УНИВЕРЗИТЕТ У БЕОГРАДУ
ФАКУЛТЕТ ОРГАНИЗАЦИОНИХ НАУКА

Катедра за софтверско инжењерство
Лабораторија за софтверско инжењерство (**СИЛАБ**)

Семинарски рад из предмета:
СОФТВЕРСКИ ПАТЕРНИ

Професор:	Студент:	Број индекса:
Проф. др Синиша Влајић	**Петар Јеремић**	**93/15**

Београд - 2020.

Садржај

1. УВОД

У последње време, системи који се праве постају све комплекснији. Поред нових понашања система, програмери морају да воде рачуна и о наслеђеном коду тј. коду који је направио неко из старије школе програмера. Да би разумео тај код, програмеру је потребан ментор, по могућности неко ко је направио или учествовао у прављењу тог система. Временом, програмери су се запитали „Шта је потребно да онај који чита мој код разуме шта сам ја хтео да урадим?“. Да би смо одговорили на то питање, прво морамо дати одговор на то шта код који направимо чини неразумљивим. У следећем тексту, прећићемо један од најзанимљивијих делова прављења система тј. пројектовање система. Када програмер осмишља изглед и организацију свог кода, заједница му даје јасне назнаке на шта да обрати пажњу и у ком смеру да се креће да би његов код био разумљивији, како и заједници, тако и њему. Управо те назнаке се зову Патерни пројектовања.

2. СОФТВЕРСКИ ПАТЕРНИ

2.1 Основне дефиниције патерна

Софтверски патерни представљају неке од најбољих пракси како би код требало да буде организован. Након великог броја система које су програмери креирали, одређен код се понављао више пута и схваћено је да испреплетани људски умови морају да нађу заједничке тачке гледишта. Патерни су оно што већина заједнице сматра као тренутно најбоље решење организације кода. Управо то је и суштина софтверских патерна. Они не представљају оптимално решење, већ неку врсту хеуристике. Да би се нашло оптимално решење, прво употпуности треба знати шта је проблем што већ прелази у проблем математичког доказивања.

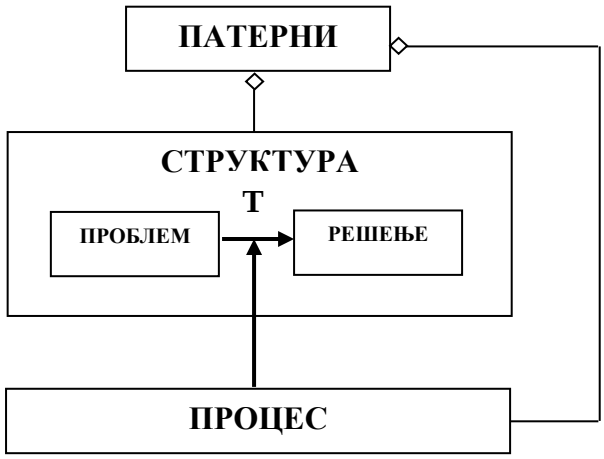
Када кренемо да развијамо софтвер, појављују се разна искушења. Једно од њих свакао јесте да нађемо тренутно најбоље решење за задати(локални) проблем или још горе, прво решење које можемо да смислимо. Ту се јавља проблем тзв. локалног оптимума где, ако нам се чини да је то најбоље решење што за тај мали проблем можда и јесте, то може да нам направи проблеме при испуњавању основног циља, развијања целокупног система. При развијању софтвера важи једно неписано правило које гласи „Мучимо се сада да би нам после било лакше.“. Управо је то улога патерна у развоју софтвера, њиова примена у систем не значи да ће нам баш у том тренутку олакшати живот али ће нам свакако у будућности бити драго да их имамо.

Патерне користимо када при развијању совтвера дођемо у још једно искушење, а то је жеља да наш систем буде савршен. На први поглед, то и не изгледа као лоша жеља и томе свакао теба тежити, али у већини случајева програмер ће бити оптерећен роковима и неће имати времена да сваку слагалицу свог система стави на своје место. Патерни дају јасан знак да сваки систем, иако треба да буде јединствена креација свог творца, има свој калуп који управо своме творцу улива сигурност га да његова жеља савршене креације не удаљи од циља и исту не растави у фрактале.

Можемо рећи да наш истем може имати два екстремна стања: стање потпуног реда и стање потпуног хаоса. Овде се стање потпуног реда не треба схватити као стање коме тежимо. Зашто је то тако? Замислите да, кад год изађете из куће, видите аутомобиле са само једном бојом. То решава неке проблеме, али прави хиљаду других. Сада идемо у другу крајност. Замислите да не постоје два аутомобила исте боје. Како их поредити по бојама и да ли онда боје упоште постоје? Сличан проблем као и код потпуног реда. Шта је онда циљ? Не постоји општи одговор. Некада је потребно мало више реда, некада мало више хаоса. Када се вратимо на развијање софтвера, већ смо рекли, програмер уноси хаос у свет машина. Да не би отишао предалеко, мора користити одређене алате и доскочице како би преварио своју природу. Када види да ће отићи предалеко, патерни му омогућавају да врати мало реда у свет који прави.

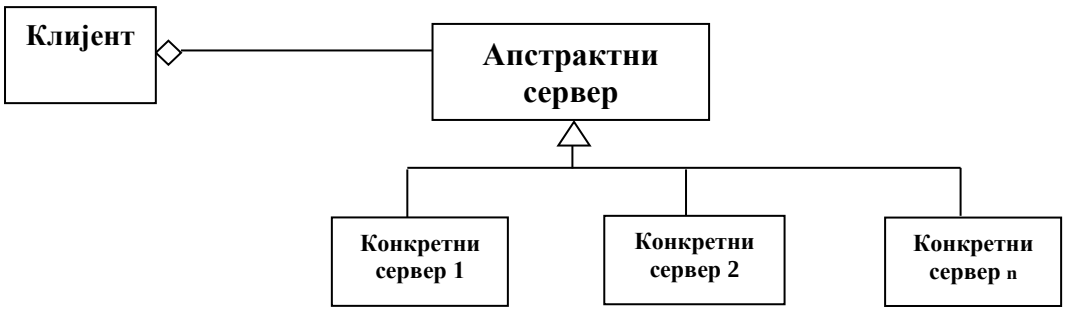
2.2 Општи облик GOF патерна пројектовања

Као што је већ речено, јака заједница чини и боље појединце. Група програмера [1] је 1994. Године објавила књигу о начину писања кода у објектно орјентисаном програмирању који може бити поново употребљив што представља праћење једног од основних принципа оо програмирања (Не понављај се) али се може рећи и сваког другог. У следећим сликама кратко је објашњено шта они заправо представљају.



Слика 1 Процес преласка из структуре проблема у структуру решења

У тој књизи је објашњена структура решења које нуди сваки од имплементираних патерна. Искусном програмеру или некоме ко већ познаје сваки од патерна је таква илустрација сасвим довољна. На слици испод је илустрована општа структура решења.

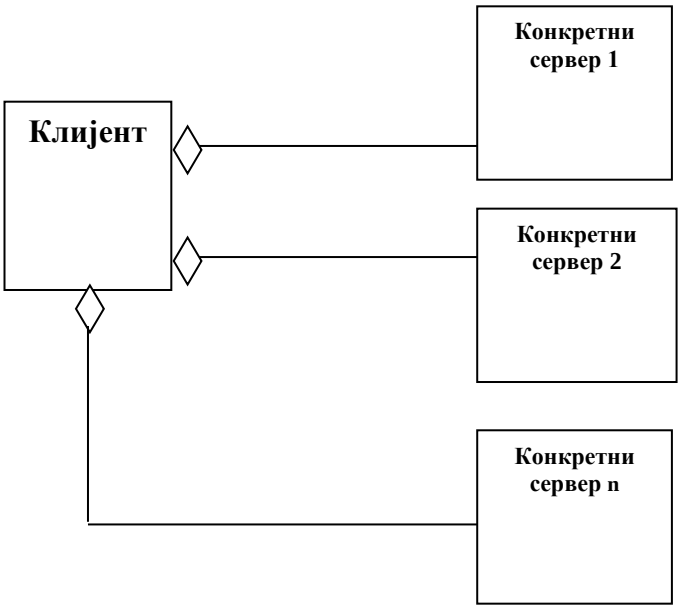


Слика 2 Структура решења GOF патерна пројектовања

Иако ова илустрација пружа довољан увид у то што желимо да постигнемо, увек је добро поћи од оног корака када треба да сазнајемо шта желимо да удрадимо. У генералном смислу, имамо клијента који може потраживати услуге од произвољног

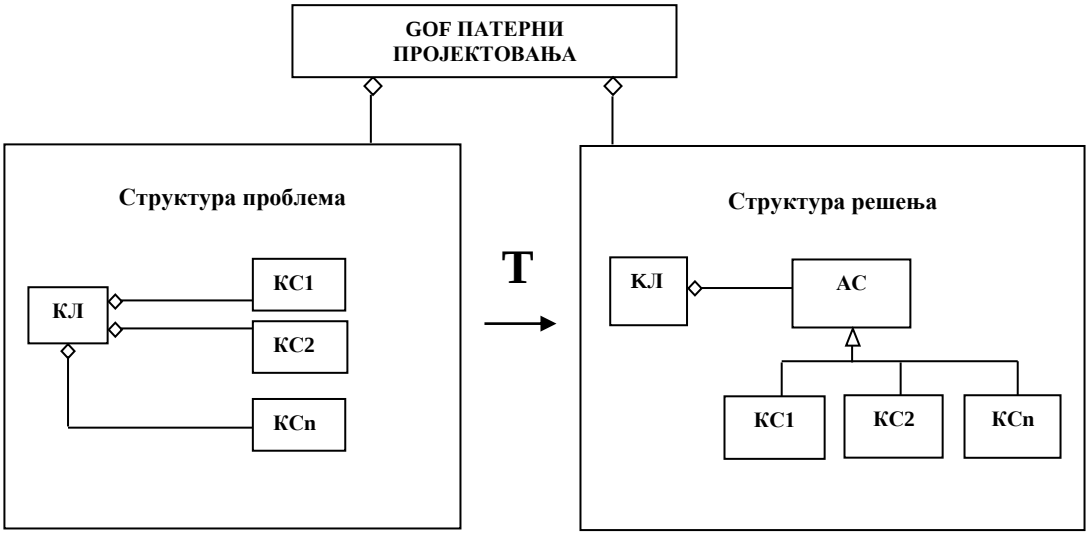
[1] https://en.wikipedia.org/wiki/Design_Patterns

броја сервера. Ако не стандардизујемо начин комуникације клијента и сервера, добијамо слику испод. На први поглед, то се не чини као икаква грешка јер клијент успешно комуницира са сваким сервером. Проблем настаје у модификовању. Ова тесна веза између клијента и сервера променом имплементације сервера узрокује и промену клијента што дефинитивно потискује значај одвајања сервера и клијента.



Слика 3 Структура проблема GOF патерна пројектовања

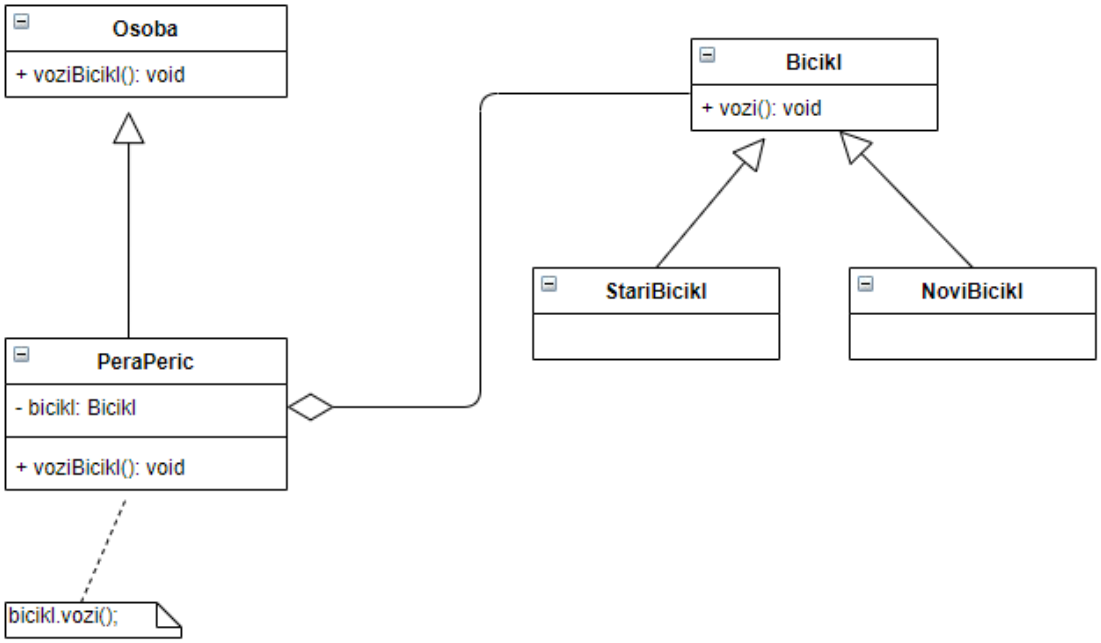
Стандардизација комуникације доноси велики број погодности. Један од њих је свакако могућност промене имплементације сервера без икаквог знања клијента. Клијент само мора да зна какву услугу очекује од сервера. Друге од погодности јесу боља организација кода, код који је разумљивији човеку, лакше одржавање и надограђивање система... Слика испод показује како патерни у генералном смислу постижу трансформацију из структуре онога што је идентификовано као проблем у структуру решења.



Слика 4 Општи облик GOF патерна пројектовања

2.3 Примењивост патерна у животу

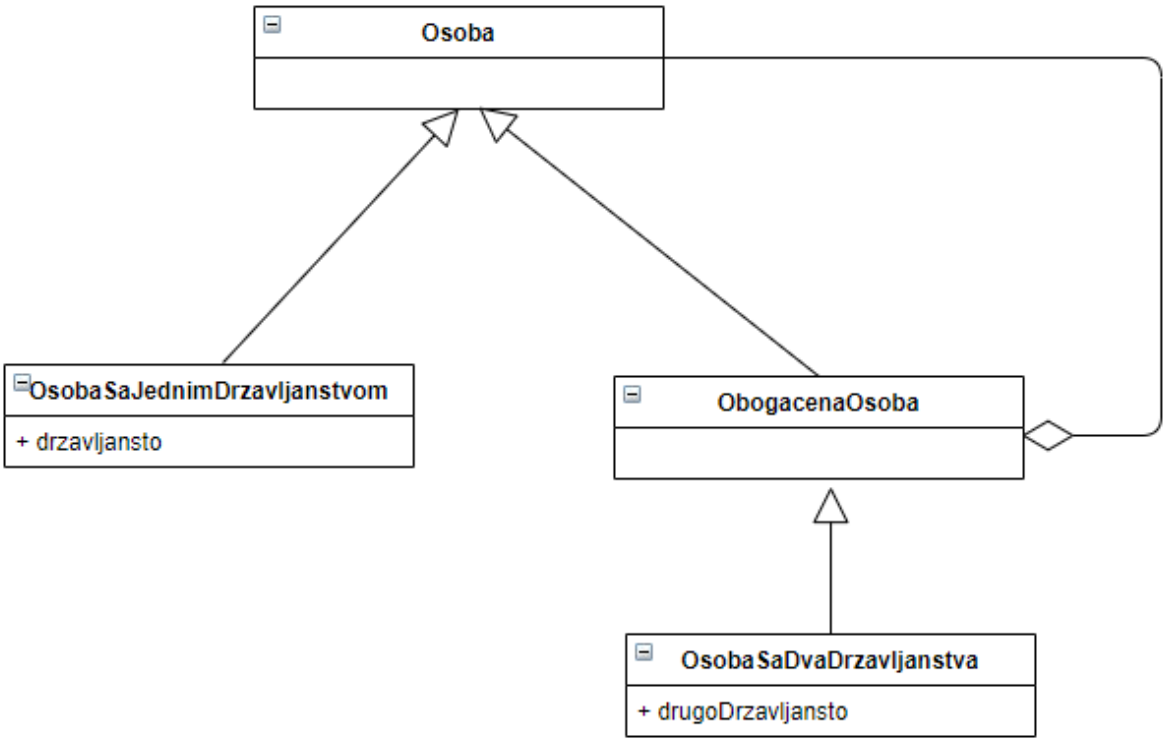
Патерне виђамо свуда и често их несвесно примењујемо. На пример, доста нас се сећа прве самосталне вожње бициклом. Тада смо се добро осећали јер, не само да смо научили нешто ново, него смо испунили циљ, победили смо. Замислите да смо са наученом новом вештином ограничени само на бицикл на којем смо научили да возимо. Замислите да куповином новог бицикла морате поново да учите да га возите. Управо та вештина је патерн који може изнова и изнова да се користи.



Слика 5 Примери патерна - бицикл

У претходном примеру дискутабилно је да ли особа може да има бицикл тј. да ли је потребно рећи да баш конкретна особа има бицикл у свом стању. У нашем случају имамо ту флексибилност да конкретна осова може да наследи нека друга стања које је можда теже памтити за сваки конкретни ентитет. У сваком случају, оба решења су прихватљива.

Још један, мало ређи пример је добијање другог држављанства. Замислите да сте добили држављанство ваше друге омиљене државе (прва је ваша, наравно). Сада се поставља питање да ли се одрећи свог држављанства и примити друго тј. „уништити“ себе и направити новог или обогатити себе другим држављанством?



Слика 6 Примери патерна - држављанство

Ово је врло добро решење јер обогаћена особа може да има држављанстава колико хоће (ако их добије).

Било како било, патерни су сваки вид поновне примене стеченог знања.

3. ПРИМЕНА СОФТВЕРСКИХ ПАТЕРНА У РАЗВОЈУ СОФТВЕРСКОГ СИСТЕМА

3.1 Опис софтверског система

Софтверски систем треба да обезбеди анализирање кроз игру разних игара из области теорије игара. Прва ствар коју корисник треба да види је екран на којем пише да треба да изабере игру. Када изабере игру, екран се мења и види основни интерфејс. Са доње стране започиње игру, селектује потезе и прави исте. Противник је комјуптер који прави своје стратегије у складу са опцијом коју је корисник одабрао.

3.2 Патерни за креирање објеката

ПК1 – Abstract Factory
ПК2 – Factory method
ПК3 – Singleton
ПК4 – Builder
ПК5 – Prototype

3.3 Структурни патерни

СП1: Bridge
СП2: Decorator
СП3: Proxy
СП4: Facade

3.4 Патерни понашања

ПП1: Command
ПП2: Strategy
ПП3: State
ПП4: Observer
ПП5: Chain of responsibility
ПП6: Template method
ПП7: Visitor
ПП8: Mediator
ПП9: Iterator
ПП10: Memento
ПП11: Interpreter

ПК1: Abstract Factory патерн

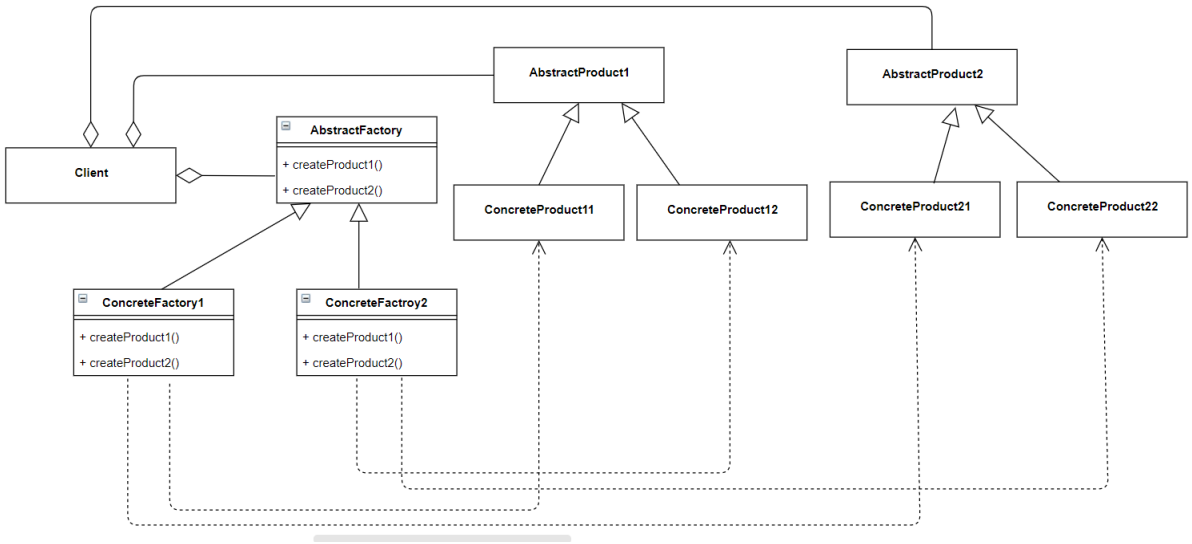
Дефиниција

Обезбеђује интерфејс 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



Слика 7 Abstract Factory патерн

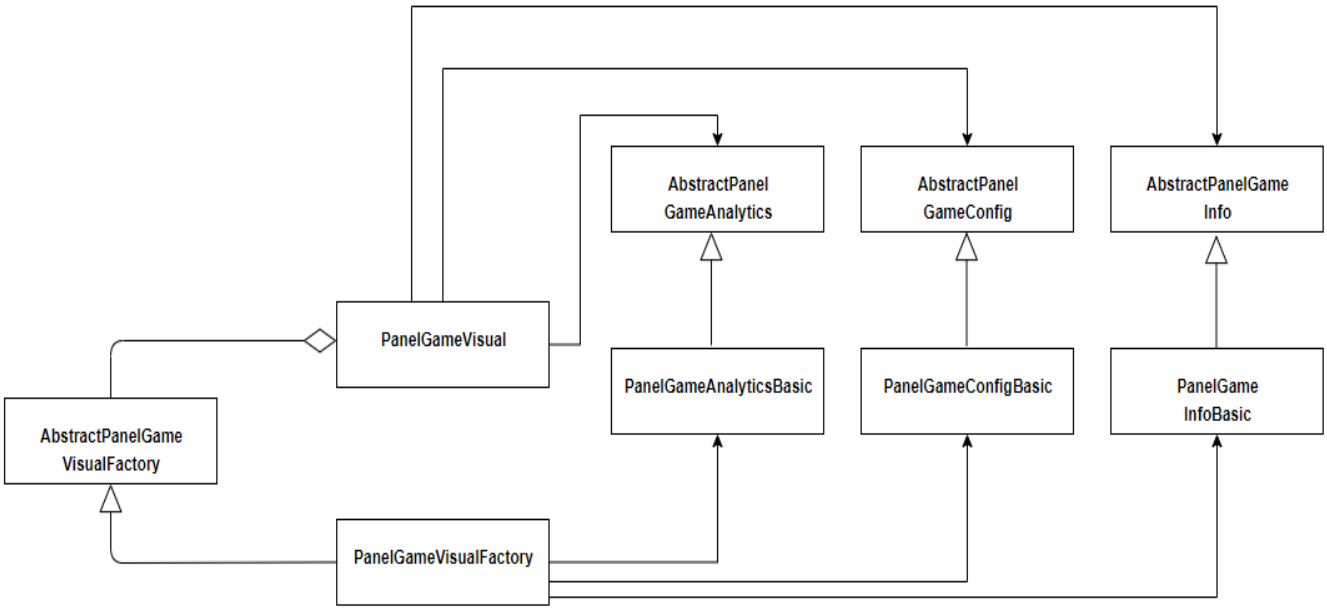
Учесници

- **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.
- **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.

Кориснички захтев PAF :

Панел за главни приказ игре тражи од фабрике за израду панела за главни приказ игре да изгради његове делове.

Дијаграм класа примера PAF



Слика 8 Дијаграм класа - Abstract factory

ПК2: Factory method патерн

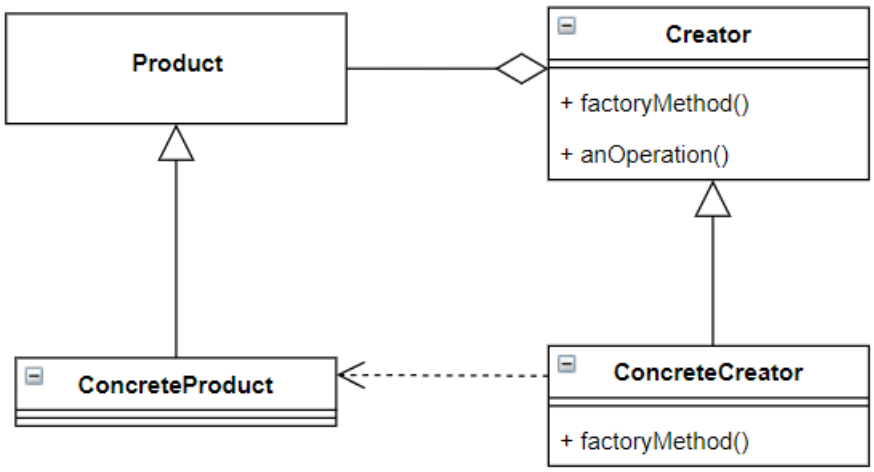
Дефиниција

Дефинише интерфејс за креирање објекта (производа), али преноси на подкласе одлуку коју ће класу истанцирати. 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).*

Структура Factory method патерна



Слика 9 Factory method патерн

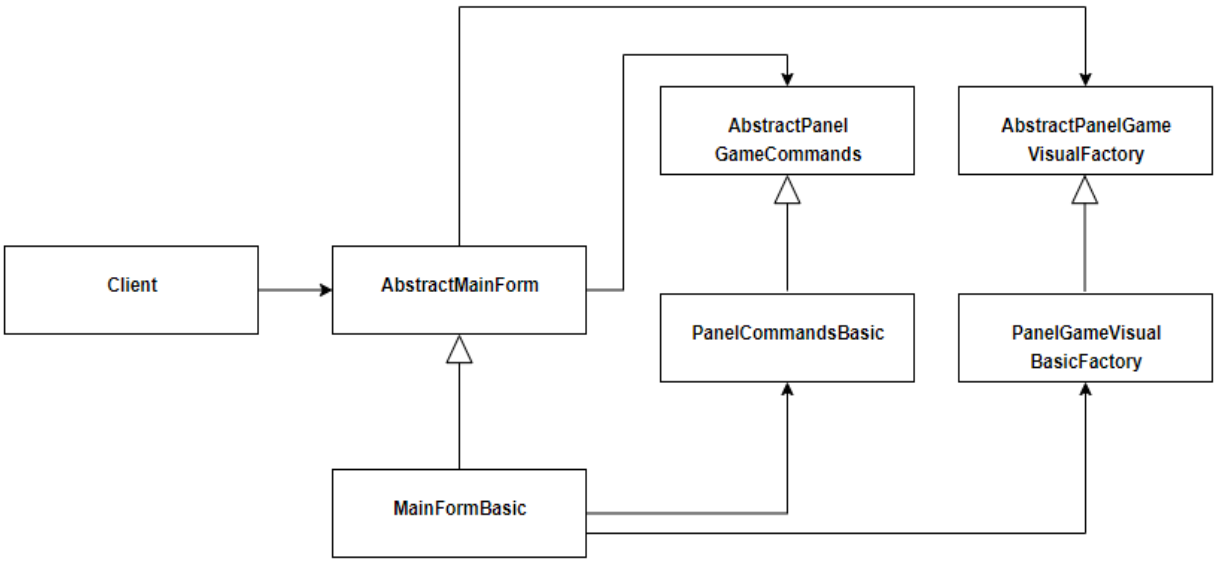
Учесници

- **Creator** - Дефинише интерфејс 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.

Кориснички захтев PFM2:

Клијент тражи главну форму која има панел за команде и фабрику за прављење главног панела за игру.

Дијаграм класа примера PFM2



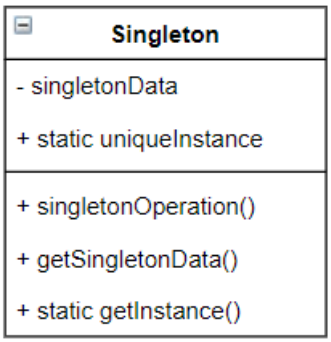
Слика 10 Дијаграм класа - Factory method

ПКЗ: Singleton патерн

Дефиниција

Обезбеђује класи само једно појављивање и глобални приступ до ње.

Структура Singleton патерна



Слика 11 Singleton патерн

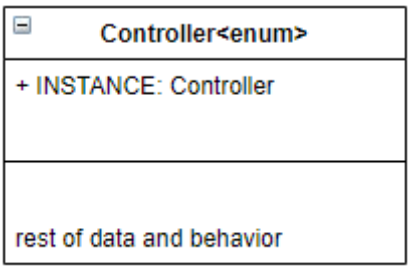
Учесници

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

Кориснички захтев PSI:

Онемогућити да се креира више од једног контролера.

Дијаграм класа примера PSI[2]



Слика 12 Дијаграм класа - Singleton

ПК4: Builder патерн

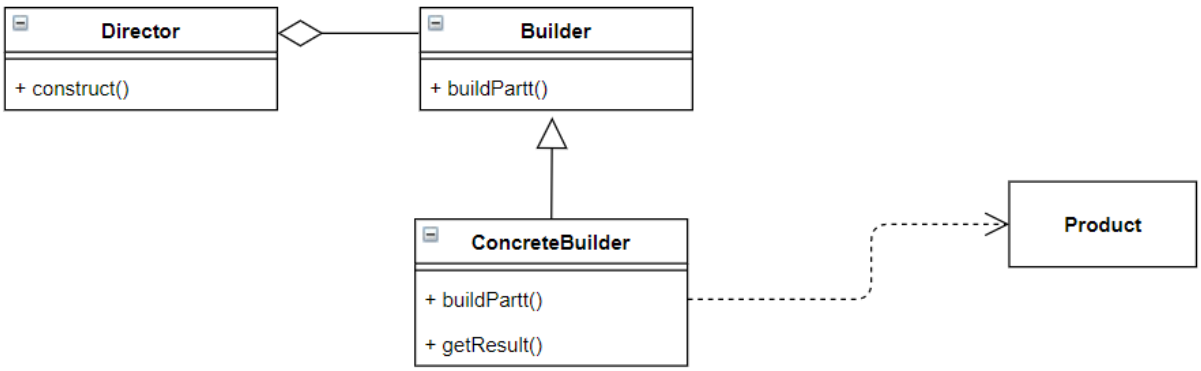
Дефиниција

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



Слика 13 Builder патерн

Учесници

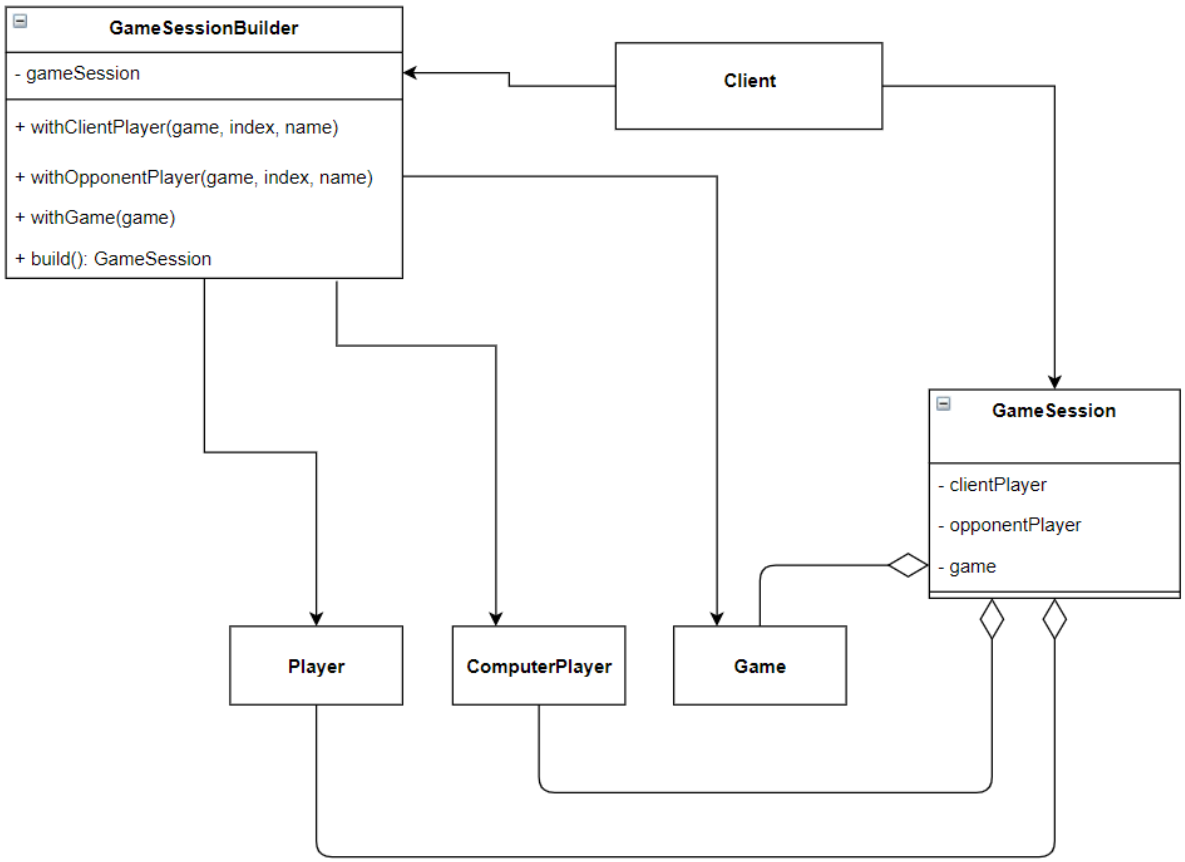
[2] У програмском језику јава, данас је најпопуларнија метода за овај патерн користити енумерацију.

- **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.

Кориснички захтев PBL2:

Омогућити да прављење сесије игре прави преко класе која је специјализована за то. Прављење сесије игре ће се вршити из више делова који је чине а то су клијент играч, компјутер играч и игра.

Дијаграм класа примера PBL2



Слика 14 Дијаграм класа Builder

ПК5: Prototype патерн

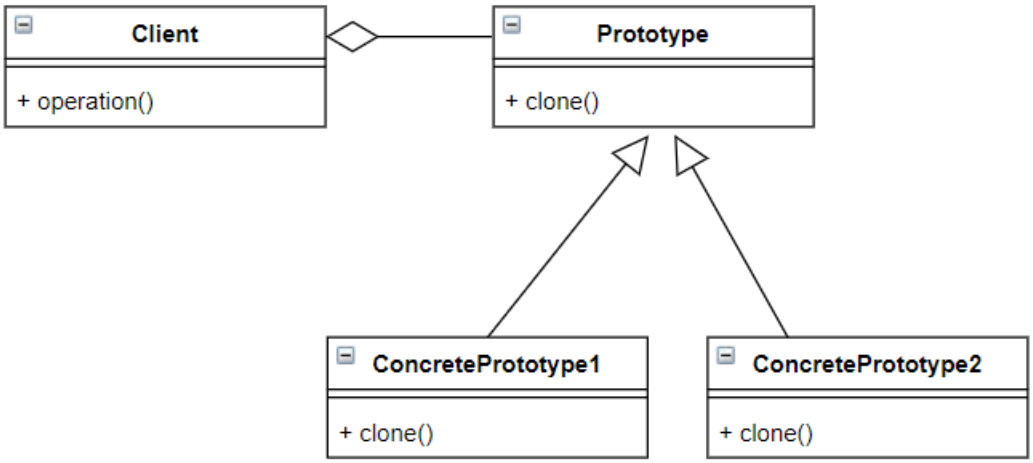
Дефиниција

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



Слика 15 Prototype патерн

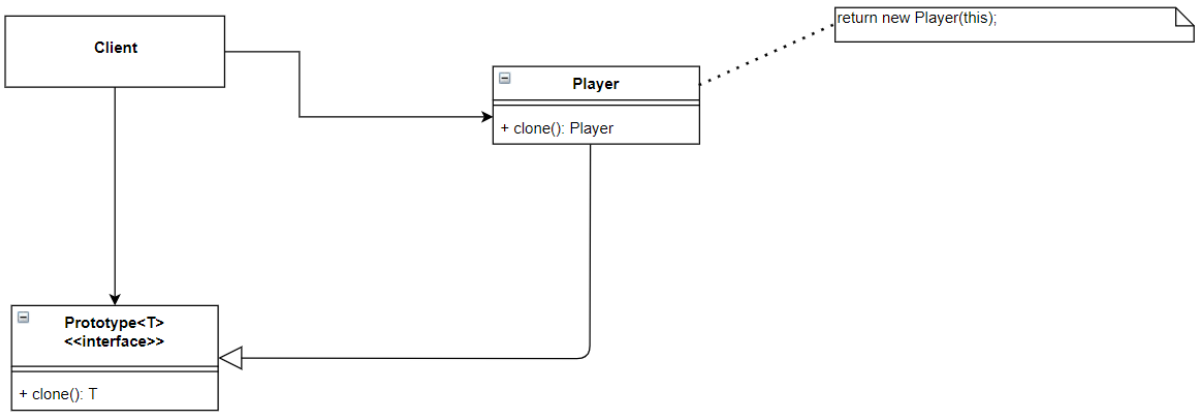
Учесници

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

Кориснички захтев PPR2:

Омогућити прављење копије играча са тренутним стањем играча.

Дијаграм класа примера PPR2



Слика 16 Дијаграм класа Prototype

СП1: Bridge патерн

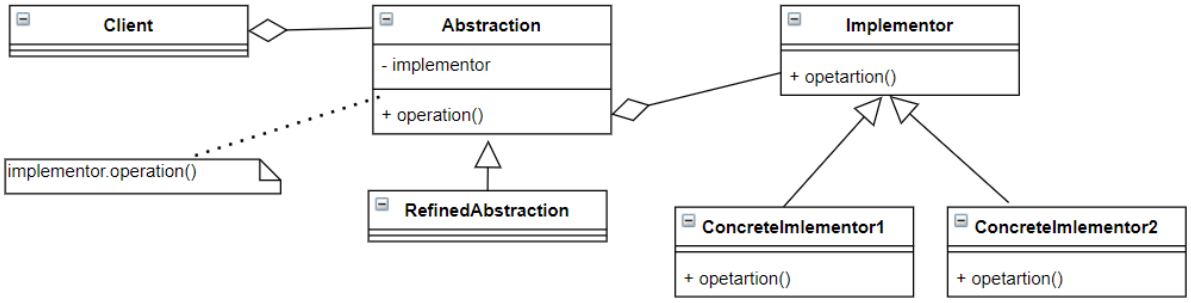
Дефиниција

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



Слика 17 Bridge патерн

Учесници:

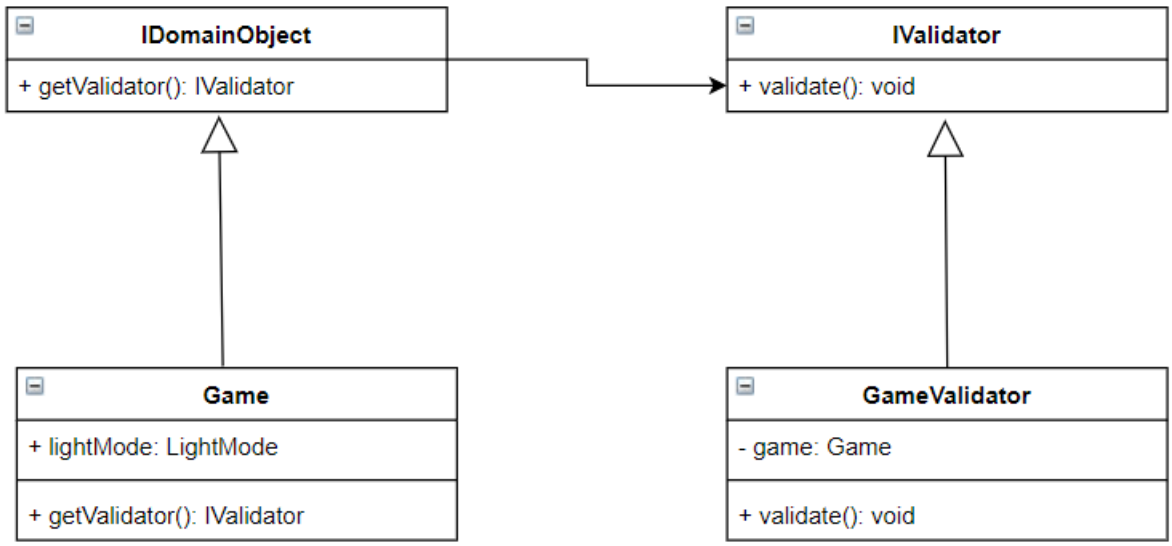
- **Abstraction** - Дефинише интерфејс апстракције.Чува референцу на објекат типа *Implementor*.
- **RefinedAbstraction** - Проширује интерфејс *Abstraction*.

- **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*.

Кориснички захтев PBR2:

Обезбедити валидатор за сваки доменски објекат.

Дијаграм класа примера PBR2



Слика 18 Дијаграм класа - Bridge

СП2: Decorator патерн

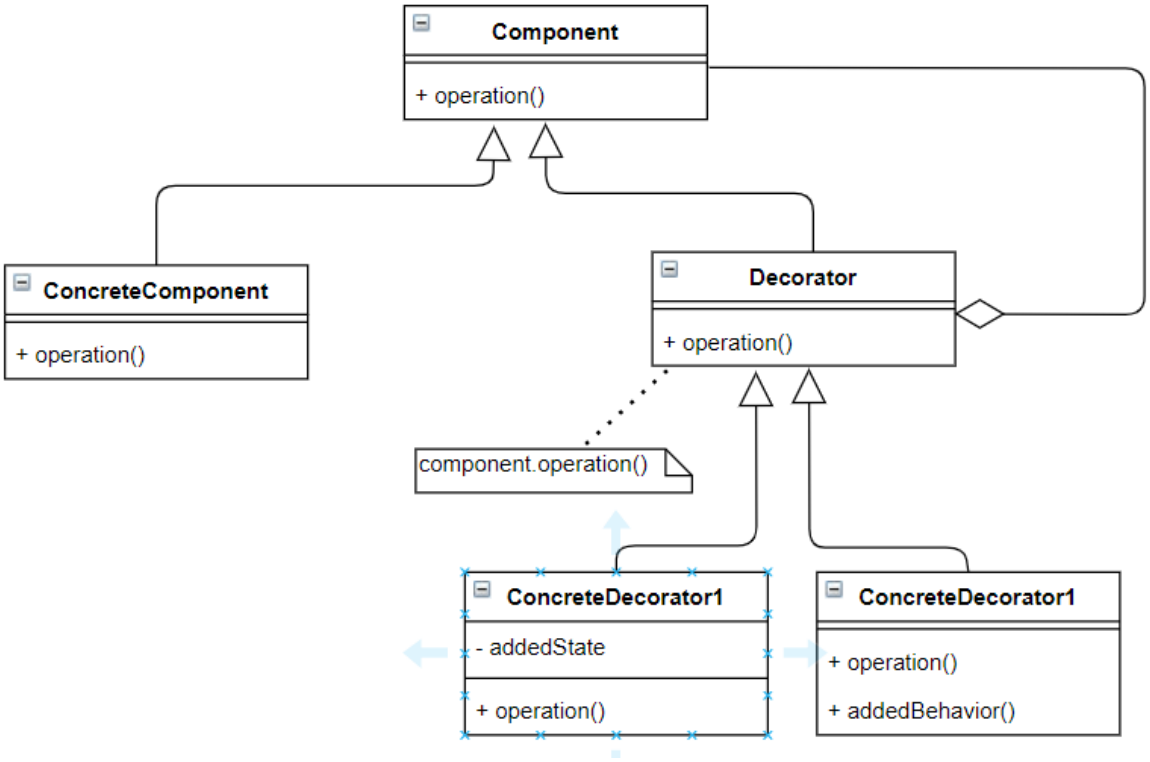
Дефиниција

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



Слика 19 Decorator патерн

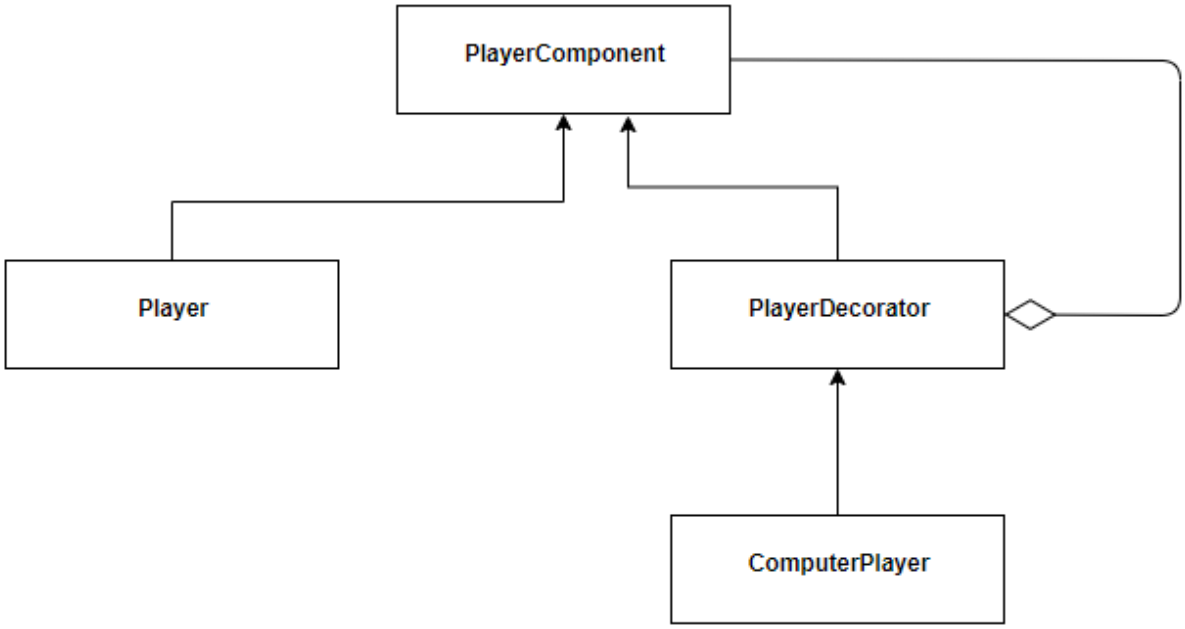
Учесници

- **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.

Кориснички захтев PDE2:

Потребно је обезбедити компјутерског играча који ће имати додатно понашање одабира стратегије као и селектора стратегије.

Дијаграм класа примера PDE2



Слика 20 Дијаграм класа - Decorator

СПЗ: Proxy патерн

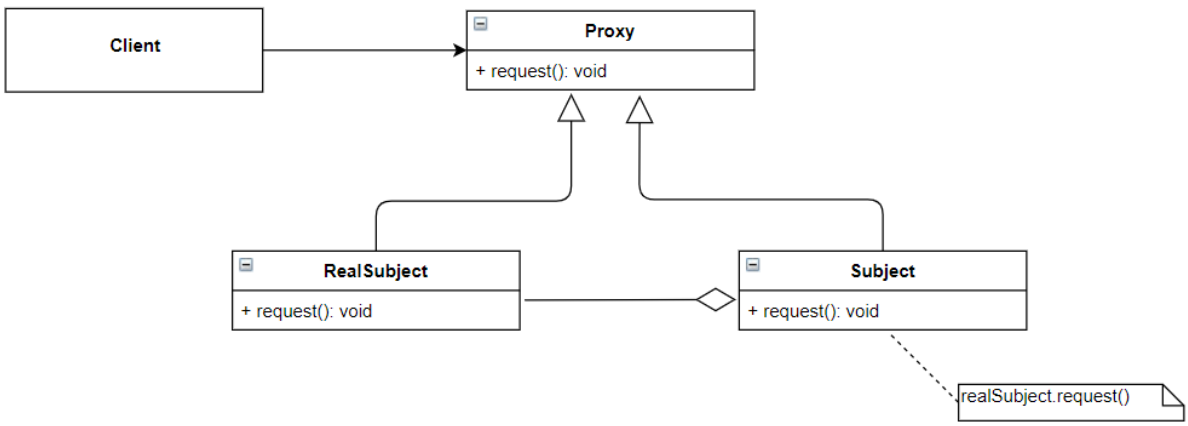
Дефиниција

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



Слика 21 Proxy патерн

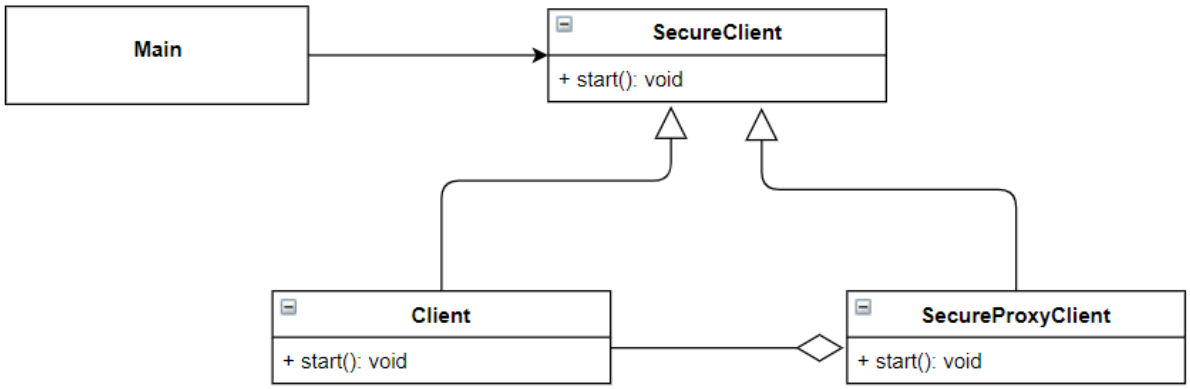
Учесници

- **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.

Кориснички захтев PPX1

Онемогућити стартовање програма ако у одређеном документу не постоји код за лиценцу.

Дијаграм класа примера PDE2



Слика 22 Дијаграм класа - Proxy

СП4: Facade патерн

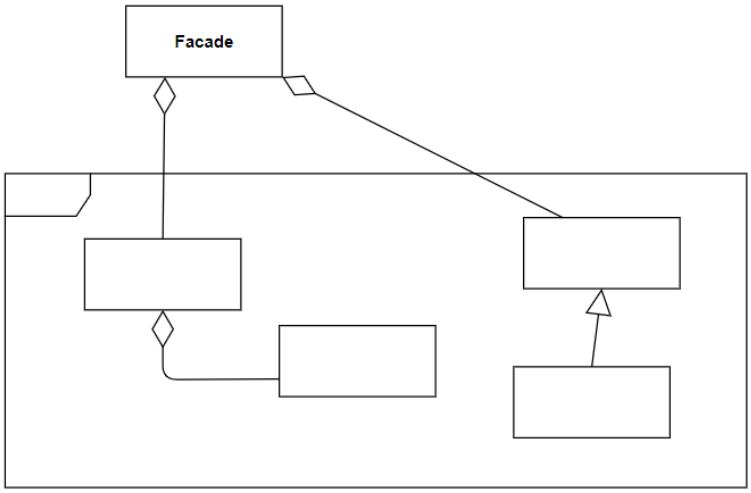
Дефиниција

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.

СтруктураFacade патерна



Слика 23 Facade патерн

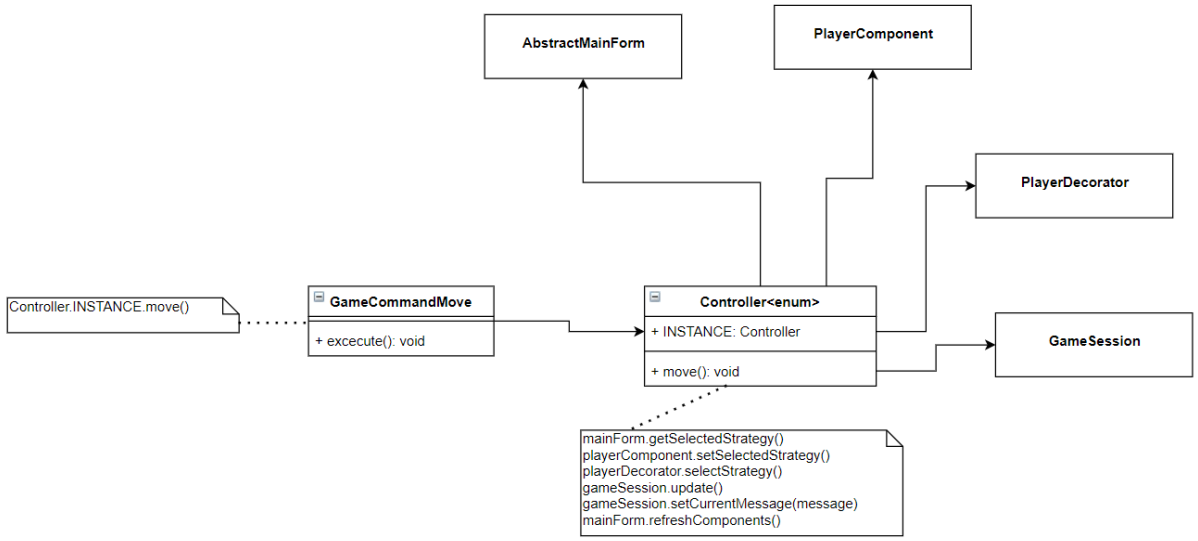
Учесници

- **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.

Кориснички захтев PFA1:

Направити контролер који ће представити интерфејс командама које корисник позове. Те команде ће позивати конролер који ће даље зати како да комуницира са осталим модулима у систему.

Дијаграм класа примера PFA1



Слика 24 Дијаграм класа Facade

ПП1. Command патерн

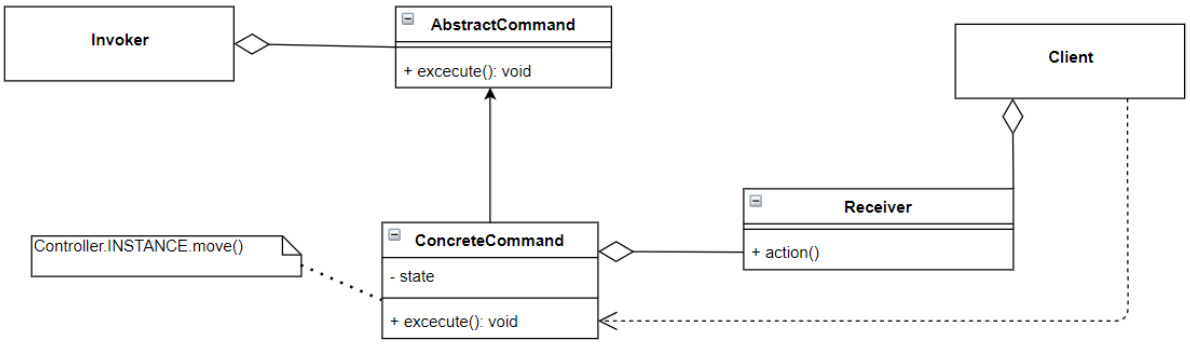
Дефиниција

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



Слика 25 Command патерн

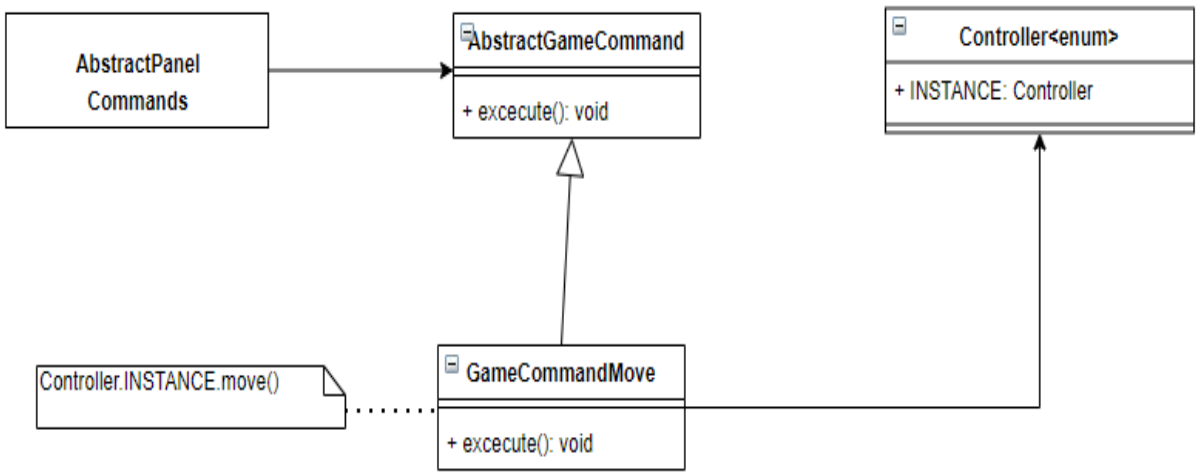
Учесници

- **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.

Кориснички захтев PCMM1:

Панел за команде игре ће позивати прављење и извршавање команде која ће својим извршавањем позивати контролер да одради акцију.

Дијаграм класа примера PCMM1



Слика 26 Дијаграм класа - Command

ПП2. Strategy патерн

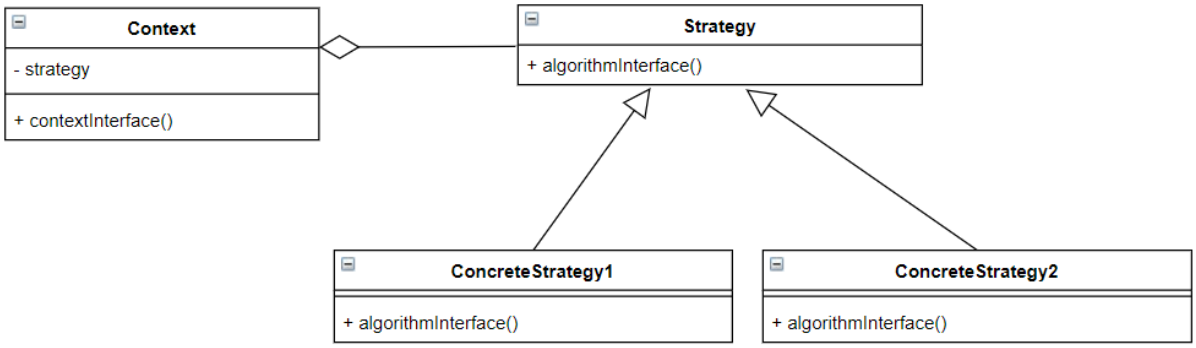
Дефиниција

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



Слика 27 Strategy патерн

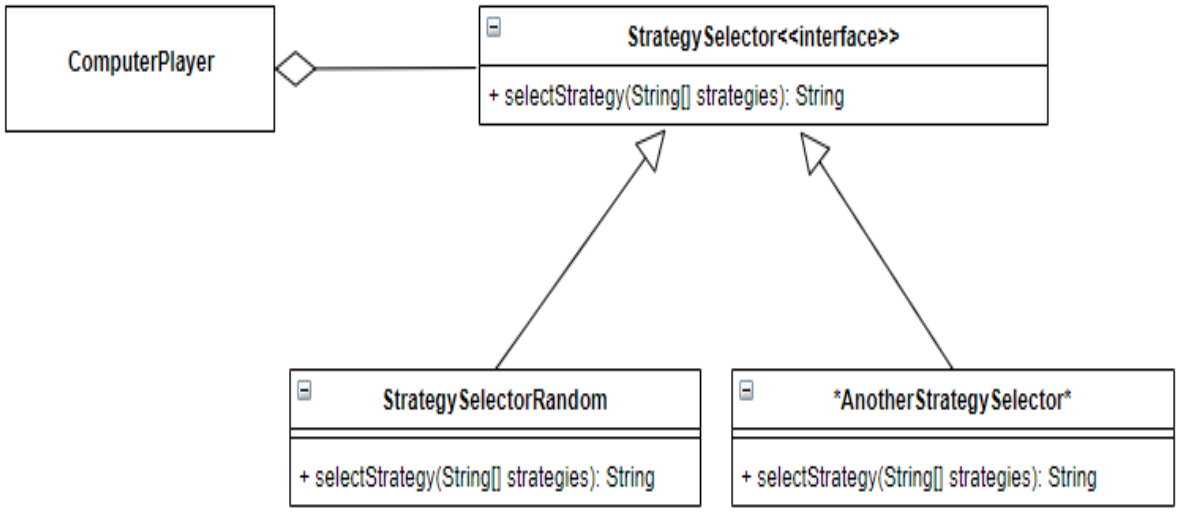
Учесници

- **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.

Кориснички захтев PSTR1:

Компјутерски играч има селектора стратегија који помоћу одређеног алгоритма бира стратегије. Тај алгоритам се може независно мењати од играча.

Дијаграм класа примера PSTR1



Слика 28 Дијаграм класа - Strategy

ППЗ. State патерн

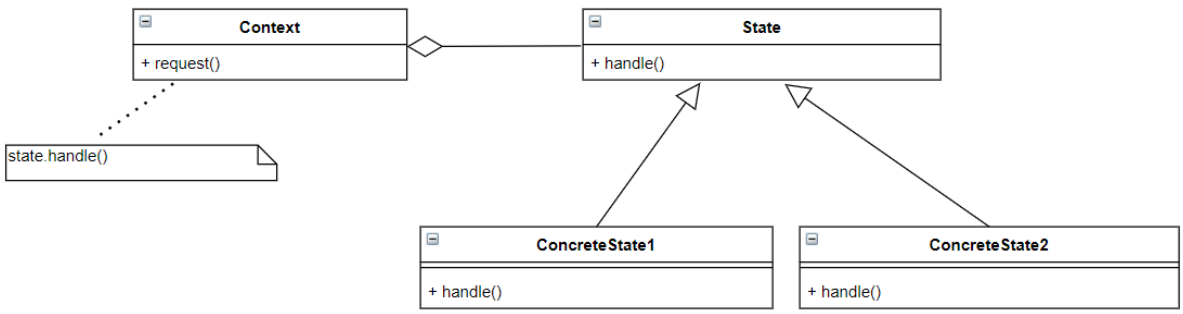
Дефиниција

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



Слика 29 State патерн

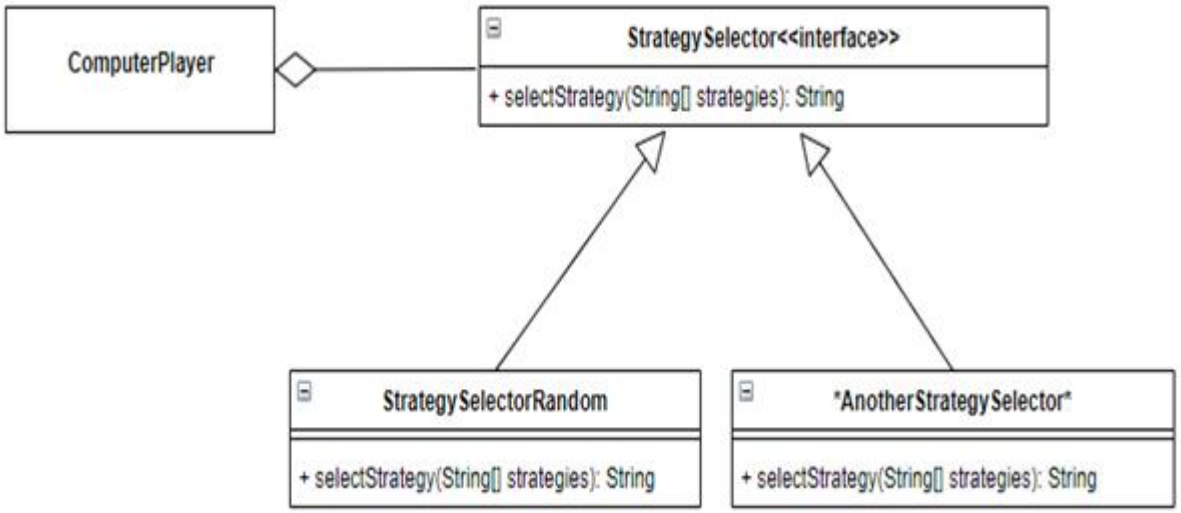
Учесници

- **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.

Кориснички захтев PST1:

Компјутерски играч може променити свог селектора стратегија и бирати стратегије у зависности од тога којег селектора има.

Дијаграм класа примера PST1[3]



Слика 30 Дијаграм класа - State

[3] Strategy і State патерн су веома слични. Разлика је у томе што State подразумева стање.

ПП4. Observer патерн

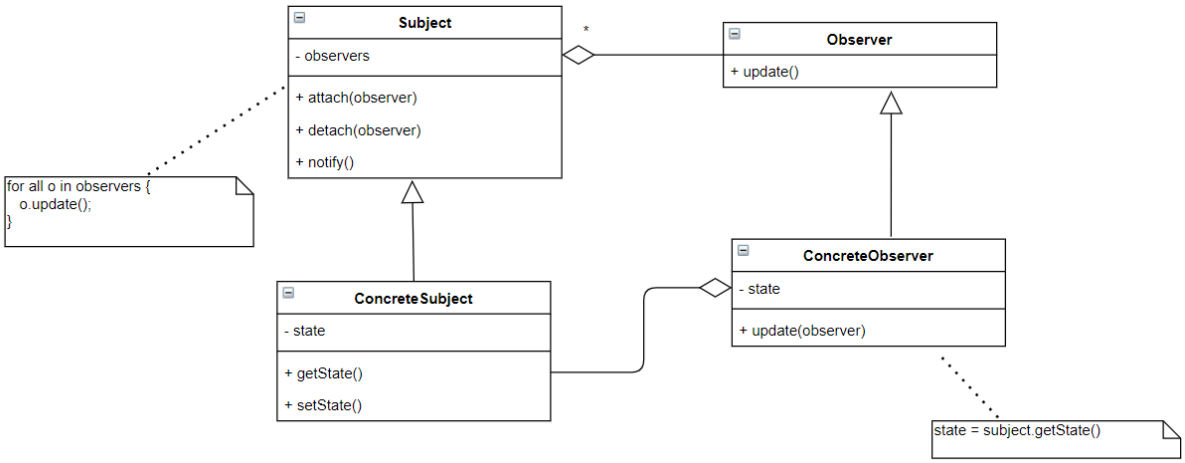
Дефиниција

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



Слика 31 Observer патерн

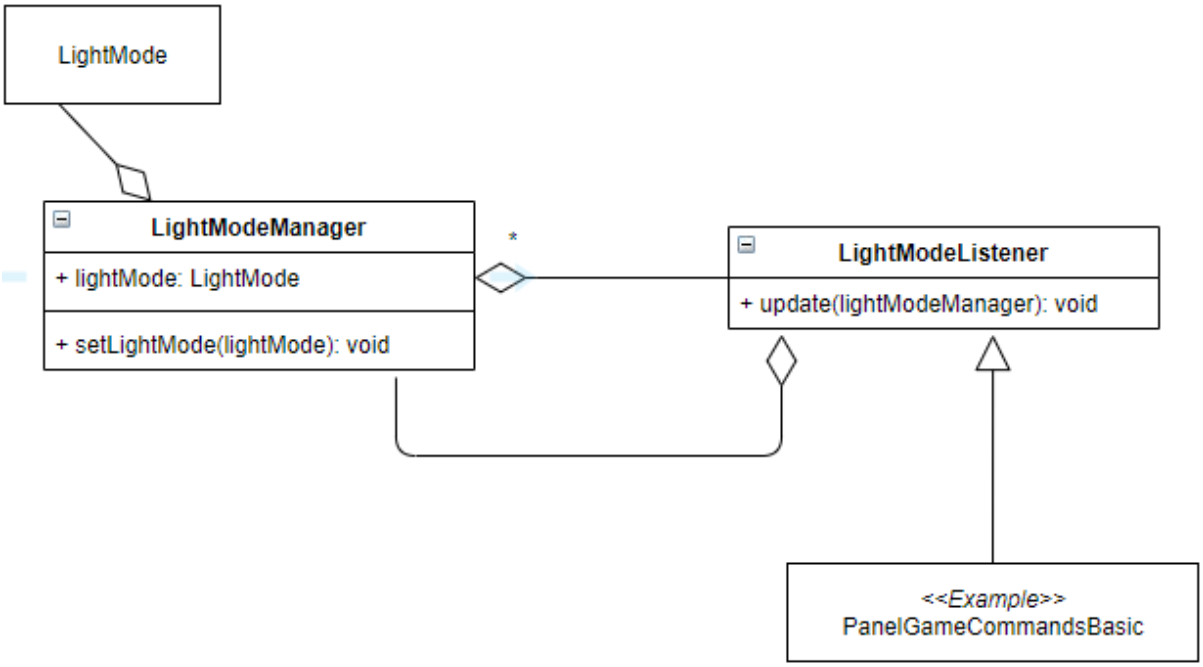
Учесници

- **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.

Кориснички захтев РОВ2:

Програм може мењати стање осветљења. Ради тако што менаџер за осветљење обавештава све панеле који га ослушкују да је корисник променио осветљење.

Дијаграм класа примера РОВ2



Слика 32 Дијаграм класа - Observer

ПП5. Chain of responsibility патерн

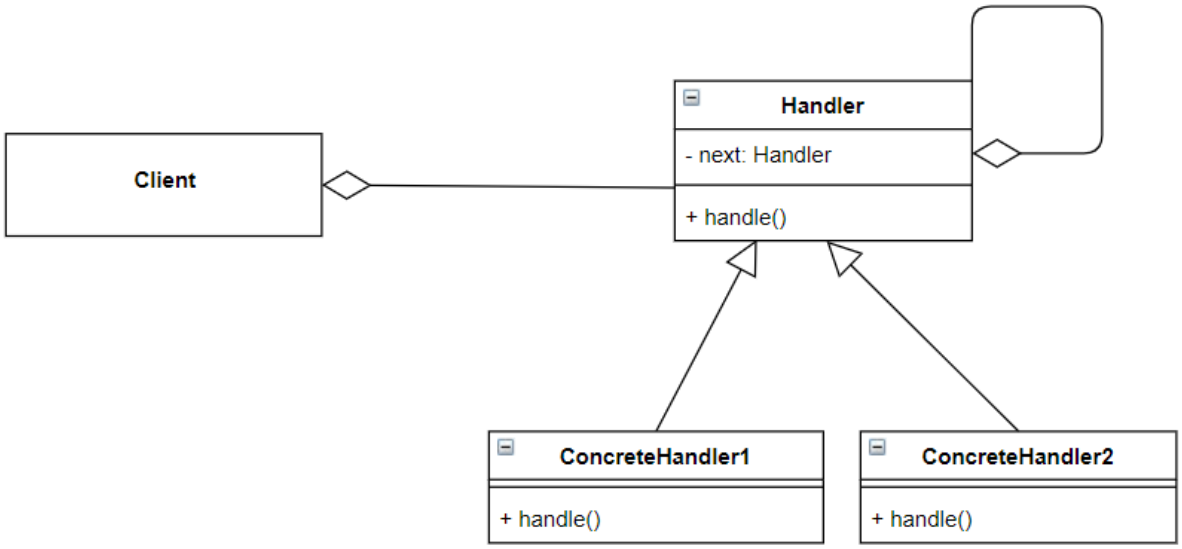
Дефиниција

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



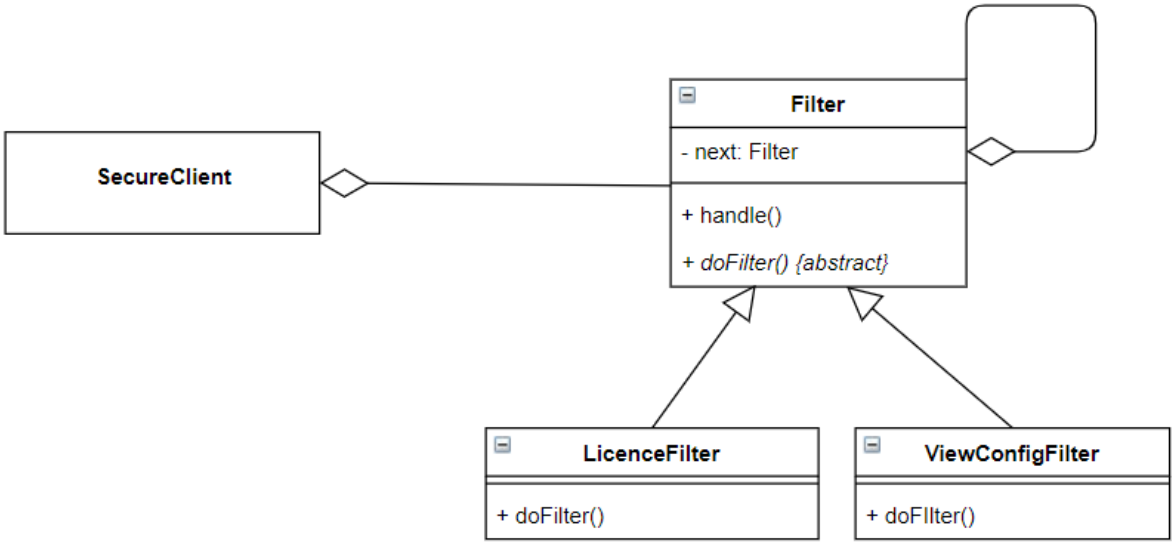
Слика 33 Chain of responsibility патерн

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.

Кориснички захтев PCOR1:

Обезбедити да се пре покретања програма изврши провера валидности конфигурационих фајлова. Пре покретања клијента, сваки конфигурациони фајл ће имати свој филтер и ако један филтер не прође проверу, програм се не може покренути.



Слика 34 Дијаграм класа - Chain of responsibility

ПП6. Template method патерн

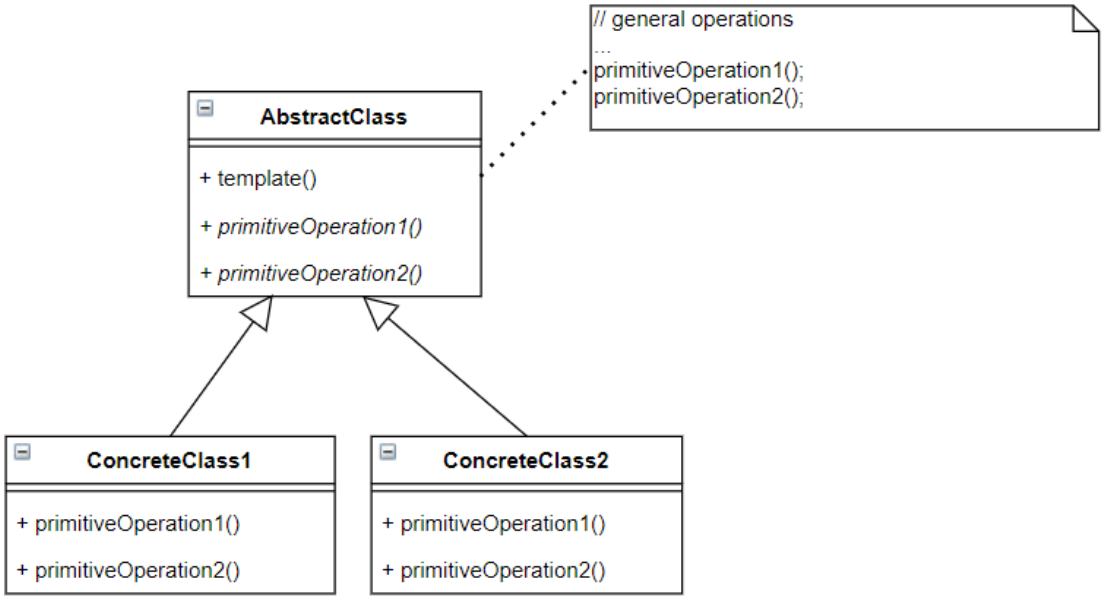
Дефиниција

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



Слика 35 Template method патерн

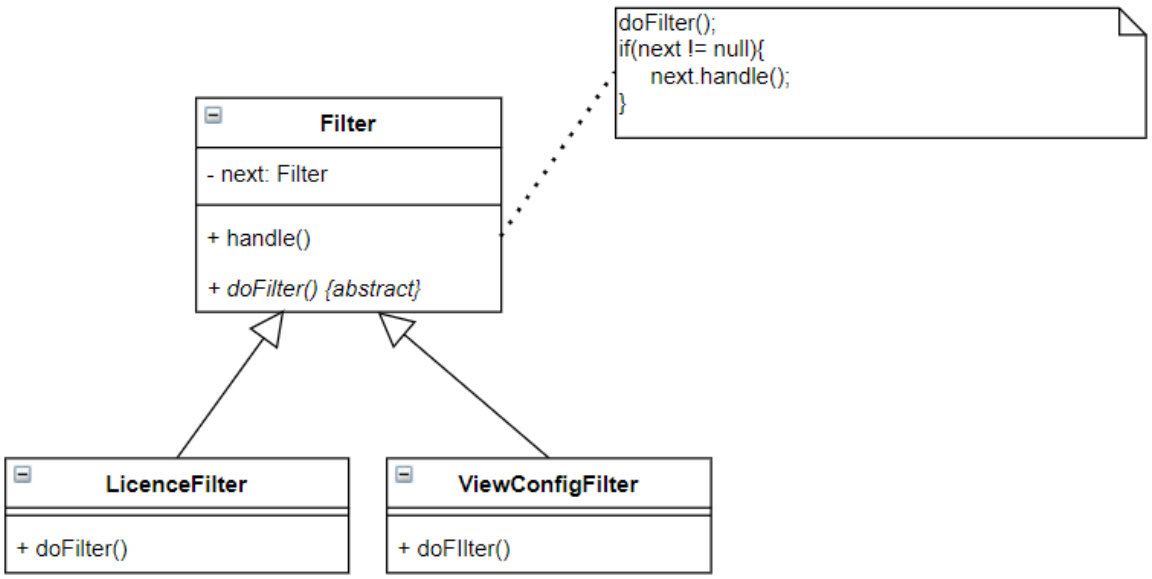
Учесници

- **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.

Кориснички захтев PTM1:

Омогућити да конкретни флитери не морају та имплементирају логику да ли постоји следећи и обрада следећег.

Дијаграм класа примера PTM1



Слика 36 Дијаграм класа Template method

ПП7. Visitor патерн

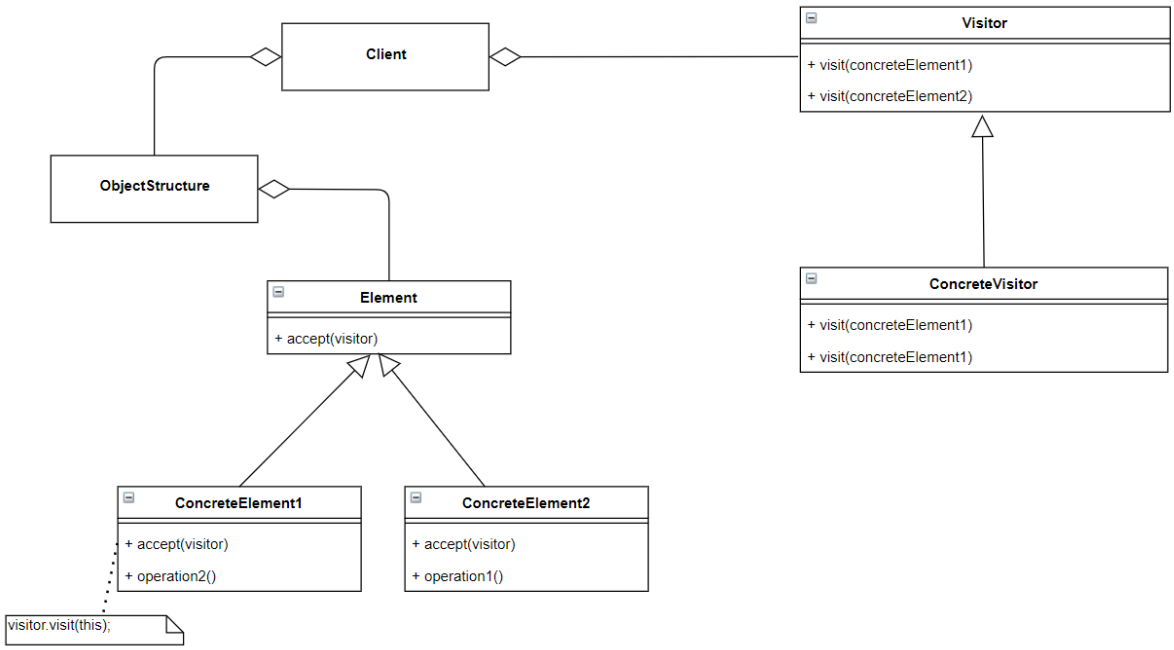
Дефиниција

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



Слика 37 Visitor патерн

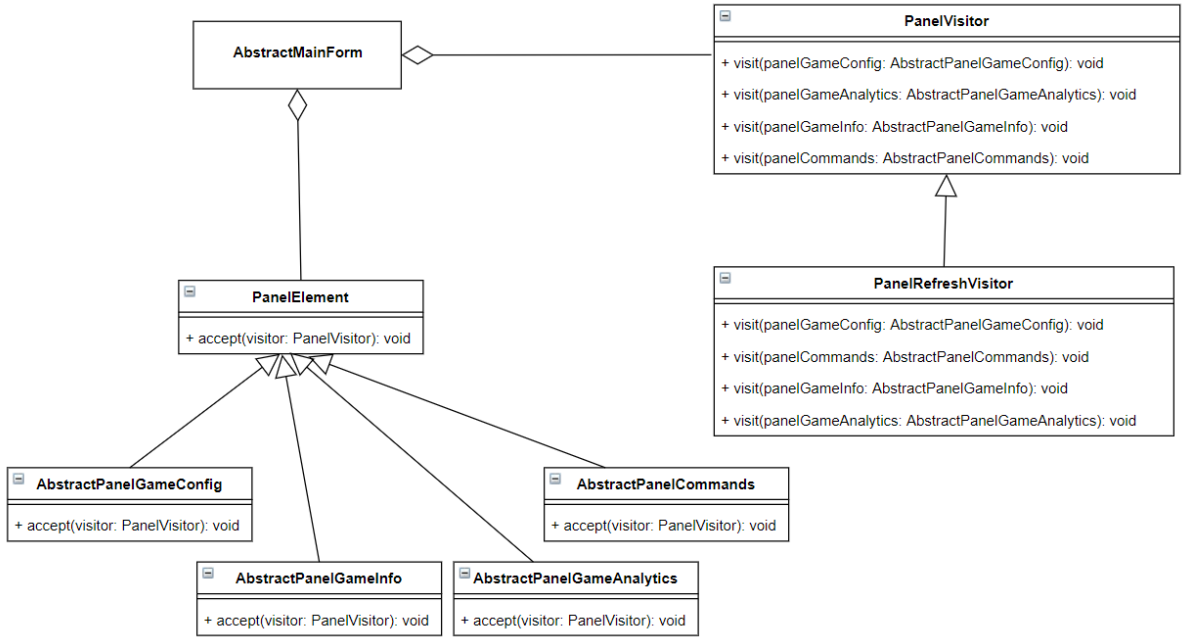
Учесници*:*

- **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*().

Кориснички захтев PVI2:

Одвојити логику освежавања панела у засебну класу и оставити простора да се додаје још класа које ће радити са панелима.

Дијаграм класа примера PVI2



Слика 38 Дијаграм класа Visitor

ПП8. Mediator патерн

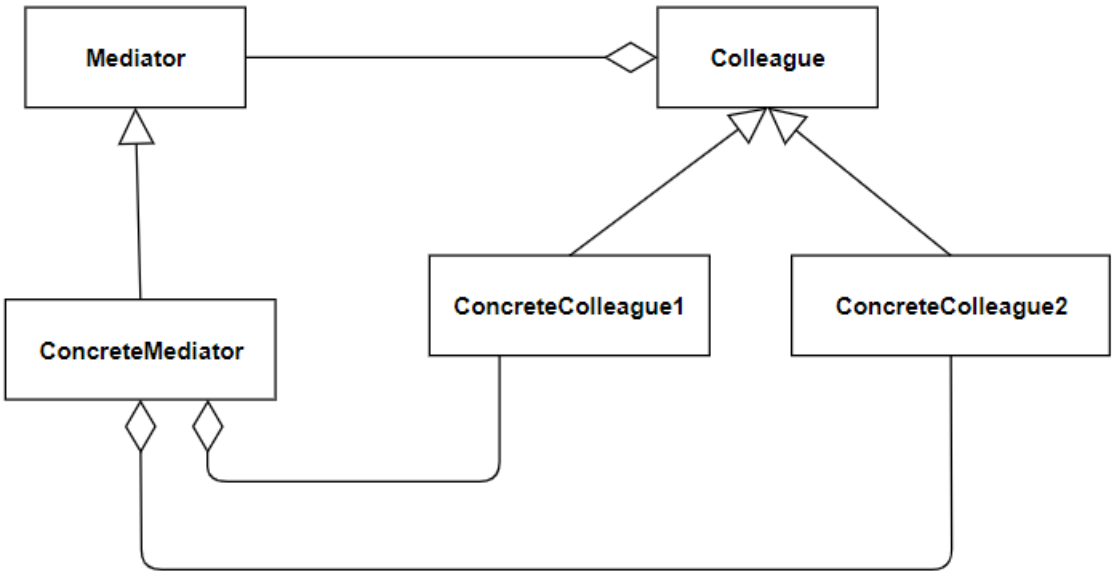
Дефиниција

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



Слика 39 Mediator патерн

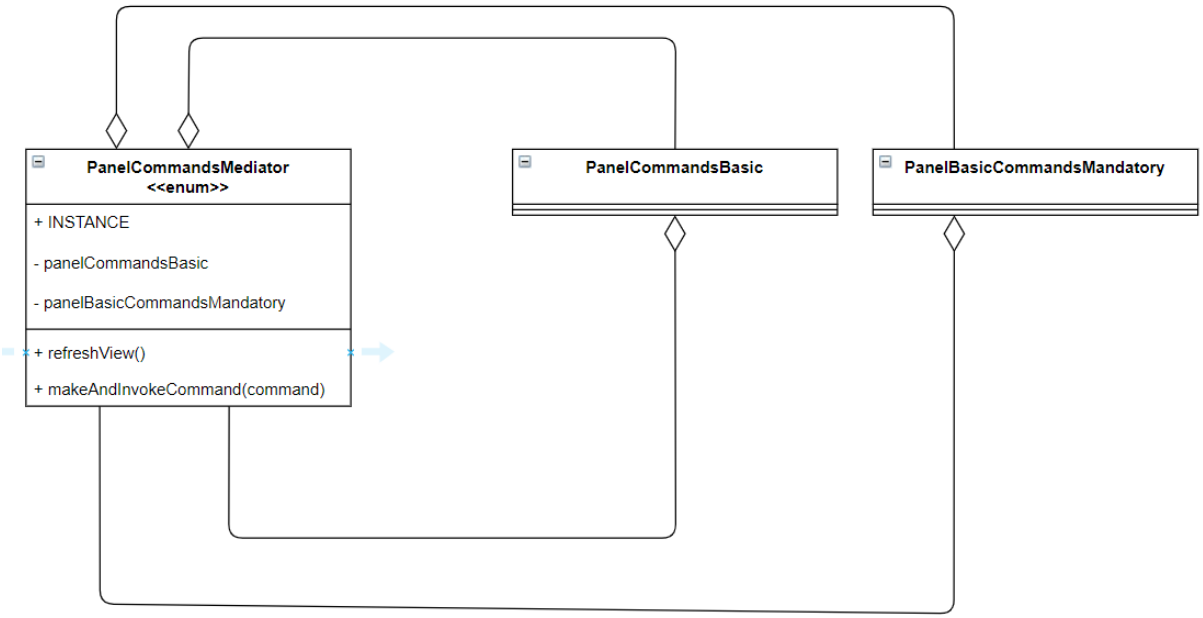
Учесници

- **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.

Кориснички захтев PMED1:

Омогућити да панел за команде и панел за обавезне команде користе класу посредника како међусобно мењале стања.

Дијаграм класа примера PMED1



Слика 40 Дијаграм класа Mediator

ПП9. Iterator патерн

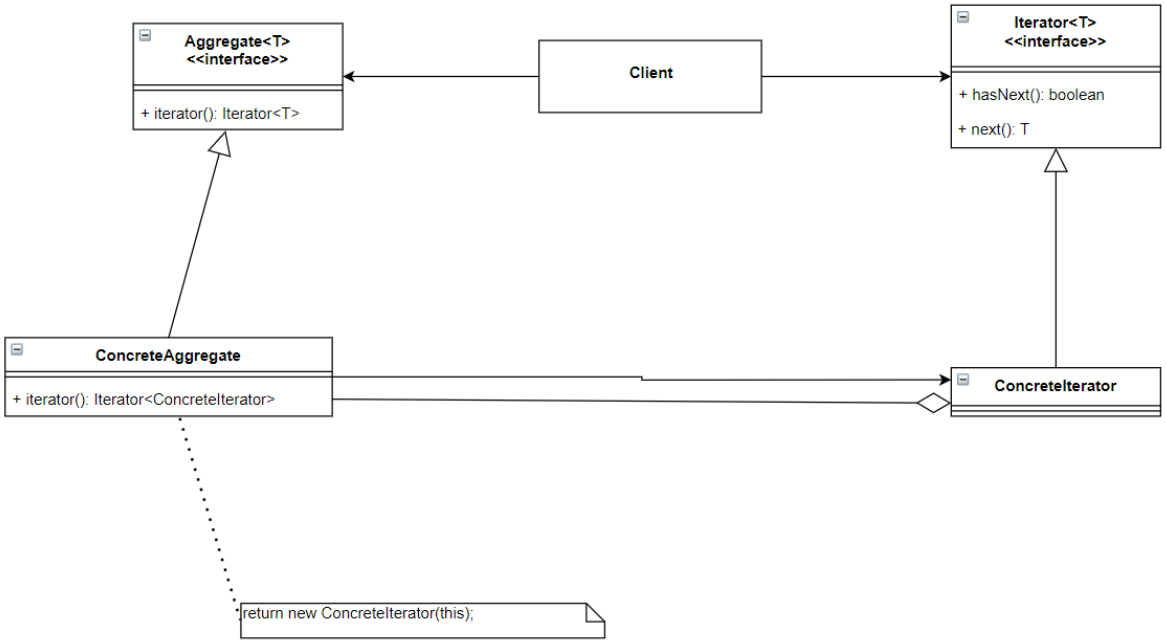
Дефиниција

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.

Структура Iterator патерна



Слика 41 Iterator патерн

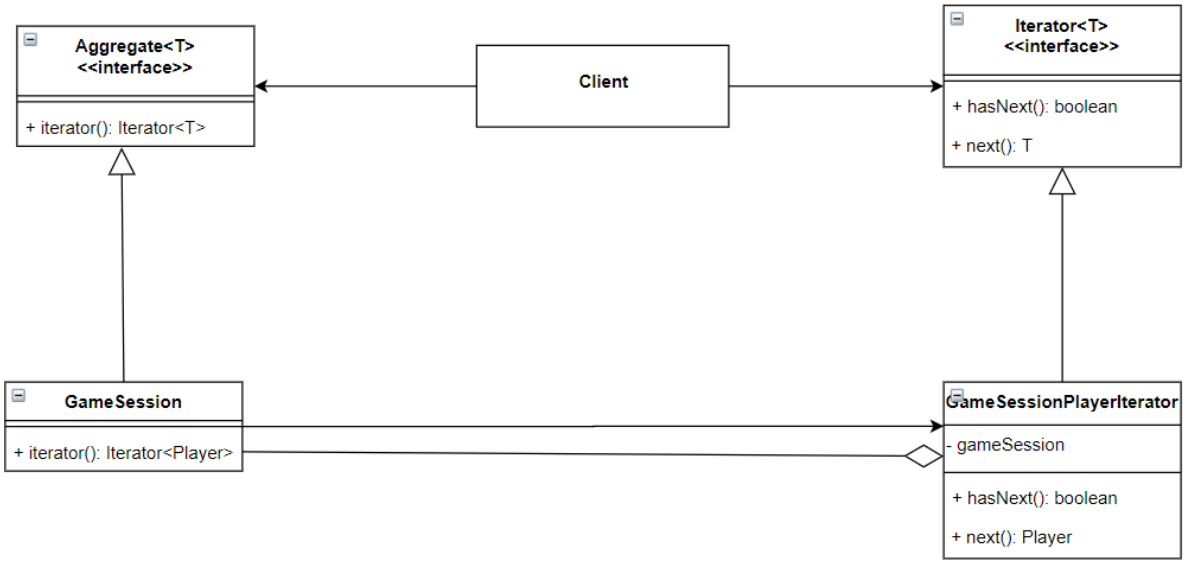
Учесници

- **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.

Кориснички захтев PIT1:

Омогућити пролаз кроз играче сесије игре.

Дијаграм класа примера PIT1



Слика 42 Дијаграм класа Iterator

ПП10. Memento патерн

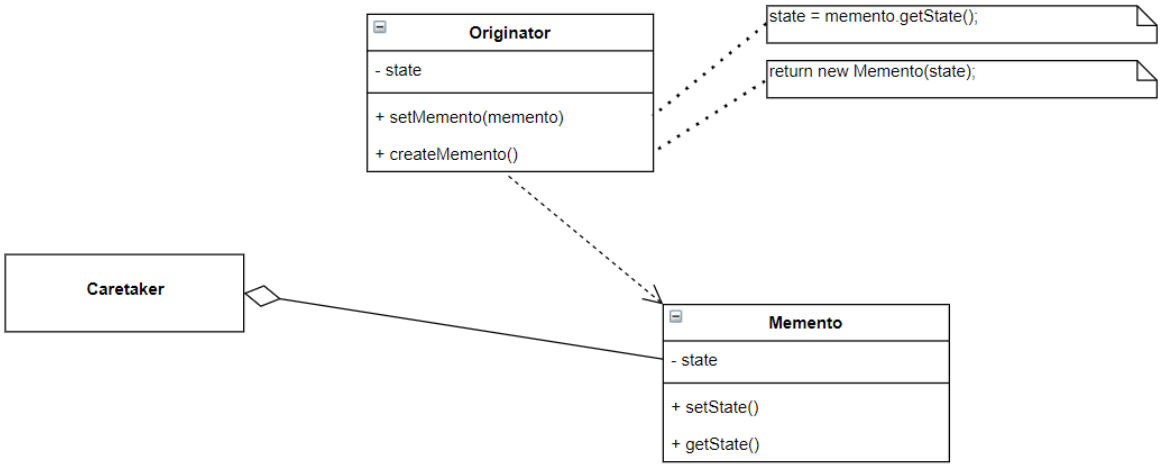
Дефиниција

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



Слика 43 Memento патерн

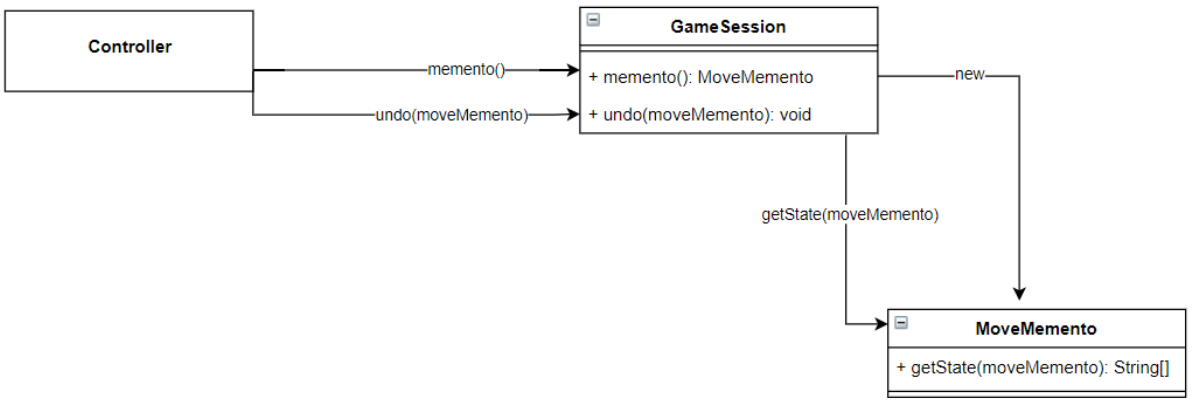
Учесници

- **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.

Кориснички захтев PMEM1:

Омогућити чување стања које узрокују повучени потези у игри како би могла да се поврате.

Дијаграм класа примера PMEM1



Слика 44 Дијаграм класа Memento

ПП11. Interpreter патерн

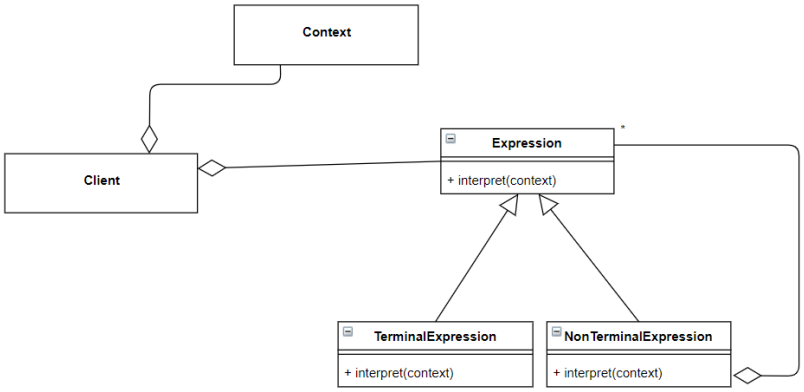
Дефиниција

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.

Структура Interpreter патерна



Слика 45 Interpreter патерн

Учесници

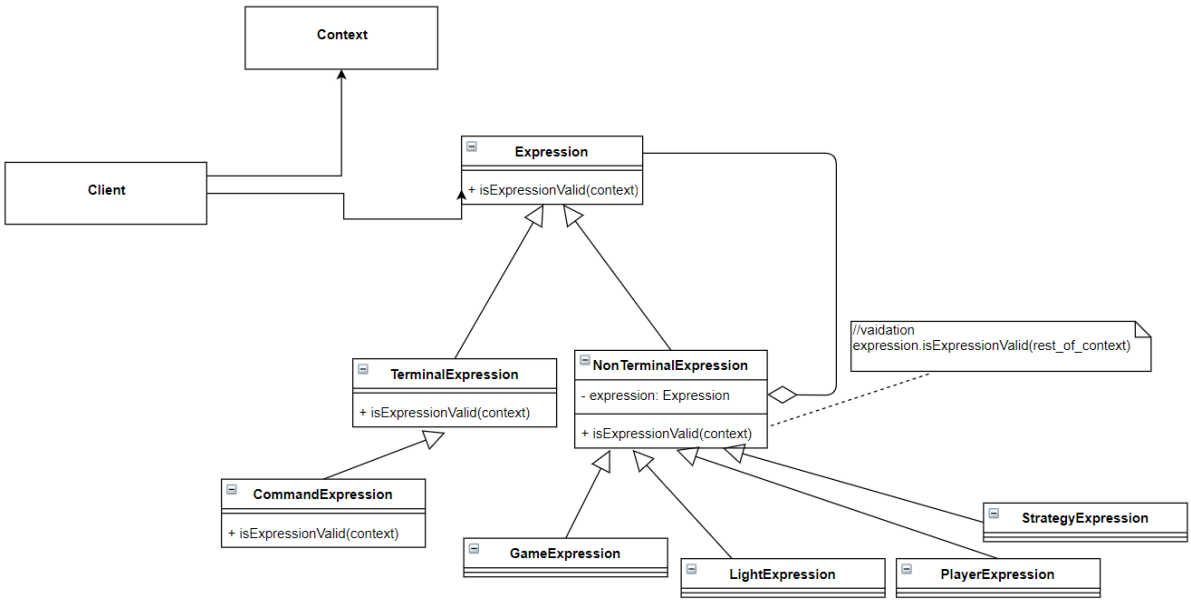
- **Client** - Гради стабло апстрактне синтаксе, репрезентујући (представљајући) реченице језика које су дефинисане граматиком. Стабло апстрактне синтаксе је састављено од *TerminalExpression* и *NonterminalExpression* класа.
- **Context** - Садржи информације које су глобалне за интерпретер.
- **AbstractExpression** - 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[4]. Имплементира *Interpret(Context)* операцију за *NonterminalExpression* симболе (објекте) у граматици.

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

Кориснички захтев PIN:

Због захтева за развијање интерфејса са командне линије, направити интерпретер који ће проверавати да ли је нека команда исправна.

Дијаграм класа примера PIN



Слика 46 Дијаграм класа Interpreter

4. ЗАКЉУЧАК

У овом раду приказани су примери неки од најчешћих патерна који се користе у пројектовању софтвера. Сваки програмер би требало да их за али само као узор. Слепо придржавање структуре патерна може довести до контраефекта. Различити стилови и знање програмера могу проузроковати на први поглед доста различит код али искусан програмер ће видети да се ту ради о једном патерну. Кроз развој софтвера, у овом примеру, показано је да заправо, у једном патерну као део може се наћи део другог патерна. Та испреплетаност заправо даје на значају патернима јер помоћу њих добијамо јасну слику.

Као што смо већ напоменули, ово су само неки од патерна који су дефинисани још давне 1994. године. До данас, број добрих патерна се толико повећао да у сваком програмском језику можемо да нађемо оквире (фрејмворке) који нам аутоматски имплементирају велики број њих. Наравно, добар програмер још увек мора бити свестан њиховог постојања.

5. ЛИТЕРАТУРА

1. Gamma E, Helm R, Johnson R, Vlissides J.: Design Patterns. Addison-Wesley: Reading MA, 1995

2. Siniša Vlajić, Vojislav Stanojević, Dušan Savić, Miloš Milić, Ilija Antović, Saša Lazarević, *The General Form of GoF Design Patterns,* The World of Computer Science and Information Technology Journal (WSCIT), Volume 6, Issue 2. pp. 12-20, 2016, ISSN: 2221-0741.

3. Синиша Влајић: *Софтверски патерни*, Издавач Златни пресек, ISBN: 978-86-86887-30-6, Београд, 2014.