Dizajn Paterni

# Creational Patterns

## Abstract Factory (5)

Interfejs za kreiranje familija slicnih ili zavisnih objekata bez specificiranja njihovih konkretnih klasa. Prakticno object maker koji moze kreirati vise od jednog tipa objekata.

Koristimo kad treba kreirati *set povezanih, slicnih objekata* koji se koriste zajedno kao familije objekata. Konkretne klase du dekaplovane od klijenta.

Razlika izmedju ovog i Method Factory paterna je sto:

1. Method Factory kreira jedan objekat
2. Abstract kreira objekte kroz kompoziciju a method kroz nasledjivanje

Klase:

* AbstractFactory
* ConcreteFactory
* AbstractProduct
* Product
* Client

## Builder (2)

Koristi se kod kreiranja kompleksnih objekata – na primer ako glavna klasa mora da kreira usput jos neke objekte kako bi zavrsila kreiranje svog objekta… U toku visestrukih kreiranja objekata, step by step pristup. Takodje se koristi kada se gradi objekat koji moze da ima vise razlicitih konfiguracija.

Klase:

* IBuilder
* ConcreteBuilder
* Director – odgovoran za korake gradjenja Product-a u raznim konfiguracijama.
* Product

## Prototype (3)

Koristi se kada zelimo da kloniramo neki objekat u novi identican objekat. Tada se deklarise zajednicki interfejs za sve objekte koji treba da omoguce svoje kloniranje. Obicno taj interfejs ima samo jedan metod *Clone*, koji sve klase koje nasledjuju taj interfejs treba da implementiraju.

Inace Prototype patern je ugradjen u C# sa *Cloneable* interfejsom.

## Factory Method (5)

Definise se interfejs za kreiranje objekata ali ostavlja podklasama da odluce koje klase objekat ce da instanciraju. Koristi se kad se ne zna unapred koji objekti ce tacno trebati da se kreiraju.

Sastoji se iz klasa:

* ConcreteCreator koja implementira Creator interfejs i ima FactoryMethod koji po ulaznim parametrima odlucuje koja ce se tacno ConcreteProduct klasa instancirati.
* Creator interfejs koji implementira ConcreteCreator
* Product interfejst, implementiraju sve ConcreteProduct klase
* ConcreteProduct klasa

## Singleton (4)

Kada zelimo da klasa ima samo jednu instancu. Ili kada zelimo da se nekim objektima pristupa bilo odakle iz programa. Obicno se to implementira na sledeci nacin:

1. Defaultni konsturktor se postavi kao private kako bi se sprecilo da se iz nekog dela koda instancira jos jedna instance ove klase.
2. Kreirati staticki metod za kreiranje koji ce da se ponasa kao konstruktor. Ovaj metod pozove privatni konsturktor da kreira objekat i sacuva ga u statickom polju. Svaki drugi poziv ovom metodu ustvari vraca ovaj isti objekat.

Problemi sa Singletonom:

* Krse prvi princip SOLIDa Single responsibility
* Problem kod multitredinga pa mora dodatna threadsafe implementacija
* Problem kod unit testinga zbog mokanja singletona.

# Structural Patterns

## Adapter (4)

Strukturalni patern koji omogucava objektima sa nekompatibilnim interfejsima da kolaboriraju. Adapter ustvari “vrepuje” jedan od objekata kako bi sakrio kompleksnost konvertovanja. Objekt koji je vrepan nije u sustini ni svestan postojanja Adaptera. Adapter funkcionise po principu:

1. Ima interfejs kompatibilan objektu A.
2. Koriscenjem ovog interfejsa, objekat A moze da poziva metode Adaptera.
3. Kada Adapter primi zahtev od objekta A, on prosledjuje taj zahtev objektu B koji je vrepan u njemu ali u format u kome ih objekat B ocekuje.

Takodje je moguce i da adapter bude dvosmeran.

## Bridge (3)

Strukturalni patern koji omogucava da se velika komplikovana klasa u koju su spojene neke korelirane funkcionalnosti razdvoji u dve hijerarhije – abstrakciju i implementaciju koje se svaka moze nezavisno dalje razvijati.

Ona u sustini prebacuje neke funkcionalnosti sa extednanja parent klase da bi mogle da se dodaju nove klase koje je nasledjuju, na kompoziciju klasa, gde ce se stvoriti dve posebne hijerarhije i nove funkcionalnosti ce biti ubacene u novu klasu koju ce prethodna da sadrzi.

Abstrakcija i implementacija se ovde posmatraju kao:

Abstrakcija (interfejs) – high level kontrolni lejer koji sam po sebi ne radi nista

Implementacija (Platform) – sav posao abstrakcija delegira ovom sloju.

Abstrakcija je kao GUI, a implementacija je kao API.

## Composite (4)

Strukturalni dizajn patern koji omogucava da se klase organizuju u razgranatu strukturu (tree structure) I onda da se radi sa ovim strukturama kao sa individualnim objektima. Koriscenje ovog paterna ima smisla samo ako osnovni model aplikacije moze da se predstavi kao razgranata struktura.

Stablo se sastoji prakticno od kompozita (grana) i od „lisca“. I postoji zajednicki interfejs koji nasledjuju i grane i listovi. Benefit ovoga je da korisnik ove strukture ne mora da se brine oko konkretnih klasa objekata koji cine drvo. Ne mora da se brine da li je objekat kompozit ili list, svi se tretiraju isto preko zajednickog interfejsa, kada se pozove odredjeni metod, objekti sami prosledjuju poziv u dubinu grananja.

Klase:

* Component – predstavlja interfejs za objekte koji cine drvo, kompozite i lisce. Deklarise metode zajednicke za sve komponente stabla, deklarise metode za pristup child komponentama u stablu, a moze da deklarise i metode za pristup parent komponentama stabla.
* Leaf – predstavlja listove u stablu, od njih nema dalje grananja. Definise ponasanje tih primitivnih objekata u stablu (koompoziciji)
* Composite – definise ponasanje objekata koji imaju children, cuva child komponente, implementira deo interefjsa vezan za child komponente.
* Client – aplikacija koja manipulise ovom strukturom

## Decorator aka. Wrapper (3)

Wrapper je objekat koji moze biti povezan sa nekim *target* objektom. Sadrzi isti set metoda koje sadrzi i target i delegira na njega sve requestove koje on sam dobije (sto se tice klijenta, ti objekti su identicni). Ali wrapper moze da izmeni rezultat tih poziva tako sto uradi jos nesto pre ili posle prosledjivanja poziva targetu. U Dekorator paternu se umesto nasledjivanja koriste principi agregacije[[1]](#footnote-1) ili kompozicije[[2]](#footnote-2)

Klase koje ucestvuju:

* Component – definise interfejs za objekte kojima se moze dinamicki dodati ponasanje
* ConcreteComponent – definise objekt kome se mogu dodati dodatna ponasanja
* Decorator – drzi referencu na komponent objekat i definise interfejs koji se slaze sa interfejsom Component
* ConcreteDecorator – dodaje ponasanja komponenti

## Facade (5)

Fasada je strukturalni patern koji pruža klasu koja je ustvari pojednostavljeni interfejs za kompleksan sistem. Takodje, fasada moze da se desi da pruzi samo limitirane mogucnosti u odnosu na koriscenje celog sistema direktno. Ona prakticno ukljucuje samo one mogucnosti koje trebaju odredjenom klijentu. Korisna kada hocemo da koristimo neku naprednu kompleksnu biblioteku u nasoj aplikaciji, ali nam treba samo mali deo mogucnosti.

Takodje, Fasada se moze koristiti kada neki kompleksan substitem hocemo da podelimo u lejere, kreiraju se fasade koje pristupaju svakom lejeru. Na taj nacin se moze umanjiti cupling izmedju razlicitih podsistema, terajuci ih da komuniciraju samo kroz fasade.

Klase koje ucestvuju:

* Facade – zna koji podsistem je zaduzen za koji zahtev i delegira zahteve podsistemima
* Subsystem classes – implementira funkcionalnosti sabsistema, hendla zahteve koji su poslati od fasade. Nemaju znanje o samoj fasadi.

## Flyweight

## Proxy (4)

Koristi se kada imamo masivan objekat koji zauzima dosta resursa a ne koristi se cesto. Umesto da se inicijalizuje jednom i nepotrebno drzi u memoriji, mogu svi klijenti da ga inicijalizuju po potrebi, ali to bi dovelo do toga da se dosta koda duplicira. Takodje, nekad je nemoguce i kod tog servisa prilagoditi jer moze da nam bude nedostupan (thirdParty)

Umesto toga, kreira se nova Proxy klasa koja ima isti interfejs kao originalni servis koji se prikriva iza proksija. Onda se aplikacija apdejtuje tako da svi klijenti koriste proxi klasu umesto relanog objekta. Kada proksi objekat dobije zahtev od klijenta, krera se realni objekat servisa i delegira mu se zahtev.

Klase koje ucestvuju:

* Proxy – drzi referencu na realni servis, ima isti interfejs servisu tako da se lako mogu zameniti, kontrolise pristup servisu i brine oko inicijalizacije i brisanja njegovog objekta.
* Subject - definise zajednicki interfejs za proxy i realni servis, tako da se lako mogu zameniti.
* RealSubject – definise realni objekat koji proxi zapravo predstavlja.

# Behavioral

## Chain of responsibility (2)

Bihejvoralni patern koji transformise odredjena ponasanja (behaviors) u standalone objekte koji se nazivaju handleri. Dalje, se ti hendleri linkaju i lanac gde svaki hendler ima polje u kome cuva referencu na sledeci hendler u lancu. Svaki hendler prima zahtev (request) i obradjuje ga ili ga prosledjuje dalje u lancu sledecem hendleru. Po jednom pristupu, zahtev putuje kroz lanac sve dok ne prodje procesuiranje svih hendlera, po drugom handler odlucuje da li ce on da procesira zahtev ili ce da ga posalje dalje kroz lanac tako da po ovom pristupu jedan hendler preocesrira zahtev ili ni jedan. Ovaj drugi pristup je uobiucajen kod obrade evenata grafickog interfejsa. (primer: kada se klikne na dugme, ovaj event putuje kroz lanac GUI elemenata od dugmeta, grupe, panela, forme i zavrsava se sa glavnim prozorom aplikacije. Event obradjuje prvi elemenat u nizu koji ima hendler za njega).

Bitno je da sve hendler klase od kojih se gradi lanac, implementiraju isti interfejs, svaki konkretan hendler mora da implementira metod *execute*. Na ovaj nacin, sami lanci se mogu kreirati u rantajmu od razlicitih hendlera bez da se „kaplaju“ konkretne klase.

Klase koje ucestvuju:

* Handler – definise interfejs za hendlanje rekvestova, opciono imeplementira link na naslednika u lancu
* ConcreteHandler – hendla zahteve za koji je odgovorna, moze da pristupi sledecem u lancu, ako moze da procesuira request, procesuira ga, inace ga prosledi sledecem u lancu.
* Client – Inicira request ConcreteHandler objektu u lancu

## Command (4)

Bihejvoral patern koji implementira princip *separation of concenrs* koji se svodi na deljenje aplikacije u slojeve (layers). Primer je GUI, gde je GUI layer odgovoran za prikaz a svaki dodatni rad osim prikaza se delegira biznis sloju. Obicno GUI poziva metod iz biznis lejera i prosledjuje mu neke parametre.

Kod *Command* paterna, se takvi zahtevi ne salju direktno vec se izvuku svi detalji zahteva (destinacija objekt, metod i parametri) i smeste se u command objekat sa jednim jedinim metodom koji dalje prosledjuju taj zahtev kome treba. Command objekti sluze kao link izmedju GUI lejera i biznis logike. GUI objekat ne treba da zna koji objekat biznis logike ce da hendluje azhtev, GUI samo trigeruje komandu, ona dalje hendla sve detalje. Takodje, vise slicnih GUI akcija koje predstavljaju istu opreaciju moze da trigeruje istu komandu, sto resava code duplication.

Sledeci korak – sve komande implementiraju jedan isti interfejs koji obicno ima samo jednu metodu i ne prima nikakve parametre. Tako jedan sender moze da koristi razne komande bez da se kapluju sa konkretnim klasama komandi. Ovako se lako mogu i promeniti komande koje su pridruzene odredjenom senderu. Takodje, posto se po ovom paternu, komandi ne prosledjuju dodatni parametri od strane GUIa nego se samo trigeruju, komanda mora da bude prekonfigurisana sa ulaznim parametrima **ili da je u stanju sama da ih pokupi**.

Komande ovako postaju middle layer koji jos dodatno umanjuju kaplovanje izmedju GUI i biznis sloja.

Klase koje ucestvuju:

* Command – deklarise interfejs za sve komande
* ConcreteCommand – definise vezu izmedju akcije i objekta koji je izvrsava (Receiver), implementira Execute tako sto poziva odredjene metode na Receiver-u
* Client – instancira ConcreteCOmmand objekat i setuje njegovog risivera
* Invoker – inicira slanje zahteva komandi
* Receiver – izvrsava operaciju vezanu za zahtev

## Iterator (5)

Glavna ideja iterator paterna je da izdvoji navigaciju po kolekciji na odredjen nacin u poseban objekat. Takodje, itereator sam prati svoju poziciju u kolekciji i koliko jos elementata u kolekciji postoji, tako da se moze koristiti vise iteratora paralelno na istoj kolekciji, skroz medjusobno nezavisnih.

Obicno iteratori sadrze jedan obavezan metod za dohvatanje elemenata iz kolekcije, ostali metodi su dodatak. Takodje, svi iteratori moraju da implementiraju jedan interfejs, sto cini klijentski kod kompatibilan sa kolekcijama svih tipova ili razlicitim algoritama za navigaciju kroz kolekciju dokle god postoji potrebni iterator. Ako je potrebno da se kroz kolekciju ide po nekoj odredjenom algoritmu (na primer ako kolekcija nije uredjena kao niz nego stablo) samo se kreira nova iterator klasa bez da se menja sama kolekcija ili mnogo promena na klijentskoj strani.

Klase koje ucestvuju:

* Iterator – definise interfejse za sve ostale iteratore za pristup i prolazak kroz elemente kolekcije
* ConcreteIterator – implementira interfejs, prati trenutnu poziciju u kolekciji dok se prolazi kroz nju
* CollectionInterface – definise interfejs za kreiranje iteratora za konkretnu kolekciju
* ConcreteCollection – implements collectionINterface and creation of the proper ConcreteIterator

## Mediator (2)

Po mediator paternu, sva direktna komunikacija izmedju komponenti za koje zelimo da su nezavisne jedna od druge treba da prestane. Umesto toga, sve komponente komuniciraju posredno pozivajuci specijalni medijator objekat koji preusmeravaju pozive ka odgovarajucim komponentama. Kao rezultat, komponente su zavisne samo od pojedinacnih medijatora klasa, ne i od ostalih komponenti.

Na ovaj nacin, mediator patern moze da enkapsulira kompleksnu mrezu veza izmedju razlicitih objekata u jedan jedini mediator objekat. Sto manje dependencia ima klasa, lakse ju je modifikovati, naslediti ili ponovo koristiti.

Klase koje ucestvuju:

* Mediator – definise interfejs za komunikaciju izmedju objekata
* ConcreteMediator – implementira kooperativno ponasanje koordinisuci izmedju objekata, zna za i odrzava vezu sa objektima koji komuniciraju preko njega
* Colleague – svaka Colleague klasa zna za svoj Mediator objekat i komunicira sa njim svaki put kada bi direktno komunicirala sa ostalim objektima direktno.

## Memento (1)

Memento patern, resava problem narusene enkapsulacije kod pokusaja da sacuvamo prethodno stanje nekog objekta. Memento patern ustvari delegira pravljenje snepshota stanja objektu cije se stanje i zeli zapamtiti u memento – originatoru. Tako da umesto da se pokusava zapamtiti snaphsot objekta sa spoljasnje strane, od strane nekog drugog objekta i time se naruse svi skriveni fildovi i metode, sam originator pravi svoj snepshot i smesta ga u specijalan objekat *memento*.

Sadrzaj memento objekta nije pristupacan ni jednom drugom objektu osim onom koji ga je kreirao. Ostali objekti mogu, uz pomoc limitiranog interfejsa, samo da procitaju metadata mementa (creation time, name of operation...) ali ne detaljne vrednosti samog stanja objekta sadrzanog u snepshotu.

Tako kreirani mementoi se cuvaju u posebnim objektima nazvanim caretakers. Caretaker radi sa mementom samo preko posebnog limitiranog interfejsa, ne moze da pristupi detaljima snepshota niti da ih menja. U isto vreme, samo originator ima pristup poljima u mementu i to mu omogucava da se vrati u svoje predjasnje stanje.

Klase koje ucestvuju:

* Memento – cuva interno stanje Originator objekta. Moze da cuva samo one podatke koje originator odluci da su bitni za cuvanje. Stiti podatke od pristupa drugih objekata koji nisu originatori. Memento efektivno ima dva interfejsa, onaj limitirani koji vidi caretaker i onaj potpuni koji vidi Originator kako bi imao pun pristup.
* Originator – kreira memento sa sadrzanim svojim snepshotom, koristi vec kreiran memento da vrati svoje predjasnje stanje
* Caretaker – odgovoran za cuvanje mementa, nikad ne pristupa detaljima mementa.

1. Agregacija – veza izmedju dva objekta takva da child postoji nezavisno od parenta i ako se parent objekat obrise, child ostaje da zivi. Parent nije ekskluzivni kontejner child-a. [↑](#footnote-ref-1)
2. Kompozicija – veza izmedju dva objekta gde postoji lifecycle dependency gde kada se objekat parent klase obrise, sa njim se brise i child objekat. [↑](#footnote-ref-2)