Softverski paterni na primeru mini aplikacije za evidenciju poslovanja preduzeća

SADRŽAJ

ŠTA SU PATERNI I ČEMU SLUŽE

Patern u suštini opisuje problem koji se javlja iznova(klasu problema) i "generičko"rešenje tako da se to rešenje može primeniti uvek na drugačiji način nad tom klasom problema.
U razvoju softvera paterni projektovanja omogućavaju da rešenje problema bude specifično kako bi se rešio konkretan problem ali i dovoljno opšte kako bi bilo rešenje ili deo rešenja za buduće probleme, da bi se minimiziralo ponovno projektovanje rešenja za neku klasu problema.
Takodje paterni projektovanja omogućavaju dinamičku izmenu funkcionalnosti u toku izršavanja.
Patern je proces koji vrši trasformaciju strukture problema(uredjene dvojke(Klijent,Koknterni server)) u strukturu rešenja (uredjena trojka(Klijent,Apstraktni server,Konkretni server). Ova definicija prestavlja opšti oblik GOF paterna projektovanja i ukazuje na jedan od osnovnih principa OO programiranja -Programiranje ka interfejsu a ne implementaciji.
Klijent je element strukture koji koristi funkcionalnosti apstraktnog i konkretnog servera kako bi ostvario sopostvenu funkcionalnost.
Apstraktni server je element strukture koji klijentu obezbedjuje apstraktnu funkcionalnost koja se može realizovati od strane više konkrentnih servera.
Konkretni server je element strukture koji klijentu daje konkretnu funkcionalnost koja prestavlja realizaciju apstraktne funkcionalnosti.
Na ovaj način svaka izmena funkcionalnosti koju klijent koristi vrši se u klasama konkrenih servera, i sve dok se interfejs (apstraktni server) ne menja klijentska klasa se ne mora menjati, što upravo pokazuje da je klijentska klasa zavisna samo od apstraktnog servera a ne i od konkretne implementacije. Takodje ukoliko se dodaju nove funkcionalnosti klijent toga ne mora biti svestan, on jednostavno kroz konstruktor ili seter metodu dobija objekat bilo koje klase koja nasledjuje apstraktni server(kompatibilnost objektnih tipova) a zatim se u vremenu izvršavanja programa umesto u vremenu kompajliranja odredjuje objekat čija će metoda biti pozvana(dinamički polimorfizam).
Zbog toga je navedena struktura rešenja održiva jer klijent u vreme kompajliranja nije vezan za jedan konkretni server što bi onemogućilo fleksibilnost programa u toku izvršavanja.
Ovaj pricip ima ogroman značaj pri razvoju velikih aplikacija gde se teži minimalnoj zavnisnosti izmedju klasa, kako bi se aplikacija što lakše održavala i razvijala, i kod aplikacija koje zahtevaju dinamičku izmenu funkcionalnosti u toku izvršavanja.

GOF parterni projektovanja se dele u tri grupe:
Kreacioni paterni koji pomažu da se izgradi sistem nezavisno od toga kako su objekti kreirni komponovani i reprezentovani
- Abstract Factory
- Builder
- Factory Method
- Prototype
- Singleton

Strukturni paterni koji opisuju složene strukture medjusobno povezanih klasa i objekata
- Adapter patern
- Bridge patern
- Composite patern
- Decorator patern
- Facade patern
- Flyweight paterna
- Proxy paterna

Paterni ponašanja koji opisuju način na koji klase ili objekti saradjuju i rasporedjuju odgovornosti.
- Chain of responsability
- Command
- Interpreter
- Iterator
- Mediator
- Memento
- Observer
- State
- Strategy
- Template method
- Visitor

Na primeru mini aplikacije za evidencuju poslovanja preduzeća biće implementirano i objašnjeno 8 od 23 GOF paterna projektovanja.

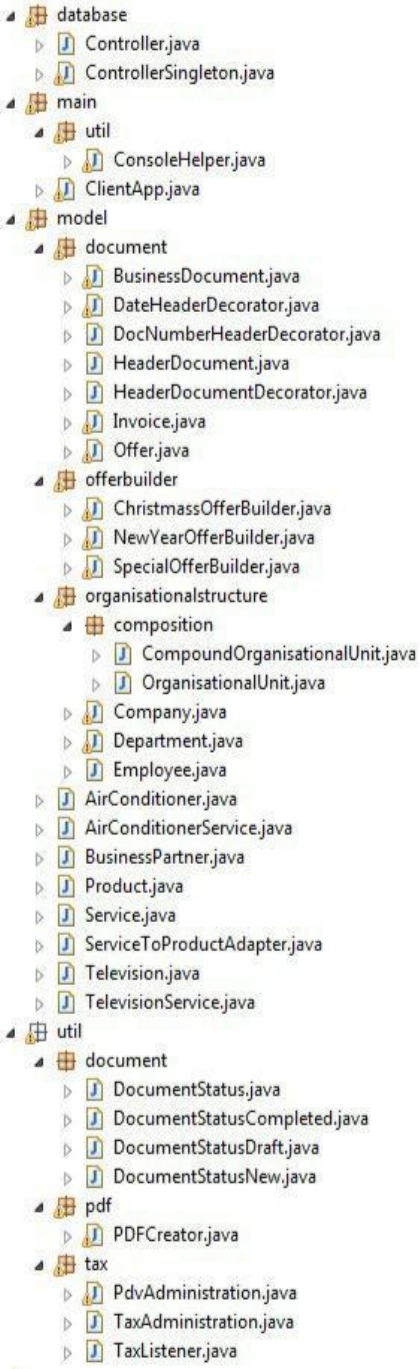
SPECIFIKACIJA APLIKACIJE

"Business document management" aplikacija služi za kreiranje ponuda i faktura u pdf formatu. Aplikacija omogućava korisniku da čuva proizvode i uluge i da zatim na osnovu tih proizvoda konstruiše pomenuta dokumenta.
Moguće je čuvati dokumenta u tri različita statusa. Inicijalno dokumenta su u statusu "new" što znači da još nisu sačuvana(čuvanje je implementirano kao lista u operativnoj memoriji). Nakon što se dokument sačuva on je u statusu "draft" što znači da još nije izvršeno procesiranje linija dokumenta. Nakon što se dokument kompletira prelazi u status "complete"(izvršeno procesiranje linija). Takodje u ovkiru aplikacije omogućeno je da se vodi evidencija o trenutnoj stopi pdv-a i da se eventualna promena pdv-a propagira na sva kreirana dokumenta i izvrši regenerisanje kreiranih pdf-ova. Pored kreiranja i obrade novih ponuda postoje i predefinisane specijalne ponude koje se mogu dobiti na zahtev.
Na početku je aplikacija radila samo sa prozvodima a zatim su dodate i usluge kako bi preduzeća koja se bave uslužnim delatnostima takodje mogla koristiti aplikaciju. Pored navedene osnovne funkcionalnosti upravljanja poslovnim dokumentima aplikacija omogućava čuvanje i prikaz organizacione strukture kompanije za koju se dokumenta obradjuju.

ARHITEKTURA APLIKACIJE

Aplikacija se sastoji iz četiri osnovna dela:
1.Database(paket database) koji je zadužen za čuvanje u operativnoj memoriji dokumenata,proizvoda i usluga. Čuvanje je implementirano "in memory", što znači da bazu predstavljaju zapravo liste u operativnoj memoriji. U ovom delu aplikacije implementiran je **Singleton** patern o čemu će biti više reči u nastavku teksta.
2.Model(paket model i njegovi podpaketi) u kome se nalaze binovi dokumenata, proizvoda i usluga kao i klase zadužene za kreiranje ovih binova. Takodje nalaze se i binovi koji predstavljaju kompaniju, sektore i zaposlene kao i pomoćna klasa sa njihovo komponovanje u organizacionu strukturu. U ovom delu aplikacije implementirana su 5 paterna: **Composite**, **Builder, Adapter, Template method i Decorator** patern.
3.Util(paket util i njegovi podpaketi) u kome se nalaze klase u kojima je implementiran status dokumenata(util.document paket), klase u kojima je implementirana evidencija zakonskih propisa , trenutno samo pdv-a(util.tax paket), kao i klasa zadužena za generisanje pdf dokumenata(util.pdf paket). U ovom delu aplikacije implementirani su **State i Observer** paterni.

4.Klijentski deo(paket main) koji zapravo predstavlja front end deo aplikacije(implelentiran konzolno).



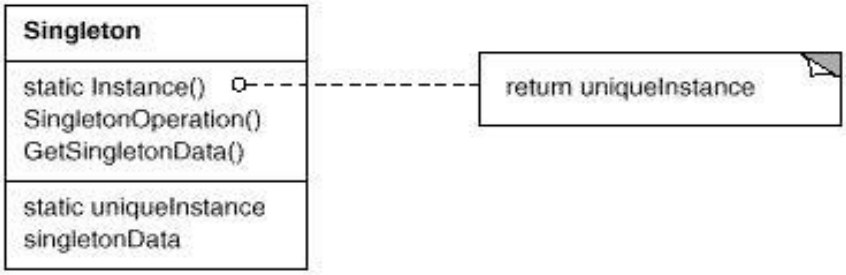
SOFTVERSKI PATERNI NA PRIMERU MINI APLIKACIJE ZA EVIDENCIJU POSLOVANJA PREDUZEĆA

PETERNI U APLIKACIJI

1.Singleton

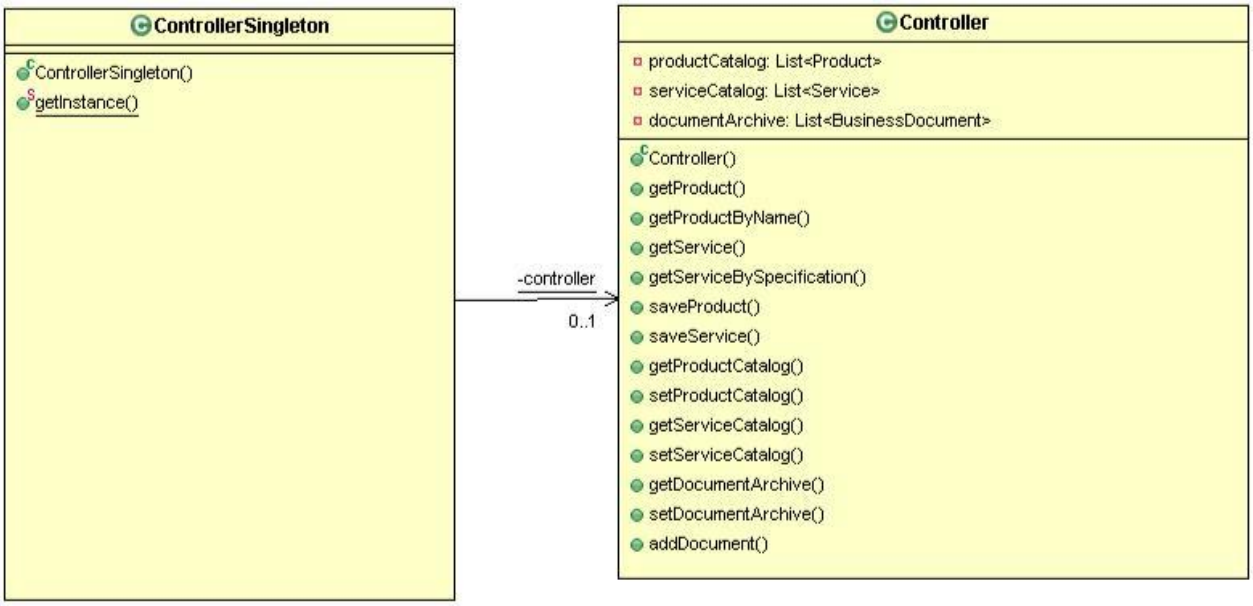
Singleton je najjednostavniji patern koji je upotrebljen u aplikaciji.
Definicija Singleton paterna: Obezbedjuje klasi samo jedno pojavljivanje i globalni pristup do nje.

Struktura Singleton paterna:



ControllerSingleton klasa koja obezbedjuje kontroleru samo jedno pojavljivanje i globalni pritup do njega:

Struktura paterna u okviru aplikacije:



Objašnjenje i korisnički zahtev:
Kao što se vidi sa slike klasa Controller ima ulogu "in memory" baze, a klasa ControllerSingleton obezbedjuje jedistveni pristup do nje preko statičke metode getInstance().
Patern je primenjen kako bi cela aplikacja radila nad jednom instancom baze, tj čuvala objekte i izvačila objekte iz isith kolekcija.
Prvi put kada neko zatraži instancu kontrolera(pozove metodu getInstance()) ControllerSigleton klasa će instancirati kontrolera i vratiti instancu, a svaki sledeci put ce vraćati postojeću instancu.

2. Adapter patern

Definicija adapter paterna: Konvertuje interfejs neke klase(Adaptee) u drugi interfejs(Target) koji klijent(Client) očekuje. Adapter patern omogućava zajednički rad klasa koje imaju nekomatibilne iterfejse.

Učesnici:

- Client(Offer)

- Saradjuje sa interfejsom Adaptee preko interfejsa Target

- Target(Product)

- Definiše domenski-specifičan iterfejs koji klasa Client koristi.

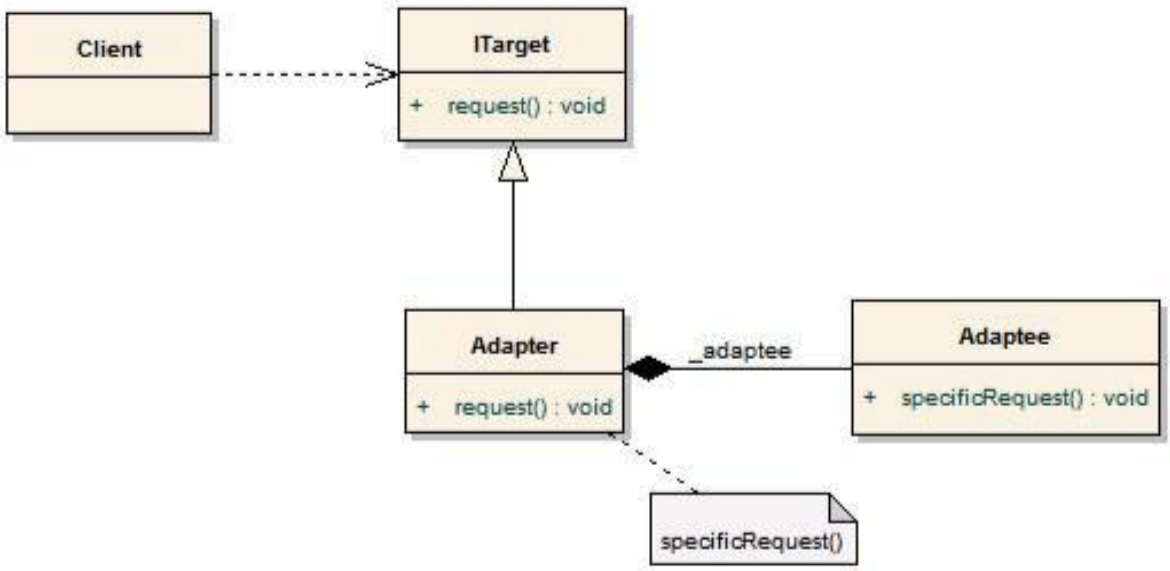
- Adapter(ServiceToProductAdapter)

- Adaptira(prilagodjava) interfejs Adaptee interfejsu Target

- Adaptee(Service)

Definiše postojeći iterfejs koji treba adaptirati

Struktura adapter paterna:



Objašnjenje i korisnički zahtev:

Kao što je pomenuto u okviru specifikacije aplikacije, aplikacija je bila prvenstveno namenjena za preduzeća koja u svom asortimanu imaju samo proizvode, pa su stoga klase dokumenata kao svoje linije čuvale listu proizvoda. Pri generisanju pdf-a dokumenata povlačile su se sledeće informacije o proizvodu: naziv(getName()), opis(getDescription()), cena(getPrize()), jedinica mere(getUnit()).

Nakon što je odlučeno da se u okviru aplikacije pored proizvoda omogući evidencija i usluga kreirane su klase usluga koje implementiraju interfejs Service sa sledećim metodama: getType(), getSpecification(), getPrize(), I getDurationUnit().

Zatim je stigao zahtev da se na dokumentima pored proizvoda kao linije dokumenta nadju I usluge, kao I da se pri generisanju pdf-a dokumenta ispisuju informacije I o uslugama.

Ovaj zahtev mogao se rešiti na više načina. Jedan od načina bio je menjanje klasa dokumenata tako da čuvaju I listu usluga, kao I metoda getLines() kako bi generisale pored linija koje sadrže informacije o proizvodima na dokumentu, I linije koje sadrže infromacije o uslugama na dokumentu.

Kako su za generisanje linija potrebne slične informacije I za proizvode I za usluge, samo se metode koje te informacije izvlače razlikuju mogao se izbeći problem menjanja svih klasa dokumenata upotrebom adapter paterna koji bi prilagodio interfejs usluga interfejsu product koji dokumenta već koriste.

Prilikom generisanja linija izvlačila se informacija o nazivu proivoda, a u slučaju usluga to će biti informacija o tipu usluge. Na mestu gde se za proizvod povlači opis, za uslugu će se povlačiti specifikacija. Metoda koja vraća cenu je ista I za proizvod I za uslugu, a metoda koja vraća jedinicu mere razlikuje se samo po nazivu.

Dakle potrebno je samo pri pozivu navedenih metoda proizvoda pozvati odgovarajuće metode usluge, tj. konvertovati interfejs Servcie u interfejs Product.

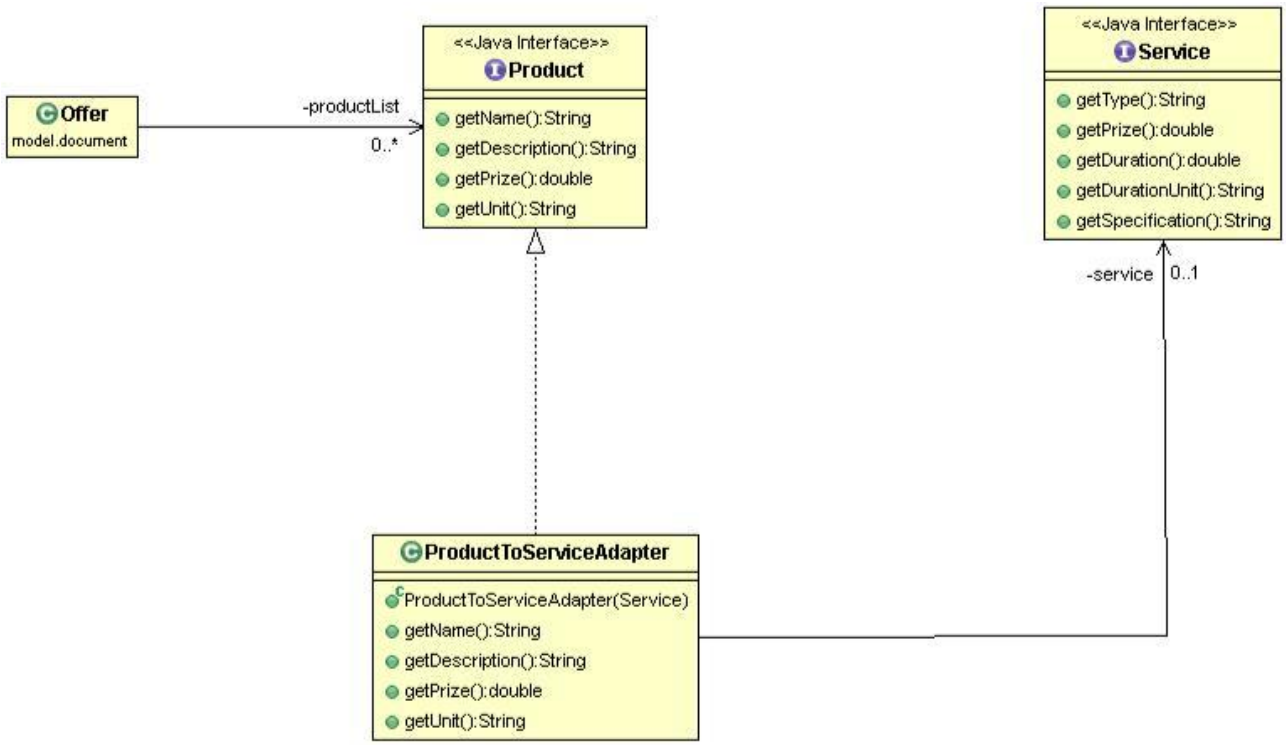
Upravo klasa ServiceToProductAdapter(adapter) konvertuje interfejs Service(adaptee) u interfejs Product(target). Klase dokumenata su u ovom slučaju klijenti.

Na ovaj način se klase dokumenata i dalje vezuju za interfejs proizvoda, ali se sada dokumentima može proslediti pored proizvoda i usluga "zapakovana" unutar klase ServiceToProduct adapter jer ta klasa implementira interfejs proizvoda.

Klasa ServiceToProductAdapter:

```
public class ServiceToProductAdapter implements Product {

    private Service service;

    public ServiceToProductAdapter(Service service) {
        this.service = service;
    }

    @Override
    public String getName() {
        return service.getType();
    }

    @Override
    public String getDescription() {
        return service.getSpecification();
    }

    @Override
    public double getPrize() {
        return service.getPrize();
    }

    @Override
    public String getUnit() {
        return service.getDurationUnit();
    }
```

Struktura Adapter paterna u okviru aplikacije:



3. State patern

Definicija state paterna: D o p u š t a o b j e k t u (C o n t e x t) d a p r o m e n i p o n a š a n j e (ConcreteStateA,ConcreteStateB,ConcreteStateC) kada se menja njegovo stanje (state).

Učesnici:

- Context(BusinessDocument)

- Definiše interfejs za klijenta. Sadrži pojavljivanje State podklase koja definiše tekuće stanje Context objekta.

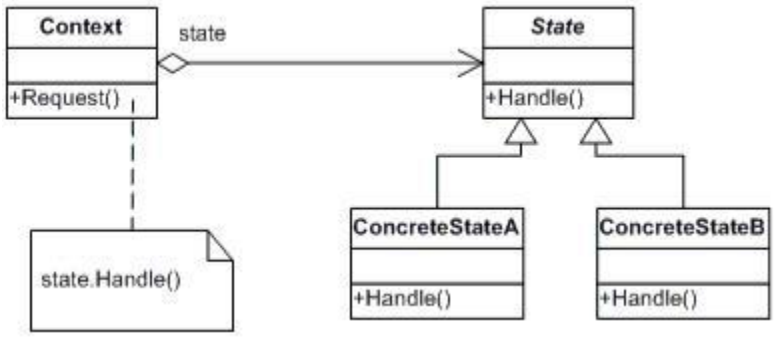
- State(DocumentStatus)

- Definiše interfejs za Context objekat, odnosno ponašanje koje se menja u zavisnosti od promene stanje Context objekta.

- ConcreteState(DocumentStatusNew, DocumentStatusDraft,DocumenStatusCompleted)

- Svaka ConcreteState podklasa implementira odredjeno ponašanje u zavisnosti od stanja Context objekta.

Struktura State paterna:



Objašnjenje i korisnički zahtev:

State patern je implementiran iz razloga što se iste operacije nad dokumentima izvode na različite načine u zavisnosti od toga u kom trenutku tj u kojem stanju(koje sve operacije su do tog momenta izvrsene nad dokumentom) se dokument nalazi.

U zavisnosti od stanja u kojem se dokument nalazi razlikuje se štampa, validacije koje će biti pokrenute pri čuvanju, kompletiranju,...

Dokumena(Context) su u aplikaciji implelentirana tako da kroz svoj životni vek(životni vek objekata tipa BusinessDocument) menjaju svoje stanje(Klasa DocumentStatus - state). Dokument kada se kreira inicjalno je u stanju "new"(Klasa DocumentStatusNew - ConcreteStateA). Nako što se sačuva u operativnoj memoriji prelazi u stanje "draft"(Klasa DocumentStatusDraft - ConcreteStateB) što znači da još nije izvršeno procesiranje njegovih linija i da nije spreman za slanje poslovnim partnerima. Nakon što se kompletira dokument prelazi u stanje "completed"(Klasa DocumentStatusCompleted - ConcreteStateC) što znači da je izvršeno procesiranje linija(sračunat pdv,...) i da je dokument spreman za slanje biznis partnerima.

Takodje sama stanja su zadužena za prebacivanje dokumenata iz tekućeg u naredno stanje.

Na ovaj način dokument je vezan za apstraktno stanje i za izvrsenje svojih operacija (Request()) uvek poziva operacije apstraktnog stanja. Koja operacija(operacija kog objekta) će biti izvrsena zavisi od toga u kom se stanju dokument nalazi. Tako dokument ostaje nezavisan od konkretnih stanja, koja se u budućnosti mogu dodavati i menjati.

Stanje new:

```
public class DocumentStatusNew extends DocumentStatus {

    public DocumentStatusNew(BusinessDocument businessDocument) {
        super(businessDocument);
    }

    @Override
    public String getStatus() {
        return "NEW";
    }

    @Override
    public void complete() throws Exception {
        throw new Exception("Document can't be completed because it is not yet saved!");
    }

    @Override
    public void save() throws Exception {
        businessDocument.saveValidation();
        businessDocument.setDocumentStatus(new DocumentStatusDraft(businessDocument));
        ControllerSingleton.getInstance().addDocument(businessDocument);
    }

}
```

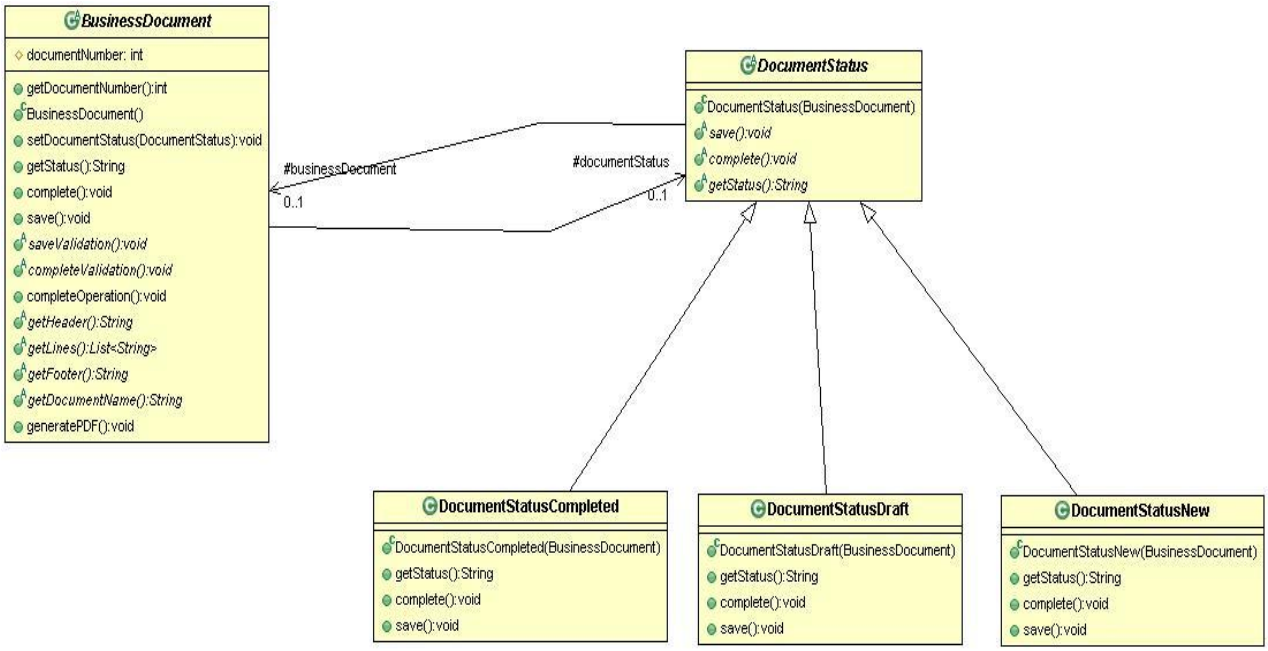
Stanje draft:

```
public class DocumentStatusDraft extends DocumentStatus{

    public DocumentStatusDraft(BusinessDocument businessDocument) {
        super(businessDocument);
    }

    @Override
    public String getStatus() {
        return "DRAFT";
    }

    @Override
    public void complete() throws Exception {
        businessDocument.completeValidation();
        businessDocument.completeOperation();
        this.businessDocument.setDocumentStatus(new DocumentStatusCompleted(businessDocument));
    }

    @Override
    public void save() throws Exception {
        throw new Exception("Document already saved!");
    }

}
```

Stanje completed:

```
 */
public class DocumentStatusCompleted extends DocumentStatus {

    public DocumentStatusCompleted(BusinessDocument businessDocument) {
        super(businessDocument);
    }

    @Override
    public String getStatus() {
        return "COMPLETED";
    }

    @Override
    public void complete() throws Exception {
        throw new Exception("Document already completed");
    }

    @Override
    public void save() throws Exception {
        throw new Exception("Completed document could not be edited");
    }

}
```

Struktura state paterna u okviru aplikacije:



Na dijagramu se vidi da i dokument(context) i status dokumenta(state) imaju metode complete(),save(),getStatus(). Pri pozivu ovih metoda objekta klase BusinessDocument, objekat klase BusinessDocument ce pozivati metodu nekog od objekata klasa DocumentStatusCompleted, DocumentStatusNew ili DocumentStatusDraft(ConcreteStates) u zavisnosti od toga u kom se stanju objekat nalazi.

U zavisnosti od operacije trenutni status može prevesti dokument u neki drugi status. Zbog toga, ali i zbog pozivanja odredjenih validacionih metoda dokumenta i statusi imaju referencu ka dokumentu kao što se i vidi na dijagramu.

4.Builder patern

Definicija Builder paterna: Deli konstrukciju složenog objekta(proizvoda) od njegove reprezentacije(builder), tako da isti konstrukcioni proces(direktor.construct()) može da kreira različite reporezentacije(složene proizvode).

Učesnici:

- Director(ClientApp)

- Kontroliše proces konstrukcije složnog proizvoda korišćenjem Builder interfejsa.

- Builder(SpecialOfferBuilder)

- Specifira interfejs za kreiranje složenog proizvoda.

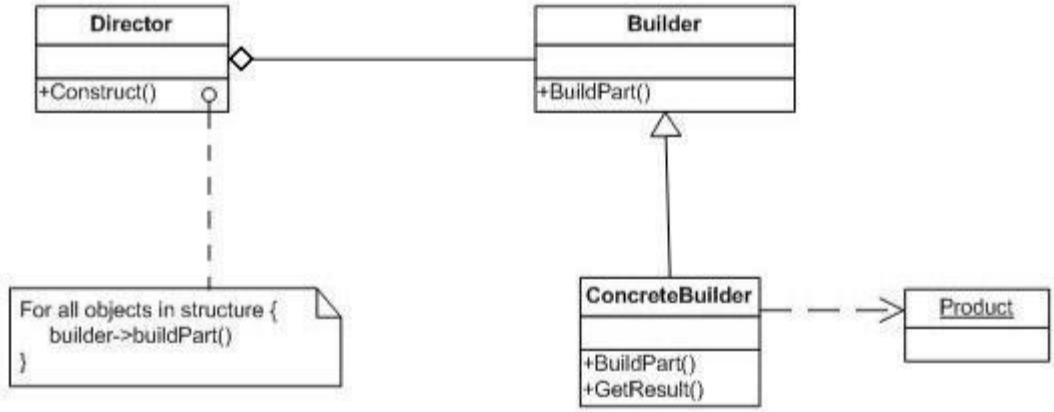
- ConcreteBuilder(NewYearOfferBuilder, ChristmasOfferBuilder)

- Konstruiše i grupiše proizvode u složeni proizvod implementirajući Builder interfejs

- Product(Offer)

- Reprezentuje složeni proizvod koji se konstruiše

Struktura builder paterna:



Objašnjenje i korisnički zahtev:

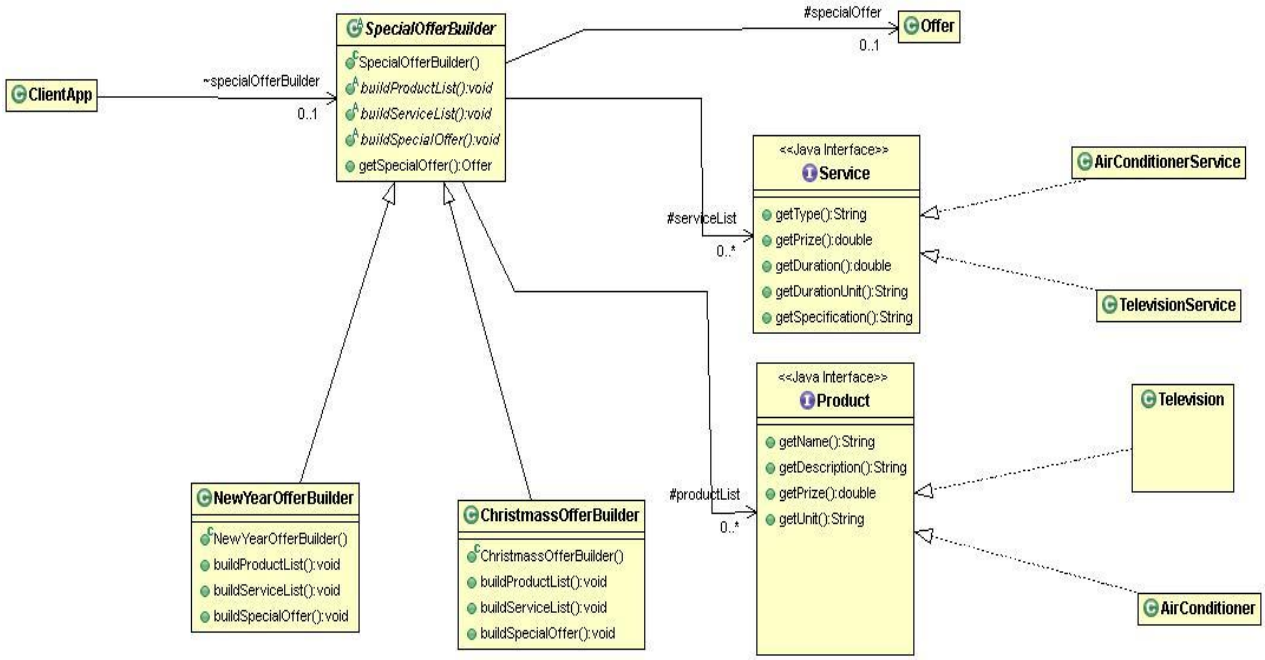
Kao što je bilo pomenuto u specifikaciji aplikacije, u okviru aplikacije postoje predefinsane specijalne ponude koje korisnik aplikacije može da kreira. Ova funkcionalnost implementirana je upravo pomocu builder paterna. Klijent(klasa ClientApp – Director u definiciji paterna) se vezuje za apstraktnog kreatora specijalnih ponuda(SpecialOfferBuilder – Builder u definiciji builder patern-a) koji se dalje specijalizuje na kreatora novogodišnje i kreatora božićne ponude(ConcreteBuilders u definiciji paterna), tako da njegov konstrukcioni proces(ClientApp.prepareOffer()) može kreirati bilo koju od dve postojeće, a takodje i bilo koju novu specijalnu ponudu(složen proizvod u definiciji paterna) koja će se dodati u budućnosti(nova klasa koja nasledjuje SpecialOfferBuilder klasu, odnosno novi concrete builder).

Na ovaj način ceo proces kreiranja specijalne ponude(delova, kao i ponude u celini) prenet je na SpecialOfferBuilder klasu. Klijent nadzire preko svoje konstrukcione metode koja poziva metode builder klase za kreiranje delova kao i sastavljanje kompletne ponude.

ClientApp(Director):

```
public class ClientApp {

    Company company;
    SpecialOfferBuilder specialOfferBuilder;

    public ClientApp(SpecialOfferBuilder sop){
        this.specialOfferBuilder = sop;
    }

    public void prepareOffer(){
        specialOfferBuilder.buildProductList();
        specialOfferBuilder.buildServiceList();
        specialOfferBuilder.buildSpecialOffer();
    }

    public void getSpecialOffer() {
        specialOfferBuilder.getSpecialOffer();
    }
```

Struktura Builder paterna u okviru aplikacije:



5. Decorator patern

Definicija dekorator paterna: Pridružuje dodatne odgovornosti(funkcionalnosti) do objketa(ConcreteComponent) dinamički. Dekorator obezbedjuje fleksibilnost u izboru podklasa(ConcreteDecoratorA, ConcreteDecoratorB) koje proširuku funkcionalnost ConcreteComponent objekta.

Učesnici:

- Component(HeaderDocument)

- Definiše interfejs za ConcreteComponent objekte kojima se odgovornost dodaje dinamički.

- ConcreteComponent(BusinessDocument)

- Definiše objekat kome će biti dodata odgovornost dinamički

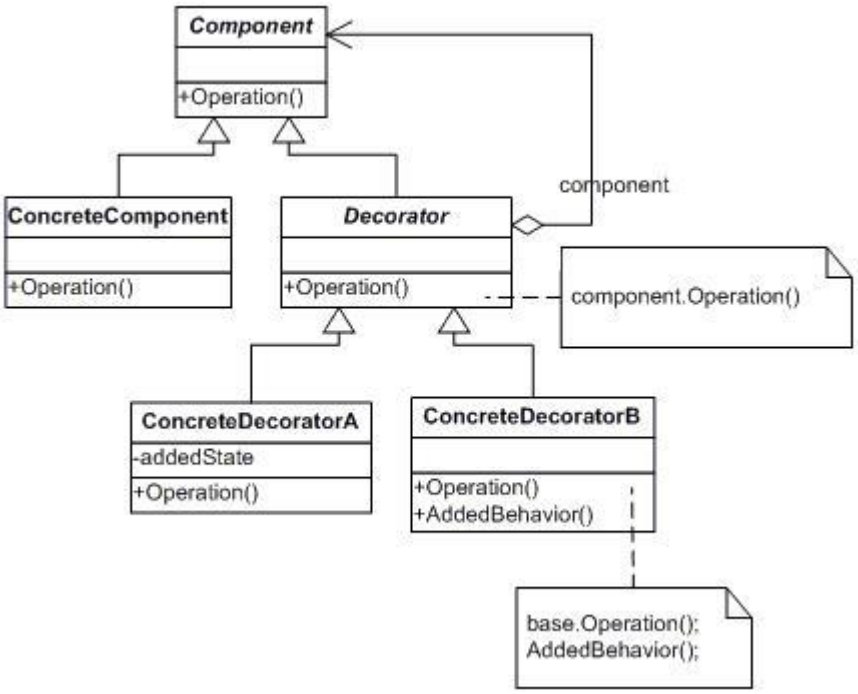
- Decorator(HeaderDocumentDecorator)

- Čuva referencu na Component objekat. Definiše interfeks koji je u skladu sa interfejsom Component.

- ConcreteDecorator(DocNumberHeaderDecorator i DateHeaderDecorator)

-Dodaje odgovornost do ConcreteComponent objekta

Struktura decorator paterna:



Objašnjenje i korisnički zahtev:

Štampa dokumenata je inicijalno bila bez datuma i broja dokumenta u zaglavlju. Nakon što je stigao zahtev za ubacivanje broja dokumenta i datuma u zaglavlje morao se izabrati način implementacije razmišljajući o tome da se sutra taj zahtev može opet promeniti. Kako decorator patern dodatne funkcionalnosti do objekta(ConcreteComponent – BusinessDocument u našem sluaju) pridružuje dinamički i omugućuje fleksibilnost u izboru podklasa (ConcreteDecoratorA – DocNumberHeaderDecorator i ConcreteDecoratorB – DateHeaderDecorator) koje proširuju funkcionalnosti ConcreteComponent objekta to je bio odličan izbor za ovu situaciju.

In t er fe j s Co m po n en t (H e ad e rD oc u me n t) ob e zb e dj u j e i nt e r fe j s z a d ob i j a nj e zaglavlja(getHeader():String). Decorator(HeaderDocumentDecorator) definiše interfejs u skladu sa Component interfejsom, čuva referencu na objekat ConcreteComponent(BusinessDocument) i implementira metodu getHeader() tako što u njoj poziva metodu konkretne komponente. Nikakvu dodatnu funckionalnost ne pridružuje već samo obezbedjuje interfejs za fleksibilno dodavanje funkcionalnosti. Konkretni dekoratori nasledjuju dekorator klasu i u implementaciji metode getHeader() pozivaju metodu super.getHeader() i dodaju novu funkcionalnost. I na taj način se može dodavati više novih funkcionalnosti i pri pozvivu poslednjeg objekta u nizu bice pozvane sve metode getHeader() svih dekorator klasa kao i klase konkretne komponente.

Component:

```
1  package model.document;
2
3  public interface HeaderDocument{
4      public String getHeader();
5  }
6
```

ConcreteComponent:
(getHeader() operacija je implementirana u podklasama konkrente komponente)

```
public abstract String getHeader();

public void generatePDF() throws Exception{
    if(getStatus().equalsIgnoreCase("NEW")){
        throw new Exception("Document must be saved before print!");
    }
    PDFCreator pdfCreator = new PDFCreator();
    HeaderDocumentDecorator headerDocumentDecorator = new HeaderDocumentDecorator(this);
    DocNumberHeaderDecorator docNumberHeaderDecorator = new DocNumberHeaderDecorator(headerDocumentDecorator);
    docNumberHeaderDecorator.setDocumentNumber(documentNumber);
    DateHeaderDecorator dateHeaderDecorator = new DateHeaderDecorator(docNumberHeaderDecorator);
    pdfCreator.buildHeader(dateHeaderDecorator.getHeader());
    pdfCreator.buildFooter(getFooter());
    pdfCreator.buildLines(getLines());
    pdfCreator.printDocument(getDocumentName()+""+getDocumentNumber()+""+getStatus());
}
```

Decorator:

```
1  package model.document;
2
3  public class HeaderDocumentDecorator implements HeaderDocument {
4
5      HeaderDocument headerDocument;
6
7      public HeaderDocumentDecorator(HeaderDocument headerDocument){
8          this.headerDocument = headerDocument;
9      }
10
11     @Override
12     public String getHeader() {
13         return headerDocument.getHeader();
14     }
15
16 }
17
```

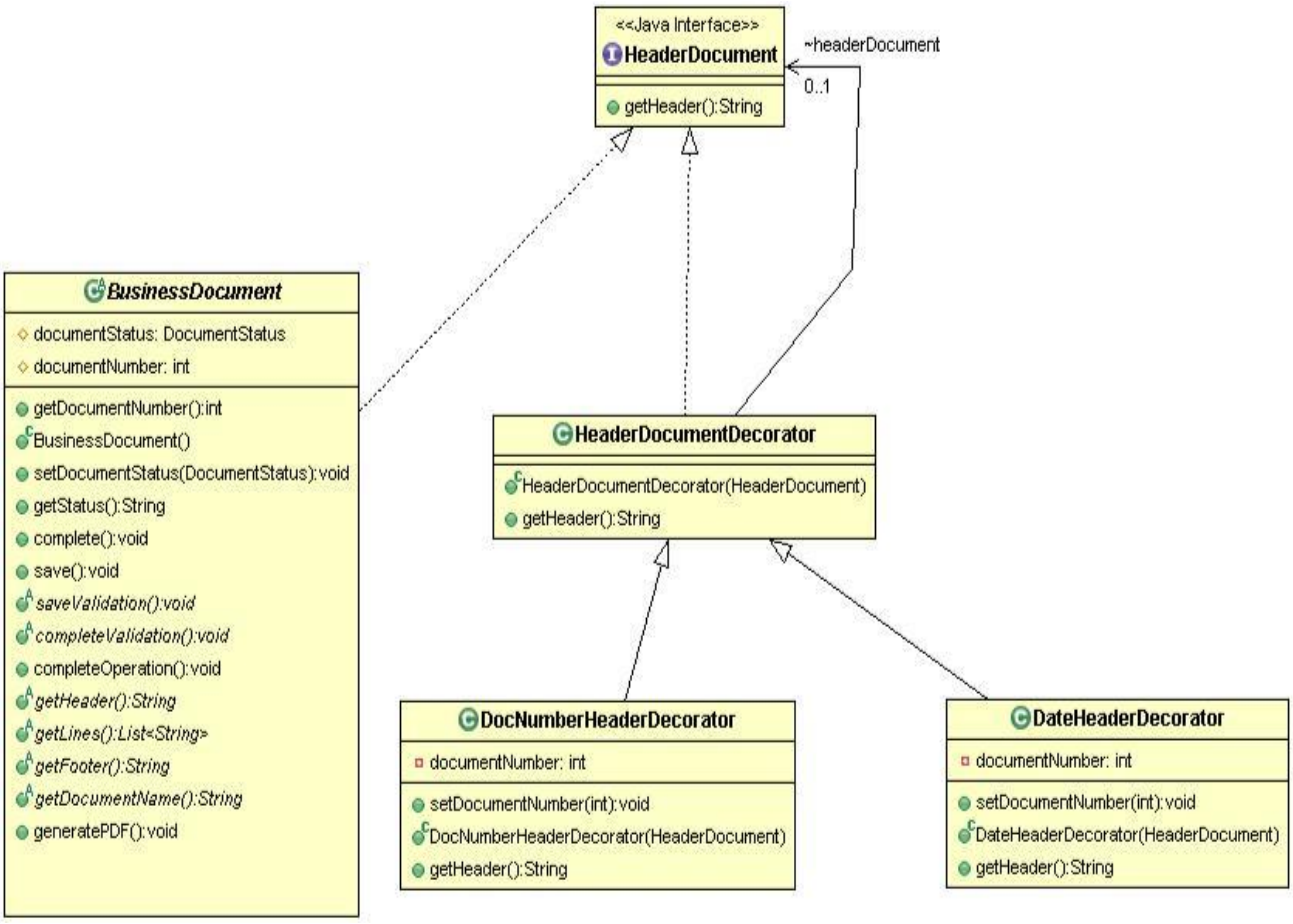
ConcreteDecoratorA:

```java
package model.document;

public class DocNumberHeaderDecorator extends HeaderDocumentDecorator {

    private int documentNumber;

    public void setDocumentNumber(int documentNumber){
        this.documentNumber = documentNumber;
    }

    public DocNumberHeaderDecorator(HeaderDocument headerDocument) {
        super(headerDocument);
    }
    @Override
    public String getHeader() {
        return super.getHeader()+"\nDocumentNumber: "+documentNumber;
    }


}
```

ConcreteDecoratorB:

```java
package model.document;

import java.text.SimpleDateFormat;

public class DateHeaderDecorator extends HeaderDocumentDecorator {

    private int documentNumber;

    public void setDocumentNumber(int documentNumber){
        this.documentNumber = documentNumber;
    }

    public DateHeaderDecorator(HeaderDocument headerDocument) {
        super(headerDocument);
    }
    @Override
    public String getHeader() {
        return super.getHeader()+"\nDate: "+ new SimpleDateFormat("dd.MM.YYYY").format(new Date());
    }


}
```

Struktura decorator paterna u okviru aplikacije:



6. Template method patern

Definicija: Definiše skelet algoritma u operaciji(TemplateMethod()), prepuštajući izvršenje nekih koraka operacije(PrimitiveOperation1(),PrimitiveOperation2()...) podklasama(ConcreteClass). Template method patern omogućava podklasama da redefinišu neke od koraka algoritma(PrimitiveOperation1(),PrimitiveOperation2()...) bez promene algoritamske strukture(TemplateMethod()).

Učesnici:

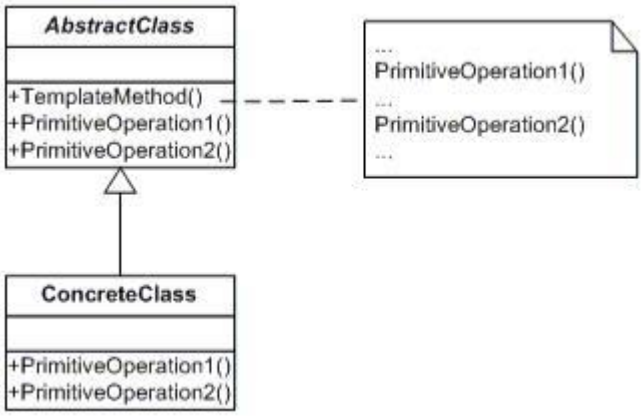
- AbstractClass(BusinessDocument)

- Definiše apstraktne primitivne operacije(PrimitiveOperation1(),PrimitiveOperation2()...) koje ConcreteClass podklasa implementira. Implementira TemplateMethod() operaciju definisanjem skeleta algoritma. TemplateMethod() operacija poziva primitivne operacije (PrimitiveOperation1(), PrimitiveOperation2()) koje su definisane u klasi AbstractClass.

- ConcreteClass(Invoice,Offer)

- Implementira primitivne operacije koje opisuju specifična ponašanja podklasa.

Struktura Template method paterna:



Objašnjenje i korisnički zahtev:

Štampa svih poslovnih dokumenata obavlja se po istom postupku. Potrebno je generisati zaglavlje dokumenta, linije dokumenta, futer dokumenta i naziv dokumenta, utvrditi status dokumenta i broj dokumenta a zatim tako prikupljene informacije prosledjivati PDFCreator utility klasi i na kraju pozvati metodu generatePDF klase PDFCreator. Sobzirom da je struktura algoritma za generisanje pdf-a svih dokumenata ista , a samo se operacije generisanja zaglavlja, linija, futera i naziva dokumenta razlikuju od dokumenta do dokumenta(konkrente klase koja nasledjuje BusinessDocument klasu) idelano rešenje za ovakav zahtev bio je template method patern. Klasa BusinessDocument predstavlja AbstractClass učesnika u Template method paternu , a klase konkretnih dokumenata Invoice i Offer predstavljaju ConcreteClass učesnike. U klasi BusinessDocument je definisana i implementirana operacija generatePDF() koja predstavlja TemplateMethod() operaciju. Takodje klasa BusinessDocument samo definise(bez implementacije) operacije getHeader(),getFooter(), getLines(), getDocumentName() koje predstavljaju primitivne operacije PrimitiveOperation1(),PrimitiveOperation2(),PrimitiveOperation3(), i PrimitiveOperation4() koje se pozivaju u template metodi getHeader() a bice implementirane tek u konkretnim klasama dokumenata(Invoice,Offer). Na taj način ustanovljena je i unificirana za sva dokumenta struktura algoritma generisanja pdf dokumenta a u isto vreme dopušteno je da pojedine primitivne operacije tog algoritma implementiraju konkretne klase dokumenata.

BusinessDocument(AbstractClass):

```
public abstract List<String> getLines();
public abstract String getFooter();
public abstract String getDocumentName();
public abstract String getHeader();

public void generatePDF() throws Exception{
    if(getStatus().equalsIgnoreCase("NEW")){
        throw new Exception("Document must be saved before print!");
    }
    PDFCreator pdfCreator = new PDFCreator();
    HeaderDocumentDecorator headerDocumentDecorator = new HeaderDocumentDecorator(this);
    DocNumberHeaderDecorator docNumberHeaderDecorator = new DocNumberHeaderDecorator(headerDocumentDecorator);
    docNumberHeaderDecorator.setDocumentNumber(documentNumber);
    DateHeaderDecorator dateHeaderDecorator = new DateHeaderDecorator(docNumberHeaderDecorator);
    pdfCreator.buildHeader(dateHeaderDecorator.getHeader());
    pdfCreator.buildFooter(getFooter());
    pdfCreator.buildLines(getLines());
    pdfCreator.generatePDF(getDocumentName()+""+getDocumentNumber()+""+getStatus());
}
```

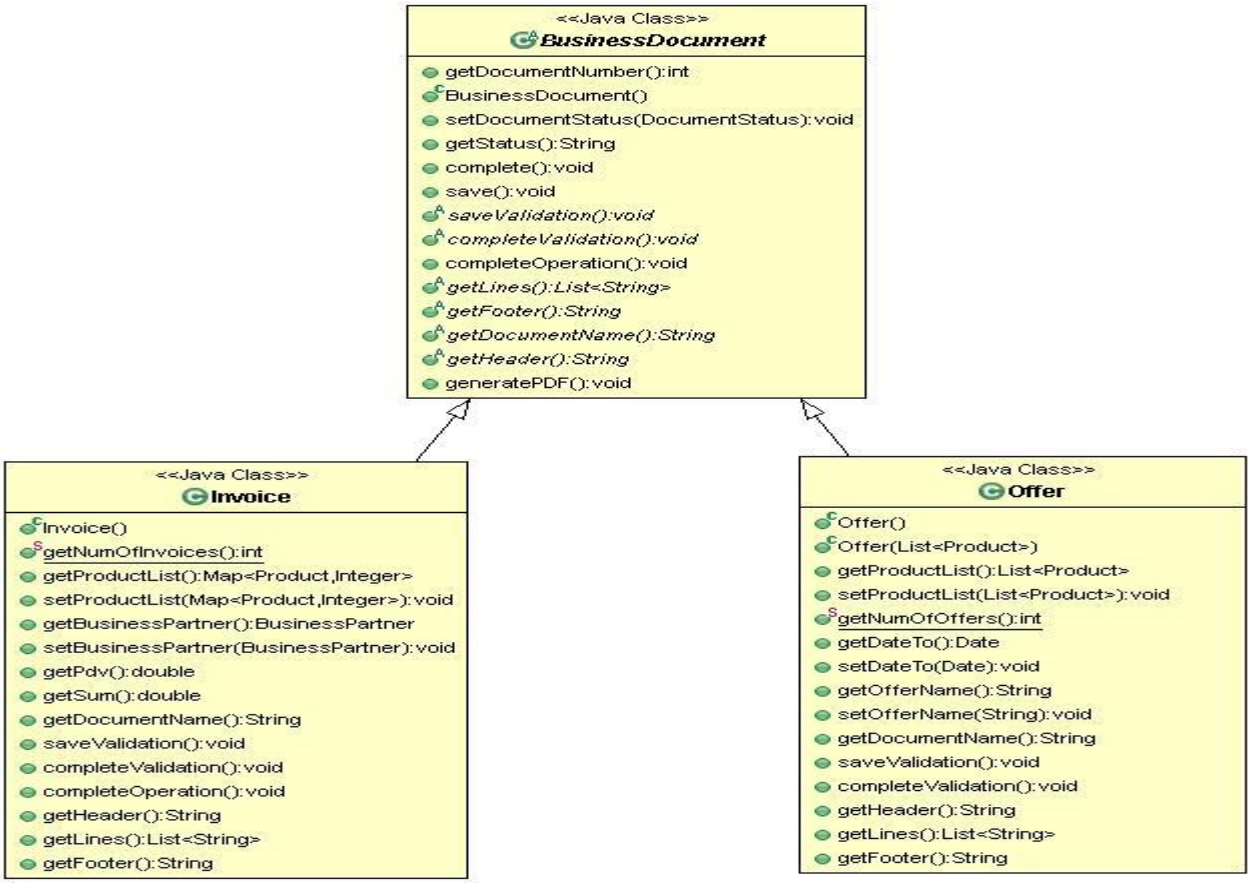
Invoice(ConcreteClass1):

```
93
94      @Override
95      public String getHeader() {
96          String header = "INVOICE(" + getStatus() + ")\n\n";
97          header += "Business Parnter\n";
98          header += businessPartner.getName() + "\n";
99          header += businessPartner.getAddress() + "\n";
100         return header;
101     }
102
103     @Override
104     public List<String> getLines() {
105         List<String> lines = new ArrayList<>();
106         for (Map.Entry<Product, Integer> entry : productList.entrySet()) {
107             String line = "";
108             Product product = entry.getKey();
109             line += product.getName() + ", ";
110             line += product.getPrize() + " per " + product.getUnit() + ", ";
111             line += entry.getValue() + " " + product.getUnit() + "s ";
112             if (sum > -1) {
113                 line += ", total: " + sum;
114                 line += ", pdv: " + pdv;
115             }
116             lines.add(line);
117         }
118         return lines;
119     }
120
121     @Override
122     public String getFooter() {
123         String footer = "Signature:";
124         if(getStatus().equalsIgnoreCase("COMPLETED"))
125             footer+="\nPDV Rate: " + currentPDVRate;
126         return footer;
127     }
128
129     @Override
130     public String getDocumentName() {
131         return "Invoice";
132     }
133
134 }
135
```

Offer(ConcreteClass2):

```
92
93      @Override
94      public String getHeader() {
95          String header = "OFFER(" + getStatus() + ")\n"+getOfferName();
96          return header;
97      }
98
99      @Override
100     public List<String> getLines() {
101         List<String> lines = new ArrayList<>();
102         for (Product product : productList) {
103             String line = "";
104             line += product.getName() + ", ";
105             line += product.getPrize() + " per " + product.getUnit();
106             if (getStatus().equalsIgnoreCase("COMPLETED")) {
107                 line += ", pdv: " + (product.getPrize() * currentPDVRate) / 100.00;
108             }
109             lines.add(line);
110         }
111         return lines;
112     }
113
114     @Override
115     public String getFooter() {
116         if (getStatus().equalsIgnoreCase("COMPLETED")) {
117             return "Valid to: " + dateTo + "\n"+"PDV: "+currentPDVRate;
118         }
119         return "";
120     }
121
122     @Override
123     public String getDocumentName() {
124         return "Offer";
125     }
126
127
```

Struktura paterna u okviru aplikacije:



7.Composite patern

Definicija: Objekti se sastavljaju(komponuju) u strukturu stabla kako bi predstavili hijerarhiju celine i delova. Composite patern omogućava da se jednostavni (Leaf) i složeni (Composite) objekti tretiraju jedinstveno. Jednostavni i složeni objekti su komponente(Component).

Učesnici:

- Client(ClientApp)

- Manipuliše objektima(komponentama) u strukturi pomoću Component interfejsa.

- Component(OrganisationalUnit)

- Deklariše interfejs za objekte koji će da obrazuju strukturu. Deklariše interfejs za pristupanje i upravljanje objektima strukture.

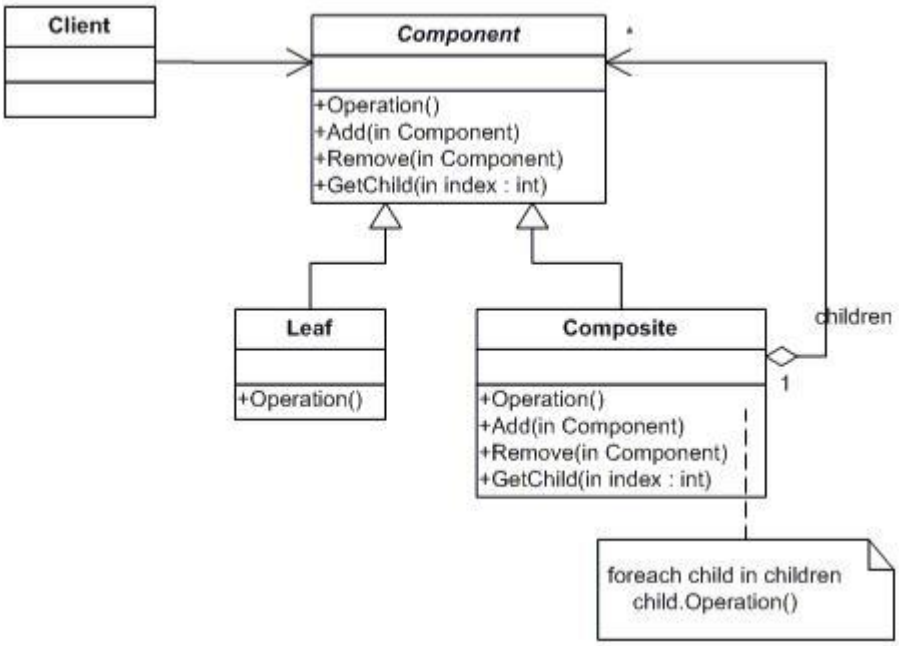
- Leaf(Employee)

- Predstavlja proste objekte (Leaf) u strukturi i definiše njihovo ponašanje. Prosti objekti nemaju decu-objekte.

- Composite(CompoundOrganisationalUnit,Company,Department)

Definiše ponašanje za složene objekte (Composition) koji imaju decu-objekte. Čuva decu-objekte. Implementira operacije interfejsa Component.

Struktura Composite paterna:

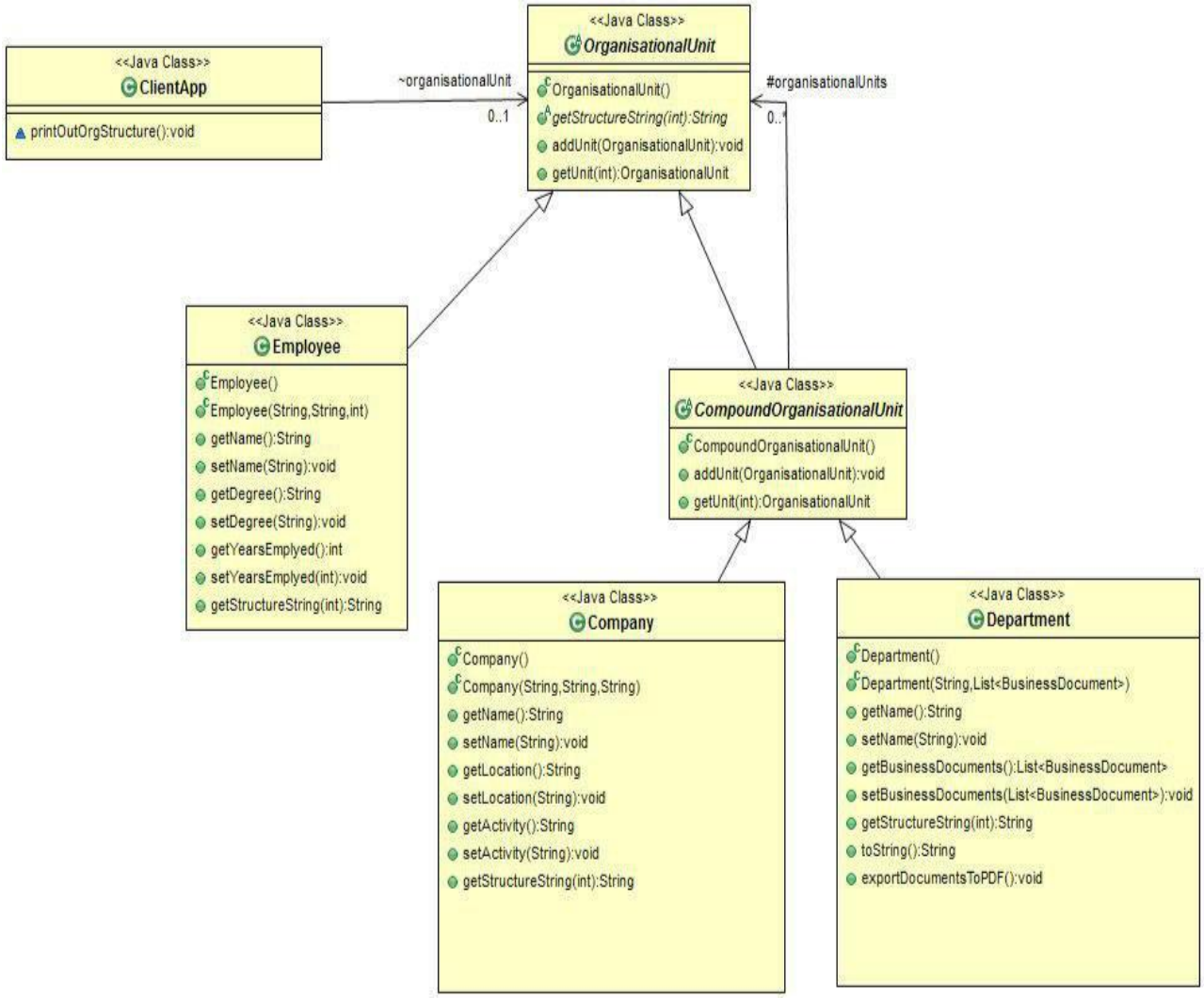


Objašnjenje i korisnički zahtev:

Jedna of funkcionalnosti aplikacije je i čuvanje kao i prikaz organizacione strukture kompanije. Pored prikaza cele organizacione strukture bilo je potrebno omogućiti i prikaz organizacione strukture pojedinih delova kompanije. Na vrhu organizacione strukure se nalazi naravno kompanija(Company klasa). Kompanija se sastoji iz sektora(Department klasa), a u sektorima rade radnici(Employee klasa). Takodje radnici mogu raditi u preduzecu i van sektora. Optimalno rešenje za implementiranje ove funkcionalnosti je Composit patern, jer je omogućava da radnike, odeljenja i kompaniju tretiramo jednistveno – kao jednu organizacionu jedinicu, koja je zadužena za čuvanje informacije o sebi uključujući i druge organizacione jedinice iz kojih se sastoji. Upravo klasa

OrganisationalUnit(Component) definiše iterfejs za ove organizacione jedinice(operacije dodavanja i vraćanja poedinih delova , kao i ispis strukure te organizacione jedinice). U klasi CompoundOrganisationalUnit su implementirane operacije dodavanja i vraćanja organizacionih jedinica iz kojih se data organizaciona jedinica sastoji(čuvanje i vraćanje dece-objekata) koje su zajedničke za sve složene organizacione jedinice(Company,Department). Company i Department klase(Composition) će naslediti CompoundOrganisationalUnit klasu i dodatno implementirati operaciju ispisa svoje strukture(ispis infomracija o sebi i svoj deci preko poziva njihovih metoda za ispis organizacione strukture). Klasa Employee(leaf) nasledjuje OrganisationalUnit klasu(Component) i implementira samo operaciju za ispis svoje organizacione strukture, što je u slučaju leaf klasa samo informacija o tom objektu.
Na ovaj način omogućeno je da se bilo kojoj složenoj organizacionoj jedinici(klase okje nasledjuju CompoundOrganisationalUnit) dodaju druge proste i složene organizacione jedinice a zatim pozivom metode za ispis organizacione strukture ispišu sve informacije o toj organizacionoj jedinici kao i svim organizacionim jedinicama iz kojih se sastoji(preko poziva njihovih metoda za ispis)

Struktura paterna u okviru aplikacije:



OrganisationalUnit(Component):

```java
package model.organisationalstructure.composition;

import java.util.ArrayList;

//composite
public abstract class CompoundOrganisationalUnit extends OrganisationalUnit  {
    protected List<OrganisationalUnit> organisationalUnits = new ArrayList<OrganisationalUnit>();


    public void addUnit(OrganisationalUnit organisationalUnit) {
        this.organisationalUnits.add(organisationalUnit);
    }

    public OrganisationalUnit getUnit(int i) {
        return this.organisationalUnits.get(i);
    }
}
```

Employe(Leaf):

```java
    }
    @Override
    public String getStructureString(int level) {
        return "Employee (name= "+name+")";
    }
```

CompoundOrganisationalUnit(Composite):

```java
package model.organisationalstructure.composition;

import java.util.ArrayList;

//composite
public abstract class CompoundOrganisationalUnit extends OrganisationalUnit  {
    protected List<OrganisationalUnit> organisationalUnits = new ArrayList<OrganisationalUnit>();


    public void addUnit(OrganisationalUnit organisationalUnit) {
        this.organisationalUnits.add(organisationalUnit);
    }

    public OrganisationalUnit getUnit(int i) {
        return this.organisationalUnits.get(i);
    }
}
```

Department(Composite):

```java
    }
    @Override
    public String getStructureString(int level) {
        String children = "";
        String prefiks = "";
        for (int i = 0; i <= level; i++) {
            prefiks+="\t";
        }
        for (int i = 0; i < organisationalUnits.size(); i++) {
            children+=prefiks+organisationalUnits.get(i).getStructureString(level+1);
            if(i!=organisationalUnits.size()-1){
                children+=",\n ";
            }else{
                children+="]";
            }
        }
        return "Department (name=" + name +"[\n"+children+")";
    }
```

Company(Composite):

```
    @Override
    public String getStructureString(int level) {
        String children = "";
        String prefiks = "";
        for (int i = 0; i <= level; i++) {
            prefiks+="\t";
        }
        for (int i = 0; i < organisationalUnits.size(); i++) {
            children+=prefiks+organisationalUnits.get(i).getStructureString(level+1);
            if(i!=organisationalUnits.size()-1){
                children+=",\n ";
            }else{
                children+="]";
            }
        }
        return "Company (name=" + name +"[\n"+children+")";
    }
}
```

Client(Client):

```
public ClientApp(OrganisationalUnit organisationalUnit){
    this.organisationalUnit = organisationalUnit;
}

void printOutOrgStructure(){
    System.out.println("Organizaciona struktura preduzeća: \n"+organisationalUnit.getStructureString(0));
}
```

8. Observer Patern

Definicija: Definiše jedan-veiše zavisnot izmedju objekata (Subject->Observer) tako da promena stanja nekog objekata(ConcreteSubject) utiče automatski na promenu stanja svih drugih objekata koji su povezani sa njim(ConcreteObserver)

Učesnici:

- Subject(TaxAdministration)

- Zna ko su njegovi Observer objekti. Obezbedjuje interfejs za povezivanje i razvezivanje Observer objekata.

- ConcreteSubject(PdvAdministration)

- Čuva stanje na koje se postavljaju ConcreteObserver objekti. Šalje obaveštenja do njegovih Observer objekata kada se promeni njegovo stanje(subjectState)

- Observer(TaxListener)

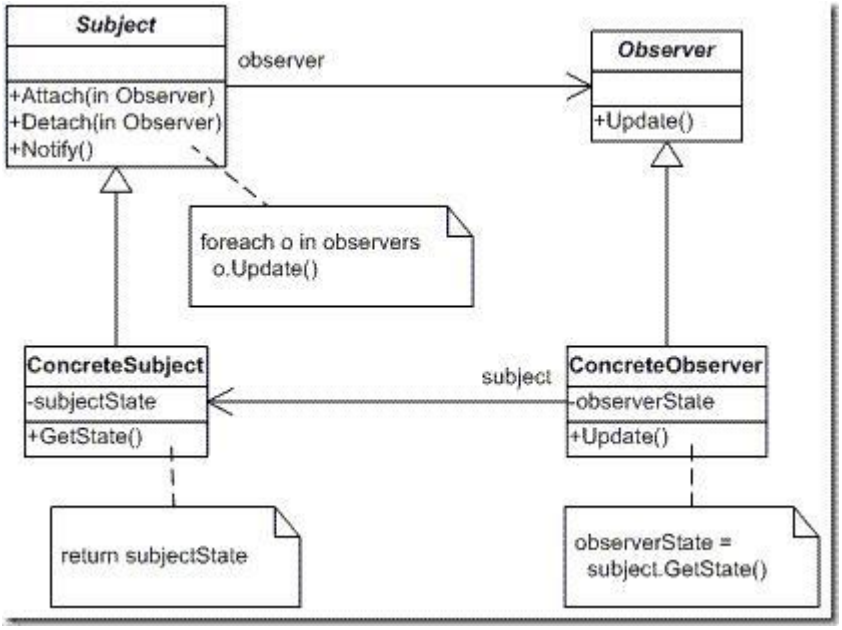
- Definiše interfejs za promenu objekata (Update()) koj treba da budu obevešteni kada se promeni Subject objekat.

- ConcreteObserver(BusinessDocument)

- Čuva referencu na ConcreteSubject objekat. Čuva stanje koje treba da ostane konzistetno sa stanjem ConcreteSubject objekta. Implementira Observer interfejs kako bi sačuvao njegovo stanje

konzistentno sa stanjem ConcreteSubject objekta.

Struktura Observer paterna:



Objašnjenje i korisnički zahtev:

Kako bi bilo omogućeno da preduzeća u sistemu pdv-a koriste aplikaciju bilo je neophodno da se za sva dokumenta sračunava pdv po tekućoj stopi pdv-a. Veliki problem bi bio ukoliko bi pri svakoj promeni stope pdv-a bilo neophodno prolaziti kroz sva dokumenta za koja se želi promeniti stopa i ručno menjati. Kako bi se izbegao ovaj problem implementiran je obzerver patern, tako da je samo jedna klasa zaduzena za evidenciju i promenu trenutne stope pdv-a(PdvAdministration-ConcreteSubect) a sve klase(ConcreteObserver) koje žele da budu obaveštene o eventualnim promenama mogu osluškivati ovu klasu. U aplikaciji BusinessDocument(ConcreteObserver) klasa implementira interfejs TaxListener(Observer) i operaciju pdvChanged() sto omogućava da njeni objekti budu dodati u listu objekata klase TaxAdministration(Subject) koji će biti obavešteni kada klasa PdvAdministration(ConcreteSubject) promeni stopu pdv-a(klasa PdvAdministration poziva metodu definisanu u klasi TaxAdministration koja poziva metodu pdvChanged() svih objekata u listi objekata koji osluškuju promene).

TaxAdministration(Subject):

```
//#####################OBSERVER PATTERN#####################################
public class TaxAdministration {

    protected List<TaxListener> taxListeners = new ArrayList<TaxListener>();

    public void notifyTaxChanged(){
        for(TaxListener taxListener : taxListeners){
            taxListener.pdvChanged();
        }
    }

    public void addTaxListener(TaxListener taxListener){
        this.taxListeners.add(taxListener);
    }
}
```

PdvAdministration(ConcreteSubject):

```
1  package util.tax;
2
3  import java.util.ArrayList;
4  import java.util.List;
5
6  public class PdvAdministration extends TaxAdministration {
7      private double currentPDV = 0.00;
8
9
10     public void setCurrentPDV(double PDVRate){
11         this.currentPDV = PDVRate;
12         notifyTaxChanged();
13     }
14
15     public double getCurrentPDV(){
16         return currentPDV;
17     }
18
19
20 }
21
```

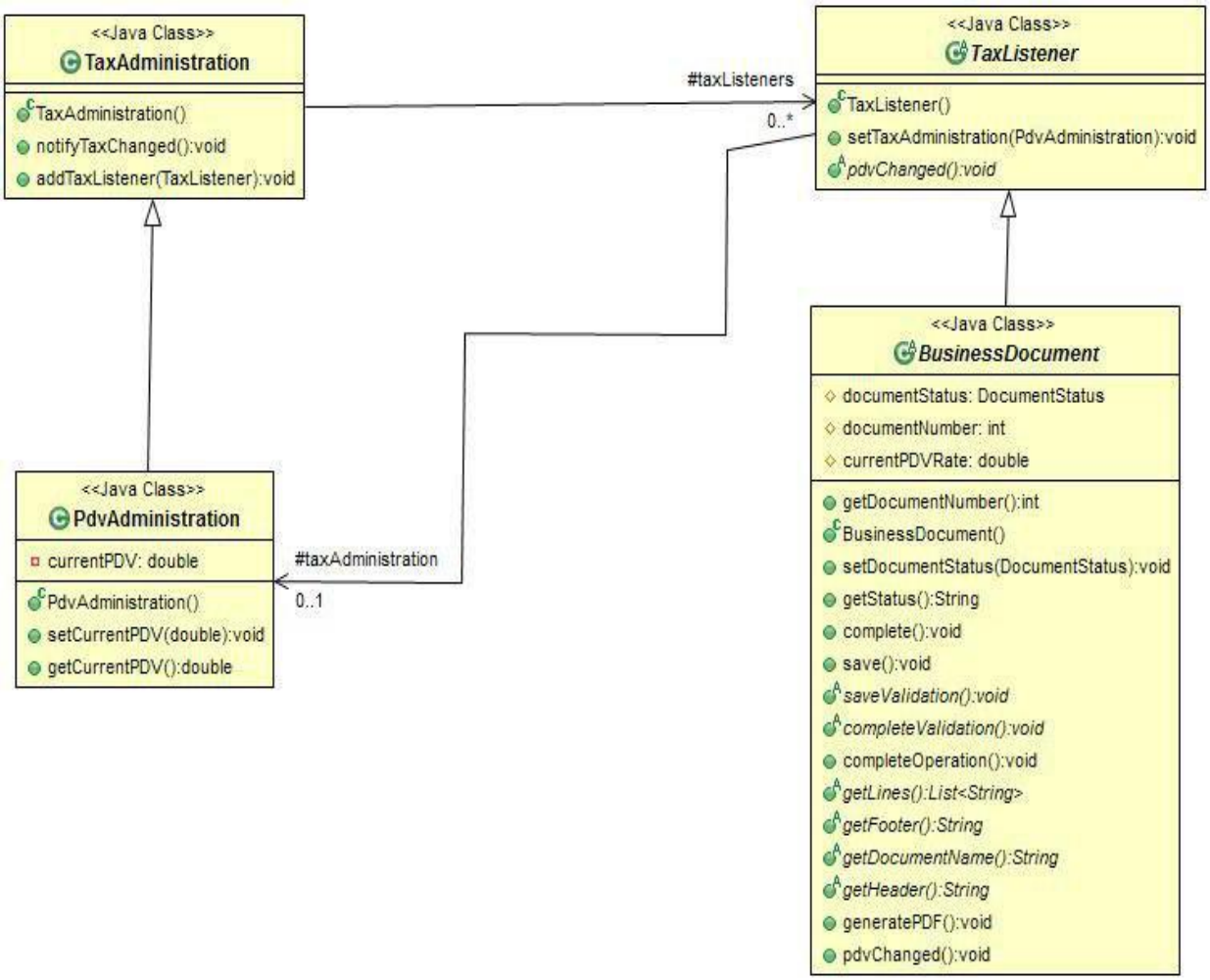
TaxListener(Observer):

```
package util.tax;

public abstract class TaxListener {

    protected PdvAdministration taxAdministration;

    public TaxListener(){

    }

    public void setTaxAdministration(PdvAdministration taxAdministration){
        this.taxAdministration = taxAdministration;
        taxAdministration.addTaxListener(this);
        pdvChanged();
    }

    public abstract void pdvChanged();

}
```

BusinessDocument(ConcreteObserver):

```
    }

    @Override
    public void pdvChanged() {
        this.currentPDVRate = taxAdministration.getCurrentPDV();
    }
```

Struktura paterna u aplikaciji:



ZAKLUČAK

Poznavanje opšte strukture paterna projektovanja je samo prvi korak u shvatanju suštine softverskih paterna. Sledeći korak u razumevanju paterna je studiranje konkretnih aplikacija I pronalaženje mesta na kojima bi odgovarajući patern unapredio strukturu aplikacije I omogućio njeno lakše održavanje I unapređivanje.
Efikasna upotreba paterna zahteva razmišljanje o problemima koji se mogu pojaviti tek kasnije u razvoju softvera, za šta je neophodno iskustvo.
Smatram da su paterni nezaobilazni pri razvoju velikih I skalabilnih aplikacija, jer primena pravih paterna na pravom mestu ubrzava razvoj, omogućava da buduće izmene budu jednostavnije, kao I nezavisno dodavanje novih funkcionalnosti koje ne zahtevaju ponovno testiranje postojećih.

Literatura

- “Softverski paterni“, Siniša Vlajić, Beograd – 2014