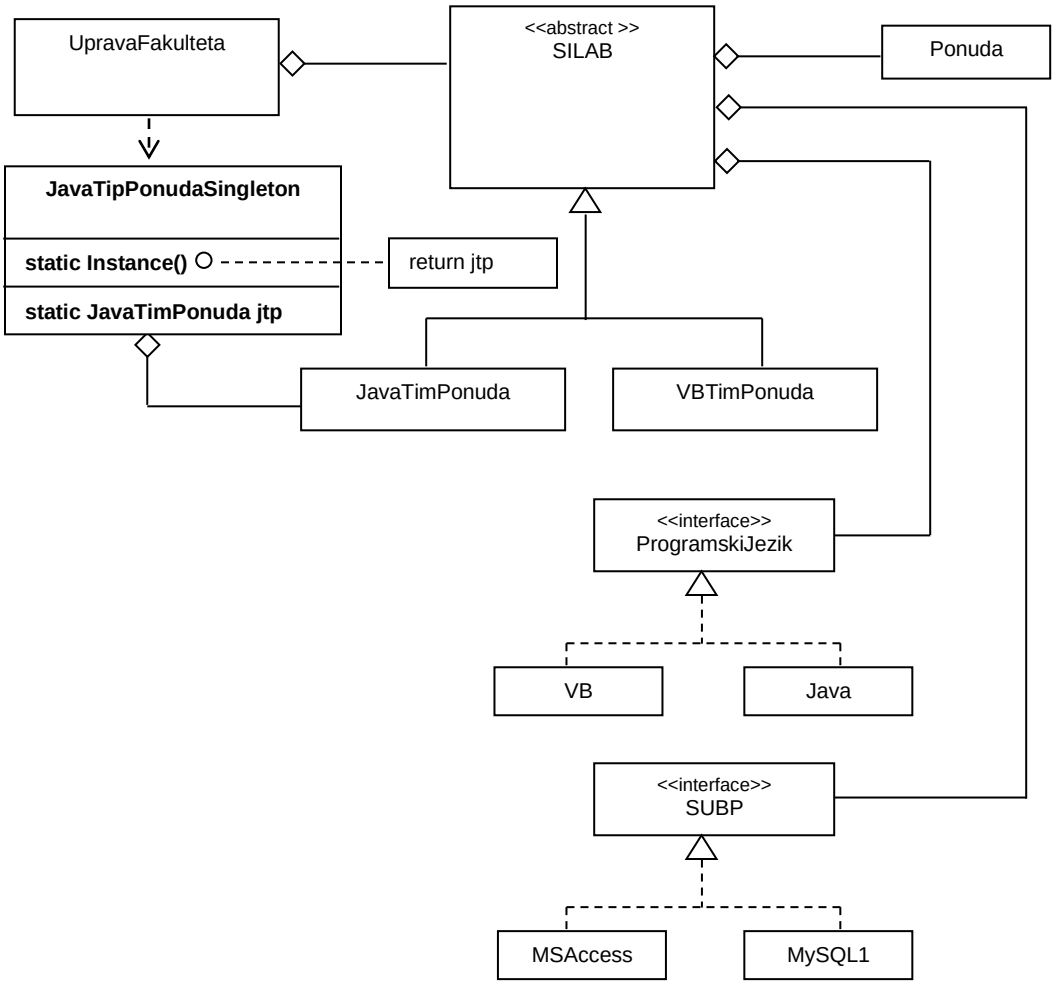
```
class JavaTimPonuda extends SILAB
{
  JavaTimPonuda() {pon = new Ponuda();}
  public void kreirajProgramskiJezik(){pj = new Java();}
  public void kreirajSUBP() { subp = new MySQL();}
  public void kreirajPonudu() { pon.ponuda = "Programski jezik-" + pj.vratiProgramskiJezik() + "
                                SUBP-" + subp.vratiSUBP();}
  public String vratiPonudu(){return pon.ponuda;}
}
```

// Улога: Дефинише Instance() операцију која омогућава клијентима приступ до
// њеног јединственог појављивања (jtp).

```
class JavaTimPonudaSingleton
{ static boolean jedinstvenoPojavljivanje=false;
  static JavaTimPonuda jtp;
  static JavaTimPonuda Instance()
       {    if (jedinstvenoPojavljivanje==false)
               { System.out.println("Kreira se nova ponuda");
                 jedinstvenoPojavljivanje=true;
                 jtp = new JavaTimPonuda();
                 }
               else
                 { System.out.println("Ponuda je vec kreirana"); }
               return jtp;
       }
}
```

```
class VBTimPonuda extends SILAB  {
  VBTimPonuda(){pon = new Ponuda();}
  public void kreirajProgramskiJezik(){pj = new VB();}
  public void kreirajSUBP() {subp = new MSAccess();}
  public void kreirajPonudu() { pon.ponuda = "Programski jezik-" + pj.vratiProgramskiJezik() + "
                                            SUBP-" + subp.vratiSUBP();}
  public String vratiPonudu(){return pon.ponuda;}
}

interface ProgramskiJezik  {String vratiProgramskiJezik();}

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

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

interface SUBP {String vratiSUBP();}

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

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

Дијаграм класа примера PSI1



Веза Singleton патерна и општег облика патерна

Код *Singleton* патерна не постоји ни једна **СРП**.

СП: Структурни патерни

Структурни патерни *описују сложене структуре међусобно повезаних класа и објеката.*

Постоје следећи структурни патерни:

1. Adapter патерн - Конвертује интерфејс неке класе у други интерфејс који клијент очекује. Адаптер патерн омогућава заједнички рад класа које имају некомбатибилне 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 патерн омогућава да се једноставни и сложени објекти третирају јединствено.

4. Decorator патерн - Придружује додатне одговорности (функционалности) до објекта динамички. Decorator патерн обезбеђује флексибилност у избору подкласа које проширују функционалност.

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

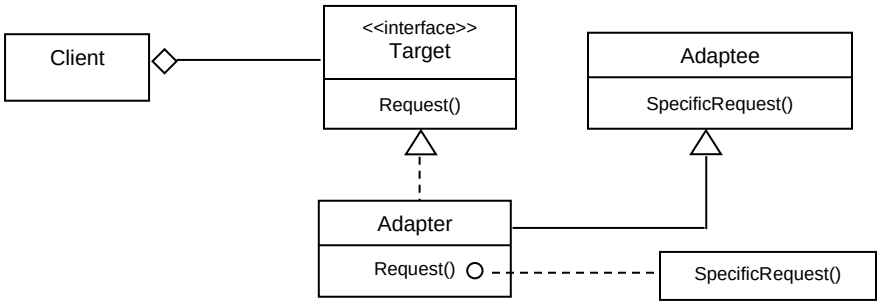
6. Flyweight патерн - Користи дељење да ефикасно подржи велики број ситних објеката.

7. Proxy патерн - Обезбеђује посредника за приступање другом објекту како би се омогућио контролисани приступ до њега.

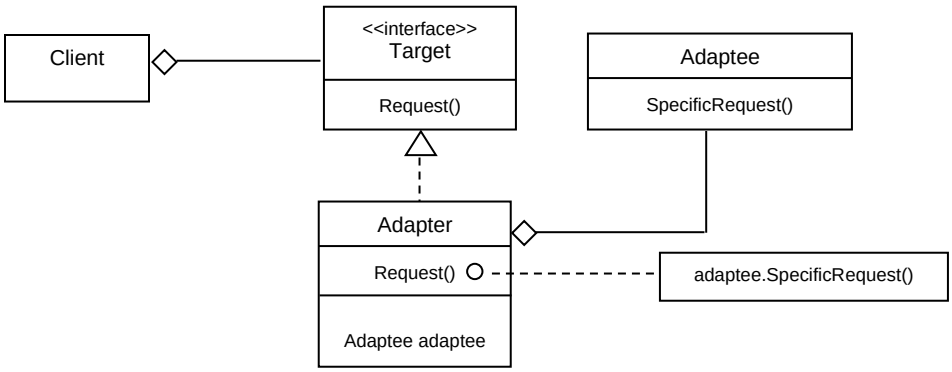
СП1: Adapter патерн
Дефиниција
Конвертује интерфејс неке класе у други интерфејс који клијент очекује. Адаптер патерн омогућава заједнички рад класа које имају некомбатибилне интерфејсе.

Појашњење дефиниције: Конвертује интерфејс неке класе (Adaptee) у други интерфејс (Target) који клијент (Client) очекује. Адаптер патерн омогућава заједнички рад класа (Adaptee, Target) које имају некомбатибилне интерфејсе .

Структура Adapter патерна се може јавити у два облика:
а) Класа *Adapter* користи вишеструко наслеђивање код прилагођавања некомпатибилних интерфејса.



б) Класа Adapter користи композицију код прилагођавања некомпатибилних интерфејса.



Учесници

- **Client**
  Сарађује са интерфејсом *Adaptee* преко интерфејса *Target*.
- **Target**
  Дефинише доменски-специфичан интерфејс који класа *Client* користи.
- **Adapter**
  Адаптира (прилагођава) интерфејс *Adaptee* интерфејсу *Target*.
- **Adaptee**
  Дефинише постојећи интерфејс који треба адаптирати.

Пример Adapter paterna[49]

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.*

[49] Наведени пример представља проширење примера који је урађен код Builder патерна.

```
//Улога: Дефинише доменски-специфичан интерфејс који класа UpravaFakulteta користи.
interface SILABTarget    // Target
{ void KrProgramskiJezik();
  void KrSUBP();
  void KrPonudu();
  String VrPonudu();
}

//Улога: Адаптира (прилагођава) интерфејс SILAB интерфејсу SilabTarget.
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: Сарађује са интерфејсом SILAB преко интерфејса SILABTarget.
class UpravaFakulteta   // Client
{
  SILABTarget silta;

  UpravaFakulteta(SILABTarget silta1){silta = silta1;}

  // Контролише конструкцију понуде коришћењем интерфејса 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 постојећи интерфејс који треба адаптирати.
abstract class  SILAB // Adaptee
{  ProgramskiJezik  pj;
    SUBP subp;
    Ponuda pon;

   abstract void kreirajProgramskiJezik();
   abstract void kreirajSUBP();
   abstract void kreirajPonudu();
   abstract String vratiPonudu();
 }

class Ponuda {String ponuda;}

class JavaTimPonuda extends SILAB
{ JavaTimPonuda() {pon = new Ponuda();}
  public void kreirajProgramskiJezik(){pj = new Java();}
  public void kreirajSUBP() { subp = new MySQL();}
  public void kreirajPonudu() { pon.ponuda = "Programski jezik-" + pj.vratiProgramskiJezik() + " SUBP-" +
                                  subp.vratiSUBP();}
  public String vratiPonudu(){return pon.ponuda;}
}
```

```
class VBTimPonuda extends SILAB
{
  VBTimPonuda(){pon = new Ponuda();}
  public void kreirajProgramskiJezik(){pj = new VB();}
  public void kreirajSUBP() {subp = new MSAccess();}
  public void kreirajPonudu() { pon.ponuda = "Programski jezik-" + pj.vratiProgramskiJezik() + " SUBP-" +
                                subp.vratiSUBP();}
  public String vratiPonudu(){return pon.ponuda;}
}
```

```
interface ProgramskiJezik  {String vratiProgramskiJezik();}
```

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

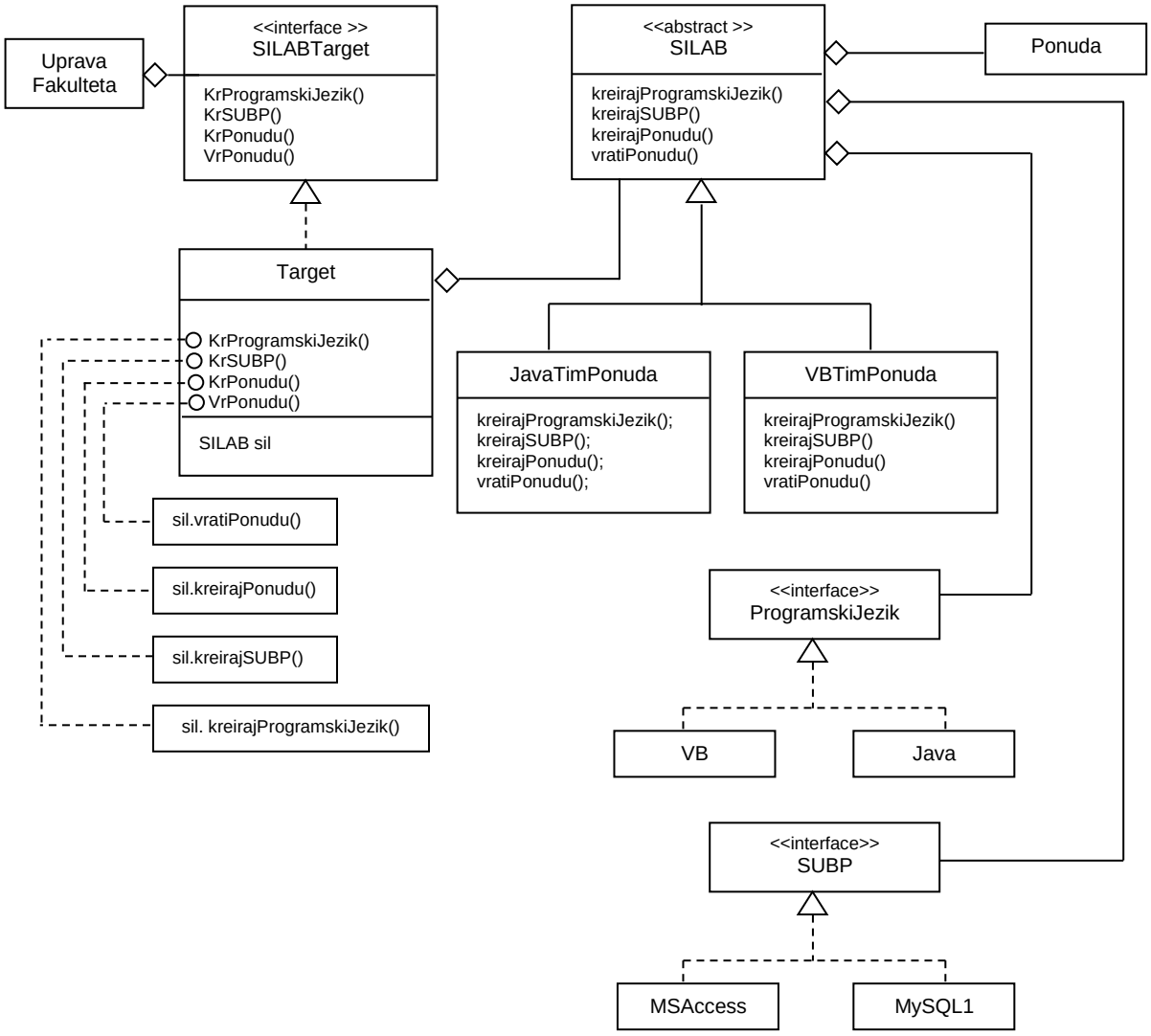
```
class VB implements ProgramskiJezik { public String vratiProgramskiJezik(){return "VB";}}
```

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

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

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

Дијаграм класа примера PAD1:



Објашњење примера:

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

```
abstract class  SILAB // Adaptee
{  ProgramskiJezik  pj;
    SUBP subp;
    Ponuda pon;
   abstract void kreirajProgramskiJezik();
   abstract void kreirajSUBP();
   abstract void kreirajPonudu();
   abstract String vratiPonudu();
 }
```

интерфејсу *SILABTarger*:

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

То се постиже помоћу класе *Adapter*:

```
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 преко интерфејса *SilabTarget*.

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

Клијент је поставио захтев да се промени интерфејс. Треба да се постигну два циља помоћу *Adapter* патерна:

а) Да се прилагоди постојећи интерфејс интерфејсу који клијент очекује.

б) Не треба да се мења 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 операција.

Када се повезују објекти наведеног патерна то се ради на следећи начин: Прво се креира објекат који од никога не зависе (**jat**). Затим се креира адаптер објекат (**sil**), који се преко конструктора повезује са објектом који се адаптира (**jat**). На крају се креира клијентски објекат (**uf**), који се преко конструктора повезује са адаптер објектом (**sil**).

Важно правило: *Уколико имамо два класе, при чему нека класа X користи класу Y (класа Y је независна у односу на класу X), за класу X можемо да кажемо да је клијент класа док је класа Y сервер класа. При креирању клијентске и серверске класе, увек се прво формира серверска па тек онда клијентска класа.*

```
class X // Клијент класа
{Y y;

 X(Y y1){y=y1;}
}
class Y // Сервер класа
  {…}

public static void main(String args[])
   { Y y = new Y();
     X x = new X(y);     …
   }
```

Веза Адаптер патерна и општег облика патерна

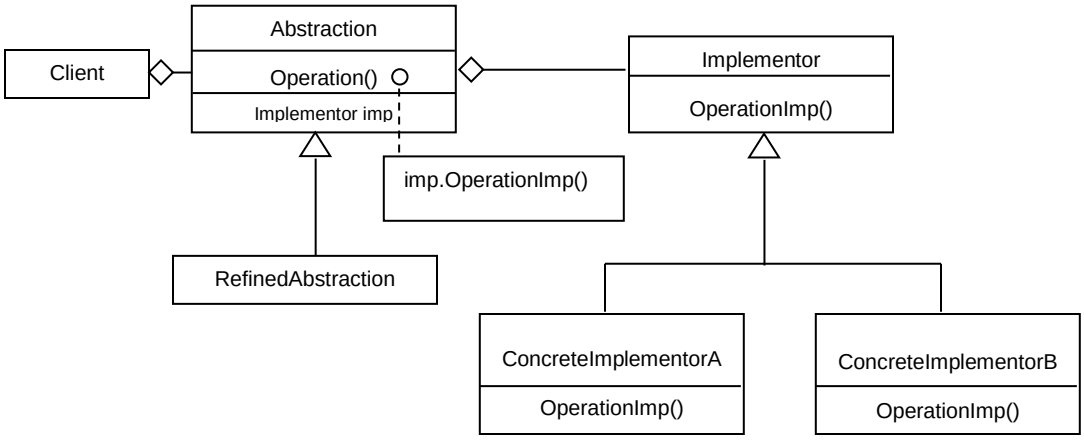
Код *Adapter* патерна постоји једна **СРП**: (*Client, Target,Adapter*).

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.

Структура Bridge патерна



Учесници:

- **Abstraction**
  Дефинише интерфејс апстракције.Чува референцу на објекат типа *Implementor*.
- **RefinedAbstraction**
  Проширује интерфејс *Abstraction*.
- **Implementor**
  Дефинише интерфејс за имплементационе класе (*ConcreteImplementorA, ConcreteImplementorB*). Овај интерфејс не мора да одговара интерфејсу *Abstraction* и они могу бити веома различит. Обично *Implementor* интерфејс обезбеђује само примитивне операције док интефејс *Abstraction* дефинише операције високог нивоа које су засноване на наведеним примитивним операцијама.
- **ConcreteImplementor**
  Имплементира интерфејс *Implementor*.

Пример Bridge узора

Кориснички захтев PBR1[50]: *Управа Факултета је тражила од тимова да промене формат Понуде тако што ће се прво навести СУБП па тек онда програмски језик.*
Java тим у понуди треба да наведе ко су аутори Понуде. Управа Факултета је поставила захтев да и стари формат понуде остану расположив.

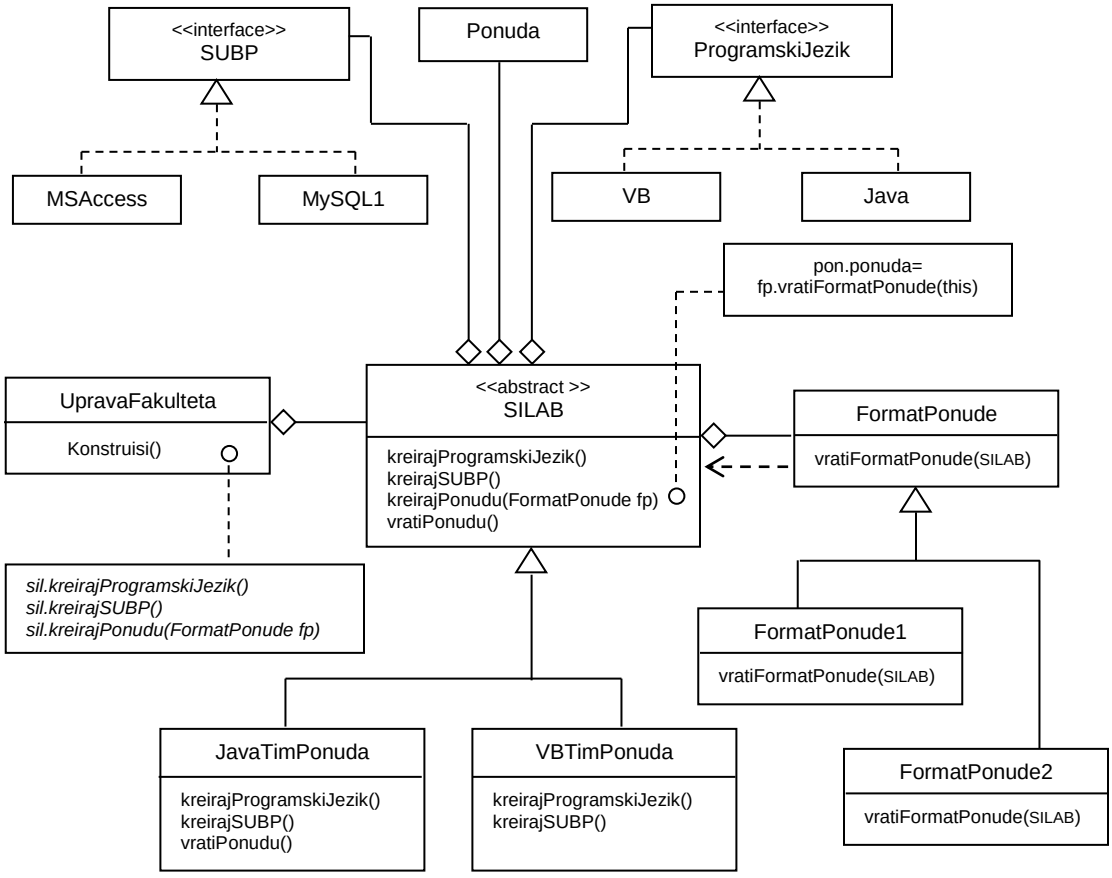
```
class UpravaFakulteta   // Client
{
  SILAB sil;
  UpravaFakulteta(SILAB sil1){sil= sil1;}
  void Konstruisi(FormatPonude fp)
    { sil.kreirajProgramskiJezik();
      sil.kreirajSUBP();
      sil.kreirajPonudu(fp);
    }
```

[50] Наведени пример представља проширење примера који је урађен код Builder патерна.

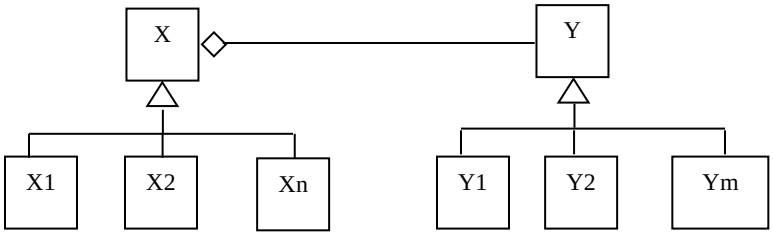
```
  public static void main(String args[])
            {
              UpravaFakulteta uf;
              FormatPonude fp = null;
              if (args[0].equals("1")) fp = new FormatPonude1();
              if (args[0].equals("2")) fp = new FormatPonude2();

              JavaTimPonuda jat = new JavaTimPonuda();
              uf = new UpravaFakulteta(jat);
              uf.Konstruisi(fp);
              System.out.println("Ponuda java tima: " + jat.vratiPonudu());

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

// Улога: Дефинише интерфејс апстракције.
abstract class  SILAB  // Abstracion
{  ProgramskiJezik  pj;
   SUBP subp;
   Ponuda pon;

   abstract void kreirajProgramskiJezik();
   abstract void kreirajSUBP();

   public String vratiPonudu(){return pon.ponuda;}
   public void kreirajPonudu(FormatPonude fp) { pon.ponuda = fp.vratiFormatPonude(this);}
}

class JavaTimPonuda extends SILAB // RefinedAbstraction1
{ JavaTimPonuda() {pon = new Ponuda();}
  public void kreirajProgramskiJezik(){pj = new Java();}
  public void kreirajSUBP() { subp = new MySQL();}
  public String vratiPonudu(){return "Autor: Lab za soft. inzenjerstvo: " + pon.ponuda;}
}

class VBTimPonuda extends SILAB // RefinedAbstraction2
{ VBTimPonuda(){pon = new Ponuda();}
  public void kreirajProgramskiJezik(){pj = new VB();}
  public void kreirajSUBP() {subp = new MSAccess();}
}

class Ponuda {String ponuda;}

// Улога: Дефинише интерфејс за имплементационе класе  (FormatPonude1, FormatPonude2).
abstract class FormatPonude   // Implementor
{  abstract String vratiFormatPonude(SILAB sil);
}

// Улога: Имплементира интерфејсFormatPonude.
class FormatPonude1 extends FormatPonude   // Concrete Implementor A
{  String vratiFormatPonude(SILAB sil)
     {  return  "Programski jezik-" + sil.pj.vratiProgramskiJezik() + "  SUBP-" + sil.subp.vratiSUBP();}
}

class FormatPonude2 extends FormatPonude // Concrete Implementor B
{
String vratiFormatPonude(SILAB sil)
   { return  "SUBP-" + sil.subp.vratiSUBP() + " Programski jezik-" + sil.pj.vratiProgramskiJezik();}
}

interface ProgramskiJezik {String vratiProgramskiJezik();}

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

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

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

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

Дијаграм класа примера PBR1:



У наведеном примеру је омогућено да додавање нових тимова Лабораторије за софтверско инжењерство (нпр. *CTimPonuda*) и нових формата понуде (нпр. *FormatPonude3*) буде независно. То значи да било који тим Лабораторије за софтверско инжењерство и било који формат понуде могу бити повезани. На основу наведеног изводи се следећи закључак:
Уколико имамо класе или интерфејсе X и Y, где X садржи Y и где су из X изведене класе X1,X2,...,Xn, док су из класе Y изведене класе Y1,Y2,...,Ym, могуће је да се повеже било која класа која је изведена из X са класом која је изведена из Y.



То значи да било које Xi, i = (1,..,n) може бити повезано са било које Yj, j = (1,..,m).

Веза Bridge патерна и општег облика патерна

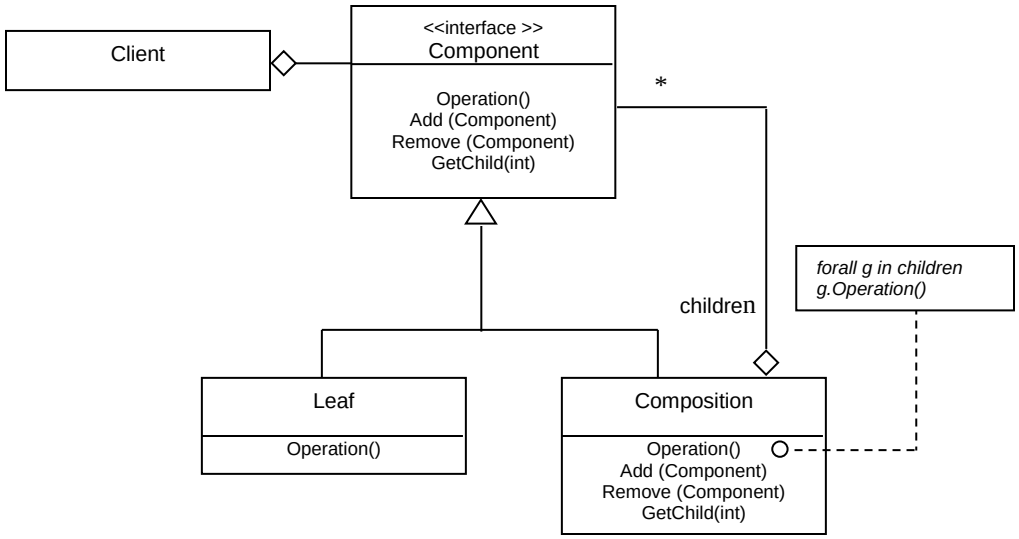
Код *Bridge* патерна постоје две **СРП**: (*Client, Abstraction,RefinedAbstraction*) и (*Abstraction, Implementor,ConcreteImplementor*).

СП3. Composite патерн

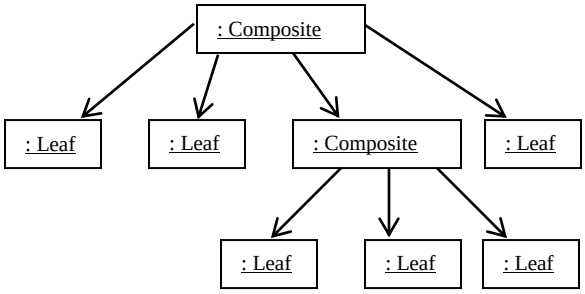
Дефиниција
Објекти се састављају (компонују) у структуру стабла како би представили хијерархију целине и делова. *Composite* патерн омогућава да се једноставни и сложени објекти третирају јединствено.

Појашњење дефиниције
Објекти се састављају (компонују) у структуру стабла како би представили хијерархију целине и делова. *Composite* патерн омогућава да се једноставни (*Leaf*) и сложени (*Composite*) објекти третирају јединствено. Једноставни и сложени објекти су компоненте (*Component*).

Структура Composite патерна



Типична структура *Composite* објекта има следећи изглед:



Учесници

- **Client**
  Манипулише објектима (компонентама) у структури помоћу *Component* интерфејса.
- **Component**
  Декларише интерфејс за објекте који ће да образују структуру. Декларише интерфејс за приступање и управљање објектима структуре.
- **Leaf**
  Представља просте објекте (*Leaf*) у структури и дефинише њихово понашање. Прости објекти немају децу-објекте.
- **Composite**
  Дефинише понашање за сложене објекте (*Composition*) који имају децу-објекте. Чува децу-објекте. Имплементира операције интерфејса *Component.*

Пример Composite патерна

Кориснички захтев PCO1[51]: *Управа Факултета је тражила да тим који је у понуди навео Јаву као програмски језик детаљније образложи неке од најважнијих Јава технологија које ће се користити за израду софтверског система ПДС-а на ФОН-у, ту се посебно мисли на J2EE технологију.*

```
class UpravaFakulteta
{ SILAB sil;
  UpravaFakulteta(SILAB sil1){sil = sil1;}
  void Konstruisi()
    { sil.kreirajProgramskiJezik();
      sil.kreirajSUBP();
      sil.kreirajPonudu();
    }
   public static void main(String args[])
                {  UpravaFakulteta uf;
                   JavaTimPonuda jat = new JavaTimPonuda();
                   uf = new UpravaFakulteta(jat);
                   uf.Konstruisi();
                  System.out.println("Ponuda java tima: " + jat.vratiPonudu());
                   VBTimPonuda vbt = new VBTimPonuda();
                    uf = new UpravaFakulteta(vbt);
                    uf.Konstruisi();
                    System.out.println("Ponuda VB tima: " + vbt.vratiPonudu());
                }
}
abstract class  SILAB
{  ProgramskiJezik  pj;
    SUBP subp;
    Ponuda pon;
   abstract void kreirajProgramskiJezik();
   abstract void kreirajSUBP();
   abstract void kreirajPonudu();
   abstract String vratiPonudu();
 }
class Ponuda {String ponuda;}
class JavaTimPonuda extends SILAB
{ JavaTimPonuda() {pon = new Ponuda();}
  public void kreirajProgramskiJezik(){pj = new Java();}
  public void kreirajSUBP() { subp = new MySQL();}
  public void kreirajPonudu() { pon.ponuda = "\n Programski jezik:" + pj.vratiProgramskiJezik() + " \n SUBP:" + subp.vratiSUBP();}
  public String vratiPonudu(){return pon.ponuda;}
}
class VBTimPonuda extends SILAB
{ VBTimPonuda(){pon = new Ponuda();}
  public void kreirajProgramskiJezik(){pj = new VB();}
  public void kreirajSUBP() {subp = new MSAccess();}
  public void kreirajPonudu() { pon.ponuda = "\n Programski jezik:" + pj.vratiProgramskiJezik() + "  \n SUBP:" + subp.vratiSUBP();}
  public String vratiPonudu(){return pon.ponuda;}
}
interface ProgramskiJezik  {String vratiProgramskiJezik();}
class VB implements ProgramskiJezik { public String vratiProgramskiJezik(){return "VB";}}
interface SUBP {String vratiSUBP();}
class MySQL implements SUBP { public String vratiSUBP(){return "MySQL";}}
class MSAccess implements SUBP  { public String vratiSUBP(){return "MS Access";}}
// Улога: Манипулише објектима (компонентама) у структури помоћу Komponenta  интерфејса.
class Java implements ProgramskiJezik  // Client
{  Komponenta j;
   static String ponudaJava;
   Java()
    {  ponudaJava="\n";
       j = new JavaPlatforma("JavaPlatforma");
      J2SE j2se = new J2SE("J2SE");
      j.dodajKomponentu(j2se,"JavaPlatforma");
      J2EE j2ee = new J2EE("J2EE");
      j.dodajKomponentu(j2ee,"JavaPlatforma");
      EJB ejb = new EJB("EJB");
      j.dodajKomponentu(ejb,"J2EE");
      Servlet sr = new Servlet("Servlet");
      j.dodajKomponentu(sr,"J2EE");
```

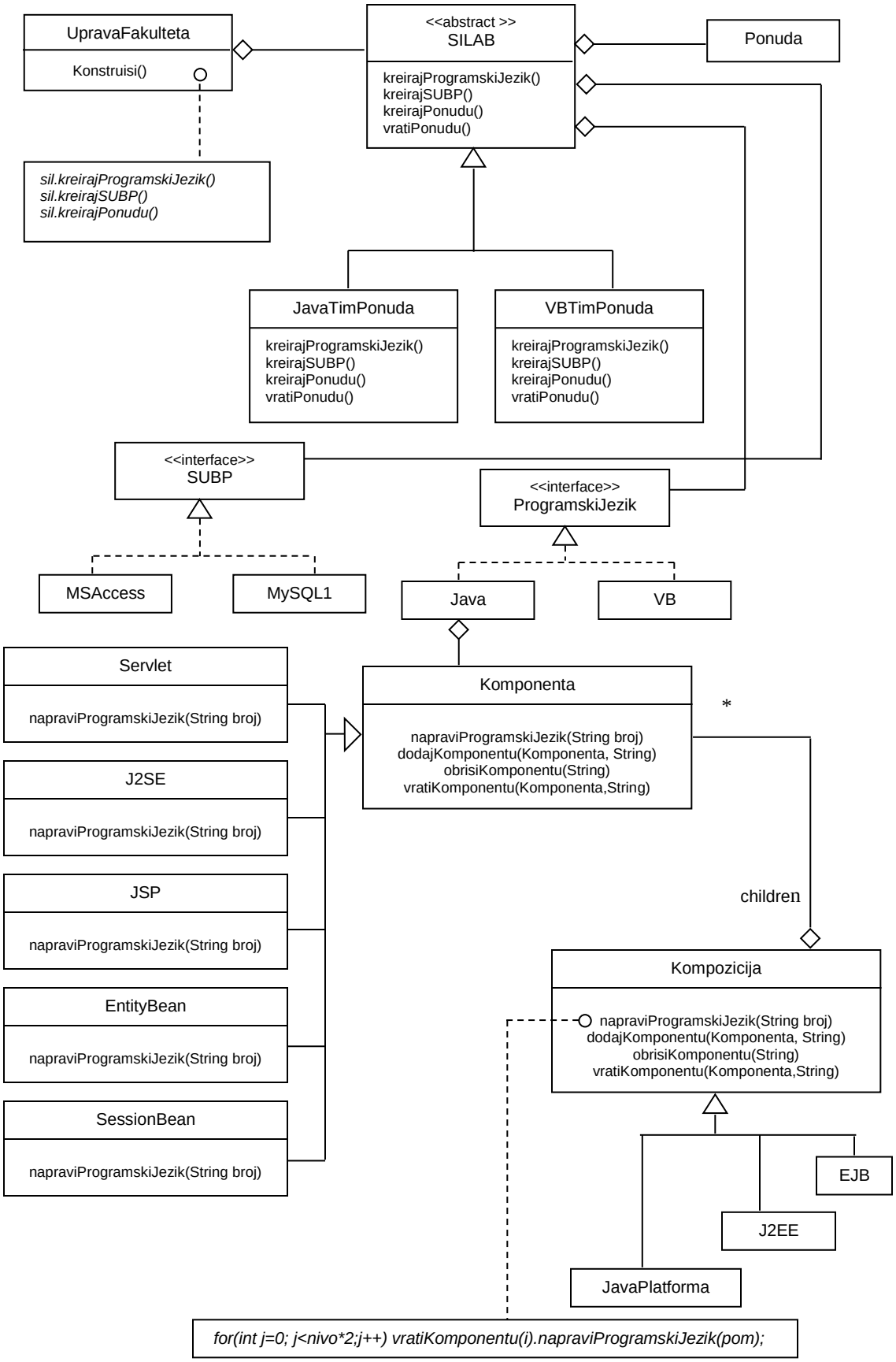
[51] Наведени пример представља проширење примера који је урађен код Builder патерна.

```
    JSP jsp = new JSP("JSP");
     j.dodajKomponentu(jsp,"J2EE");
    EntityBean eb = new EntityBean("EntityBean");
    j.dodajKomponentu(eb,"EJB");
    SessionBean sb = new SessionBean("SessionBean");
    j.dodajKomponentu(sb,"EJB");
  }
  public String vratiProgramskiJezik()
  { j.napraviProgramskiJezik("1");
    return Java.ponudaJava;
  }
}
// Улога: Декларише интерфејс за објекте који ће да образују структуру. Декларише интерфејс за
// приступање и управљање објектима структуре.
class Komponenta  // Component
{ String sifraKomponente;
  static int nivo = 1;
  Komponenta(String sifraKomponente1){sifraKomponente = new String(sifraKomponente1);}
  String vratiSifruKomponente() {return sifraKomponente;}
  Komponenta vratiKomponentu(int i) {return null;}
  Komponenta vratiKomponentu(Komponenta Komponenta,String sifraKomponente1){return null;}
  void napraviProgramskiJezik(String broj) {}
  void dodajKomponentu(int i, Komponenta deteKomponenta) {}
  void dodajKomponentu(Komponenta deteKomponenta, String sifraRoditelja) {}
  void obrisiKomponentu(String sifraKomponente){}
  String opisTehnologije(){return "";}
}
// Улога: Дефинише понашање за сложене објекте који имају децу-објекте. Чува  децу-објекте. Имплементира операције
// интерфејса Komponenta.
class Kompozicija extends Komponenta // Composite
{ Komponenta kom[];
  Kompozicija(String sifraKomponente1){super(sifraKomponente1); kom = new Komponenta[5]; }
  Komponenta vratiKomponentu(int i) {return kom[i];}
  void dodajKomponentu(int i, Komponenta deteKomponenta) {kom[i]=deteKomponenta;}
  public void napraviProgramskiJezik(String broj) // parametar broj oznacava broj koji stoji ispred Java tehnologija
      {   // formatiranje Java ponude
          Java.ponudaJava = Java.ponudaJava + " " + broj + ":" + vratiSifruKomponente() + opisTehnologije();
          //**************************
          for (int i=0; i<5; i++)
           {   if (vratiKomponentu(i) != null)
                 { nivo++;
                   // formatiranje Java ponude
                  Java.ponudaJava = Java.ponudaJava + "\n";
                   for(int j=0; j<nivo*2;j++) Java.ponudaJava = Java.ponudaJava + " " ;
                    String pom = broj + "." + (i+1);
                   //**************************
                   vratiKomponentu(i).napraviProgramskiJezik(pom);
                   nivo--;
                  }
           }
      }
 public void dodajKomponentu(Komponenta deteKomponenta, String sifraRoditelja)
  { int i;
    Komponenta roditeljKomponenta = vratiKomponentu(this,sifraRoditelja);   // ako nema roditelja vraca null
    if (roditeljKomponenta != null)
         { for(i=0; i<5;i++)
              if (roditeljKomponenta.vratiKomponentu(i) == null)
                 { roditeljKomponenta.dodajKomponentu(i, deteKomponenta);
                   break;
                 }
              if (i==5)
                 { System.out.println("Ne moze da se unese dete-komponenta, jer je njena komponenta-roditelj popunila svu
                                       decu-komponente (maksimalno 5 dece):");
                 }
           }
        else
          {  System.out.println("Roditelj sa sifrom:" + sifraRoditelja + " ne postoji: ");
          }
    }
```

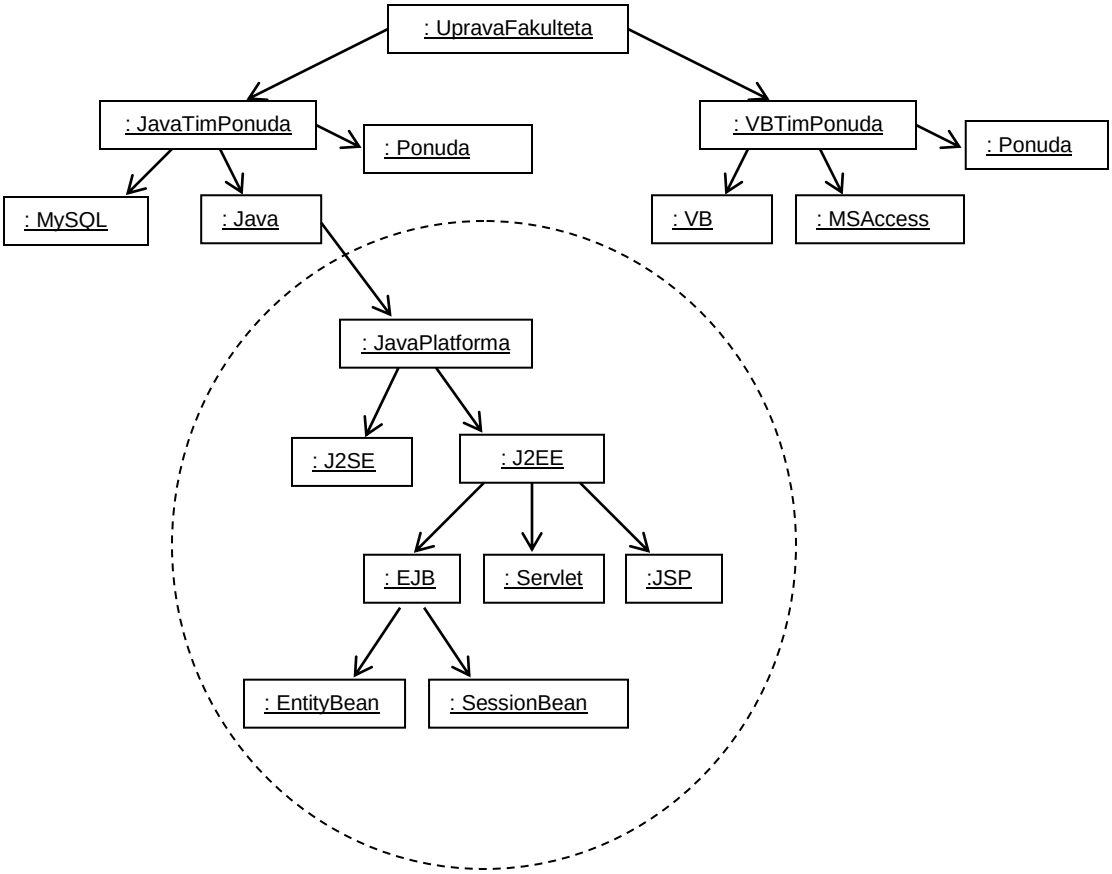
```
 Komponenta vratiKomponentu(Komponenta Komponenta,String sifraKomponente1)
  {  if (Komponenta.vratiSifruKomponente().equals(sifraKomponente1))
         return Komponenta;
      for(int i=0; i<5;i++)
        {  if(Komponenta.vratiKomponentu(i) != null)
              { if (Komponenta.vratiKomponentu(i).vratiSifruKomponente().equals(sifraKomponente1))
                   return Komponenta.vratiKomponentu(i);
                else
                   { Komponenta pom = vratiKomponentu(Komponenta.vratiKomponentu(i),sifraKomponente1);
                     if (pom!=null)
                          return pom;
                    }
              }
         }
     return null;
   }
}

// ********************************************************************************
class EJB extends Kompozicija // Composite
{ EJB(String sifraKomponente1) {super(sifraKomponente1);}
  String opisTehnologije() {return " - Java serverska tehnologija";}
}
class J2EE extends Kompozicija // Composite
{ J2EE(String sifraKomponente1) {super(sifraKomponente1);}
  String opisTehnologije() {return " - Java tehnologija za razvoj slozenih aplikacija";}
}
class JavaPlatforma extends Kompozicija // Composite
{ JavaPlatforma(String sifraKomponente1) {super(sifraKomponente1);}
}
// ********************************************************************************
// Улога: Представља просте објекте (Leaf) у  структури и дефинише њихово понашање. Прости
// објекти немају децу-објекте.
class EntityBean extends Komponenta   // Leaf
{ EntityBean(String sifraKomponente1) {super(sifraKomponente1);}
  public void napraviProgramskiJezik(String broj)
   { Java.ponudaJava = Java.ponudaJava + " " + broj + ":" + vratiSifruKomponente() + " - perzistentni podaci";}
}
class SessionBean extends Komponenta  // Leaf
{ SessionBean(String sifraKomponente1) {super(sifraKomponente1);}
public void napraviProgramskiJezik(String broj)
   {Java.ponudaJava = Java.ponudaJava + " "  + broj + ":" + vratiSifruKomponente() + " - poslovna logika";}
}
class Servlet extends Komponenta // Leaf
{ Servlet(String sifraKomponente1) {super(sifraKomponente1);}
 public void napraviProgramskiJezik(String broj)
 {Java.ponudaJava = Java.ponudaJava + " " + broj + ":" +  vratiSifruKomponente() + " - Web tehnologija (korisnici interfejs
  + poslovna logika)";}
}

class JSP extends Komponenta  // Leaf
{ JSP(String sifraKomponente1) {super(sifraKomponente1);}
public void napraviProgramskiJezik(String broj)
 { Java.ponudaJava = Java.ponudaJava + " " + broj + ":" + vratiSifruKomponente() + " - Web tehnologija (kor. interfejs)";}
}
class J2SE extends Komponenta // Leaf
{ J2SE(String sifraKomponente1) {super(sifraKomponente1);}
  public void napraviProgramskiJezik(String broj)
  { Java.ponudaJava = Java.ponudaJava + " " + broj + ":" + vratiSifruKomponente() + " - Standardna Java tehnologija";}
}
// ********************************************************************************
```

ZCO1: *Урадити за пример PCO1 методу obrisiKomponentu(String sifraKomponente), где ће се на основу параметра sifraKomponente наћи објекат са том шифром, након чега ће објекат бити обрисан из структуре.*

Дијаграм класа примера PCO1



Објектни дијаграм примера PCO1



Веза Composite патерна и општег облика патерна

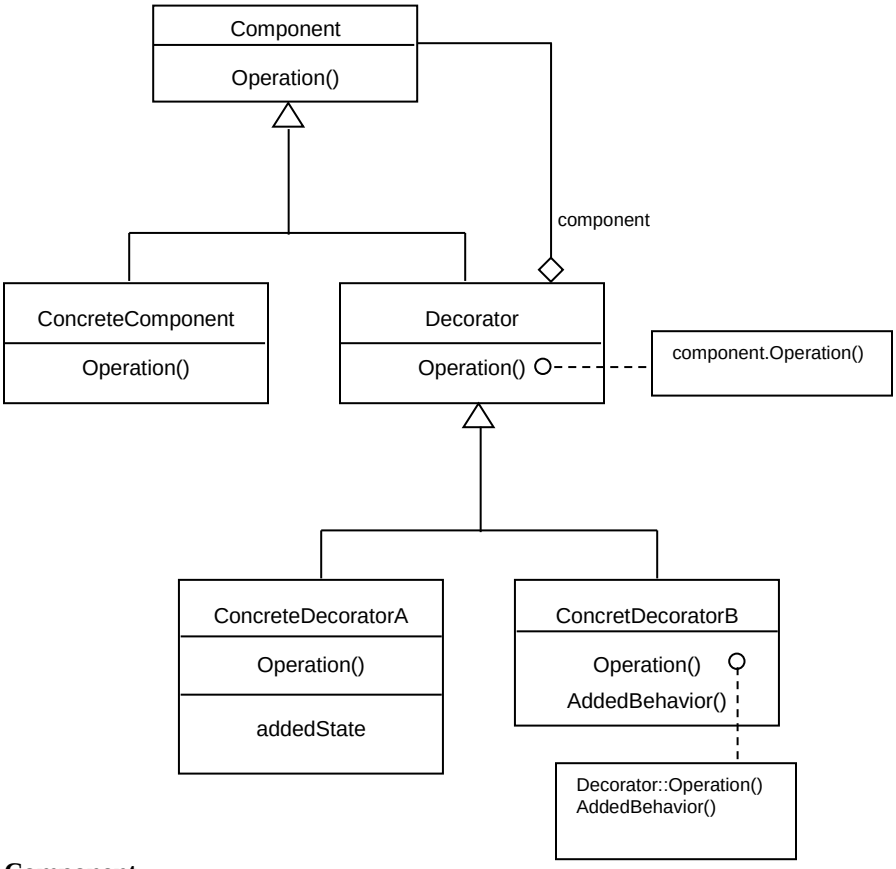
Код *Composite* патерна постоје две **СРП**: (*Client, Component,ConcreteComponent*) и (*Composition, Component, ConcreteComponent*). *ConcreteComponent* може бити *Leaf* или *Composition.*

СП4: Decorator патерн

Дефиниција
Придружује додатне одговорности (функционалности) до објекта динамички. Decorator патерн обезбеђује флексибилност у избору подкласа које проширују функционалност.

Појашњење дефиниције
Придружује додатне одговорности (функционалности) до објекта (*ConcreteComponent*) динамички. Декоратор обезбеђује флексибилност у избору подкласа *(ConcreteDecoratorA, ConcreteDecoratorB)* које проширују функционалност *ConcreteComponent* објекта.

Структура Decorator патерна



Учесници

- **Component**

Дефинише интерфејс за *ConcreteComponent* објекте којима се одговорност додаје динамички.

- **ConcreteComponent**

Дефинише објекат коме ће бити додата одговорност динамички.

- **Decorator**

Чува референцу на *Component* објекат.Дефинише интерфејс који је у складу са интерфејсом *Component*.

- **ConcreteDecorator**

Додаје одговорност до *ConcreteComponent* објекта.

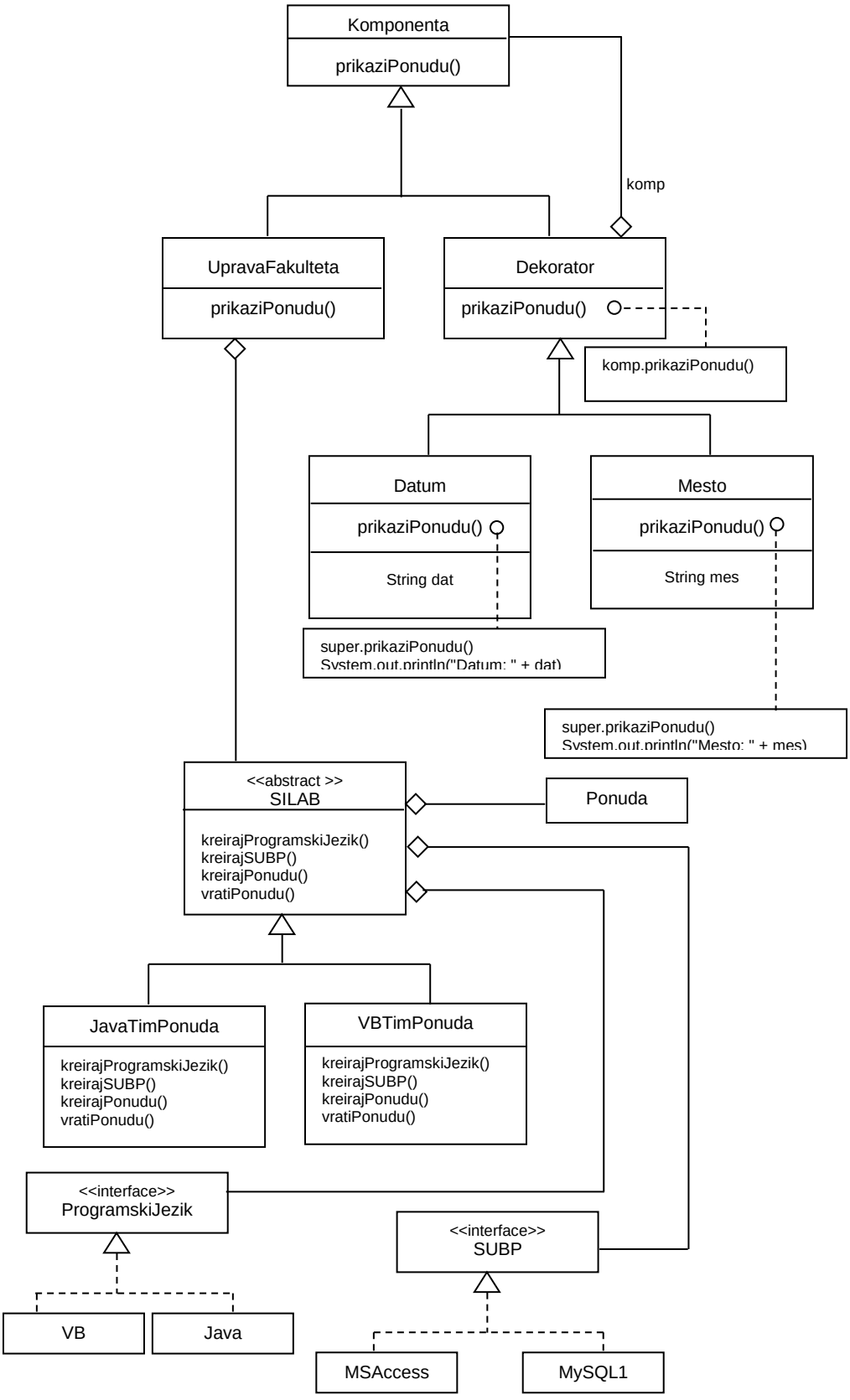
Пример Decorator патерна
Кориснички захтев PDR1[52]: *Управа Факултета треба да прошири понуду Јава тима са датумом када је издата понуда. Након тога понуду треба проширити са местом где је издата понуда.*

```
// Улога: Дефинише интерфејс за UpravaFakulteta објекат коте се одговорност додаје динамички.
interface Komponenta  // Component
{ void prikaziPonudu();}
```

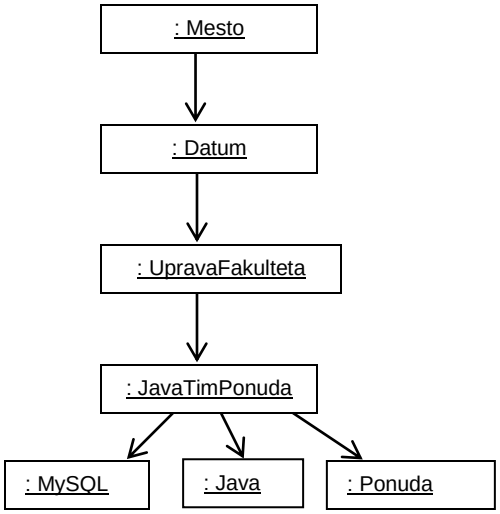
[52] Наведени пример представља проширење примера који је урађен код Builder патерна.

```
// Улога: Чува референцу на  Komponenta објекат. Дефинише интерфејс који је у складу са интерфејсом
/  Komponenta.
class  Dekorator implements Komponenta // Decorator
{   Komponenta komp;
    Dekorator(Komponenta komp1) {komp = komp1;}
    public void prikaziPonudu(){komp.prikaziPonudu();}
}
// Улога: Додаје одговорност до UpravaFakulteta објекта.
class Datum extends Dekorator  // Concrete Decorator 1
{   String dat;
    Datum(Komponenta komp1) {super(komp1); dat = new String (”28.08.2014”);}
    public void prikaziPonudu(){super.prikaziPonudu(); System.out.println("Datum: " + dat);}
}
class Mesto extends Dekorator  // Concrete Decorator 2
{   String mes;
    Mesto(Komponenta komp1) {super(komp1);mes=new String(“Beograd”);}
    public void prikaziPonudu(){super.prikaziPonudu(); System.out.println("Mesto: " + mes);}
}
// Улога: Дефинише  објекат коме ће бити додата одговорност  динамички.
class UpravaFakulteta implements Komponenta  // ConcreteComponent
{ SILAB sil; // Builder
  UpravaFakulteta(SILAB sil1){sil = sil1;}
  void Konstruisi()
    { sil.kreirajProgramskiJezik();
      sil.kreirajSUBP();
      sil.kreirajPonudu();
    }
   public static void main(String args[])
     {  UpravaFakulteta uf;
        JavaTimPonuda jat = new JavaTimPonuda();  // ConcreteBuilder1
         uf = new UpravaFakulteta(jat);
         uf.Konstruisi();
         Datum dat = new Datum(uf);
         Mesto mes = new Mesto(dat);
          // Moglo je i ovako da se napise: Mesto mes = new Mesto(new Datum(uf));
          mes.prikaziPonudu();
          // u ovom primeru se prvo pojavljuje mesto pa datum na ponudi
          //  mes = new Mesto(uf);dat = new Datum(mes); dat.prikaziPonudu();
      }
   public void prikaziPonudu(){System.out.println("Ponuda java tima: \n" + sil.vratiPonudu());}
}
abstract class  SILAB
{  ProgramskiJezik  pj;  SUBP subp;
    Ponuda pon;
   abstract void kreirajProgramskiJezik();
   abstract void kreirajSUBP();
   abstract void kreirajPonudu();
   abstract String vratiPonudu();
 }
class Ponuda {String ponuda;}
class JavaTimPonuda extends SILAB
{ JavaTimPonuda() {pon = new Ponuda();}
  public void kreirajProgramskiJezik(){pj = new Java();}
  public void kreirajSUBP() { subp = new MySQL();}
  public void kreirajPonudu() { pon.ponuda = "Programski jezik-" + pj.vratiProgramskiJezik() + "
                                SUBP-" + subp.vratiSUBP();}
  public String vratiPonudu(){return pon.ponuda;}
}
class VBTimPonuda extends SILAB {
  VBTimPonuda(){pon = new Ponuda();}
  public void kreirajProgramskiJezik(){pj = new VB();}
  public void kreirajSUBP() {subp = new MSAccess();}
  public void kreirajPonudu() { pon.ponuda = "Programski jezik-" + pj.vratiProgramskiJezik() + "
                                SUBP-" + subp.vratiSUBP();}
  public String vratiPonudu(){return pon.ponuda;}
}
interface ProgramskiJezik  {String vratiProgramskiJezik();}
class Java implements ProgramskiJezik { public String vratiProgramskiJezik(){return "Java";}}
class VB implements ProgramskiJezik { public String vratiProgramskiJezik(){return "VB";}}
interface SUBP {String vratiSUBP();}
class MySQL implements SUBP { public String vratiSUBP(){return "MySQL";}}
class MSAccess implements SUBP  { public String vratiSUBP(){return "MS Access";}}
```

Дијаграм класа примера PDR1



Објектни дијаграм примера PDR1



Веза Decorator патерна и општег облика патерна

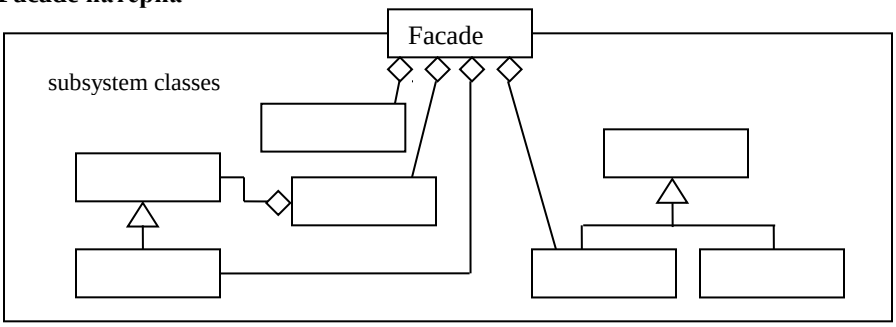
Код Decorator патерна постоји једна **СРП**: (*Decorator, Component, ConcreteC*).
ConcreteC може бити *ConcreteComponent* или *Decorator.*

СП5: Facade патерн

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

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

Структура Facade патерна



Учесници

- **Facade**
  Зна које класе подсистема (*Sybsystem classes*) су одговорне за послате захтеве од клијента. Преноси одговорност за извршење клијентских захтева до објеката подистема.
- **Sybsystem classes**
  Имплементирају подсистемске функционалности. Обрађују захтеве које су добили од *facade* објекта. Не знају ко је *facade* објекат, јер не чувају референцу на њега.

Пример Facade патерна
Кориснички захтев PFA1[53]: *Управа Факултета треба да управља процесом прављења понуде тако што ће на високом нивоу да издаје задатке Комисији за понуде која треба оперативно да реализује сваки од постављених задатака.*

```
class UpravaFakulteta // client
{ static KomisijaZaPonude kzp;
  UpravaFakulteta(){kzp = new KomisijaZaPonude();}

  public static void main(String args[])
            {  UpravaFakulteta uf = new UpravaFakulteta();
               kzp.odrediFormatPonude(args[0]);
               kzp.kreirajPonuduJavaTima();
               kzp.Konstruisi();
               kzp.prikaziPonudu();
   }
}
```

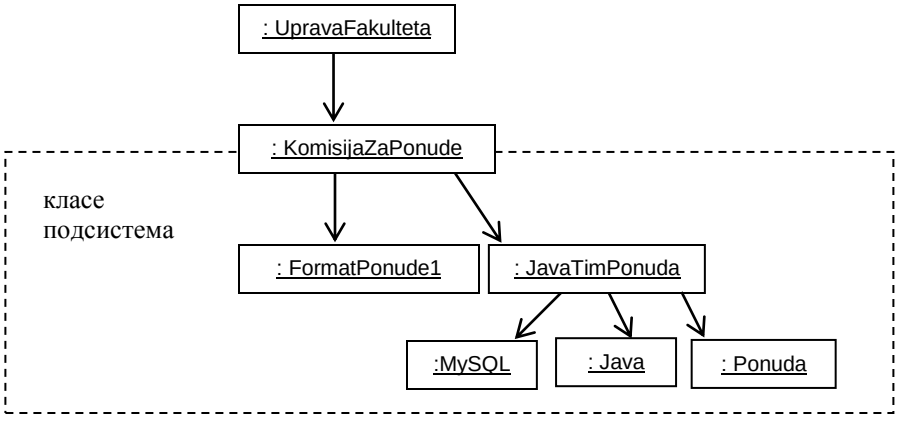
```
// Улога: Зна које класе подсистема (FormatPonude, SILAB) су одговорне за послате захтеве од клијента. Преноси
одговорност за извршење клијентских захтева до објеката подистема.
class KomisijaZaPonude  // Facade
{   FormatPonude fp;
    SILAB sil;
    void odrediFormatPonude(String arg)
      { if (arg.equals("1")) fp = new FormatPonude1();
        if (arg.equals("2")) fp = new FormatPonude2();
      }
    void kreirajPonuduJavaTima(){sil = new JavaTimPonuda(fp);}
    void Konstruisi(){  sil.kreirajProgramskiJezik();  sil.kreirajSUBP();sil.kreirajPonudu(fp);}
    void prikaziPonudu(){System.out.println("Ponuda java tima: " + sil.vratiPonudu());}
}
```

[53] Наведени пример представља проширење примера који је урађен код Bridge патерна.

```
// Улога: Имплементирају подсистемске функционалности. Обрађују захтеве које су добили од facade (KomisijaZaPonude) // објекта. Не
знају ко је facade објекат, јер не чувају референцу на њега.
abstract class SILAB
{ ProgramskiJezik pj;
  SUBP subp;
  Ponuda pon;
  abstract void kreirajProgramskiJezik();
  abstract void kreirajSUBP();
  public String vratiPonudu(){return pon.ponuda;}
  public void kreirajPonudu(FormatPonude fp) { pon.ponuda = fp.vratiFormatPonude(this);}
 }

class JavaTimPonuda extends SILAB
{ JavaTimPonuda() {pon = new Ponuda();}
  public void kreirajProgramskiJezik(){pj = new Java();}
  public void kreirajSUBP() { subp = new MySQL();}
  public String vratiPonudu(){return "Autor: Lab za soft. inzenjerstvo: " + pon.ponuda;}
}

class VBTimPonuda extends SILAB { VBTimPonuda(){pon = new Ponuda();}
  public void kreirajProgramskiJezik(){pj = new VB();}
  public void kreirajSUBP() {subp = new MSAccess();}
}

class Ponuda {String ponuda;}

// Улога: Дефинише интерфејс за имплементационе класе (FormatPonude1, FormatPonude2).
abstract class FormatPonude
{ abstract String vratiFormatPonude(SILAB sil);
}

// Улога: Имплементира интерфејсFormatPonude.
class FormatPonude1 extends FormatPonude
{ String vratiFormatPonude(SILAB sil)
    { return "Programski jezik-" + sil.pj.vratiProgramskiJezik() + " SUBP-" + sil.subp.vratiSUBP();}
}

class FormatPonude2 extends FormatPonude
{
String vratiFormatPonude(SILAB sil)
   { return "SUBP-" + sil.subp.vratiSUBP() + " Programski jezik-" + sil.pj.vratiProgramskiJezik();}
}

interface ProgramskiJezik {String vratiProgramskiJezik();}
class Java implements ProgramskiJezik { public String vratiProgramskiJezik(){return "Java";}}
class VB implements ProgramskiJezik { public String vratiProgramskiJezik(){return "VB";}}
interface SUBP {String vratiSUBP();}
class MySQL implements SUB P{ public String vratiSUBP(){return "MySQL";}}
class MSAccess implements SUBP { public String vratiSUBP(){return "MS Access";}}
```

ZPFA: *Нацртати дијаграм класа за пример PFA1.*

Објектни дијаграм примера PFA1



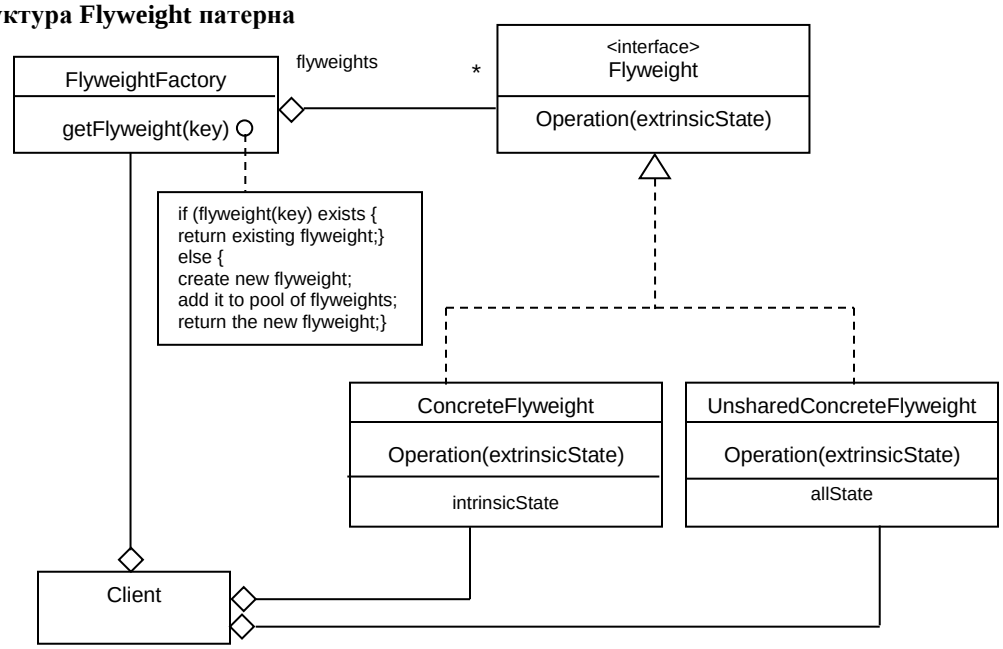
Веза Facade патерна и општег облика патерна

Код Facade патерна не постоји ни једна **СРП**.

СП6: Flyweight патерн
Дефиниција
Користи дељење да ефикасно подржи велики број ситних објеката.

Појашњење ГОФ дефиниције
Користи дељење (*ConcreteFlyweight*) да ефикасно подржи велики број ситних објеката.

Структура Flyweight патерна



Учесници

- **Flyweight –** декларише интерфејс преко кога *Flyweight* објекти могу да прихвате и делују на спољашње стање.
- **ConcreteFlyweight** – имплементира *Flyweight* интерфејс и додаје простор за унутрашње стање (*intrinsicState*) ако постоји. *ConcreteFlyweight* објекат мора бити дељив. Стање које се чува мора бити унутрашње; т.ј., не сме зависити од контекста у коме се налази *ConcreteFlyweight*.
- **UnsharedConcreteFlyweight** – поред дељивих *Flyweight* подкласа постоје и *Flyweight* подкласе које нису дељиве (*UnsharedConcreteFlyweight*). *Flyweight* интерфејс омогућава дељење али га не намеће. Заједничко за недељиве *Flyweight* подкласе је да имају, на неком нивоу, *ConcreteFlyweight* објекте као децу у објектној структури.
- **FlyweightFactory** – креира и управља *Flyweight* објектима. Он омогућава дељивост *Flyweight* објектима. Када клијент захтева *Flyweight* објекат, *FlyweightFactory* објекат враћа постојећи *Flyweight* објекат или креира нови ако исти не постоји.
- **Client** – садржи референце на *Flyweight* објекте. Израчунава или чува спољашња стања *Flyweight* објеката.

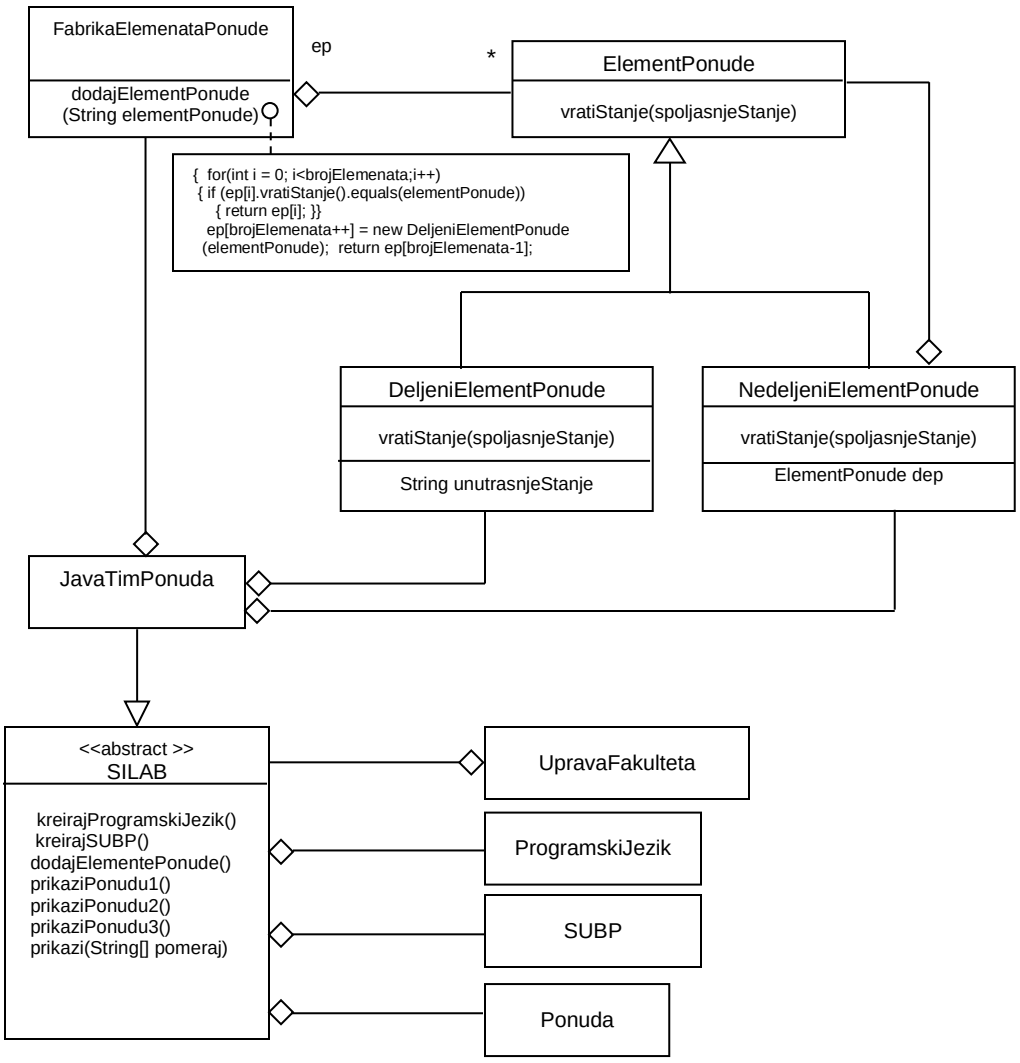
Пример Flyweight патерна
Кориснички захтев PFW1: *Управа Факултета је тражила од Јава тима да припреми своју понуду у следећа три различита облика:*
а) Понуда Јава тима.
Аутор: Лаб. за софтверско инжењерство.
Програмски језик: Java.
СУБП: MySQL.
б) Понуда Јава тима. Аутор: Лаб. за софтверско инзењерство.
Програмски језик: Java. СУБП: MySQL.
ц) Аутор: Лаб. за софтверско инзењерство.
Понуда Јава тима.
Програмски језик: Јава.
СУБП: MySQL.

Шеф Лабораторије за софтверско инжењерство је тражио од Јава тима да елементе понуде чува на једном месту, како се не би десила редуданса истих елемената понуде у различитим захтеваним облицима понуде.

```
class UpravaFakulteta
{ static SILAB sil;
  public static void main(String args[])
              {  sil = new JavaTimPonuda();
                 sil.kreirajProgramskiJezik();
                 sil.kreirajSUBP();
                 sil.dodajElementePonude();
                 sil.prikaziPonudu1();
                 sil.prikaziPonudu2();
                 sil.prikaziPonudu3();
              }
}

class Ponuda {String ponuda;}

abstract class SILAB
{ ProgramskiJezik  pj;  SUBP subp;  Ponuda pon;
  abstract void kreirajProgramskiJezik();
  abstract void kreirajSUBP();
  abstract void dodajElementePonude();
  abstract void prikaziPonudu1();
  abstract void prikaziPonudu2();
  abstract void prikaziPonudu3();
  abstract void prikazi(String[] pomeraj);
}

// Улога: садржи референце на ElementPonude објекте. Израчунава или чува спољашња стања ElementPonude објеката.
class JavaTimPonuda extends SILAB  // Client
{ FabrikaElemenataPonude fep;
  NedeljeniElementPonude nep[];
  ElementPonude ep;

  JavaTimPonuda()
    { fep = new FabrikaElemenataPonude();pon = new Ponuda();
      nep  =new NedeljeniElementPonude[4];}
      void kreirajProgramskiJezik(){pj = new Java();}
      void kreirajSUBP() {subp = new MySQL();}
      void dodajElementePonude()
          { fep.dodajElementPonude("Ponuda Java tima.");
            fep.dodajElementPonude("Autor: Lab. za softversko inzenjerstvo.");
            fep.dodajElementPonude("Programski jezik: " + pj.vratiProgramskiJezik() + ".");
            fep.dodajElementPonude("SUBP:" + subp.vratiSUBP() + ".");
          }
  void prikaziPonudu1()
     { String [] pomeraj = { "","\n","\n","\n"};
       ep = fep.dodajElementPonude("Ponuda Java tima.");
       nep[0] = new NedeljeniElementPonude(ep);
       ep = fep.dodajElementPonude("Autor: Lab. za softversko inzenjerstvo.");
       nep[1] = new NedeljeniElementPonude(ep);
       ep = fep.dodajElementPonude("Programski jezik: " + pj.vratiProgramskiJezik() + ".");
       nep[2] = new NedeljeniElementPonude(ep);
       ep = fep.dodajElementPonude("SUBP:" + subp.vratiSUBP() + ".");
              nep[3] = new NedeljeniElementPonude(ep);
        prikazi(pomeraj);

     }
    void prikaziPonudu2()
     {   String [] pomeraj = { "","\t","\n",""};
         ep = fep.dodajElementPonude("Ponuda Java tima.");
         nep[0] = new NedeljeniElementPonude(ep);
         ep = fep.dodajElementPonude("Autor: Lab. za softversko inzenjerstvo.");
         nep[1] = new NedeljeniElementPonude(ep);
         ep = fep.dodajElementPonude("Programski jezik: " + pj.vratiProgramskiJezik() + ".");
         nep[2] = new NedeljeniElementPonude(ep);
         ep = fep.dodajElementPonude("SUBP:" + subp.vratiSUBP() + ".");
         nep[3] = new NedeljeniElementPonude(ep);
         prikazi(pomeraj);
     }
```

```
void prikaziPonudu3()
    {  String [] pomeraj = { "","\n\t","\n\t\t","\n\t\t"};
       ep = fep.dodajElementPonude("Autor: Lab. za softversko inzenjerstvo.");
       nep[0] = new NedeljeniElementPonude(ep);
       ep = fep.dodajElementPonude("Ponuda Java tima.");
       nep[1] = new NedeljeniElementPonude(ep);
       ep = fep.dodajElementPonude("Programski jezik: " + pj.vratiProgramskiJezik() + ".");
       nep[2] = new NedeljeniElementPonude(ep);
       ep = fep.dodajElementPonude("SUBP:" + subp.vratiSUBP() + ".");
       nep[3] = new NedeljeniElementPonude(ep);
       prikazi(pomeraj);
     }
void prikazi(String pomeraj[])
   {  pon.ponuda = "";
      for(int i=0;i<4;i++)  { pon.ponuda = pon.ponuda + nep[i].vratiStanje(pomeraj[i]); }
              System.out.println(pon.ponuda);
         System.out.println("*********************************");
    }
 }
interface ProgramskiJezik
{String vratiProgramskiJezik();}
class Java implements ProgramskiJezik
{ public String vratiProgramskiJezik(){return "Java";}}
interface SUBP
{String vratiSUBP();}
class MySQL implements SUBP
{ public String vratiSUBP(){return "MySQL";}}
//Улога:Креира и управља ElementPonude објектима. Он омогућава дељивост ElementPonude објектима. Када клијент
// захтева ElementPonude објекат, FabrikaElemenataPonude  објекат враћа постојећи Flyweight објекат  или креира нови ако
// исти не постоји.
class FabrikaElemenataPonude   // FlyweightFactory
{ ElementPonude ep[];
  int brojElemenata;
  FabrikaElemenataPonude(){ep = new DeljeniElementPonude[4];brojElemenata=0;}
  ElementPonude dodajElementPonude(String elementPonude)
     {  for(int i = 0; i<brojElemenata;i++)
              { if (ep[i].vratiStanje().equals(elementPonude))
                  { return ep[i]; }
              }

           ep[brojElemenata++] = new DeljeniElementPonude(elementPonude);
           return ep[brojElemenata-1];
     }
}
// Улога: Декларише интерфејс преко кога ElementPonude  објекти могу да прихвате и делују на спољашње стање.
abstract class ElementPonude // Flyweight
{ abstract String vratiStanje(String spoljasnjeStanje);
  String vratiStanje() {return "";}
}
// Улога: Имплементира ElementPonude интерфејс и додаје простор за унутрашње стање (unutrasnjeStanje) ако постоји.
DeljeniElementPonude објекат мора бити дељив. Стање које се чува мора бити унутрашње; т.ј., не сме зависити од
контекста у коме се налази DeljeniElementPonude објекат.
class DeljeniElementPonude extends ElementPonude // ConcreteFlyweight
{ String unutrasnjeStanje;
  DeljeniElementPonude (String unutrasnjeStanje1){unutrasnjeStanje = unutrasnjeStanje1;}
  public String vratiStanje(String spoljasnjeStanje){ return spoljasnjeStanje + " " + unutrasnjeStanje;}
  String vratiStanje(){return unutrasnjeStanje;}
  }
// Улога: Поред дељивих ElementPonude подкласа  постоје и ElementPonude подкласе које нису дељиве
(NedeljeniElementPonude). ElementPonude интерфејс омогућава дељење али га не намеће. Заједничко за недељиве
ElementPonude подкласе је да имају, на неком нивоу, DeljeniElementPonude  објекте као децу у објектној структури.
class NedeljeniElementPonude  extends ElementPonude // UnsharedConcreteFlyweight
{ ElementPonude dep;
  NedeljeniElementPonude(ElementPonude dep1) {dep = dep1; }
  public String vratiStanje(String spoljasnjeStanje){return dep.vratiStanje(spoljasnjeStanje);}
}
```

Дијаграм класа примера PFW1



ZPFA: *Нацртати објектни дијаграм за пример PFW1.*

Веза Flyweight патерна и општег облика патерна

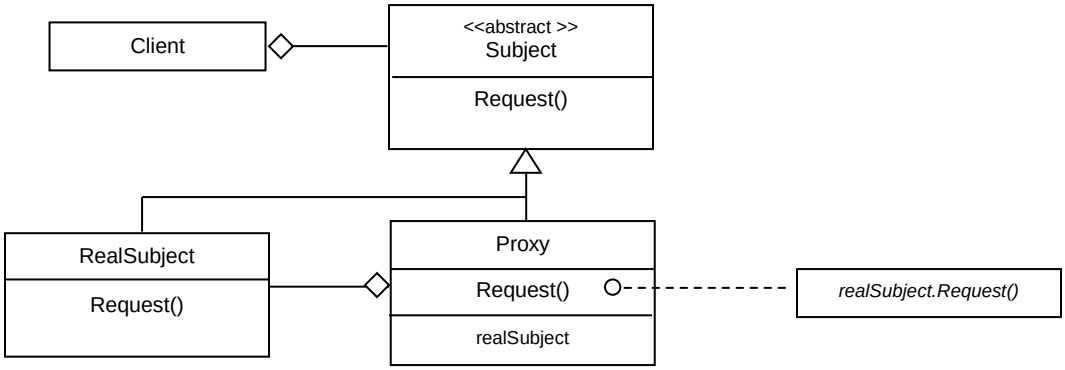
Код Flyweight патерна постоји једна **СРП**: (*FlyweightFactory, Flyweight, ConcreteF*).
ConcreteF може бити *ConcreteFlyweight* или *UnsharedConcreteFlyweight.*

СП7: Proxy патерн

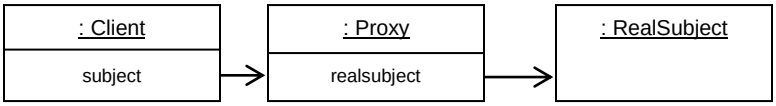
Дефиниција
Обезбеђује посредника за приступање другом објекту како би се омогућио контролисани приступ до њега.

Појашњење дефиниције
Обезбеђује посредника (*Proxy*) за приступање другом објекту (*RealSubject*) како би се омогућио контролисани приступ до њега.

Структура Proxy патерна



Приказујемо могући објектни дијаграм proxy структуре у реалном времену.



Учесници

- **Proxy**
  - Садржи референцу које омогућава *Proxy* објекту приступ до *RealSubject* објекта.
  - Обезбеђује интерфејс идентичан са интерфејсом *Subject* тако да *Proxy* објекат може заменити *RealSubject* објекат.
  - Контролише приступ до *RealSubject* објекта и може бити одговоран за његово креирање и брисање.
- **Subject**
  - Дефинише заједнички интерфејс за *RealSubjec*t и *Proxy* класе тако да се *Proxy* објекат може користити свуда где се очекује RealSubject објекат.
- **RealSubject**
  - Дефинише *RealSubject* објекат који репрезентује *Proxy* објекат.

Пример Proxy патерна
Кориснички захтев PPX1: *Управа Факултета је послала захтев Јава тиму Лабораторије за софтверско инжењерство да направи (састави) понуду за израду софтверског система последипломских студија ФОН-а. У понуди треба се наведе:*
а) Програмски језик у коме ће се развијати програм.
б) Систем за управљање базом података у коме ће се чувати подаци.
Управа Факултета ће надзирати (контролисати) израду понуда. Јава тим са ФОН-а је послао захтев Јава тиму фирме JavaGroup да направи наведену понуду.

```
class UpravaFakulteta   // Client
{  SILAB sil;
   UpravaFakulteta(SILAB sil1){sil = sil1;}

  void Konstruisi()
    { sil.kreirajProgramskiJezik();
      sil.kreirajSUBP();
      sil.kreirajPonudu();
    }
  public static void main(String args[])
            {  UpravaFakulteta uf;
               JavaTimGroupPonuda jtg = new JavaTimGroupPonuda();
               JavaTimPonuda jat = new JavaTimPonuda(jtg);
               uf = new UpravaFakulteta(jat);
               uf.Konstruisi();
               System.out.println("Ponuda java tima: " + jat.vratiPonudu());
            }
 }
```

// Улога: Дефинише заједнички интерфејс за JavaTimGroupPonuda и JavaTimPonuda класе тако да се JavaTimPonuda
// објекат може користити свуда где се очекује JavaTimGroupPonuda објекат.

```
abstract class  SILAB // Subject
{  ProgramskiJezik  pj;
    SUBP subp;
    Ponuda pon;
   abstract void kreirajProgramskiJezik();
   abstract void kreirajSUBP();
   abstract void kreirajPonudu();
   abstract String vratiPonudu();
 }
class Ponuda {String ponuda;}
```

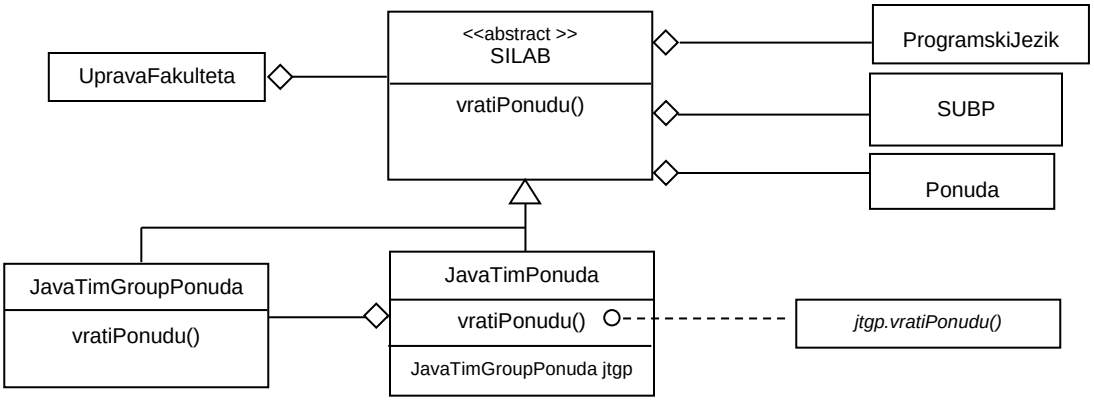
// Улога: Садржи референцу које омогућава JavaTimPonuda објекту приступ до JavaTimGroupPonuda објекту.
// Обезбеђује интерфејс идентичан са интерфејсом SILAB тако да JavaTimPonuda објекат може заменити
// JavaTimGroupPonuda објекат Контролише приступ до JavaTimGroupPonuda објекта и може бити одговоран за његово //креирање и брисање.

```
class JavaTimPonuda extends SILAB //  Proxy
{ JavaTimGroupPonuda jtgp;
  JavaTimPonuda(JavaTimGroupPonuda jtgp1) {jtgp=jtgp1;}
  public void kreirajProgramskiJezik(){jtgp.kreirajProgramskiJezik();}
  public void kreirajSUBP() { jtgp.kreirajSUBP();}
  public void kreirajPonudu() { jtgp.kreirajPonudu();}
  public String vratiPonudu(){return jtgp.vratiPonudu();}
}
```

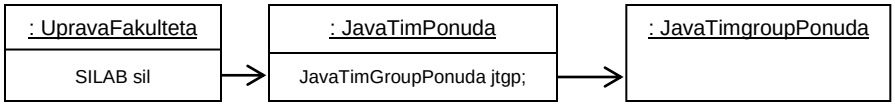
// Улога: Дефинише JavaTimGroupPonuda објекат који репрезентује JavaTimPonuda објекат.

```
class JavaTimGroupPonuda extends SILAB // RealSubject
{  JavaTimGroupPonuda() {pon = new Ponuda();}
   public void kreirajProgramskiJezik(){pj = new Java();}
   public void kreirajSUBP() { subp = new MySQL();}
   public void kreirajPonudu() { pon.ponuda = "Programski jezik-" + pj.vratiProgramskiJezik() + " SUBP-" +
                                  subp.vratiSUBP();}
   public String vratiPonudu(){return pon.ponuda;}
}

interface ProgramskiJezik {String vratiProgramskiJezik();}
class Java implements ProgramskiJezik { public String vratiProgramskiJezik(){return "Java";}}
class VB implements ProgramskiJezik { public String vratiProgramskiJezik(){return "VB";}}
interface SUBP {String vratiSUBP();}
class MySQL implements SUBP { public String vratiSUBP(){return "MySQL";}}
class MSAccess implements SUBP { public String vratiSUBP(){return "MS Access";}}
```

Дијаграм класа примера PPX1



Објектни дијаграм примера PPX1



Веза Proxy патерна и општег облика патерна

Код Proxy патерна постоји једна **СРП**: (*Client,Subject,ConcreteSubject*).
ConcreteSubject може бити *Proxy* или *RealSubject.*

ПП: Патерни понашања

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

Постоје следећи патерни понашања:

1. Chain of responsability - 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, редове или дневнике захтева, и подржава повратне (undoable) операције чији се ефекат може поништити.

3. Interpreter - За дати језик, дефинише репрезентацију граматике језика заједно са интерпретером, који користи ту репрезентацију да интерпретира (тумачи) 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. Помоћу медијатора се успоставља слаба веза између објеката, тако што се онемогућава њихово међусобно експлицитно референцирање, што омогућава независно мењање њиховог међудејства.

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 његово интерно стање.

9. Strategy - Дефинише фамилију алгоритама, учаурује сваки од њих и обезбеђује да они могу бити замењиви. Strategy патерн омогућава промену алгоритма независно од клијената који га користе.

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

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

ПП1. Chain of responsibility патерн

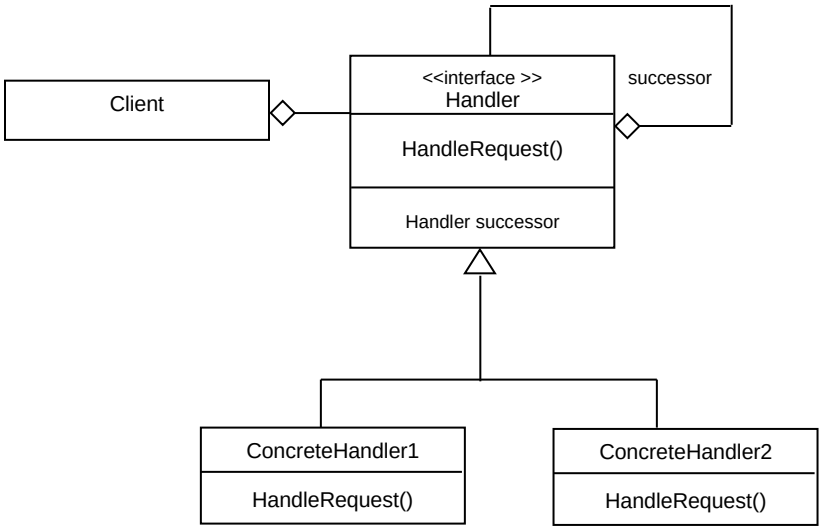
Дефиниција

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

Појашњење дефиниције

Избегава чврсто повезивање између пошиљаоца захтева (*Client*) и његовог примаоца (*Handler*), обезбеђујући ланац повезаних објеката (*ConcreteHandler1*, *ConcreteHandler2*), који ће да обрађују захтев све док се он не обради.

Структура Chain of responsibility патерна



Учесници

- **Handler**
  Дефинише интерфејс за обраду захтева. Садржи линк (*successor*) ка следећем *Handler* објекту.
- **ConcreteHandler**
  Обрађује захтев за који је одговоран. Може приступити његовом следбенику. Уколико може да обради захтев он га обрађује, иначе га прослеђује до следбеника.
- **Client**
  Иницира захтев који треба да се обради.

Пример Chain of responsibility патерна:

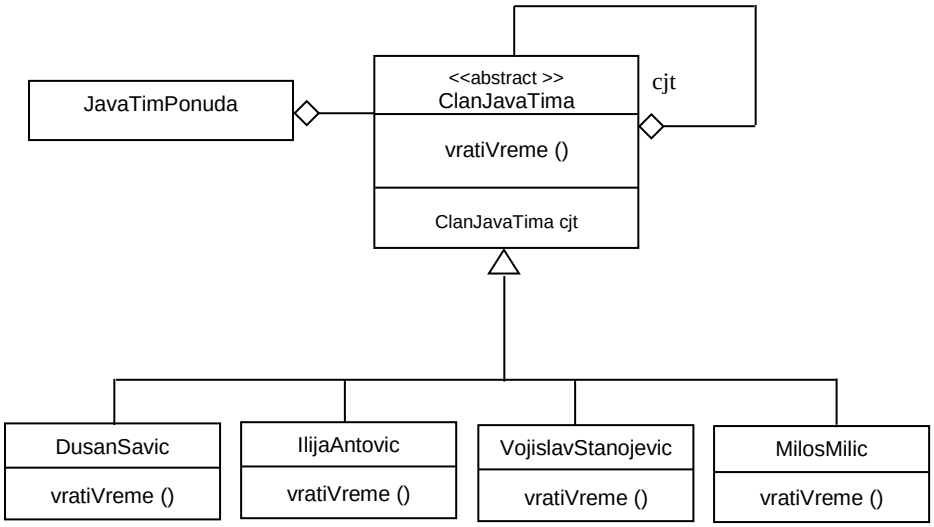
Кориснички захтев COR1: *Шеф Лабораторије за СИ је послао захтев члановима Јава тима да свако од њих прикаже утрошено време у писању понуде.*
Ово је варијанта када више Handler објеката обрађује исти захтев.

```
class UpravaFakulteta
{ SILAB sil;
  UpravaFakulteta(SILAB sil1){sil = sil1;}

  void Konstruisi()
    { sil.kreirajProgramskiJezik();
      sil.kreirajSUBP();
      sil.kreirajPonudu();
    }
 public static void main(String args[])
    {  UpravaFakulteta uf;
       DusanSavic ds = new DusanSavic(null,true);
       IlijaAntovic ia = new IlijaAntovic(ds,true);
       VojislavStanojevic vs = new VojislavStanojevic(ia,true);
       MilosMilic mm = new MilosMilic(vs,true);
       JavaTimPonuda jat = new JavaTimPonuda(mm);
       uf = new UpravaFakulteta(jat);
       uf.Konstruisi();
       System.out.println(jat.vratiPonudu()); }}
```

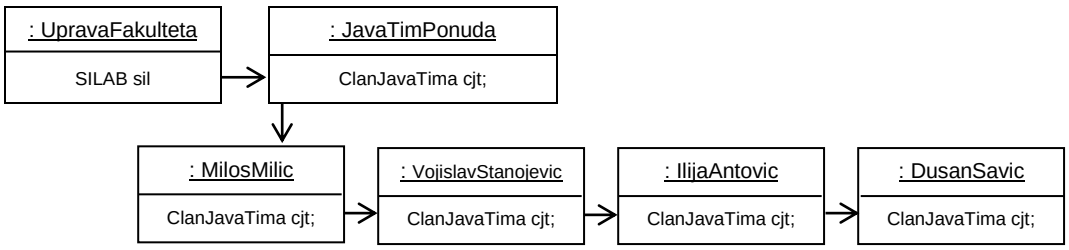
```
abstract class  SILAB
{  ProgramskiJezik  pj;
    SUBP subp;
    Ponuda pon;

   abstract void kreirajProgramskiJezik();
   abstract void kreirajSUBP();
   abstract void kreirajPonudu();
   abstract String vratiPonudu();
 }
class Ponuda {String ponuda;}
// Uloga: Inicira zahtev koji treba da se obradi.
class JavaTimPonuda extends SILAB  // Client
{ ClanJavaTima cjt;
  JavaTimPonuda() {pon = new Ponuda();}
  JavaTimPonuda(ClanJavaTima cjt1){cjt = cjt1;pon = new Ponuda();}
  public void kreirajProgramskiJezik(){pj = new Java();}
  public void kreirajSUBP() { subp = new MySQL();}
  public void kreirajPonudu(){pon.ponuda = "Programski jezik-" + pj.vratiProgramskiJezik() + "  SUBP-" + subp.vratiSUBP()
                                  + " " + utrosenoVreme();}
  public String vratiPonudu(){return pon.ponuda;}
  String utrosenoVreme(){return cjt.utrosenoVreme();}
}
// Uloga: Definiše interfejs za obradu zahteva. Sadrži link (cjt) ka sledećem ClanJavaTima objektu.
abstract class ClanJavaTima // Handler
{  ClanJavaTima cjt;
   boolean radioNaPonudi;
   ClanJavaTima(ClanJavaTima cjt1,boolean radioNaPonudi1){cjt = cjt1;
                    radioNaPonudi=radioNaPonudi1;}
   String utrosenoVreme()
   {       String uvreme = "";
          if (radioNaPonudi == true) uvreme = "\nUtroseno vreme:" + vratiVreme() + " ";
          if (cjt!=null ) uvreme = uvreme + cjt.utrosenoVreme(); // Lifo lista
          // uvreme = cjt.utrosenoVreme() + uvreme; - Fifo lista
         return uvreme;
   }
   abstract String vratiVreme();
}
// Uloga: Obrađuje zahtev za koji je odgovoran. Može pristupiti njegovom sledbeniku. Ukolико može da obradi zahtev on
// ga obrađuje, inače ga prosleđuje do sledbenika.
class DusanSavic extends ClanJavaTima  // ConcreteHandler1
{ DusanSavic(ClanJavaTima cjt1,boolean radioNaPonudi1){super(cjt1,radioNaPonudi1);}
  String vratiVreme(){return " Dusan Savic - 2h";}}
class IlijaAntovic extends ClanJavaTima  // ConcreteHandler2
{ IlijaAntovic(ClanJavaTima cjt1,boolean radioNaPonudi1){super(cjt1,radioNaPonudi1);}
  String vratiVreme(){return " Ilija Antovic - 1h 30'";}}
class VojislavStanojevic extends ClanJavaTima  // ConcreteHandler3
{ VojislavStanojevic(ClanJavaTima cjt1,boolean radioNaPonudi1){super(cjt1,radioNaPonudi1);}
  String vratiVreme(){return " Vojislav Stanojevic - 1 25'";}}
class MilosMilic extends ClanJavaTima  // ConcreteHandler4
{MilosMilic(ClanJavaTima cjt1,boolean radioNaPonudi1){super(cjt1,radioNaPonudi1);}
 String vratiVreme(){return " Milos Milic - 1h 20'";}}

interface ProgramskiJezik {String vratiProgramskiJezik();}
class Java implements ProgramskiJezik public String vratiProgramskiJezik(){return "Java";}}
class VB implements ProgramskiJezik { public String vratiProgramskiJezik(){return "VB";}}
interface SUBP {String vratiSUBP();}
class MySQL implements SUBP{ public String vratiSUBP(){return "MySQL";}}
class MSAccess implements SUBP{ public String vratiSUBP(){return "MS Access";}}
```

ZCOR1: *Написати програм код кога само један Handler објекат обрађује захтев(први на који наиђе захтев). Кориснички захтев је исти као код примера COR1.*

Дијаграм класа примера PCOR1



Објектни дијаграм примера PCOR1



Веза Chain of responsibility патерна и општег облика патерна

Код *Chain of responsibility* патерна постоје две **СРП**: (*Client,Handler, ConcreteHandler*) и (*Handler,Handler, ConcreteHandler*).

ПП2. Command патерн

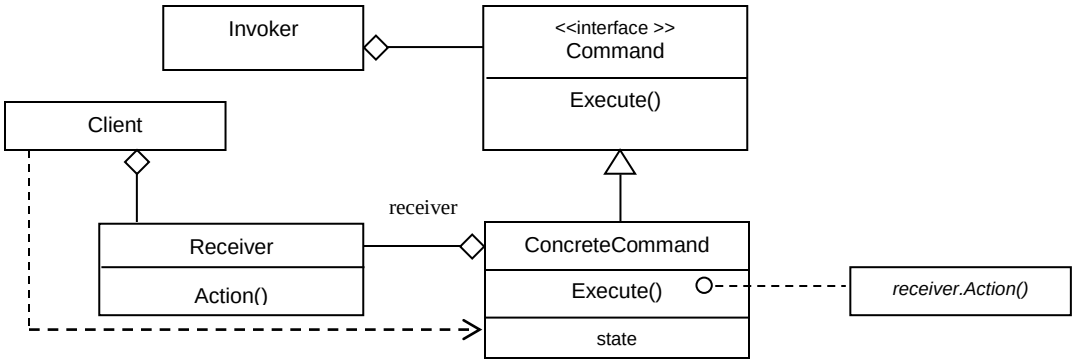
Дефиниција

Захтев се учаурује као објекат, што омогућава клијентима да параметризују различите захтеве, редове или дневнике захтева, и подржава повратне (undoable) операције чији се ефекат може поништити.

Појашњење дефиниције

Захтев (*Execute()*) се учаурује као објекат (*ConcreteCommand*), што омогућава клијентима (*Client*) да параметризују различите захтеве (*ConcreteCommand.execute(),Receiver.Action()*), редове или дневнике захтева, и подржава повратне (undoable) операције чији се ефекат може поништити.

Структура Command патерна



Учесници

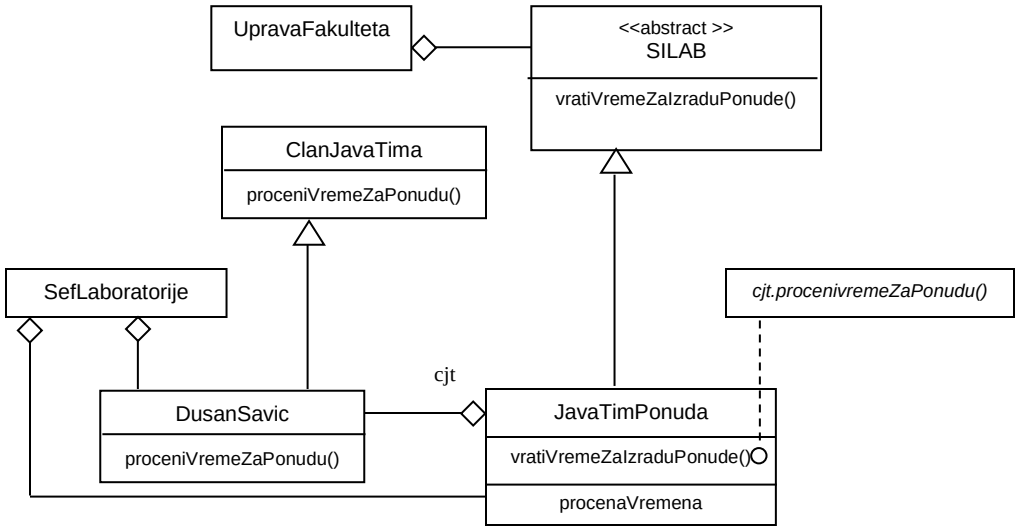
- **Invoker**
  Позива *ConcreteCommand* објекат да изврши постављени захтев (*Execute()*).
- **Command**
  Декларише интерфејс за извршење операције (*Execute()*).
- **ConcreteCommand**
  Дефинише везу између *Receiver* објекта и акције (*Action()*). Имплементира *Execute()* методу позивајући методу *Action() Receiver* објекта.
- **Client**
  Креира *ConcreteCommand* објекат и дефинише његов *Receiver* објекат .
- **Receiver**
  Извршава методу (*Action()*) која је придружена постављеном захтеву(*Execute()*). Било која класа може да буде *Receiver* класа.

Пример Command патерна

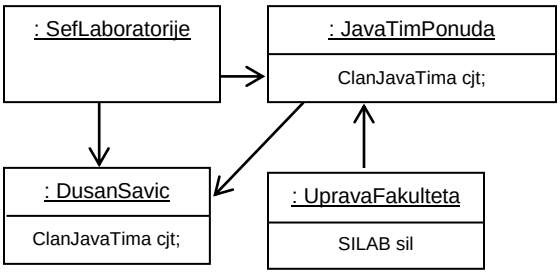
Кориснички захтев PCOMM1: *Шеф лабораторије одређује: а) да је Јава тим задужен да врати Управи Факултета време потребно за израду понуде б) Душана Савића да процени колико је времена потребно да се направи понуда. Сваки пут када Управа Факултета позове Јава тим да врати време потребно да се направи понуда, Јава тим позива Душана Савића да он да процену о потребном времену. Наведену процену Јава тим шаље до Управе Факултета.*

```
// Улога: Креира JavaTimPonuda објекат и дефинише његов Receiver (DusanSavic) објекат .
class SefLaboratorije   // Client
{  DusanSavic ds;
  SILAB sil;
  SefLaboratorije(){ds = new DusanSavic();sil = new JavaTimPonuda(ds);}
  SILAB vratiTimPonuda(){return sil;}
}

// Улога: Позива SILAB објекат да изврши постављени захтев (vratiVremeZaIzraduPonude()).
class UpravaFakulteta  //Invoker
{
  SILAB sil;
  UpravaFakulteta(SILAB sil1){sil = sil1;}

  public static void main(String args[])
    { UpravaFakulteta uf;
      SefLaboratorije sl = new  SefLaboratorije();
      SILAB sil1 = sl.vratiTimPonuda();
      uf = new UpravaFakulteta(sil1);
      System.out.println(uf.vratiVremeZaIzraduPonude());
    }

  String vratiVremeZaIzraduPonude(){return sil.vratiVremeZaIzraduPonude();}
}


// Улога: Декларише интерфејс за извршење операције Execute().
abstract class SILAB // Command
{
  abstract String vratiVremeZaIzraduPonude();
}


// Улога: Дефинише везу између  ClanJavaTima објекта (cjt)  и акције(proceniVremeZaPonudu(). Имплементира
//методу  vratiVremeZaIzraduPonude ()  позивајући metodu proceniVremeZaPonudu() ClanJavaTima објекта.
class JavaTimPonuda extends SILAB   // ConcreteCommand
{ ClanJavaTima cjt;
  String procenaVremena;
  JavaTimPonuda(ClanJavaTima cjt1){cjt=cjt1;}
  String vratiVremeZaIzraduPonude(){procenaVremena = cjt.proceniVremeZaPonudu(); return procenaVremena; }
}


abstract class ClanJavaTima
{abstract String proceniVremeZaPonudu();
}

// Улога:   Извршава методу proceniVremeZaPonudu ()  која је придружена постављеном
// захтеву (vratiVremeZaIzraduPonude()).
class DusanSavic  extends ClanJavaTima// Receiver
{ String proceniVremeZaPonudu(){return "Potrebno je 2 dana da se napravi ponuda";}
}
```

Дијаграм класа примера PCOMM1



Објектни дијаграм примера PCOMM1



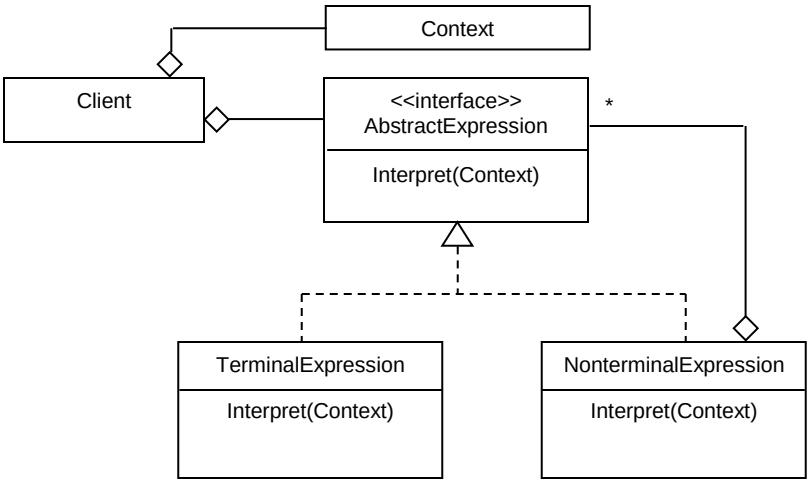
Веза Command патерна и општег облика патерна

Код Command патерна постоји једна **СРП**: (*Invoker,Command, ConcreteCommand*).

ППЗ. Interpreter патерн

Дефиниција
За дати језик, дефинише репрезентацију граматике језика заједно са интерпретером, који користи ту репрезентацију да интерпретира (тумачи) реченице у језику.

Појашњење дефиниције
За дати језик (*Context*), дефинише репрезентацију граматике језика заједно са интерпретером (*AbstractExpression*, *TerminalExpression*, *NonterminalExpression*). Interpreter користи ту репрезентацију да интерпретира, помоћу операције *Interpret(Context)* реченице у језику.



Учесници

- **Client**
  Гради стабло апстрактне синтаксе, репрезентујући (представљајући) реченице језика које су дефинисане граматиком. Стабло апстрактне синтаксе је састављено од *TerminalExpression* и *NonterminalExpression* класа.
- **Context**
  Садржи информације које су глобалне за интерпретер.
- **AbstractExpression**
  Декларише апстрактну *Interpret(Context)* операцију која је заједничка за све чворове стабла апстрактне синтаксе.
- **TerminalExpression**
  Имплементира *Interpret(Context)* операцију која је придружена *TerminalExpression* симболу (објекту) граматике. Један објекат се захтева за сваки *TerminalExpression* симбол у реченици.
- **NonterminalExpression**
  Једна класа се захтева за свако правило у граматици. Чува референцу на низ *AbstractExpression* објеката[54]. Имплементира *Interpret(Context)* операцију за *NonterminalExpression* симболе (објекте) у граматици.

[54] Када кажемо *AbstractExpression* објекат, мислимо на објекте класа које су изведене из интерфејса *AbstractExpression*.

Пример Interpreter патерна

Кориснички захтев PINT1: *Управа Факултета је поставила захтев Java и VB тиму да припреме понуду која ће бити у складу са следећом граматиком*[55]*:*

```
**********************************************************************************
Izraz = Selekcija
Selekcija = Sekvenca1 | Sekvenca2
Sekvenca1 = "Ponuda", Blanko, Tim , Blanko, "tima:", Blanko, Broj, Blanko, "dinara."
Sekvenca2 = "Java", Blanko, "platforma:", Blanko, JavaPlatforma, Blanko, Kraj
Literal =  Slovo, {Slovo|SpecijalniZnak}
Slovo = "a" | "b" | "v" | ... | Blanko
SpecijalniZnak = ":" | "." | ";"
Blanko = " "
Tim = "Java" | "VB"
JavaPlatforma = "J2ME" | "J2SE" | "J2EE"
Kraj ="." | ";"
Broj = Cifra - "0", {Cifra}
Cifra = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
**********************************************************************************
```

Java тим може да припреми понуду за обе секвенце(Sekvenca1, Sekvencu2). VB тим може да припреми понуду заједну секвенцу (Sekvenca1).

```
/*Улога: Гради стабло апстрактне синтаксе, репрезентујући (представљајући) реченице језика које су дефинисане
граматиком. Стабло апстрактне синтаксе је састављено од TerminalExpression (Literal, Selekcija) и NonterminalExpression
(Sekvenca) класа.
*/
class UpravaFakulteta // Client
 {  SILAB sil;
    ApstraktniIzraz ai;
    UpravaFakulteta(SILAB sil1) {sil = sil1;}
    public static void main(String[] arg)
        { UpravaFakulteta uf;
          String kontekst1, kontekst2;
          JavaTimPonuda jtp = new JavaTimPonuda();
          VBTimPonuda vtp = new VBTimPonuda();
          //*************************************************************************************
          uf = new UpravaFakulteta(jtp);
          kontekst1 =  uf.sil.vratiPonudu(1);
          uf.ai = new Sekvenca1();
          if (uf.ai.Interpretiraj(kontekst1)==true)  {System.out.println("Izraz 1: " + kontekst1 + " je regularan izraz.");}
          else  {System.out.println("Izraz 1: " + kontekst1 + " je neregularan izraz.");}

          kontekst2 = uf.sil.vratiPonudu(2);
          uf.ai = new Sekvenca2();
          if (uf.ai.Interpretiraj(kontekst2)==true)    {System.out.println("Izraz 2: " + kontekst2 + " je regularan izraz.");}
          else  {System.out.println("Izraz 2: " + kontekst2 + " je neregularan izraz.");}
          //*************************************************************************************
          uf = new UpravaFakulteta(vtp);
            kontekst1 =  uf.sil.vratiPonudu(1);
            uf.ai = new Sekvenca1();
            if (uf.ai.Interpretiraj(kontekst1)==true) {System.out.println("Izraz 3: " + kontekst1 + " je regularan izraz.");}
            else {System.out.println("Izraz 3: " + kontekst1 + " je neregularan izraz.");}
        }
 }

/*Улога: Садржи информације (kontekst) које су глобалне за интерпретер и које треба да се интерпретирају.*/
abstract class SILAB // Context
{ abstract String vratiPonudu(int izraz); }

class JavaTimPonuda extends SILAB
{ String vratiPonudu(int izraz)
   {   String kontekst = null;
       if (izraz == 1) kontekst ="Ponuda Java tima: 55000 dinara.";
       if (izraz == 2) kontekst ="Java platforma: J2EE .";
       return kontekst;
   }
}
```

[55] *Izraz* се испитује да ли је регуларан или не. Граматика је описана помоћу BNF (*Backus–Naur Form*) нотације.

```
class VBTimPonuda extends SILAB
{  String vratiPonudu(int izraz)
    { String kontekst = null;
      if (izraz == 1) kontekst ="Ponuda VB tima: 45000 dinara.";
      return kontekst;
    }
}
```

*/*Улога: Декларише апстрактну Interpretiraj(String) операцију која је заједничка за све чворове стабла апстрактне синтаксе. */*

```
abstract class ApstraktniIzraz // AbstractExpression
{  abstract boolean Interpretiraj(String kontekst);}
```

*/*Улога: Имплементира Interpretiraj(String) операцију која је придружена Literal симболу (објекту) граматике.*/*

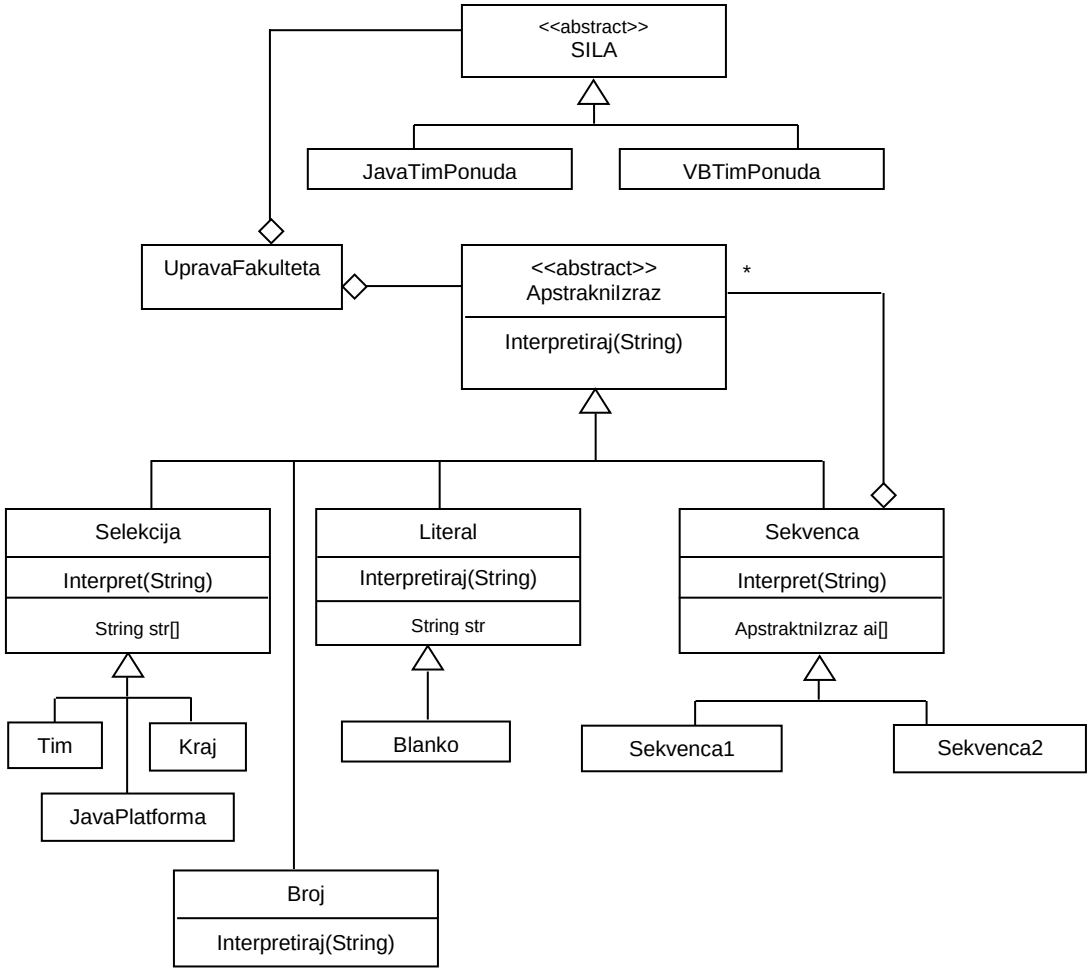
```
class Literal extends ApstraktniIzraz // TerminalExpression
{
  String str;
  Literal(String str1){ str = new String(str1);}

  boolean Interpretiraj(String kontekst)
    { boolean simbol = true;
      char niz[] = kontekst.toCharArray();
      for(int i =0; i<niz.length; i++)
        { if (i==0)
            { if (Character.isLetter(niz[i])==false)
                { simbol = false; System.out.println("Prvi znak nije slovo");return simbol;}
            }
          else
            { if ((Character.isLetter(niz[i])==false) && (niz[i]!=':' &&  niz[i]!='.' && niz[i]!=';'))
                  { simbol = false; System.out.println("Znak: " + niz[i] + " nije slovo ili specijalni znak.");return simbol;}
            }
        }
      if (kontekst.equals(str) == true)  return simbol;
      else   { simbol = false; System.out.println("Kontekst: " + kontekst + " ne odgovara literalu."); return false;}
    }
}

class Blanko extends Literal
{ Blanko () {super(" ");}
  boolean Interpretiraj(String kontekst)
     { boolean simbol = true;
       if (kontekst.equals(str) == true)  return simbol;
       else  { simbol = false; return false;}
    }
}
```

*/*Улога: Чува референцу на низ ApstraktniIzraz објеката. Имплементира Interpretiraj(String) операцију за Sekvenca симболе (објекте) у граматици. */*

```
class Sekvenca extends ApstraktniIzraz // NonterminalExpression
{   ApstraktniIzraz [] ai;

    boolean Interpretiraj(String sek)
       {  String WITH_DELIMITER = "(?<=%1$s)|(?=%1$s)";
          String[] clan = sek.split(String.format(WITH_DELIMITER, " "));
          boolean signal = true; // polazimo od predpostavke da je sekvenca regularna
          for(int i=0;i<clan.length;i++)
               { System.out.println("Clan[" + i + "]: " +  clan[i]);
                 if (ai[i].Interpretiraj(clan[i])==false)
                         { signal = false;
                           return signal;
                         }
               }
          return signal;
       }
}
```

```
/* Полазимо од предпоставке да ће вредности литерала у секвенцама бити унети исправно, у складу са дефинисаном
граматиком, када се прави стабло апстрактне синтаксе.*/
class Sekvenca1 extends Sekvenca
{ Sekvenca1 ()
    { ai = new ApstraktniIzraz[9];
      ai[0] = new Literal("Ponuda"); ai[1] = new Blanko();  ai[2] = new Tim();
      ai[3] = new Blanko(); ai[4] = new Literal("tima:"); ai[5] = new Blanko();
      ai[6] = new Broj(); ai[7] = new Blanko(); ai[8] = new Literal("dinara.");
    }
}

class Sekvenca2 extends Sekvenca
{ Sekvenca2 ()
  { ai = new ApstraktniIzraz[7];
    ai[0] = new Literal("Java"); ai[1] = new Blanko(); ai[2] = new Literal("platforma:");
    ai[3] = new Blanko(); ai[4] = new JavaPlatforma(); ai[5] = new Blanko(); ai[6] = new Kraj();
  }
}

/*Улога: Имплементира Interpretiraj(String)  операцију која је придружена Selekcija симболу (објекту) граматике.*/
class Selekcija extends ApstraktniIzraz // TerminalExpression
{ String[] element;
  boolean Interpretiraj(String tim)
    { boolean signal = false; // predpostavka da ne postoji ni jedan od elemenata.
      for(int i=0;i<element.length;i++)
          if (element[i].equals(tim)==true)
            { signal = true;
              return signal;
            }
      return signal;
    }
}

class Tim extends Selekcija
{ Tim()
  { element = new String[2];  element[0] = new String("Java");  element[1] = new String("VB"); }
}

class JavaPlatforma extends Selekcija
{ JavaPlatforma()
   { element = new String[3];  element[0] = new String("J2ME"); element[1] = new String("J2SE");
     element[2] = new  String("J2EE");
   }
}


class Kraj extends Selekcija
{
  Kraj()
   { element = new String[2]; element[0] = new String(".");  element[1] = new String(";"); }
}

/*Улога: Имплементира Interpretiraj(String)  операцију која је придружена Broj симболу (објекту) граматике.*/
class Broj extends ApstraktniIzraz // TerminalExpression
{ boolean Interpretiraj(String broj)
  { try  { int c = Integer.parseInt(broj);
           System.out.println("Broj1:" + c);
         }  catch(NumberFormatException nfe)  { return false; }

    return true;
  }
}
```

Дијаграм класа примера PINT1



ZPINT1: *Нацртати објектни дијаграм за пример PINT1*.

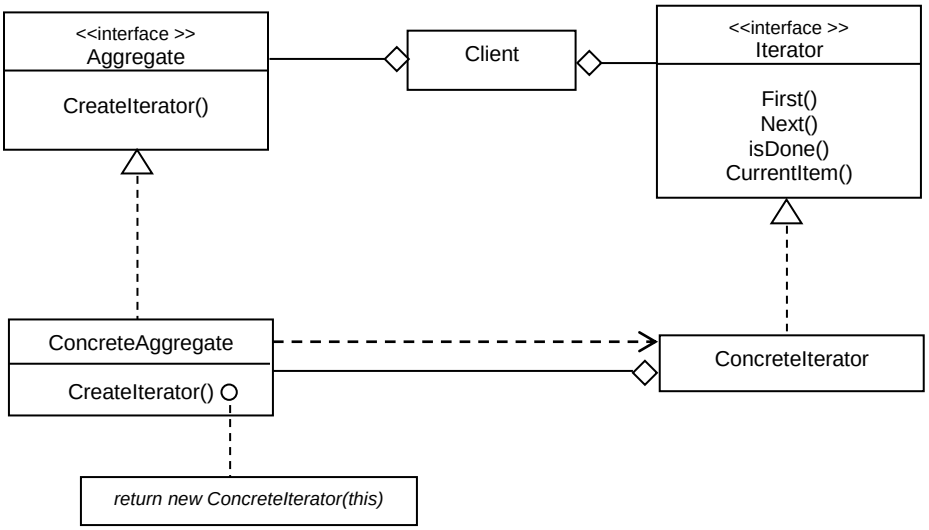
Веза Interpreter патерна и општег облика патерна

Код Interpreter патерна постоје две **СРП**: (*Client,AbstractExpression,ConcreteExpression*) и (*NoterminalExpression, AbstractExpression,ConcreteExpression*). *ConcreteExpression* може бити *TerminalExpression* или *NoterminalExpression.*

ПП4. Iterator патерн

Дефиниција
Обезбеђује начин да приступи елементима агрегатног објекта секвенцијално без излагања његове унутрашње репрезентације.

Појашњење дефиниције
Обезбеђује начин (*ConcreteIterator*) да приступи елементима агрегатног објекта (*ConcreteAggregate*) секвенцијално без излагања његове (*ConcreteAggregate*) унутрашње репрезентације.



Учесници

- **Iterator**
  Дефинише интерфејс за приступ и обилазак елемената агрегираног објекта (ConcreteAggregate).
- **ConcreteIterator**
  Имплементира *Iterator* интерфејс. Чува позицију на текући елемент агрегираног објекта (ConcreteAggregate).
- **Aggregate**
  Дефинише интерфејс за креирање *Iterator* објекта.
- **ConcreteAggregate**
  Имплементира *Aggregate* интерфејс, односно операцију *CreateIterator()* и враћа *ConcreteIterator* објекат.

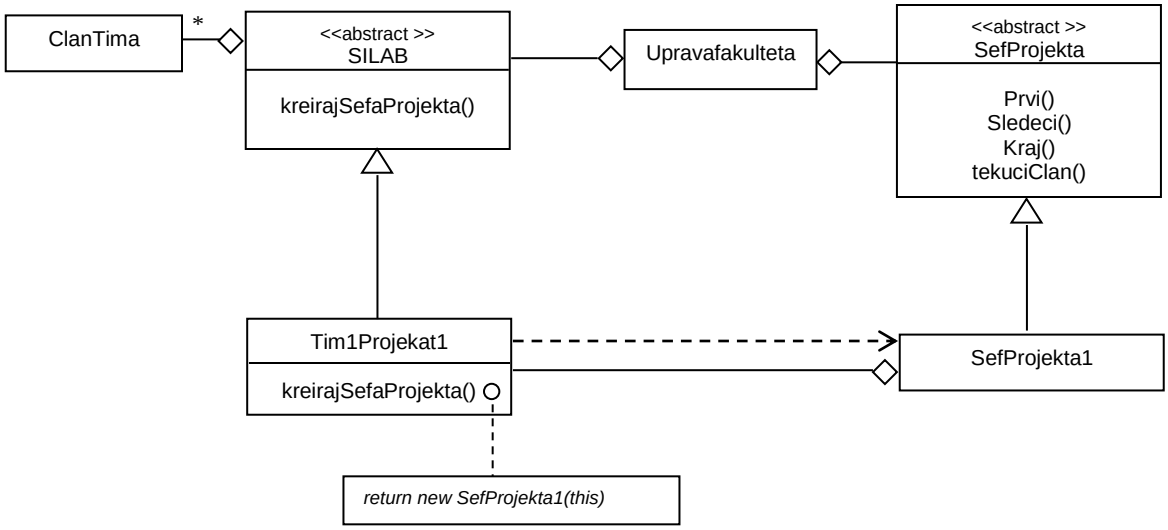
Пример Iterator патерна
Кориснички захтев PITR1: *Управа факултета је тражила од Лабораторије за софтверско инжењерство да одреди чланове тима који ће да раде пројекат и шефа пројекта који ће за сваког члана тима да одреди за шта је он одговоран у пројекту.*

```
class UpravaFakulteta   // Client
{  SILAB sil;
   SefProjekta sp;
  UpravaFakulteta(SILAB sil1){sil = sil1;}
  public static void main(String args[])
   {  Tim1Projekat1 t1p1 = new Tim1Projekat1();
      UpravaFakulteta uf = new UpravaFakulteta(t1p1);
      uf.sp = uf.kreirajSefaProjekta();
      uf.sp.Prvi();
     while (uf.sp.Kraj())
          { if (uf.sp.tekuciClan().vratiImeClana().equals("Dusan Savic"))
              {uf.sp.tekuciClan().odrediOdgovornost(" projektant baze podataka");}
            if (uf.sp.tekuciClan().vratiImeClana().equals("Milos Milic"))
               {uf.sp.tekuciClan().odrediOdgovornost(" projektant korisnickog interfejsa");}
            if (uf.sp.tekuciClan().vratiImeClana().equals("Vojislav Stanojevic"))
               {uf.sp.tekuciClan().odrediOdgovornost(" projektant aplikacione logike");}
```

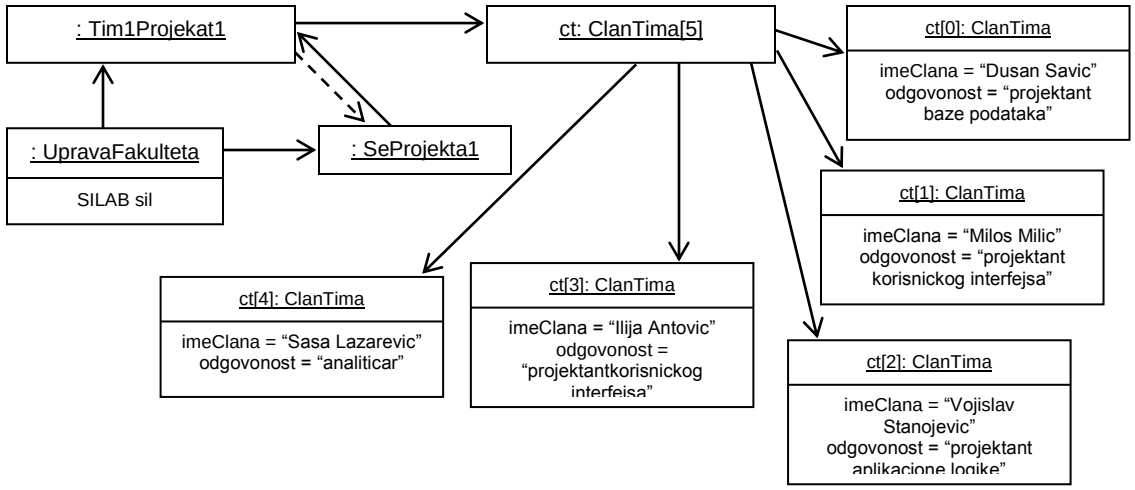
```
            if (uf.sp.tekuciClan().vratiImeClana().equals("Ilija Antovic"))
                {uf.sp.tekuciClan().odrediOdgovornost(" projektant korisnickog interfejsa");}
            if (uf.sp.tekuciClan().vratiImeClana().equals("Sasa Lazarevic"))
                {uf.sp.tekuciClan().odrediOdgovornost(" analiticar");}
            System.out.println("Clan:" + uf.sp.tekuciClan().vratiImeClana() + " Odgovornost:" +
                                        uf.sp.tekuciClan().vratiOdgovornost() );
            uf.sp.Sledeci();
          }
    }

  SefProjekta kreirajSefaProjekta () {return sil.kreirajSefaProjekta();}
 }

/*Улога: Дефинише интерфејс за креирање SefProjekta објекта[56].*/
abstract class  SILAB // Aggregate
{  ClanTima ct[];
   abstract SefProjekta kreirajSefaProjekta();
   ClanTima vratiClana(int indeks) {return ct[indeks];}
   String vratiImeClana(int indeks) {return ct[indeks].imeClana;}
   int vratiBrojClanovaTima(){return ct.length;}
}

/*Улога: Имплементира SILAB интерфејс, односно операцију kreirajSefaProjekta() и враћа ClanTIma објекат. */
class Tim1Projekat1 extends SILAB //  ConcreteAggregate
{ Tim1Projekat1 ()
   {  ct = new ClanTima[5];
      ct[0] = new ClanTima("Dusan Savic");  ct[1] = new ClanTima("Milos Milic");
      ct[2] = new ClanTima("Vojislav Stanojevic"); ct[3] = new ClanTima("Ilija Antovic");
      ct[4] = new ClanTima("Sasa Lazarevic");
   }

   SefProjekta kreirajSefaProjekta() { return new SefProjekta1(this);}
}

class ClanTima
{ String imeClana;
  String odgovornost;
  ClanTima(String imeClana1) {imeClana = new String(imeClana1);}
  void odrediOdgovornost(String odgovornost1) {odgovornost = new String(odgovornost1);}
  String vratiImeClana() {return imeClana;}
  String vratiOdgovornost() {return odgovornost;}
}

/*Улога: Дефинише интерфејс за приступ и обилазак елемената агрегираног објекта (Tim1Projekat1). */
abstract class SefProjekta // Iterator
{ abstract void Prvi();
  abstract void Sledeci();
  abstract boolean Kraj();
  abstract ClanTima tekuciClan();
}

/*Улога: Имплементира SefProjekta интерфејс. Чува позицију на текући елемент агрегираног објекта(Tim1Projekat1). */
class SefProjekta1 extends SefProjekta // ConcreteIterator
{ Tim1Projekat1 t1p1;
  int indeks;

  SefProjekta1(Tim1Projekat1 t1p11) {t1p1 = t1p11;}
  void Prvi() {indeks = 0;}
  void Sledeci() {indeks++;}
  boolean Kraj () {if (indeks < t1p1.vratiBrojClanovaTima()) return true; return false;}
  ClanTima tekuciClan() {return t1p1.vratiClana(indeks);}
  void odrediOdgovornost(String odgovornost){tekuciClan().odrediOdgovornost(odgovornost);}
}
```

[56] Када кажемо *SefProjekta* објекат, мислимо на објекте класа које су изведене из апстрактне класе *SefProjekta*.

Дијаграм класа примера PITR1



Објектни дијаграм примера PITR1



Веза Iterator патерна и општег облика патерна

Код Iterator патерна постоји две **СРП**: *(Client, Iterator, ConcreteIterator)* и *(Client, Aggregate ,ConcreteAggregate)*

ПП5. Mediator патерн

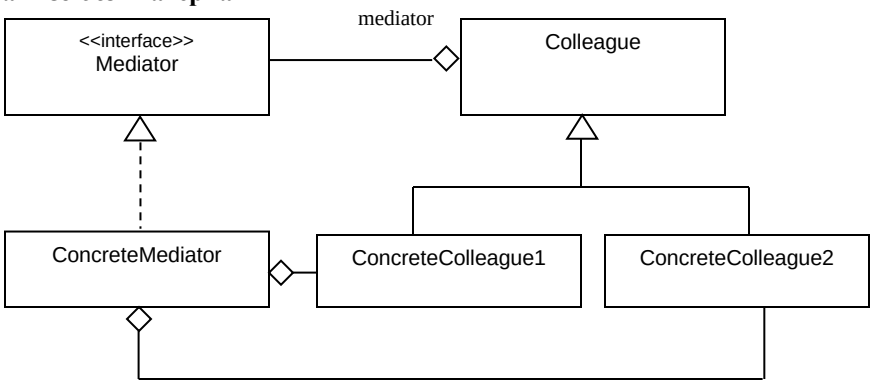
Дефиниција

Дефинише објекат који садржи скуп објеката који су у међусобној интеракцији. Помоћу медијатора се успоставља слаба веза између објеката, тако што се онемогућава њихово међусобно експлицитно референцирање, што омогућава независно мењање њиховог међудејства.

Појашњење дефиниције

Дефинише објекат (*ConcreteMediator*) који садржи скуп објеката (*ConcreteColleague1, ConcreteColleague2*) који су у међусобној интеракцији. Помоћу медијатора се успоставља слаба веза између објеката (*ConcreteColleague1, ConcreteColleague2*), тако што се онемогућава њихово међусобно експлицитно референцирање, што омогућава независно мењање њиховог међудејства.

Структура Mediator патерна



Учесници

- **Mediator**

  Дефинише интерфејс за комуникацију са *Colleaque* објектима.

- **ConcreteMediator**

  Имплементира интеракцију између *Colleaque* објеката. Зна све *Colleaque* објекте између којих настаје интеракција.

- **Colleague**

  Сваки *Colleaque* објекат зна ко је његов *Mediator* објекат. Сваки *Colleaque* објекат комуницира са његовим *Mediator* објектом сваки пут када би иначе требао да комуницира са другим *Colleaque* објектом.

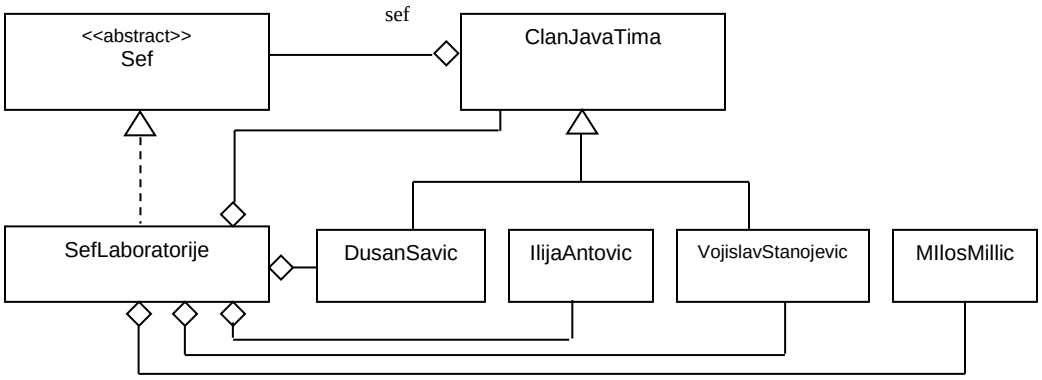
Пример Mediator патерна

Кориснички захтев PMED1: *Шеф Лабораторије за софтверско инжењерство је задужио Душана Савића да кординира процесом припреме понуду за тронивојску апликацију која има: GUI, апликациону логику и базу података. Душан Савић поставља захтев Шефу Лабораторије, у процесу припреме понуде, да пронађе чланове тима који су специјализовани за GUI, апликациону логику и базе података. Шеф Лабораторије је пронашао следеће чланове тима: Милош Милић је специјализован за GUI. Војислав Станојевић је специјализован за аликациону логику. Илија Антовић и Душан Савић су специјализовани за базе података.*

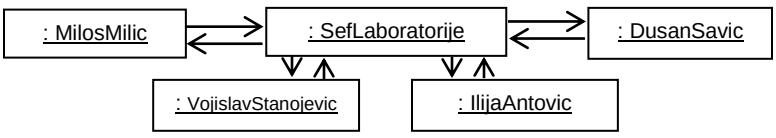
```
// Улога: Дефинише интерфејс за комуникацију са ClanJavaTima објектима.
abstract class Sef   // Mediator
{ abstract  String pripremiPonuduBP();
  abstract  String pripremiPonuduGUI();
  abstract  String pripremiPonuduAL();}

// Улога: Имплементира интеракцију између  ClanJavaTima објеката.  Зна све ClanJavaTima објекте
// између којих настаје интеракција.
class SefLaboratorije extends Sef    //ConcreteMediator
{ DusanSavic ds;  IlijaAntovic ia;  VojislavStanojevic vs;  MilosMilic mm;  ClanJavaTima cjt;
  SefLaboratorije()
    {  ds = new DusanSavic(this); ia = new IlijaAntovic(this); vs = new VojislavStanojevic(this); mm = new MilosMilic(this);}
```

```
  String pripremiPonuduGUI(){return mm.pripremiPonudu();}
  String pripremiPonuduAL(){return vs.pripremiPonudu();}
  String pripremiPonuduBP(){return ia.pripremiPonudu();}

  public static void main(String arg[])
    { SefLaboratorije sf = new SefLaboratorije();
      sf.odrediKordinatoraPripremePonude();
      System.out.println(sf.kordinirajProcesPonude());
    }
void odrediKordinatoraPripremePonude() { cjt = ds;}
String kordinirajProcesPonude(){ return cjt.kordinirajProcesPonude();}
}
// Улога: Сваки ClanJavaTima објекат зна ко је његов Sef објекат. Сваки ClanJavaTima објекат комуницира са његовим Sef
// објектом сваки пут када би иначе требао да комуницира са другим ClanJavaTima објектом.
abstract class ClanJavaTima //Colleage
{  Sef sef;
   ClanJavaTima(Sef sef1) {sef = sef1;}
   String kordinirajProcesPonude() {return "";}
   abstract String pripremiPonudu();
}
class DusanSavic extends ClanJavaTima  //ConcreteColleague1
{ DusanSavic(Sef sef1) {super(sef1);}
  String kordinirajProcesPonude(){ return sef.pripremiPonuduGUI() + " " +
     sef.pripremiPonuduAL() + " " + sef.pripremiPonuduBP();}
  String pripremiPonudu(){return "Dusan Savic: Ponuda - Baza podataka";}}
class IlijaAntovic extends ClanJavaTima  //ConcreteColleague2
{ IlijaAntovic(Sef sef1) {super(sef1);}
  String pripremiPonudu(){return "Ilija Antovic: Ponuda - Baza podataka";}}
class VojislavStanojevic extends ClanJavaTima  //ConcreteColleague3
{ VojislavStanojevic(Sef sef1) {super(sef1);}
  String pripremiPonudu(){return " Vojislav Stanojevic: Ponuda - Aplikaciona logika";}}
class MilosMilic extends ClanJavaTima  //ConcreteColleague4
{ MilosMilic(Sef sef1) {super(sef1);}
  String pripremiPonudu(){return " Milos Milic:  Ponuda - GUI";}}
```

Дијаграм класа примера PMED1



Објектни дијаграм примера PMED1



Веза Mediator патерна и општег облика патерна

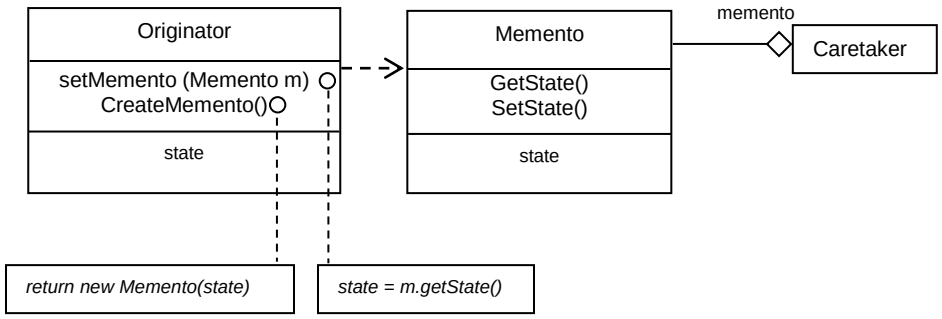
Код Mediator патерна постоји једна **СРП**: (*Colleague,Mediator, ConcreteMediator*).

ПП6. Memento патерн

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

Појашњење дефиниције
Без нарушавања учаурења memento патерн чува интерно стање објекта (*Caretaker*) тако да објекат може бити враћен у то стање касније.

Структура Memento патерна



Учесници

- **Memento**
  Чува интерно стање *Originator* објекта.
- **Originator**
  Креира *Memento* објекат који чува његово интерно стање (*Memento.state = Originator.state*).Користи *Memento* објекат да поврати запамћено стање.
- **Caretaker**
  Одговоран је за памћења *Memento* објекта. Ништа не ради са садржајем *Memento* објекта.

Пример Memento патерна

Кориснички захтев PMEM1: *Управа Факултета је тражила од Јава тима да направи понуду. Управа Факултета је послала прву понуду до Комисије за набавку која је задужена да чува текућу активну понуду (текућа понуда = прва понуда). Управа Факултета је тражила од Комисије да јој омогући да по потреби може преузети прву понуду. Након посматрања прве понуде Управа Факултета је тражила да се понуда промени у делу који се односи на процењено време израде пројекта(процењено време је 6 месеци). Управа је тражила да се смањи време израде пројекта. Јава тим је променио процењено време на 3 месеца и послао је другу понуду, где је значајно повећана цена израде пројекта. Управа Факултета је послала другу понуду до Комисије за набавке (текућа понуда = друга понуда). Након анализе прве и друге понуде (где је цена израде пројекта значајно већа од прве понуде) Управа Факултета је одлучила да изабере прву понуду коју је преузела од Комисије за набавку(текућа понуда = прва понуда).*

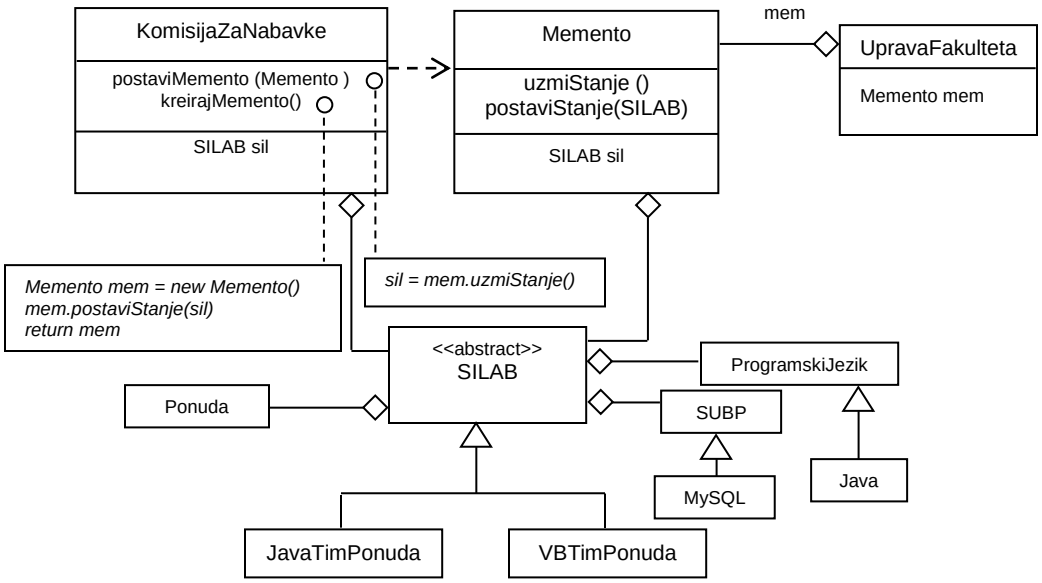
```
// Улога: Чува интерно стање KomisijaZaNabavke објекта.
class Memento  // Memento
{ SILAB sil;
  void postaviStanje(SILAB sil1) { sil=sil1;}
  SILAB uzmiStanje() { return sil;}
}
// Улога: Креира Memento објекат који чува његово интерно стање (sil =  mem.uzmiStanje()).Користи Memento
// објекат да поврати запамћено стање.
class KomisijaZaNabavke // Originator
{SILAB sil;
 KomisijaZaNabavke() {}
 void poveziSaPonudom(SILAB sil1) {sil=sil1;}
```

```
 void postaviMemento(Memento mem) { sil = mem.uzmiStanje();}
 Memento kreirajMemento()
      { Memento mem = new Memento();
        mem.postaviStanje(sil);
        return mem;
      }
}
```

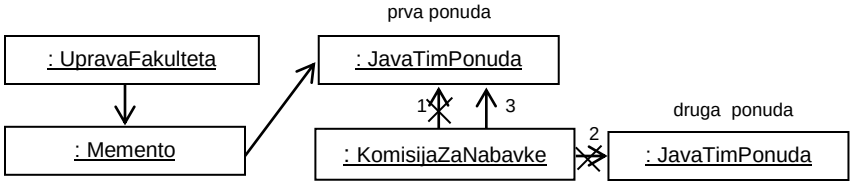
// Улога: Одговоран је за памћења Memento објекта. Ништа не ради са садржајем Memento објекта.

```
class UpravaFakulteta  // CareTaker
{ Memento mem;

  public static void main(String args[])
    {   UpravaFakulteta uf = new UpravaFakulteta();
         /* Prva ponuda */
        JavaTimPonuda jat = new JavaTimPonuda("normalno");
         KomisijaZaNabavke kzn = new KomisijaZaNabavke();
         kzn.poveziSaPonudom(jat);
         uf.mem = kzn.kreirajMemento();
         System.out.println("Ponuda java tima: " + jat.vratiPonudu());
        /* Druga ponuda */
         jat = new JavaTimPonuda("skraceno");
         kzn.poveziSaPonudom(jat);
         System.out.println("Ponuda java tima: " + jat.vratiPonudu());
        /* Vracanje na prvu ponudu */
        kzn.postaviMemento(uf.mem);
        System.out.println("Ponuda java tima: " + kzn.sil.vratiPonudu());
    }
}
abstract class  SILAB
{  ProgramskiJezik  pj;
    SUBP subp;
    String pv; // procenjeno vreme
    Ponuda pon;
    abstract void kreirajProgramskiJezik();
    abstract void kreirajSUBP();
    abstract void kreirajPonudu();
    abstract String vratiPonudu();
 }
class Ponuda {String ponuda;}
class JavaTimPonuda extends SILAB
{ JavaTimPonuda() {pon = new Ponuda();}
  JavaTimPonuda(String duz) {pon = new Ponuda(); kreirajProgramskiJezik(); kreirajSUBP();
   kreirajProcenjenoVreme(duz); kreirajPonudu(); }
  public void kreirajProgramskiJezik(){pj = new Java();}
  public void kreirajSUBP() { subp = new MySQL();}
  public void kreirajPonudu() { pon.ponuda = "Programski jezik-" + pj.vratiProgramskiJezik() + " SUBP-" +
                                    subp.vratiSUBP() + " procenjeno vreme projekta: " + pv;}
  public void kreirajProcenjenoVreme(String duz) {pv = new String(procVreme(duz));}
  String procVreme(String duz)
     { if (duz.equals("skraceno")== true) return "3 meseca";
       if (duz.equals("normalno")== true) return "6 meseci";
       return "";
     }
  public String vratiPonudu(){return pon.ponuda;}
}
interface ProgramskiJezik {String vratiProgramskiJezik();}
class Java implements ProgramskiJezik{ public String vratiProgramskiJezik(){return "Java";}}
interface SUBP {String vratiSUBP();}
class MySQL implements SUBP { public String vratiSUBP(){return "MySQL";}}
```

Дијаграм класа примера PMEM1



Објектни дијаграм примера PMEM1



1. Комисија за набавке се прво повезује са првом понудом.
2. Затим се Комисија за набавке повезује са другом понудом, прекидајући везу са првом понудом. *Memento* објекат чува везу са првом понудом.
3. На крају се Комисија за набавке опет повезује са првом понудом (помоћу *Memento* објекта), прекидајући везу са другом понудом.

Веза Memento патерна и општег облика патерна

Код Memento патерна не постоји ни једна **СРП**.

ПП7. Observer патерн

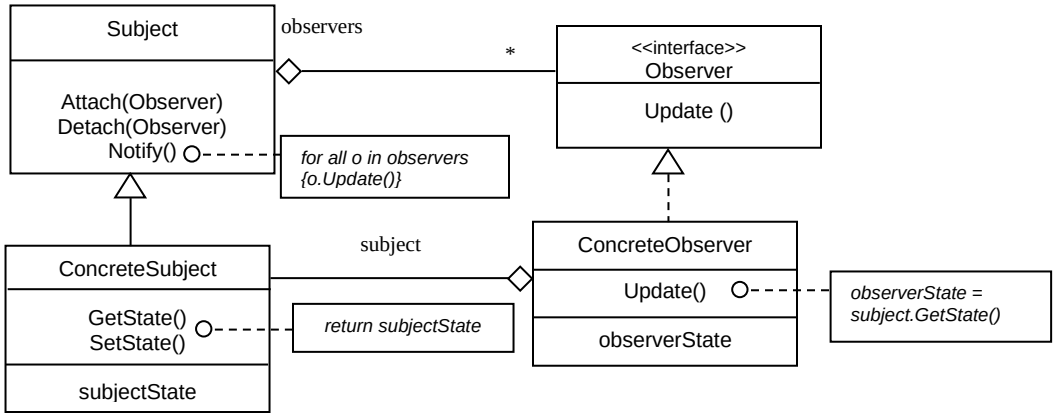
Дефиниција

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.

Појашњење дефиниције

Дефинише један-више зависност између објеката (*Subject->Observer*) тако да промена стања неког објекта (*ConcreteSubject*) утиче аутоматски на промену стања свих других објеката који су повезани са њим(*ConcreteObserver*).

Структура Observer патерна



Учесници

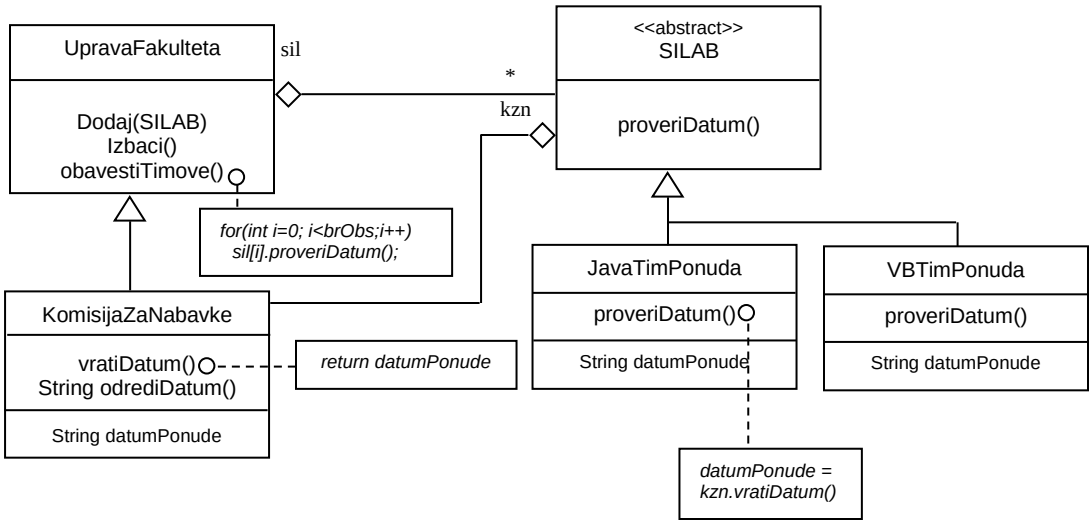
- **Subject**
  Зна ко су његови *Observer* објекти. Обезбеђује интерфејс за повезивање и развезивање Observer објеката.
- **ConcreteSubject (CS)**
  Чува стање на које се постављају *ConcreteObserver* објекти. Шаље обавештење до његових *Observer* објеката када се промени његово стање (*subjectState*).
- **Observer**
  Дефинише интерфејс за промену објеката (*Update()*) који треба да буду обавештени када се промени *Subject* објекат.
- **ConcreteObserver (CO)**
  Чува референцу на *ConcreteSubject* објекат.Чува стање које треба да остане конзистентно са стањем *ConcreteSubject* објекта. Имплементира *Observer* интерфејс како би сачувао његово стање конзистентно са стањем *ConcreteSubject* објекта.

Пример Observer патерна

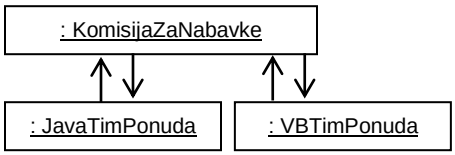
Кориснички захтев POBS1: *Управа Факултета је задужила Комисију за набавке да одреди време до када треба да се пошаљу понуде. Управа Факултета обавештава све тимове да се обрати Комисији за набавке како би видели до ког датума треба послати понуду. Тимови треба да запамте датум до када треба послати понуду.*

```
/*Улога: Зна ко су његови SILAB објекти. Обезбеђује интерфејс за повезивање и развезивање
SILAB објеката.*/
class UpravaFakulteta   // Subject
{
  SILAB sil[]; // ConcreteSubjects
  int brObs; // broj observera

  UpravaFakulteta(){sil = new SILAB[2]; brObs = 0;}
  void Dodaj(SILAB sil1) {sil[brObs++] = sil1;}
  void Izbaci(){}

  void obavestiTimove()
   { for(int i=0; i<brObs;i++)
       sil[i].proveriDatum();
   }

   public static void main(String args[])
        {
          KomisijaZaNabavke kzn = new KomisijaZaNabavke();
          kzn.odrediDatum();
          JavaTimPonuda jat = new JavaTimPonuda(kzn);
          kzn.Dodaj(jat);
          VBTimPonuda vbt = new VBTimPonuda(kzn);
          kzn.Dodaj(vbt);
          kzn.obavestiTimove();
          }
}

/*Улога: Чува стање на које се постављају SILAB објекти. Шаље обавештење до његових SILAB објеката када се промени
његово стање (datumPonude).*/
class KomisijaZaNabavke extends UpravaFakulteta  // ConcreteSubject
{ String datumPonude;
  void odrediDatum(){datumPonude = "22-dec-10";}
  String vratiDatum(){return datumPonude;}
}

/*Улога: Дефинише интерфејс за промену објеката (proveriDatum()) који треба да буду обавештени када се промени
UpravaFakulteta објекат. */
abstract class SILAB // Observer
{ KomisijaZaNabavke kzn;
  String datumPonude;
  SILAB(KomisijaZaNabavke kzn1) {kzn = kzn1;}
  abstract void proveriDatum();
}

/*Улога: Чува референцу на KomisijaZaNabavke објекат. Чува стање које треба да остане конзистентно са стањем
KomisijaZaNabavke објекта. Имплементира Observer интерфејс  како би сачувао његово стање конзистентно са стањем
KomisijaZaNabavke објекта.
*/
class JavaTimPonuda extends SILAB // ConcreteObserver1
{ JavaTimPonuda(KomisijaZaNabavke kzn1){super(kzn1);}
  void proveriDatum() { datumPonude = kzn.vratiDatum();
                  System.out.println("Java tim - Datum do kada treba poslati ponudu: " + datumPonude);}
}
class VBTimPonuda extends SILAB // ConcreteObserver2
{ VBTimPonuda(KomisijaZaNabavke kzn1){super(kzn1);}
  void proveriDatum() { datumPonude = kzn.vratiDatum();
                System.out.println("VB tim - Datum do kada treba poslati ponudu: " + datumPonude);}
}
```

Дијаграм класа примера POBS1



Објектни дијаграм примера POBS1



Веза Observer патерна и општег облика патерна

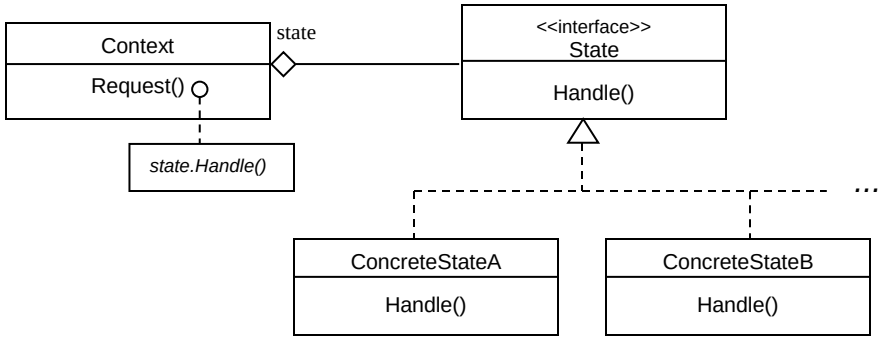
Код *Observer* патерна постоји једна **CPП**: (*Subject,Observer, ConcreteObserver*).

ПП8. State патерн

Дефиниција
Допушта објекту да промени понашање када се мења његово интерно стање.

Појашњење дефиниције
Допушта објекту (*Context*) да промени понашање (*ConcreteStateA*, *ConcreteStateB*) када се мења његово интерно стање (*state*).

Структура State патерна



Учесници

- **Context**
  Дефинише интерфејс за клијента.Садржи појављивање *State* подкласе која дефинише текуће стање *Context* објекта.
- **State**
  Дефинише интерфејс за *Context* објекат, односно понашање које се мења у зависности од промене стања *Context* објекта.
- **ConcreteState**
  Свака *ConcreteState* подкласа имплементира одређено понашање у зависности од стања Context објекта.

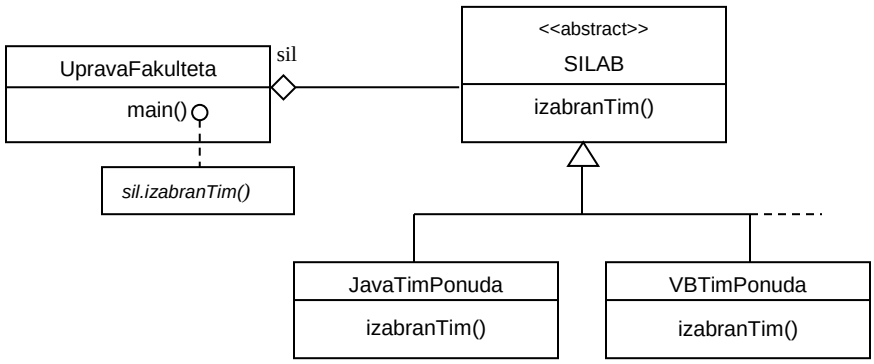
Пример State патерна
Кориснички захтев PST1: *Управа Факултета ће у зависности од броја месеци потребних за израду пројекта (контекста) да одреди да ли ће прихватити понуду Java или VB тима. Уколико је број месеци потребан за израду пројекта већи од 6 месеци, пројекат ће радити Java тим, иначе ће пројекат радити VB тим.*

```
/*Улога: Дефинише интерфејс за клијента. Садржи појављивање (sil) SILAB подкласе која дефинише текуће стање
UpravaFakulteta објекта. */
class UpravaFakulteta   // Context
{ SILAB sil; // State
  double raspSredstva;

  public static void main(String args[])
      {  UpravaFakulteta uf = new UpravaFakulteta();
         uf.raspSredstva = Double.parseDouble(args[0]);
         if (uf.raspSredstva > 6)
           { uf.sil = new JavaTimPonuda();  // ConcreteState1
           }
         else
           { uf.sil = new VBTimPonuda(); // ConcreteState2
           }
     System.out.println(uf.sil.izabranTim());
   }
}
```

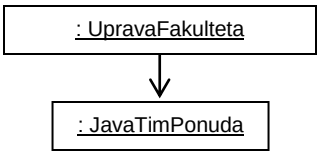
```
/*Улога: Дефинише интерфејс за UpravaFakulteta објекат, односно понашање које се мења у зависности од  промене стања
UpravaFakulteta објекта.*/
abstract class SILAB // State
{ abstract String izabranTim();}
```

```
/*Улога: Свака SILAB подкласа имплементира одређено понашање у зависности од стања  UpravaFakulteta објекта. */
class JavaTimPonuda extends SILAB // ConcreteState1
{ String izabranTim() {return "Java tim!";}}

class VBTimPonuda extends SILAB // ConcreteState2
{ String izabranTim() {return "VB tim!";}}
```

Дијаграм класа примера PST1



Објектни дијаграм примера PST1



Уколико је број месеци потребан за израду пројекта већи од 6, *UpravaFakulteta* објекат ће показивати на *JavaTimPonuda* објекат, иначе ће показивати на *VBTimPonuda* објекат. У наведеном примеру број месеци потребан за израду пројекта је већи од 6.

Веза State патерна и општег облика патерна

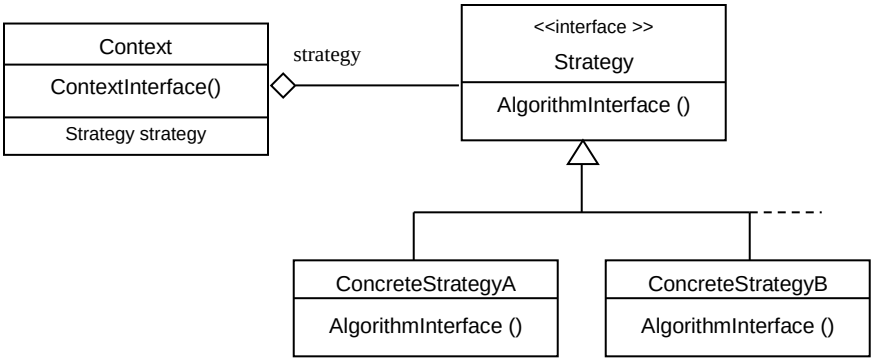
Код *State* патерна постоји једна **СРП**: (*Context,State, ConcreteState*).

ПП9. Strategy патерн

Дефиниција
Дефинише фамилију алгоритама, учаурује сваки од њих и обезбеђује да они могу бити замењиви. Strategy патерн омогућава промену алгоритма независно од клијената који га користе.

Појашњење 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 клијената (*Context*) који га користи.

Структура Strategy патерна



Учесници

- **Context**
  Он садржи референцу на *Strategy* интерфејс.*Context* објекат је конфигурисан са *ConcretyStrategy* објектом. Може да дефинише интерфејс, који омогућава *Strategy* објекту приступ до његових података.
- **Strategy**
  Декларише интерфејс који је заједнички за све подржане алгоритме. *Context* објекат користи овај интерфејс да позове алгоритам који је дефинисан преко *ConcreteStrategy* објекта.
- **ConcreteStrategy**
  Имплементира алгоритам коришћењем *Strategy* интерфејса.

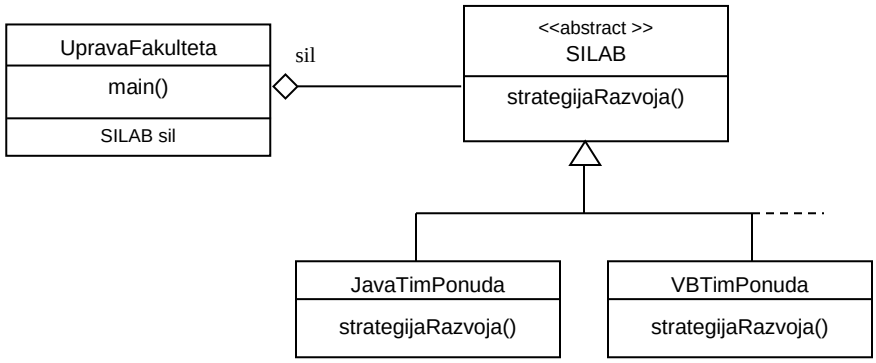
Пример Strategy патерна

Кориснички захтев PSTR1: *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.*

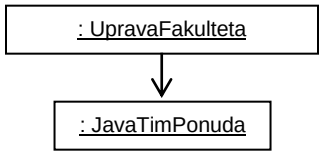
```
/*Улога: Он садржи референцу на SILAB интерфејс. UpravaFakulteta објекат је конфигурисан са JavaTipPonuda или са
VBTimPonuda објектом. Може да дефинише интерфејс, који омогућава JavaTipPonuda или VBTimPonuda објекту приступ
до његових података.*/
class UpravaFakulteta   // Context
{
  SILAB sil; // State
  UpravaFakulteta() { sil = null;}

  public static void main(String args[])
    {  UpravaFakulteta uf = new UpravaFakulteta();
       String tim = args[0];
       if (tim.equals("Java")) uf.sil = new  JavaTimPonuda();
       if (tim.equals("VB")) uf.sil = new  VBTimPonuda();
       System.out.println(uf.sil.strategijaRazvoja());
    }
}
```

```
/*Улога: Декларише интерфејс који је заједнички за све подржане алгоритме. UpravaFakulteta објекат користи овај
интерфејс да позове алгоритам који је дефинисан преко JavaTipPonuda или VBTimPonuda објекта.*/
abstract class SILAB // Strategy
{ abstract String strategijaRazvoja ();
}
/*Улога: Имплементира алгоритам коришћењем SILAB интерфејса.*/
class JavaTimPonuda extends SILAB // ConcreteStrategyA
{ String strategijaRazvoja() { String pom = "Use case driven strategija.";
                               pom = pom +   " Larmanova metoda.";
                               pom = pom +  " Iterativno-inkrementalni model.";
                               return pom;}
}

class VBTimPonuda extends SILAB // ConcreteStrategyB
{ String strategijaRazvoja() { String pom =  "Test driven strategija.";
                               pom = pom +  " Extreme programming .";
                               pom = pom +  " Iterativno-inkrementalni model.";
                               return pom;}
}
```

Дијаграм класа примера PSTR1



Објектни дијаграм примера PSTR1



UpravaFakulteta објекат ће показивати на *JavaTimPonuda* или *VBTimPonuda* објекат, у зависности од тога који тим изабере. У наведеном примеру изабран је Јава тим.

Веза Strategy патерна и општег облика патерна

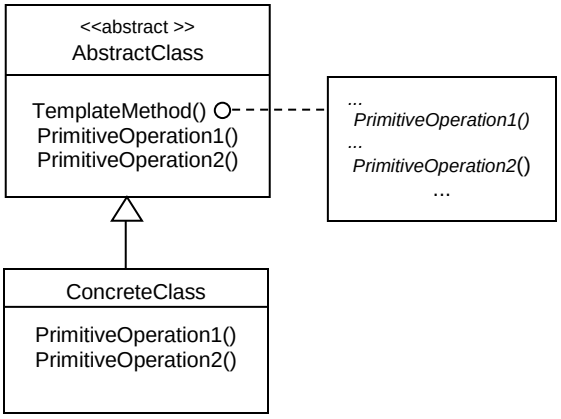
Код *Strategy* патерна постоји једна **СРП**: (*Context,Strategy, ConcreteStrategy*).

ПП10. Template method патерн

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

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

Структура Template method патерна



Учесници

- **AbstractClass**
  Дефинише апстрактне примитивне операције (*PrimitiveOperation1(), PrimitiveOperation2()*) које *ConcreteClass* подкласа имплементира. Имплементира *TemplateMethod()* операцију дефинисањем скелета алгоритма. *TemplateMethod()* операција позива примитивне операције (*PrimitiveOperation1(), PrimitiveOperation2()*) које су дефинисане у класи *AbstractClass.*
- **ConcreteClass**
  Имплементира примитивне операције које описују специфична понашања подкласа.

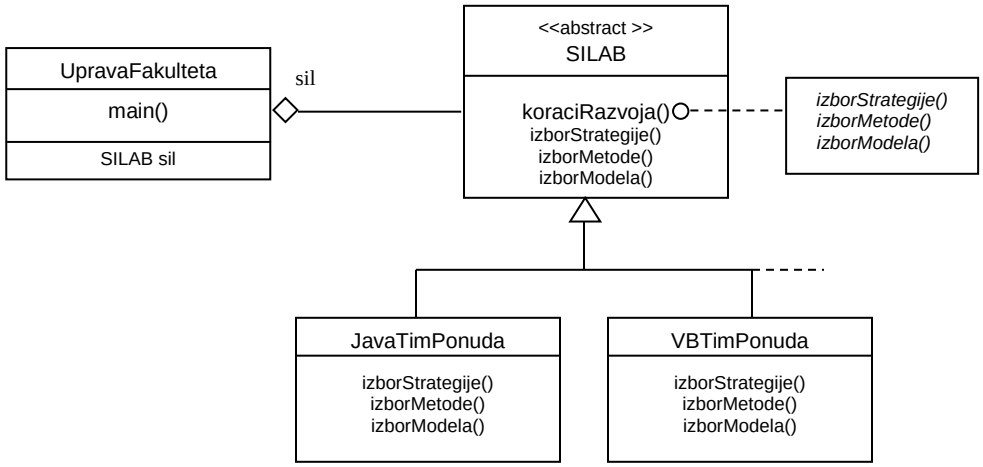
Пример Template method патерна
Кориснички захтев РТМ1: *Управа Факултета захтева од Лабораторије за софтверско инжењерство да предложи најважније кораке у развоју софтверског система. Java и VB тим ће предложити конкретне кораке који ће се извшити у развоју софтверског система. Управа Факултета одређује тим који ће развијати софтверски систем.*

```
class UpravaFakulteta   // Client
{ SILAB sil;
  UpravaFakulteta (SILAB sil1){sil=sil1;}
  public static void main(String args[])
       { JavaTimPonuda jtp = new  JavaTimPonuda();
         UpravaFakulteta uf = new UpravaFakulteta (jtp);
       System.out.println(uf.sil.koraciRazvoja());
       }
}
```

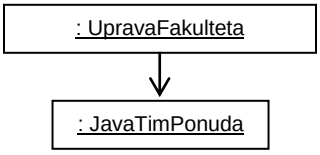
```
/* Улога: Дефинише апстрактне примитивне операције (izborStrategije(), izborMetode(), izborModela()) које SILAB подкласа имплементира. Имплементира koraciRazvoja() операцију дефинисањем скелета алгоритма.  Операција koraciRazvoja() позива  примитивне операције (izborStrategije(), izborMetode(), izborModela()) које су дефинисане у класи SILAB.*/
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();
}

/* Улога: Имплементира примитивне операције које описују специфична понашања подкласа.*/
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.";}
}
```

Дијаграм класа примера PTM1



Објектни дијаграм примера PTM1



Веза Template method патерна и општег облика патерна

Код *Template method* патерна постоји једна **СРП**[57]: (*Client*,*AbstractClass* , *ConcreteClass*).

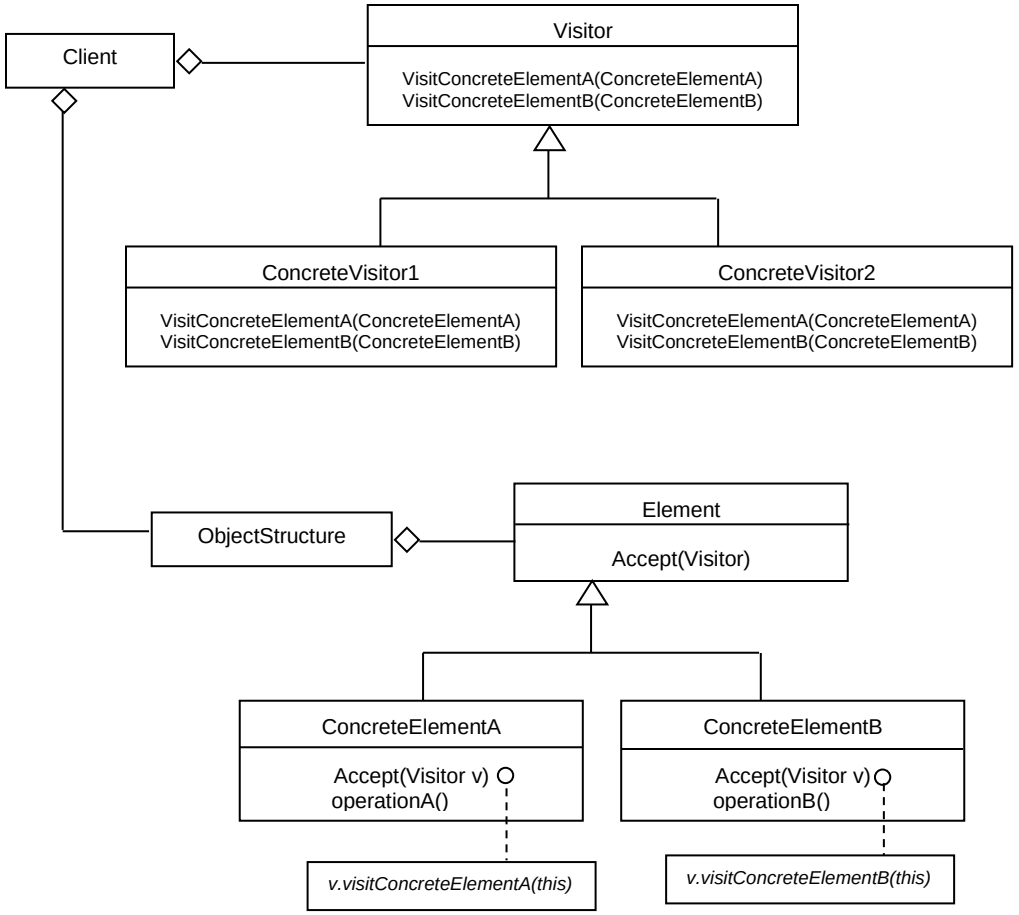
[57] У структури *Template method* патерна се не види експлицитно *Client* класа, иако она постоји као корисник *TemplateMethod()* операције.

ПП11. Visitor патерн

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

Појашњење дефиниције:
Представља операцију класе *ObjectStructure* која се извршава на елементима објектне структуре *(ConcreteElementA, ConcreteElementB).* Visitor омогућава да се дефинише нова операција (имплементирањем операција нове подкласе класе *Visitor*) без промене класа или елемената *(ConcreteElementA, ConcreteElementB)* над којима она (операција) оперише.

Структура Visitor патерна



Учесници:

- **Visitor –** декларише *Visit*() операцију *(VisitConcreteElementA(), VisitConcreteElementB())* за сваку *ConcreteElement* класу објектне структуре *(ConcreteElementA, ConcreteElementB).* Наведена операција као аргумент садржи *ConcreteElement* објекат што омогућава *Visitor* објекту да може приступати и мењати тај *ConcreteElement* објекат.
- **ConcreteVisitor** – имплементира сваку операцију *(VisitConcreteElementA(), VisitConcreteElementB())* декларисану преко Visitor интерфејса.
- **ObjectStructure** - чува елементе објектне структуре и обезбеђује интерфејс који допушта *Visitor* објекту да види наведене елементе. Може бити или композиција или колекција као што су листа или скуп.
- **Element** – дефинише операцију *Accept*() која прихвата *Visitor* објекат као аргумент.
- **ConcreteElement** - имплементира операцију *Accept*().

Пример Visitior патерна

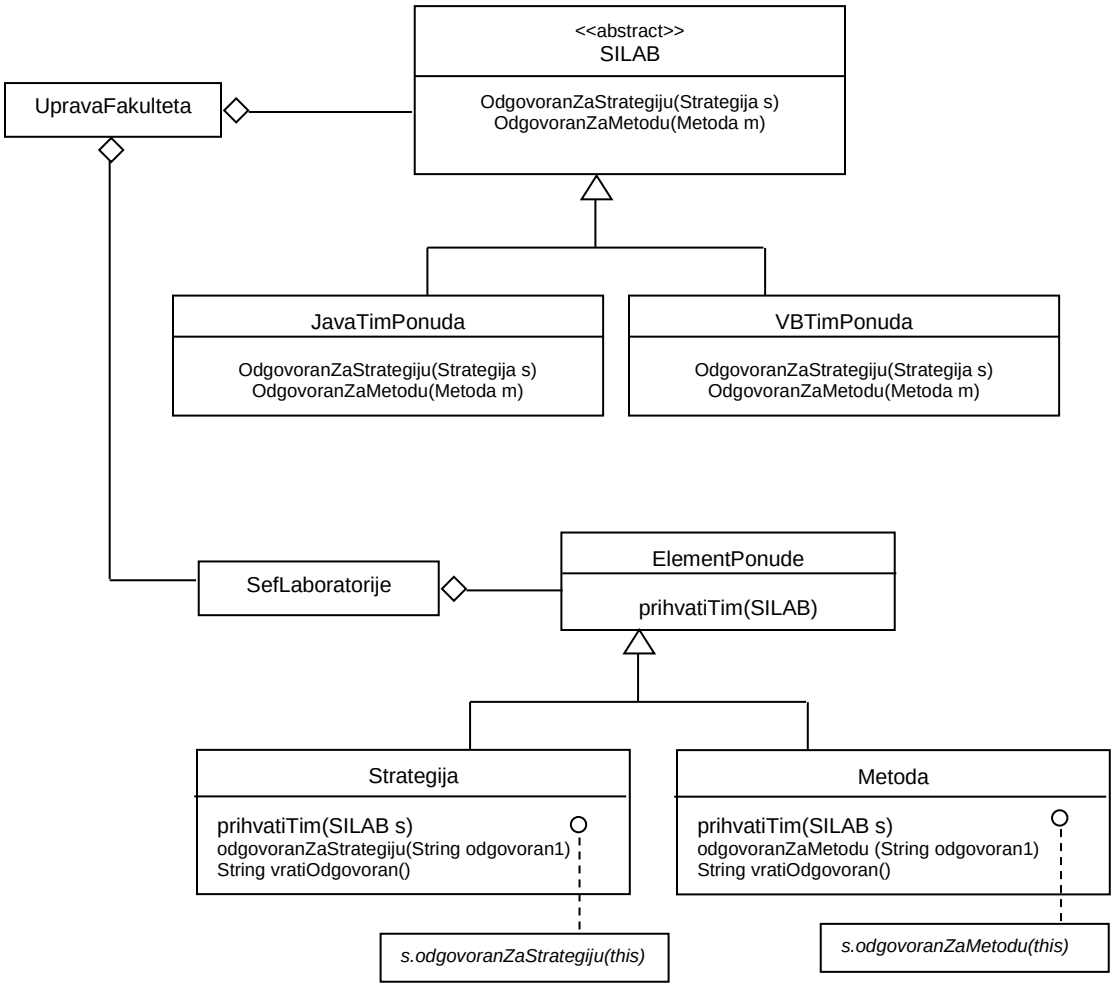
Кориснички захтев PVS1: *Управа Факултета одређује да ће понуду стратегије и методе развоја софтвера урадити Јава тим. Управа Факултета позива шефа Лабораторије за СИ да одреди чланове Јава тима који су одговорни за одређивање стратегије и методе развоја софтвера.*/*

```
class UpravaFakulteta // Client
 {
    SILAB sil;
    SefLaboratorije sl;
    UpravaFakulteta(SILAB sil1) {sil = sil1;}

    public static void main(String[] arg)
    {  UpravaFakulteta uf;
       JavaTimPonuda jtp = new JavaTimPonuda();
       uf = new UpravaFakulteta(jtp);
       uf.sl = new SefLaboratorije();
       uf.sl.odrediOdgovornost(uf.sil);
      // Ukoliko bi uprava izabrala VB tim da pripremi ponudu
      // VBTimPonuda vbt = new VBTimPonuda();
      // uf.sil = vbt;
      // uf.sl.odrediOdgovornost(uf.sil);

  }
/* Улога: чува елементе објектне структуре и обезбеђује интерфејс који допушта SILAB објекту[58] да види наведене елементе. */
class SefLaboratorije  // ObjectStructure
{ ElementPonude el[];

  SefLaboratorije()
   {  el = new ElementPonude[2];
      el[0] = new Strategija();
      el[1] = new Metoda();
   }

  void odrediOdgovornost(SILAB s)
   {  el[0].prihvatiTim(s);
      el[1].prihvatiTim(s);
      System.out.println("Odgovoran za strategiju: " + el[0].vratiOdgovoran());
      System.out.println("Odgovoran za metodu: " + el[1].vratiOdgovoran());
    }
}

/* Улога: дефинише операцију prihvatiTim() која прихвата SILAB објекат као аргумент. */
abstract class ElementPonude   // Element
{ String odgovoran;
  abstract void prihvatiTim (SILAB s);
  abstract String vratiOdgovoran();
}

/* Улога: имплементира операцију PrihvatiTim().*/
class Strategija extends ElementPonude  // ConcreteElement1
{
  void prihvatiTim (SILAB s) {s.OdgovoranZaStrategiju(this);}
  void odgovoranZaStrategiju(String odgovoran1) {odgovoran = odgovoran1;}
  String vratiOdgovoran() { return odgovoran;}
}

class Metoda extends ElementPonude // ConcreteElement2
{
  void prihvatiTim (SILAB s) {s.OdgovoranZaMetodu(this);}
  void odgovoranZaMetodu(String odgovoran1) {odgovoran = odgovoran1;}
  String vratiOdgovoran() { return odgovoran;}
}
```

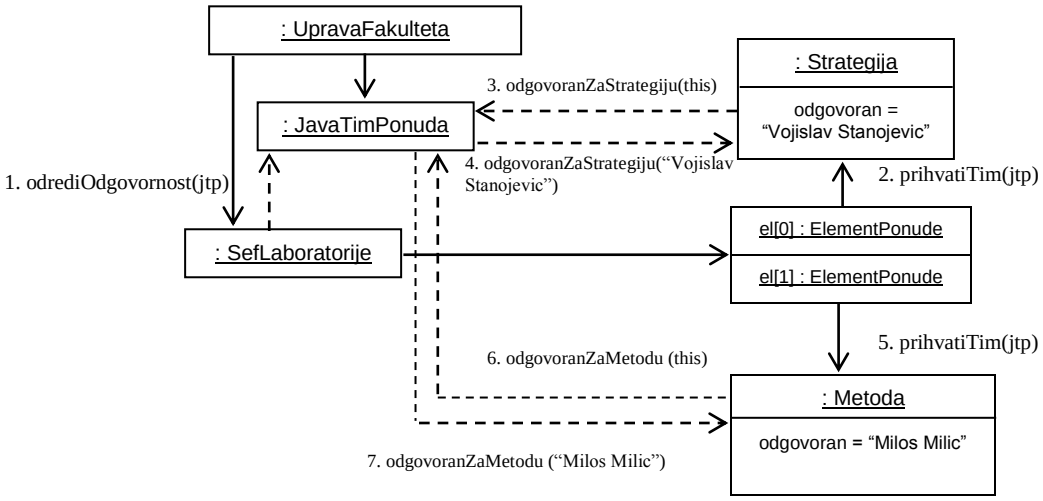
[58] Када кажемо SILAB објекат, мислимо на објекте класа које су изведене из апстрактне класе SILAB, будући да апстрактне класе не могу имати објекте.

```
/* Улога: декларише Visit() операцију (odgovoranZaStrategiju(),odgovoranZaMetodu()), VisitConcreteElementB()) за сваку класу
објектне структуре (Strategija, Metoda). Наведена операција као аргумент садржи  Stategija или Metoda објекат што
омогућава SILAB објекту да може приступати и мењати тај Stategija или Metoda објекат.*/
abstract class SILAB // Visitor
{
  abstract void OdgovoranZaStrategiju(Strategija s);
  abstract void OdgovoranZaMetodu(Metoda m);
}

/* Улога: имплементира сваку операцију (odgovoranZaStrategiju(),odgovoranZaMetodu()) декларисану преко SILAB
интерфејса.*/
class JavaTimPonuda extends SILAB // ConcreteVisitor1
{
  void OdgovoranZaStrategiju(Strategija s){s.odgovoranZaStrategiju("Vojislav Stanojevic");}
  void OdgovoranZaMetodu(Metoda m){m.odgovoranZaMetodu("Milos Milic");}
}


class VBTimPonuda extends SILAB // ConcreteVisitor2
{
  void OdgovoranZaStrategiju(Strategija s){s.odgovoranZaStrategiju("Dusan Savic");}
  void OdgovoranZaMetodu(Metoda m){m.odgovoranZaMetodu("Ilija Antovic");}
}
```

Дијаграм класа примера PVS1



Објектни дијаграм примера PVS1



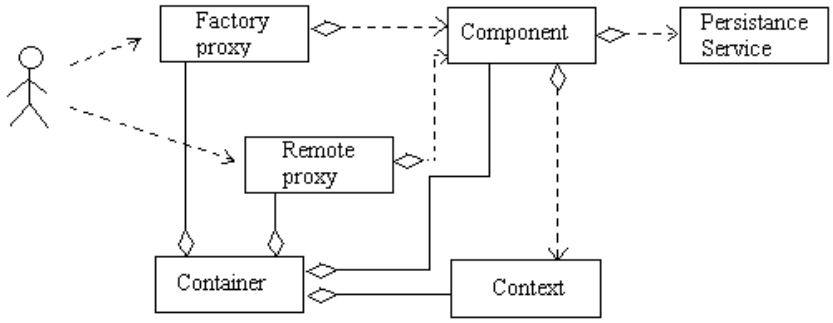
Веза Visitor патерна и општег облика патерна

Код *Visitor* патерна постоји две **СРП:** (*Client,Visitor*, *ConcreteVisitor*) и *(ObjectStructure*, *Element*, *ConcreteElement.)*

5.2 ECF и MVC патерни

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 10) 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.



Слика 10: ECF патерн

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

MVC (Model-View-Controller) патерни

MVC (Слика 11) је макроархитектурни патерн, који дели софтверски систем у три дела:
а) **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.

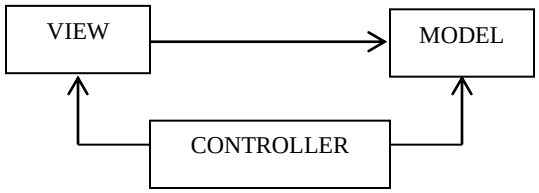
Правило 2: Контролер је диспечер који прима захтев од view-a и преусмерава га до model-a.

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 јр **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.



Слика 11: 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();
              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);
              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();}
}
```

5.3 . Имплементациони патерни – програмски идиоми

Имплементациони патерни односно програмски идиоми су патерни који се користе у конкретном програмском језику како би поједноставили програмирање. Они нису директно повезани са општим обликом GOF патерна пројектовања. Навешћу неколико примера у Јави:

1. Коришћење for петље код приказа елемената колекције

Почетна верзија програма

```
import java.util.*;

class Idiom1D
 {   public static void main(String[] arg)
      {   List list = new ArrayList();
          String a = "Pera";
          String b = "Laza";
          list.add(a);
          list.add(b);
          String s;
          for(Iterator i = list.iterator(); i.hasNext();)
             { s = (String) i.next(); System.out.print(s + " ");}
     }
}
```

Поједностављена верзија програма

```
import java.util.*;

class Idiom1P
 {  public static void main(String[] arg)
     {   List<String> list  = new ArrayList <String>();
         String a = "Pera";
         String b = "Laza";
         list.add(a);
         list.add(b);
         for(String s : list)  System.out.print(s + " ");
    }
}
```

2. Скраћивање наредбе System.out.println

Почетна верзија програма

```
class Idiom2D
 {  public static void main(String[] arg)
     {  System.out.println("Danas je lep dan!");  }
 }
```

Поједностављена верзија програма

```
import static java.lang.System.out;

class Idiom2P
 {   public static void main(String[] arg) { out.println("Danas je lep dan!");  }
 }
```

3. Коришћење Scanner класе уместо BafferedReader класе

Почетна верзија програма

```
import java.io.*;

class Idiom3D
 {   public static void main  (String[] arg) throws Exception
      {    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
           System.out.println("Unesi rec:");
           String rec = br.readLine();
           System.out.println("Uneta rec je : " + rec);
      }
}
```

Поједностављена верзија програма

```
import java.util.Scanner;
import static java.lang.System.out;

class Idiom3P
 {   public static void main  (String[] arg)
       {  Scanner sc = new Scanner(System.in);
          out.println("Unesi rec:");
          String rec = sc.nextLine();
          out.println("Uneta rec je : " + rec);
       }
 }
```

4. Штампање садржаја низа

Почетна верзија програма

```
import java.util.Arrays;
import java.util.List;
import java.util.Iterator;

class Idiom4D
 {   public static void main  (String[] arg) throws Exception
       {  String[] strArray = new String[]{"Pera","Mika","Laza"};
          List list = Arrays.asList(strArray);
          Iterator itr = list.iterator();
          while(itr.hasNext())  { System.out.print(itr.next() + " ");}
       }
}
```

Поједностављена верзија програма

```
import java.util.Arrays;

class Idiom4P
  {   public static void main  (String[] arg) throws Exception
        {  String[] imena = new String[]{"Pera","Mika","Laza"};
           System.out.print(Arrays.asList( imena ));
        }
}
```

6. Веза општег облика патерна и других концепата

6.1 Фаза прикупљања захтева и патерни

У процесу развоја софтвера фаза прикупљања захтева дефинише својства и услове које софтверски систем или шире гледајући пројекат треба да задовољи [Larman]. Захтеви се могу поделити као функционални и нефункционални. Функционални захтеви описују функције које софтверски систем треба да обезбеди а које се односе на логику пословног система (за који се развија софтверски систем). Нефункционални захтеви су *употребљивост, поузданост, подржаност система, перформансе ,..., итд.* Функционални захтеви се могу описати преко **случаја коришћења *и*** корисничких прича.

На основу искустава које смо имали у коришћењу случајева коришћења у развоју бројних софтверских пројеката направили смо један скуп текстуалних генеричких случаја коришћења (*Креирај, Обриши, Претражи, Промени, Сторнирај,...*) који се могу користити у фази прикупљања захтева у развоју различитих софтверских система. Овде ћу навести генерички случај коришћења који се односи на креирање објеката произвољне класе (*Рачун*, *Производ, Партнер*...).

Генерички случај коришћења: ***Креирање новог објекта***

_____

Назив СК
Креирање _____

Актори СК
_____

Учесници СК
_____ и систем (програм)

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

Основни сценарио СК

1. _____ **позива** систем да креира _____. (АПСО)
2. Систем **креира** _____. (СО)
3. Систем **приказује** _____ _____ и поруку: "Систем је креирао _____". (ИА)
4. _____ **уноси** податке у _____. (АПУСО)
5. _____ **контролише** да ли је коректно унео податке у _____. (АНСО)
6. _____ **позива** систем да запамти податке о _____. (АПСО)
7. Систем **памти** податке о _____. (СО)
8. Систем **приказује** _____ запамћени _____ и поруку: "Систем је запамтио _____". (ИА)

Алтернативна сценарија

3.1 Уколико систем не може да креира _____ он приказује _____ поруку: "Систем не може да креира _____". Прекида се извршење сценариа. (ИА)

8.1 Уколико систем не може да запамти податке о _____ он приказује _____ поруку "Систем не може да запамти _____". (ИА)

Уколико се на месту празних поља унесу конкретни актор (Продавац) и објекат који се креира (Рачун) добија се следећи случај коришћења.

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

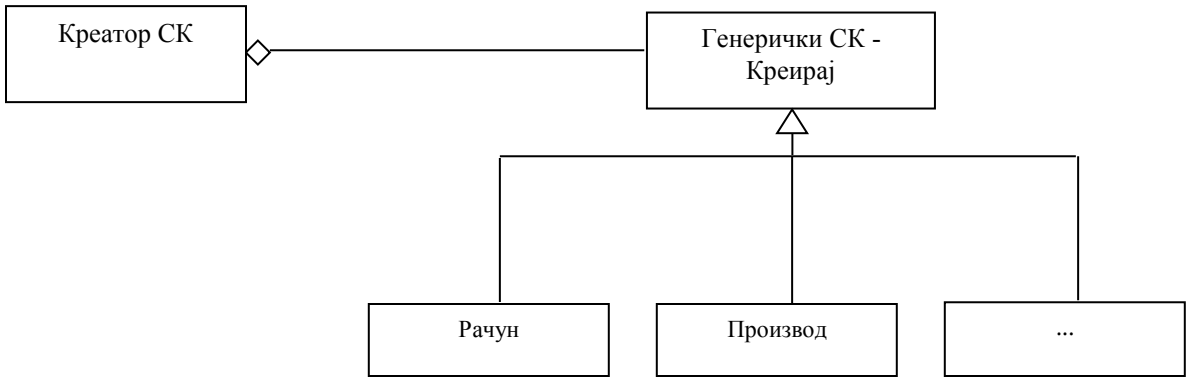
1. Продавац **позива** систем да креира нови рачун. (АПСО)
2. Систем **креира** нови рачун. (СО)
3. Систем **приказује** продавцу нови рачун и поруку: "Систем је креирао нови рачун". (ИА)
4. Продавац **уноси** податке у нови рачун. (АПУСО)
5. Продавац **контролише** да ли је коректно унео податке у нови 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)

Алтернативна сценарија

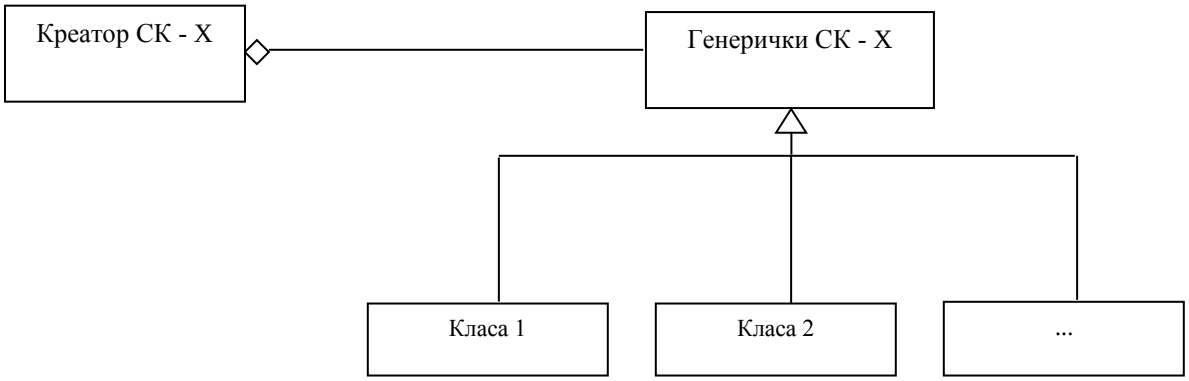
3.1 Уколико систем не може да креира нови рачун он приказује продавцу поруку: "Систем не може да креира нови рачун". Прекида се извршење сценариа. (ИА)

8.1 Уколико систем не може да запамти податке о рачуну он приказује продавцу поруку "Систем не може да запамти рачун". (ИА)

Уколико би генерички случај коришћења *Креирај* представили преко структуре решења патерна у општем смислу добили би следеће:



Уколико би желели да представимо било који генерички случај коришћења преко структуре решења GOF патерна пројектовања добили би следеће:

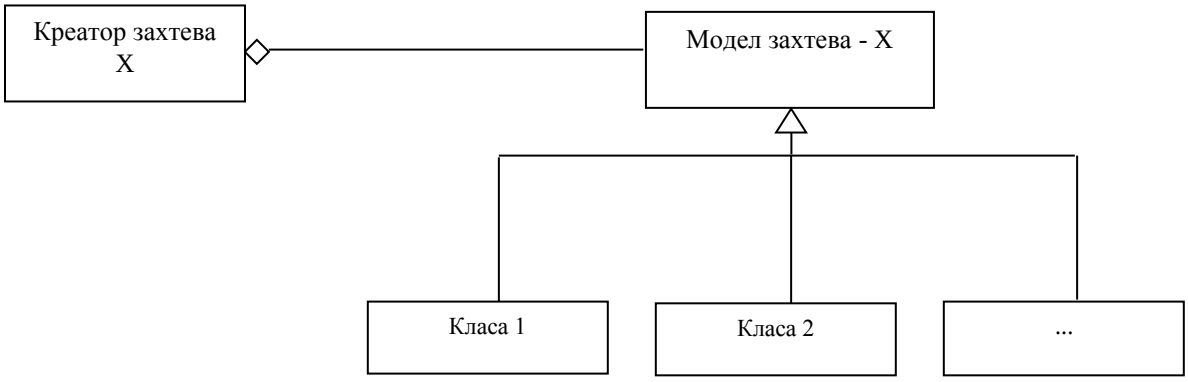


X може бити *Креирај, Обриши, Претражи, Промени, Сторнирај,...*, или неки други случај коришћења.

Класа 1, Класа 2, ... може бити *Рачун*, *Производ, Партнер*,..., или нека друга доменска класа[59].

[59] Доменска класа може бити сложена, што значи да се може састојати из више других доменских класа. Проста доменска класе се не састоји из других доменских класа.

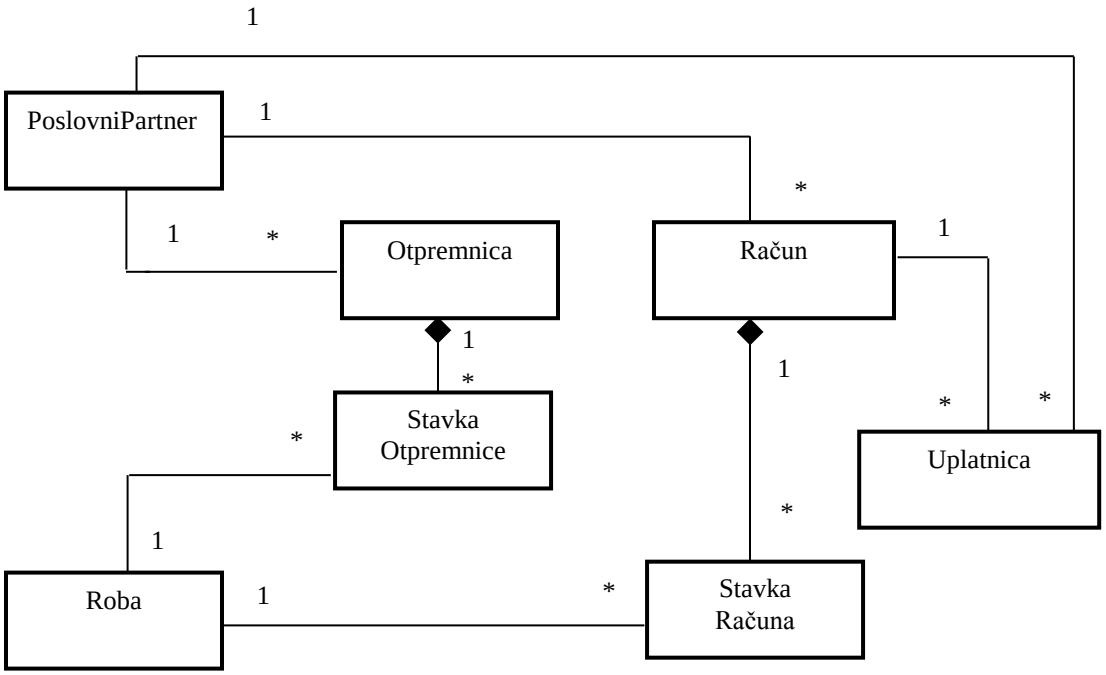
На сличан начин се могу представити и други модели захтева који описују функционалне захтеве, преко структуре решења патерна у општем смислу, ако имају генеричко својство да се могу применити за различите доменске класе.



X може бити било који функционални захтев.
Класа 1, Класа 2, ... може бити било која проста или сложена доменска класа.

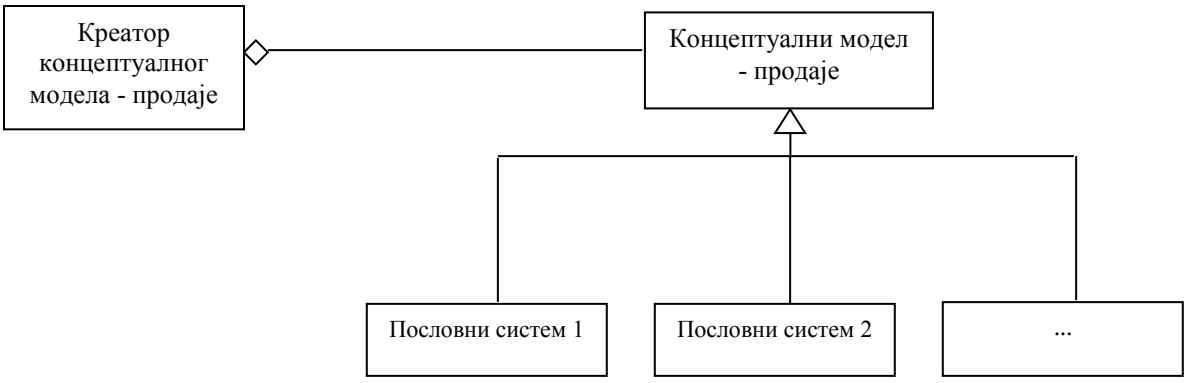
6.2 Фаза анализе и патерни

Фаза анализе описује логичку структуру и понашање софтверског система (пословну логику софт. система). Код упрошћене Ларманове методе развоја софтвера [Projektovanje softvera – knjiga u pripremi] понашање софтверског система је описано помоћу системских дијаграма секвенци и преко системских операција. Структура софтерског система се може описати помоћу **концептуалног** и **релационог модела**. Било који концептуални односно релациони модел који има генеричко својство које се може применити за различита појављивања истог домена проблема представља патерн[60]. Уколико на пример имамо упрошћени концептуални модел Продаје,



он се може применити за све пословне системе који имају продају која је у складу са наведеним концептуалним моделом продаје.

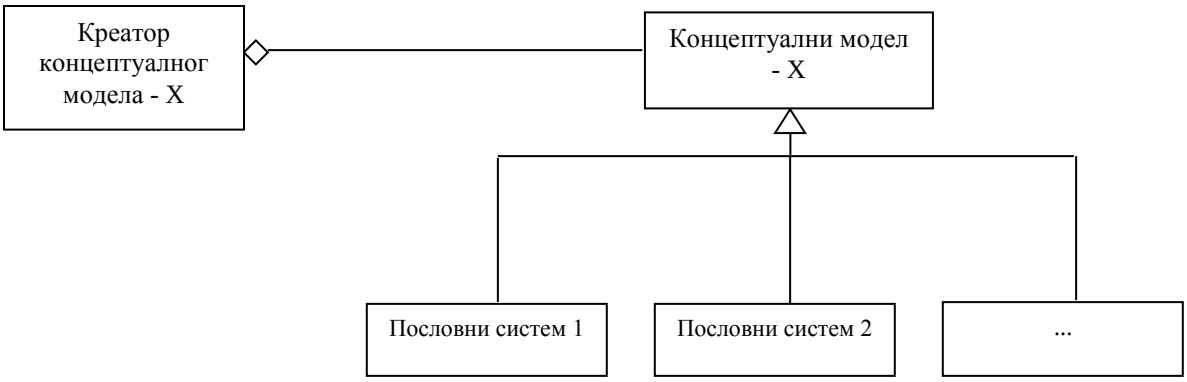
Уколико би концептуални модел продаје представили преко структуре решења GOF патерна пројектовања добили би следеће:



[60] У књизи Патерни Анализе [AP] Мартин Фаулер такође говори о генеричким концептуалним моделима који се могу применити за различите домене проблема при моделирању пословних система.

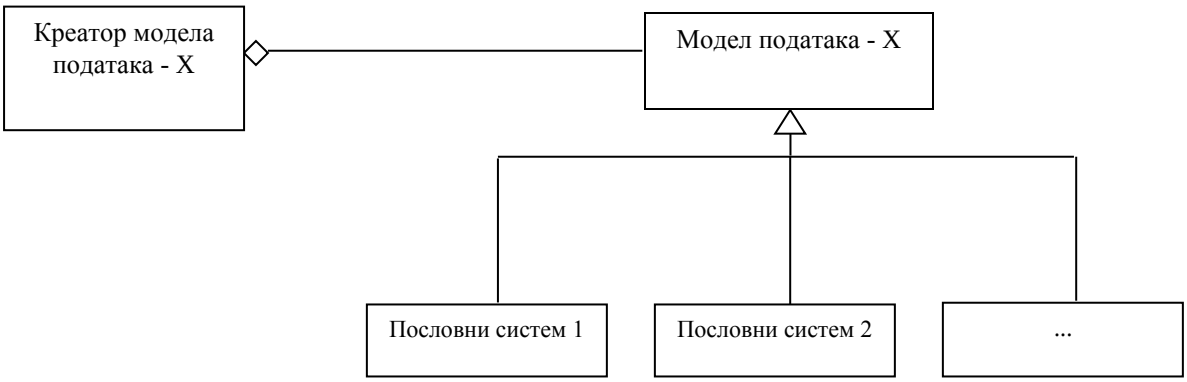
Наведени концептуални модел продаје има генеричко својство да се може применити за различите пословне системе (*Пословни систем 1, Пословни систем 2, ...*).

Уколико би желели да представимо концептуални модел функција пословног система (*Продаја, Набавка, Производња,...*) преко структуре решења GOF патерна пројектовања, добили би следеће:



X може бити *Продаја, Набавка, Производња,...,* или нека друга функција пословног система.
Наведени концептуални модел функције X има генеричко својство да се може применити за различите пословне системе (*Пословни систем 1, Пословни систем 2, ...*).

На сличан начин се може представити релациони модел или било који други модел података (модел објекти-везе, хијерархијски модел, мрежни модел, ...) функција пословног система преко структуре решења GOF патерна пројектовања:



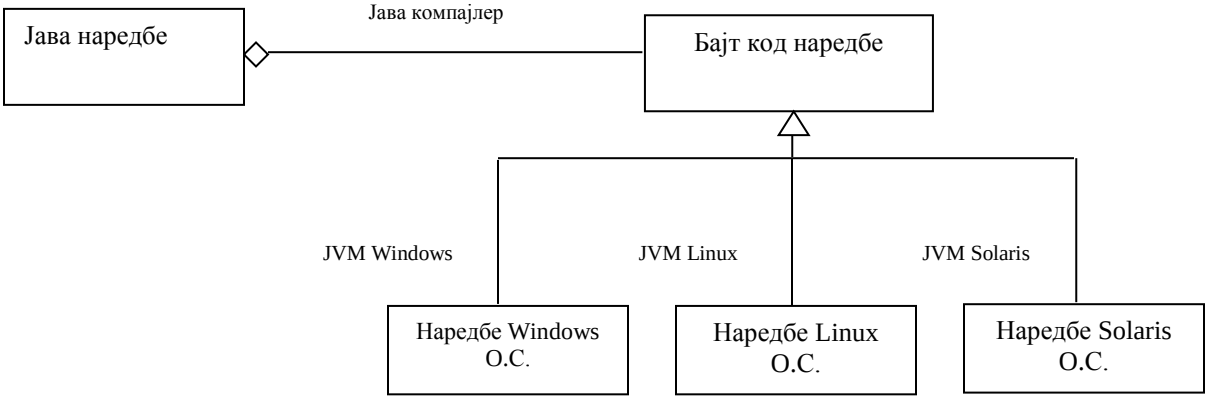
X може бити *Продаја, Набавка, Производња,...,* или нека друга функција пословног система.
Наведени модел података функције X има генеричко својство да се може применити за различите пословне системе (*Пословни систем 1, Пословни систем 2, ...*).

6.3 Софтверске технологије и патерни

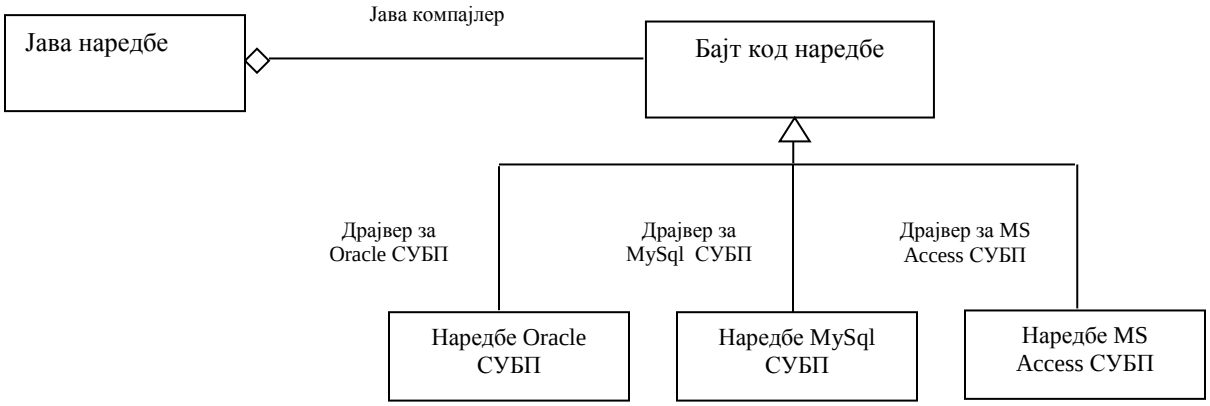
Навешћу неколико софтверских технологија које су повезане са патернима:
а) Јава програмски језик
б) Web сервиси

Јава програмски језик биће објашњен у контексту његовог а) извршавања на различитим оперативним системима и б) повезивању са различитим системима за управљање базом података (СУБП).

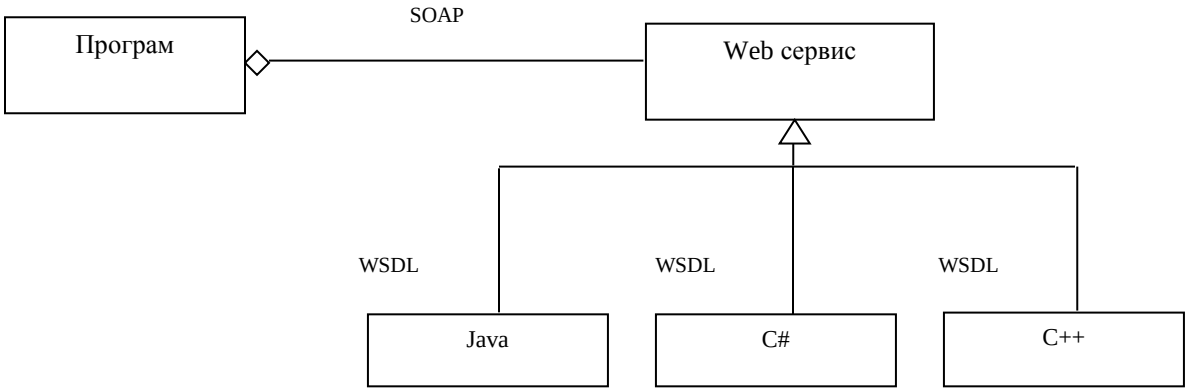
Јава наредбе се не компајлирају (преводе) директно у наредбе оперативног система на коме се извршавају. Јава наредбе се преводе у бајт код наредбе, које се затим преко Јава виртуелене машине преводе у наредбе конкретног оперативног система. Јава виртуелне машине се посебно праве за сваки оперативни систем.



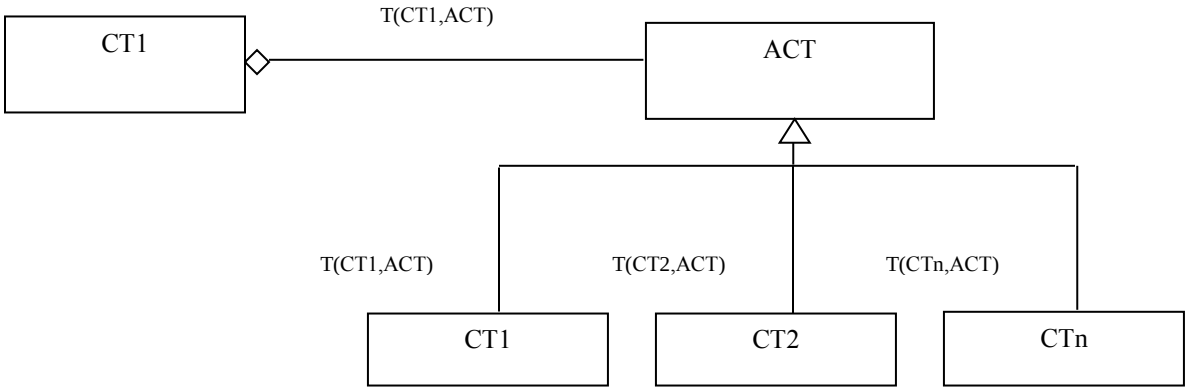
Повезивање Јава наредби са наредбама различитих СУБП се ради на сличан начин. Јава наредбе се преводе у бајт код наредбе, које се затим преко управљачких програма (драјвера) преводе у наредбе конкретног СУБП. Драјвери се посебно праве за сваки оперативни систем.



Позивање Web сервиса који је реализован у произвољном програмском језику (Java, C#,...) из програма који је такође реализован у произвољном програмском језику се обавља на следећи начин. Програм преко SOAP (Simple Object Access Protocol) протокола позива извршење Web сервиса. Програм не зна у ком програмском језику је реализован Web сервис. Web сервис се описује преко језика за описивање web сервиса WSDL-a (Web Service Description Language). То значи да различити програмски језици користе WSDL како би описали и изложили Web сервисе.



Структура наведених примера је у складу са структуром решења GOF патерна пројектовања. Свака од наведених структура на једном општијем нивоу покушава да прекине зависност између једне софтверске технологије и скупа софтверских технологија истога типа У првом случају прекида се зависност између Јава програма и различитих оперативних система. У другом случају се прекида зависност између Јава програма и различитих СУБП. У трећем случају се прекида зависност неког програмског језика са различитим програмским језицима (имплементационим технологијама)[61].
Уколико би покушали у хипотетичком смислу да прекинемо зависности између софтверске технологије CT1 и неког скупа софтверских технологија истог типа (CT1,CT2,...,CTn) потребно је дефинисати трансформације сваке од наведених технологија у апстрактну софтверску технологију (ACT). Апстрактне софтверске технологије (нпр. бајт код наредбе или Web сервиси) омогућавају да се различите софтверске технологије истог типа (програмски језици, оперативни системи, СУБП,...) описују на исти начин. Прецизније речено, из различитих софтверских технологија истог типа уочава се скуп заједничких својстава који се описују на исти начин.



[61] Програмски језик уколико подржава Web сервисе прекида зависност са самим собом.

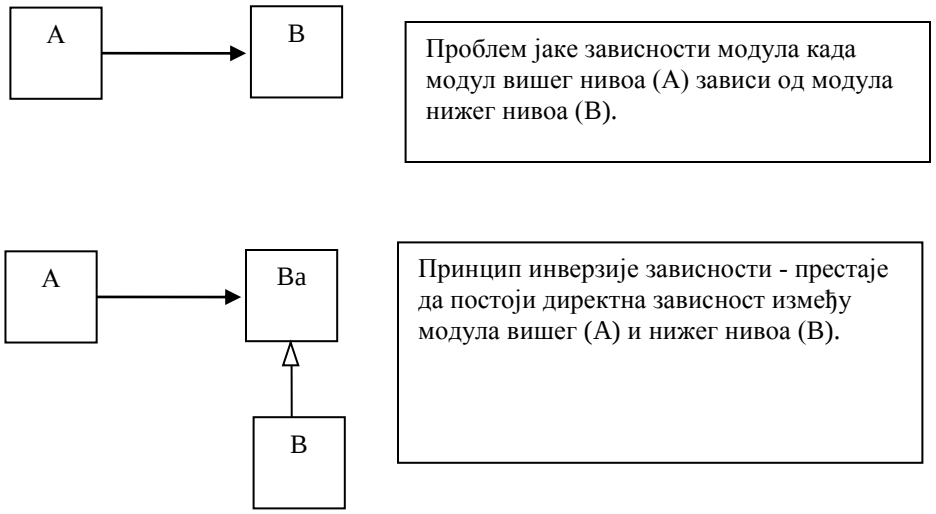
6.4 Принципи објектно-оријентисаног пројектовања и патерни

Навешћу неке од принципа објектно-оријентисаног пројектовања класа који су повезани са патерном у општем смислу:

1. Принцип инверзије зависности
2. Принцип уметања зависности

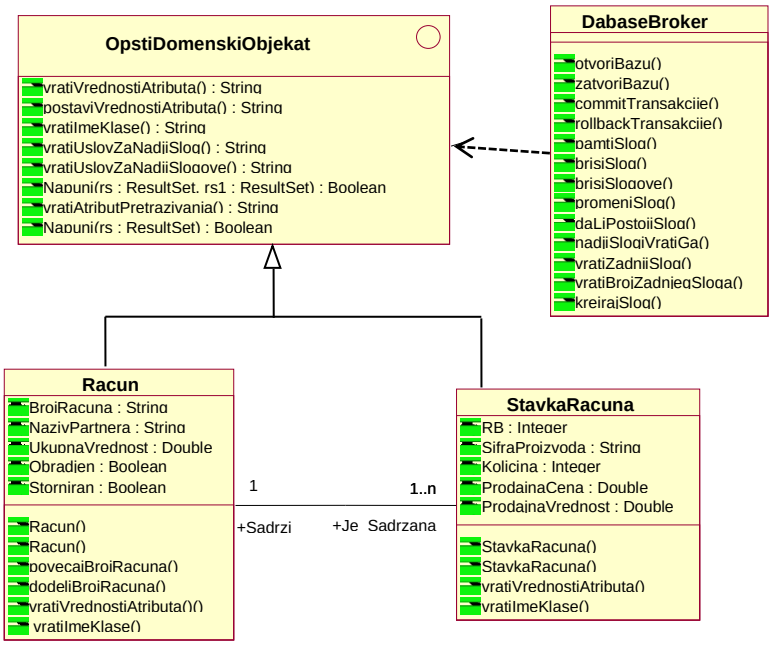
Принцип инверзије зависности (The Dependency Inversion Principle)

Када се говори о принципу инверзије зависности каже се да ***модул треба да зависи од апстракције а не од конкретизације***. То значи да модул вишег нивоа (A) не треба да зависе од модула нижег нивоа (B). Оба модула треба да зависе од апстракције (Ba). Абстракција (Ba) не треба да зависе од детаља (B). Детаљи (B) треба да зависе од апстракције (Ba). [Robert Martin]



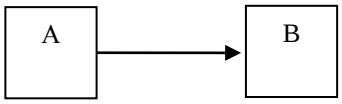
Проблем јаке зависности модула у суштини представља структуру проблема код GOF патерна пројектовања, док принцип инверзије зависности представља структуру решења код GOF патерна пројектовања.

У следећем примеру се избегава директна зависност између брокера базе података и доменских класа (Racun, StavkaRacuna) 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.

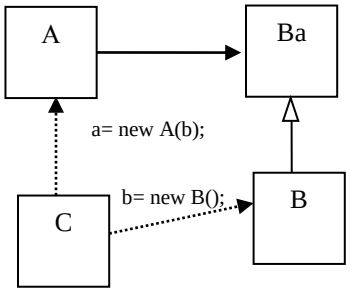


Принцип уметања зависности (The Dependency Injection Principle)

Принцип уметања зависности каже да се зависности између 2 модула програма успостављају у време извршења програма преко неког трећег модула. Принцип уметања зависности је повезан са принципом инверзије зависности, јер оба принципа решавају проблем јаке зависности између модула.



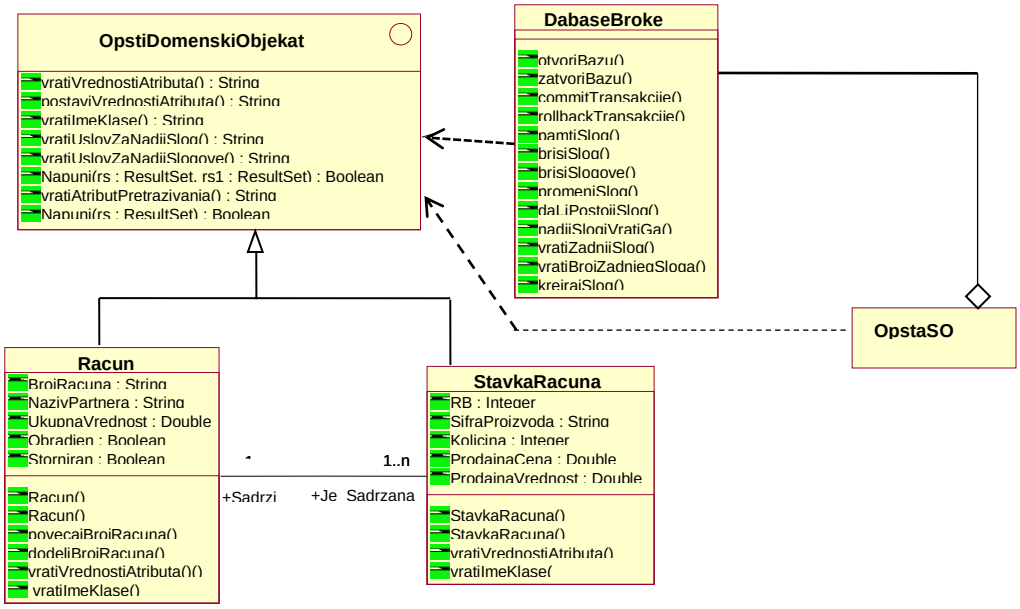
Проблем јаке зависности модула када модул вишег нивоа (A) зависи од модула нижег нивоа (B).



Принцип уметања зависности (модул C повезује у време извршења модуле A и B)

Проблем јаке зависности модула у суштини представља структуру проблема код GOF патерна пројектовања, док принцип уметања зависности указује на структуру решења код GOF патерна пројектовања.

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



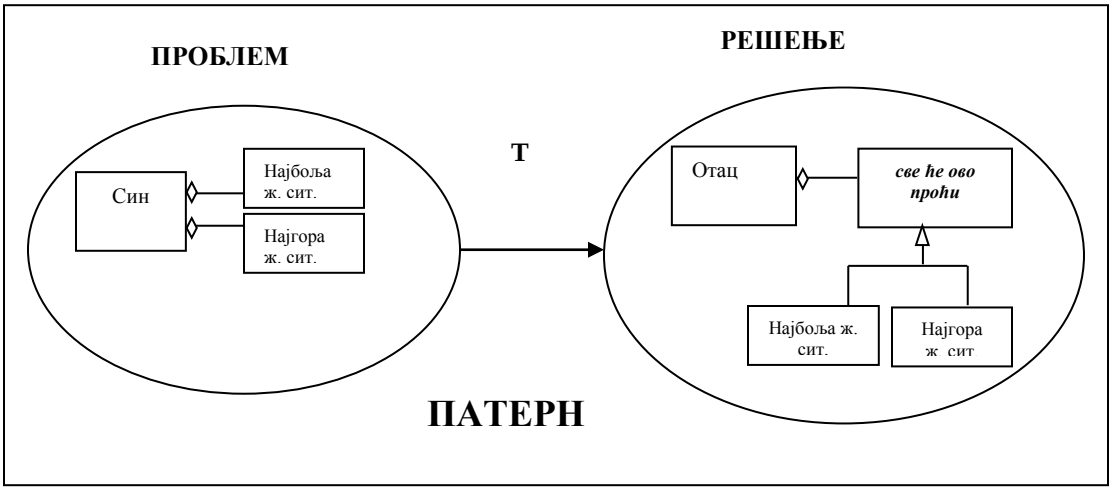
6.5 Мудре приче и патерни

Релативизација догађаја

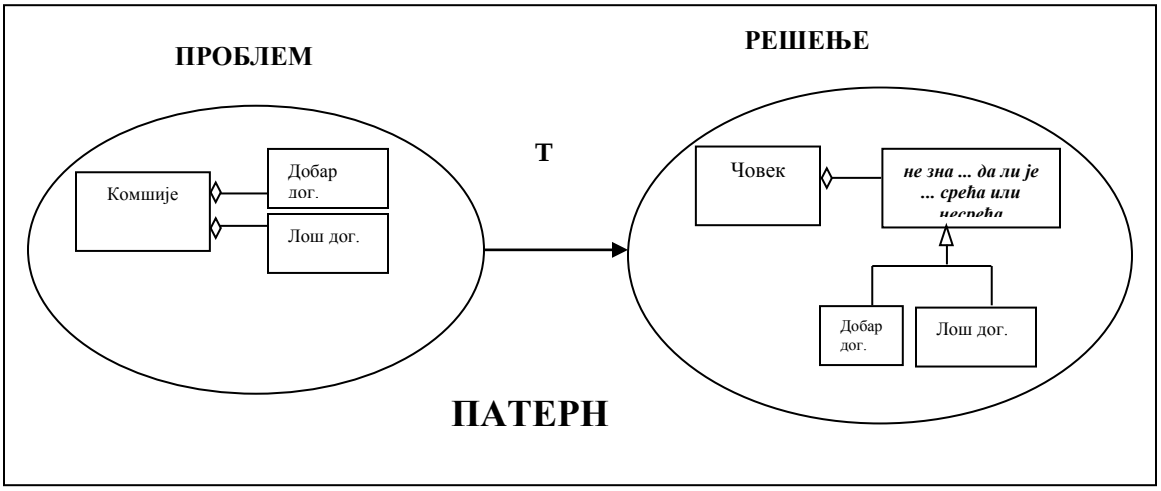
Сваки догађај ... онда када се дешава ... можеш да оцениш као добар или лош. Међутим, тек онда ... када прође време и видиш последице тог догађаја ... можеш непристрасно да процениш ... да ли је некадашња оцена о догађају била добра или лоша.

Постоје две приче у којима се релативизује тренутни догађај ... како би се трезвено и спремно дочекао следећи догађај који је узрокован предходним догађајем ... а који је његова супротност. Тако имамо причу о оцу и сину ... у којој отац у наслеђу сину није оставио никакво материјално богатство ... осим што га је научио да пише ... у нека стара времена ... када је било мало писмених људи. На самрти отац је сину оставио два кутијице ... црну и бело ... и рекао му је да црну отвори када буде у најгорој животној ситуацији ... а белу да отвори када буде био у најбољој животној ситуацији. И тако је син кренуо у бели свет након очеве смрти ... доста је радио ... и веома тешко је живео ... и у тренутку нервног растројства реши да се убије ... тада се сетио оца и отворио је црну кутијицу у којој је била парче папира са поруком ... **све ће ово проћи**. Та га је порука примирила и он одуста од самоубиства ... прикључи се првом каравану на који је наишао ... тамо упозна вођу каравана ... коме је требао неко писмен ... и временом он поста вођи каравана десна рука ... да би на крају оженио његову ћерку и постао веома имућан човек. У тренуцима највеће среће и благостања ... он се сети оца и његовог савета да отвори белу кутијици ... он је отвори и тамо пронађе парче папира на којој је писала иста порука ... као и у црној кутијици ... **све ће ово проћи.**

...

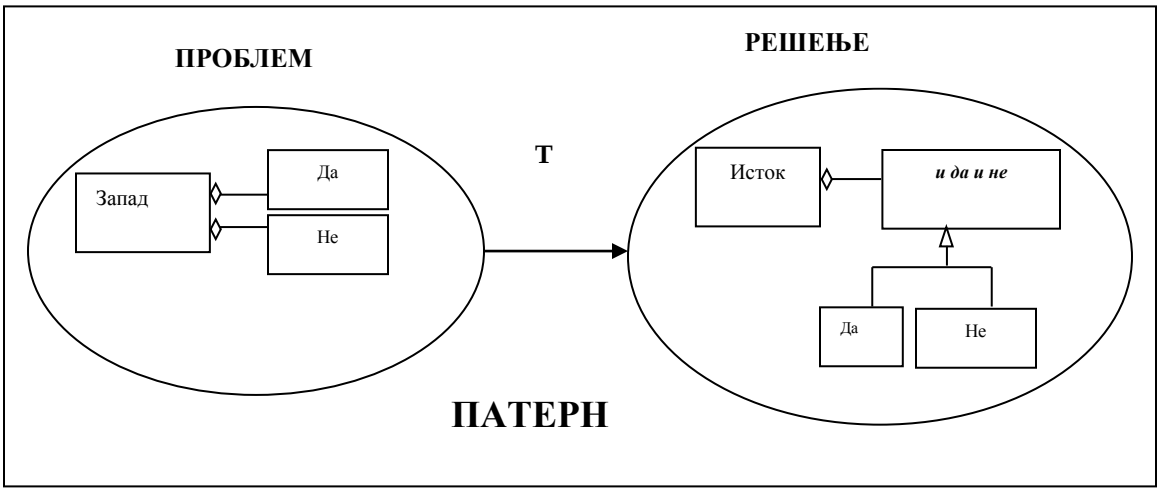


У другој причи имамо неког човека који је био веома сиромашан ... једино вредно што је имао била је кобила. Једног дана кобила је нестала ... комшије су рекле оцу да је он несрећан јер му је једини коњ нестао ... он им је одговорио да не пренагљују у оцени догађаја и рекао је ... **ја не знам да ли је то срећа или несрећа.** После неког времена ... кобила се вратила са пуно коња ... преко ноћи отац је постао богат ... комшије су тада рекле оцу да је он срећан јер има пуно коња... он им је одговорио да не пренагљују у оцени догађаја и рекао је ... **ја не знам да ли је то срећа или несрећа.** Након тога ... његов син јединац ... покушавајући да дресира коње ... пао са једног и постао хром ... комшије су рекле оцу да је он несрећан јер му је син јединац постао хром ... он им је одговорио да не пренагљују у оцени догађаја и рекао је ... **ја не знам да ли је то срећа или несрећа.** Недуго затим поче рат и поче мобилизација војно способних младића из тог села. Све младиће су мобилисали осим хромог младића ... комшије су рекле оцу да је он срећан јер му син јединац није мобилисан ... он им је одговорио да не пренагљују у оцени догађаја и рекао је ... **ја не знам да ли је то срећа или несрећа.**



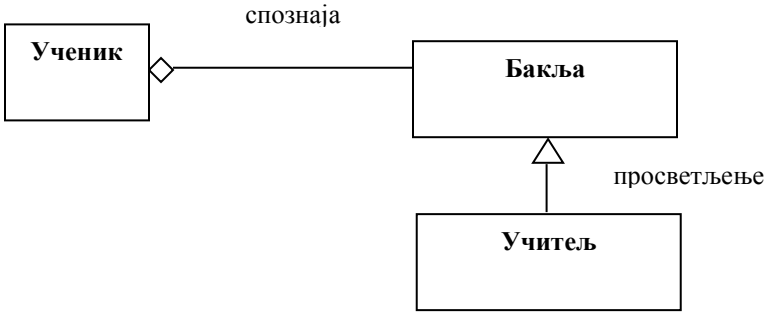
Филозофија истока и запада

Грчки филозоф Аристотел је у значајној мери допринео да се мишљење западне цивилизације сведе на бинарну логику ... **да или не.** На истоку кинески мудрац Мумон је рекао да ко објашњава разлику између да и не ... живи у подручју те разлике. Исток је поставио логику која омогућава да се нека појава може описати са **да и не**. Запад има тенденцију да сваку појаву постави или оцени на некој апсолутној основи ... док исток покушава да релативизује сваку појаву. Трансформација логике **да или не** у логику **и да и не ...** омогућава да апослутни и релативни став о некој појави буду увек у равнотежи.



Спознаја и просветљење

У једној источњачкој причи ученик је питао учитеља ... каква је разлика између спознаје и просветљења. Учитељ му је одговорио ... када спознајеш ... користиш бакљу да ти осветли пут ... када си просветљен ти сам постајеш бакља.

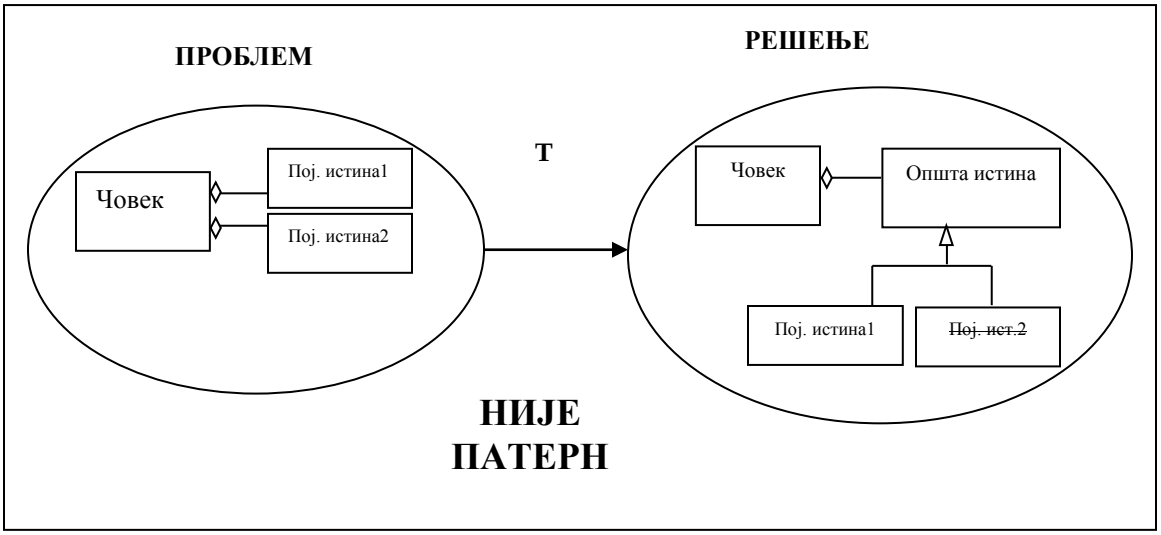


Исток каже да се истина не може ни негирати ни потврдити. То значи да се са друге стране истина може и негирати и потврдити (ово је моје мишљење).

Општа истина

У једној источњачкој причи ... зли Мара ... отелотворење зла ... је објаснио свом следбенику ... да он не мора увек људе да доводи у заблуду и странпутицу ... чак и онда када неко од људи сагледа неку истину. Он је рекао ... да људи имају особину да од појединачне истине ... направе уверење (општу истину) ... која их касније доводи у заблуду када сагледавају друге појединачне истине.

Свако ко покуша од појединачне истине да направи општу истину о некој појави ... може ући у проблем да та општа истина није у складу са другим појединачним истинама.



7. Филозофија патерна – субјективни осврт

Стојимо у облаку стварања и назиремо на хоризонту наше спознаје склад који се дешава у ритму пулсирања нашега живота. Однекуд долази хармонија, која се својим бићем усељава у наша срца и ту остаје да чека да се десе све оне лепе ствари које трају дуже од нас. Немогуће се дешава када природа проговори језиком непролазног, када наговештај отвори пут а догађај потврди наше путовање кроз мисли ка почетку. И тада не постоји ништа што може да заустави да будемо у стварању, да будемо унутар кретања, да будемо ток који траје, на врху таласа, на врху далеких простора која су увек ту, као радост када дете први пут обрадује своје родитеље. Понекад се крећемо, уназад од тачке ослонца ка истоку, како би видели себе непристрасно у боји која са нијансама плавог, са сунцем у средини, чека да нас прожме и испуни топлином вечног кретања. Тада највише постајемо своји, независни од себе, спремни да будемо искра која ће да разгори незнање која нас наводи на грешке и снове који немају корене. Сада је увек, без предрасуде тренутка који се бори за своје место под сунцем, да схватимо да не постоји ништа од живота што је без смисла, без склада са редом који нас покреће, који нас збуњује могућностима, који не дозвољава да осећај немоћи загосподари простором нашег стварања. Једног дана када се деси мај у души, онога који је смиренога духа, процветаће сви они облици и енергије које су тињале и чекале да се пробуде собом без притиска од споља, без пресије да нешто мора да се деси када ми то желимо. Тада не постоји граница докле може да се иде и одсуство времена је право стање свести које свеприсутно око нас обухвата једним обликом, једном структуром која је препозната и која води ка законима природе. Како је лепо када се после пуно година схвати да је све око нас једна савршена божанска пропорција која се може описати преко образаца који ће се увек приближавати симетрији и кроз то приближавање дешаваће се суштина. Долазиће из даљине и мењаће облике али ће увек остати иста, без намера да стане, без намере да нас остави саме. Желим да и даље будем сведок и учесник нечега, што ће надам се, помоћи људима да се врате извору наше суштине и наше судбине.

8. Литература

[AC1] Alexander Cristopher*: Notes on the Synthesis of Form.;* Cambridge, MA: Harvard University Press, 1974.

[AC2] Christopher Alexander,Sara Ishikawa, Murray Silverstein, max Jacobson, Ingrid Fiksdahl-King, and Shlomo Angel. *A Pattern Language.* Oxford University Press, New York, 1977.

[AC3] Alexander Cristopher*: The Timeless Way of Building;* New York: Oxford University Press ,1979.

[AP] Martin Fowler: *Analysis Patterns: Reusable Object Models,* Addison Wesley, USA, 1997.

[Cop1] James O. Coplien: *Software Patterns,* Bell Labs, The Hillside Group, 2000.

[Cop2] James O. Coplien: *A Development Process Generative Pattern Language. Pattern Language of Program Design*, Reading MA: Addison-Wesley, 1995.

[Cop3] J. Coplien: *Advanced C++ Programming Styles and Idioms,* Reading, MA: Addison-Wesley, 1992.

[Cop4] James O. Coplien, Liping Zhao: *Symmetry Breaking in Software Patterns,* Lucent Technologies and UMIST, 2001.

[Cop5] James O. Coplien: *The Futureof Language: Symetry or Broken Symetry?*, Bell Laboratories, USA.

[Dirk1] Dirk Riehle, Heinz Züllighoven: *Understanding and using patterns in software development,* Theory and Practice of Object Systems - Special issue on patterns, *Volume 2, Issue 1,* John Wiley & Sons, Inc., ISSN:1074-3227, New York, USA, 1996.

[DPFT] Toufik Taibi: *Design Pattern Formalisation Techniques,* IGI Publishing, Hershey, New York, 2007.

[DRSV] Докторска дисертација: "*Формализација јединственог процеса развоја софтвера помоћу узора*"., ментор проф. Др Видојко Ћирић; Факултет организационих наука, Универзитет у Београду, 2003.

[Eden1] Amnon H. Eden: *Precise Specification of Design Patterns and Tool Support in Their Application*, PhD Dissertation, The Department of Computer Science, Tel Aviv University, Tel Aviv, Israel, May 9, 2000.

[Eden2] Amnon H. Eden: *A theory of object oriented software architecture*, Department of Computer Science, Concordia University, Montreal, Quebec H3G 1M8, Canada.

[Eden3] Amnon H. Eden: *Towards a Mathematical Foundation for Design Patterns,* http://www.math.tau.ac.il/~eden/bibliography.html#towards_a_mathematical_foundation_for_design_patterns

[Eden4] Petar Grogone, Amnon H. Eden: *Concise and Formal Description of Architectures and Patterns,* Department of Computer Science, Concordia University, Montreal.

[Eden5] Amnon H. Eden: *A Visual Formalism for Object-Oriented Architecture,* Integrated Design and Process Technology, IDPT-2002, June 2002.

[Eden6] Amnon H. Eden, Y. Hirshfeld: *Principles in Formal Specification of Object Oriented Design and Architecture,* CASCON 2001, November 5-8, Toronto, Canada.

[FOW1] FOWLER, M.2006: *Writing software patterns*.
http://www.martinfowler.com/articles/writingPatterns.html

[GABR1] Gabriel, Richard P., *Patterns of Software: Tales from the Software Community*, Oxford University Press, 1998.

[GOF] Erich Gamma, Richard Helm, Ralph Johnson, JohnVlissides: *Design patterns*, Addison : Wesley, 18th Printing, September 1999.

[JPRS] Ivar Jacobson, Grady Booch, James Rumbaugh: *The Unified Software Development Process*, Rational Software Corporation, Addison-Wesley, 1999.

[KAM1] Kampffmeyer, H., Zschaler, S. 2007: *Finding the Pattern You Need: The Design Pattern Intent Ontology*, in *MoDELS*, Springer-Verlag, volume 4735/2007, 211-225.

[Koe1] R. Koenig: *Patterns and Antipatterns.* Journal of Object-Oriented Programming, 1995.

[Pree] W. Pree: *Design Patterns for Object-Oriented Software Development*, Wokingham, England, Addison-Wesley, 1995.

[Ros1] ROSEN, J. *Symmetry rules – how science and nature are founded on symmetry*, Springer-Verlag Berlin Heidelberg, 2008.

[SV1] Синиша Влајић, *Формализација GOF узора преко транзитивне релације и симетријских концепата*, Info M, вол. 3, бр. 11, стр. 22-27, 2004, ISSN 1451-4397, Београд, СЦГ.

[SV2] Синиша Влајић, Видојко Ћирић: *Нова дефиниција узора на основу концепата симетрије*, Зборник радова ИНФО ФЕСТ ‘2002, Стране 175-184, 22-28 септембра 2002. год., Будва, СР Југославија. (проглашен за најбољи ауторски рад)

[SV3] S. Vlajić, V. Ćirić: *Formalizacija uzora na osnovu komutativnog aksioma i koncepata simetrije*, interni projekat, FON, Beograd 2002 (Interni projekat je izabran među prvih pet u SR Jugoslaviji, na takmičenju DiskoBolos u Beogradu 2002 god.)

[SV4] Синиша Влајић, Видојко Ћирић:*Формализација узора на основу комутативног аксиома и концепата симетрије,* Зборник радова YuInfo 2003, Март 2003, СР Југославија. (M63) - (Navedeni rad je izabran među prvih pet u SR Jugoslaviji, na takmičenju DiskoBolos u Beogradu 2002 god.)

[SV5] Siniša Vlajić, Dušan Savić, Ilija Antović: *The Explanation of the Design Patterns by the Symmetry Concepts*, The Fourteenth IASTED International Symposium on Artificial Intelligence and Soft Computing (ASC 2011), Proceedings of the IASTED International Conference on Signal and Image Processing and Application, Pages: 363-372, ISBN: 978-0-88986-885-4, June 22 – 24, 2011, Crete, Greece. DOI: 10.2316/P.2011.716-009

[TOUF] Toufik, T. 2007. *Design Pattern Formalisation Techniques*, IGI Publishing, Hershey, Pennsylvania.

[UPLOOP] Kant Back, Ward Cunningham*: Using Pattern Languages for Object-Oriented Program.* OOPSLA '87, Sept. 1987.