Рефакторисање кода према обрасцима дизајна

Бранислава Шандрих

Београд 2015.

Апстракт

За успешан развој софтверских пројеката од великог је значаја да програмски код буде добро организован, како би се омогућило увођење нових функционалности у будућности и обезбедило лако разумевање кода, одржавање и откривање грешака, те обезбедила скалабилност и флексибилност софтвера. Рефакторисање кода према обрасцима дизајна нуди конкретне одговоре на питање како у случају коришћења различитих образаца дизајна организовати код на оптимални начин. Оно не зависи од конкретне технологије, већ садржи одређен ниво универзалности.
Рад проучава рефакторисање према обрасцима дизајна и утицај таквог приступа на конструкцију кода. Резултати рада могу бити примењени у организацији кода, а у оквиру активности на раду развијани су и програми - примери (написани у програмском језику Јава) који илуструју основне тезе рада и демонстрирају добијене закључке.

„Постоји природна веза између образаца дизајна и рефакторисања. Образац је дестинација; рефакторисање је превозно средство.”

- Design patterns: elements of reusable object-oriented software

„Наши обрасци дизајна бележе многе структуре које су управо резултат рефакторисања…Обрасци дизајна, дакле, представљају циљ рефакторисања.”

- Refactoring: Improving the Design of Existing Code

Садржај

1 Увод **5**

2 Основни обрасци дизајна **9**

3 Основи рефакторисања **11**

4 Рефакторизација код конструкционих образаца дизајна **13**

4.1 Уланчани конструктори 13
4.1.1 Објашњење 13
4.1.2 Потенцијални проблеми 13
4.1.3 Практична реализација рефакторизације 14
4.1.4 Пример 14
4.2 Метод замене вишеструких конструктора
методама креирања 17
4.2.1 Објашњење 17
4.2.2 Потенцијални проблеми 17
4.2.3 Практична реализација рефакторизације 17
4.2.4 Пример 18
4.3 Енкапсулација класа методама креирања 21
4.3.1 Објашњење 21
4.3.2 Потенцијални проблеми 21
4.3.3 Практична реализација рефакторизације 22
4.3.4 Пример 22
4.4 Екстракција класа креирања 26
4.4.1 Објашњење 26
4.4.2 Потенцијални проблеми 26
4.4.3 Практична реализација рефакторизације 26
4.4.4 Пример 27
4.5 Полиморфно креирање уместо Производне методе 31
4.5.1 Објашњење 31
4.5.2 Потенцијални проблеми 31
4.5.3 Практична реализација рефакторизације 32
4.5.4 Примери 32

5 Рефакторизација код структурних образаца дизајна 37
5.1 Замена условних рачуница обрасцем Стратегије 37
5.1.1 Објашњење 37
5.1.2 Потенцијални проблеми 37
5.1.3 Практична реализација рефакторизације 38
5.1.4 Пример 38
5.2 Замена имплицитног формирања дрвета Саставом 46
5.2.1 Објашњење 46
5.2.2 Потенцијални проблеми 46
5.2.3 Практична реализација рефакторизације 47
5.2.4 Пример 47
5.3 Енкапсулација сложеног кода помоћу Градитељ обрасца . . 52
5.3.1 Објашњење 52
5.3.2 Потенцијални проблеми 52
5.3.3 Практична реализација рефакторизације 53
5.3.4 Пример 53
5.3.5 Оптимизација 57
5.4 Формирање над-интерфејса 61
5.4.1 Објашњење 61
5.4.2 Практична реализација рефакторизације 61
5.4.3 Пример 62

6 Рефакторизација код бихејворалних образаца дизајна 64
6.1 Пребацивања украшавања у Декоратер 64
6.1.1 Објашњење 64
6.1.2 Потенцијални проблеми 65
6.1.3 Практична реализација рефакторизације 66
6.1.4 Пример 67
6.2 Замена чврсто-кодираног слања обавештења Посматрач обрасцем 77
6.2.1 Објашњење 77
6.2.2 Потенцијални проблеми 78
6.2.3 Практична реализација рефакторизације 78
6.2.4 Пример 79
6.3 Замена условних претрага Спецификацијом 84

6.3.1 Објашњење 84
6.3.2 Потенцијални проблеми 85
6.3.3 Практична реализација рефакторизације 85
6.3.4 Пример 86
6.4 Сједињавање кода за случај једног и кода за случај обраде више премерака применом Састав обрасца 99
6.4.1 Објашњење 99
6.4.2 Потенцијални проблеми 99
6.4.3 Практична реализација рефакторизације 100
6.4.4 Пример 100
6.5 Замена акумулирајућег кода Сакупљајућим параметром . . 108
6.5.1 Објашњење 108
6.5.2 Потенцијални проблеми 108
6.5.3 Практична реализација рефакторизације 108
6.5.4 Пример 109
6.6 Декомпоновање методе 114
6.6.1 Објашњење 114
6.6.2 Потенцијални проблеми 114
6.6.3 Практична реализација рефакторизације 114
6.6.4 Пример 115

7 Закључак 120

1 Увод

Задатак ефикасне организације објектно оријентисаног софтвера није нимало једноставан. Уз данас неизбежан захтев да такав софтвер буде и више пута употребљив, задатак постаје прави изазов за пројектанта. Пројекат би требало да буде довољно специфичан за конкретан проблем, али и да предвиди и подржи евентуалне будуће проблеме и захтеве. Искусни објектно оријентисани пројектанти знају да за добре пројекте не треба решавати сваки проблем од самог почетка, већ се неретко ослонити на решење које је проверено и већ употребљавано. Због тога се у многим објектно оријентисаним системима наилази на исте обрасце класа и објеката који комуницирају. Ови обрасци решавају специфичне проблеме пројектовања и омогућавају флексибилност, елеганцију и поновну употребљивост објектно оријентисаног дизајна. Они пројектантима помажу да поново употребе успешне пројекте, тако што ће нове пројекте засновати на претходном искуству.
Обрасци дизајна помажу да се изаберу алтернативе које омогућавају да систем буде више пута употребљив и да се избегну алтернативе које није могуће више пута употребити. Они идентификују класе и примерке, њихове улоге и међудејство, као и дистрибуцију њихових одговорности.
С друге стране, пројектантима који се превише ослањају на обрасце дизајна и такву организацију кода, често се дешава да развијају непотребно компликоване и гломазне системе. Пример који то демонстрира је фамозна „ Hello World" анегдота: програм написан од стране искусног програмера који познаје и радо користи разне обрасце дизајна, а исписује само уобичајни поздрав:

```
interface MessageStrategy {
  public void sendMessage();
}
abstract class AbstractStrategyFactory {
  public abstract MessageStrategy createStrategy(MessageBody mb);
}
class MessageBody {
  Object payload;
  public Object getPayload() {
    return payload;
  }
  public void configure(Object obj) {
```

```
      payload = obj;
   }
   public void send(MessageStrategy ms) {
      ms.sendMessage();
   }
}
class DefaultFactory extends AbstractStrategyFactory {
   private DefaultFactory() {;
   }
   static DefaultFactory instance;
   public static AbstractStrategyFactory getInstance() {
      if (instance == null) instance = new DefaultFactory();
      return instance;
   }
   public MessageStrategy createStrategy(final MessageBody mb) {
      return new MessageStrategy() {
         MessageBody body = mb;
         public void sendMessage() {
            Object obj = body.getPayload();
            System.out.println((String) obj);
         }
      };
   }
}
public class HelloWorld {
   public static void main(String[] args) {
      MessageBody mb = new MessageBody();
      mb.configure("Hello World!");
      AbstractStrategyFactory asf = DefaultFactory.getInstance();
      MessageStrategy strategy = asf.createStrategy(mb);
      mb.send(strategy);
   }
}
```

Појава да је код до те мере усложњен, у смислу да је флексибилнији или софистициранији него што је потребно, назива се пре-рефакторисањем. Свакако, елегантно је предвидети будуће захтеве, па и уложити нешто више (или много више) времена на побољшање и организацију кода. Међутим, уколико до тих захтева ипак не дође, протраћено је време које је неретко драгоцено.

Још је чешћа појава под-рефакторисања. Углавном се под-рефакторише када је потребно за кратко време имплементирати што више нових функционалности, без могућности да се неко време посвети и дизајну.
Рефакторисање представља споља невидиљиву трансформацију система која чува његово понашање, али пружа бољи дизајн, вишекратну употребљивост и флексибилност. Аутор књиге ,, Refactoring: Improving the Design of Existing Code" [2], рефакторисање дефинише као: ,,промену над интерном структуром софтвера, у циљу омогућавања лакшег разумевања и једноставније измене кода, без измене видљивог понашања".

Мотивација за рефакторисањем према обрасцима дизајна разликује се од мотивације за употребом образаца дизајна наведене у литератури која их изучава. Примера ради, према [2], намена Декоратер обрасца је да ,,динамички прикључује додатне одговорности објекту, то јест да пружи флексибилну алтернативу за пружање проширених функционалности употребом поткласа".
Аутори ,, Design patterns" [3], наводе да се Декоратер образац употребљава:

- за додавање одговорности појединачним објектима на динамички и транспарентан начин, тј. без утицаја на друге објекте
- за одговорности које могу да се повуку
- када проширење помоћу прављења поткласа није практично. Понекад је могућ велики број независних проширења, па би подршка за сваку комбинацију произвела превише поткласа. Постоји и случај када је дефиниција класа сакривена или је на други начин онемогућено прављење поткласа.

С друге стране, аутори JUnit алата за тестирање [5], присећају се због чега су прибегли рефакторисању према Декоратеру:
,,Неко је додао TestSetup подршку, поткласу класе TestSuite. Када смо додали RepeatedTestCase и ActiveTestCase, увидели смо да можемо да елимишемо понављање кода уколико уведемо TestSetup као декоратер класу."
Закључује се да је између мотивације за рефакторисањем према Декоратеру (у овом случају, елиминација поновљеног кода) и документованом наменом Декоратера (динамичка алтернатива проширењу помоћу поткласа), веза прилично танка.

Наредни примеру демонстрирају још оваквих разилажења:
Градитељ
Намена према [3]: Раздваја изградњу сложеног објекта од његове репрезентације да би исти процес прављења могао да произведе различите репрезентације.
Циљеви рефакторисања: Поједностављивање кода, елиминација понављања и смањивање грешака при креирању објеката.
Производни метод
Намена према [3]: Дефинише интерфејс за прављење објекта, али препушта поткласи одлуку о класи примерка. Производни метод дозвољава класи да препусти прављење примерка поткласи.
Циљеви рефакторисања: Елиминација понављања и разјашњвање намене.
Шаблонски метод
Намена према [3]: Дефинише скелет алгоритма неке операције препуштајући имплементацију неких корака поткласи. Шаблонски метод дозвољава да поткласе редефинишу неке кораке алгоритма не мењајући структуру алгоритма.
Циљеви рефакторисања: Елиминација понављања.

Вредније је познавати добре мотиве за рефакторисање према неком обрасцу дизајна, него познавати крајњи резултат који образац пружа. Према [3], обрасци дизајна су ,,структуре које настају управо као последица рефакторисања". Дакле, проучавање еволуције дизајна квалитетног софтвера је корисније него проучавање самог крајњег дизајна. Те структуре које настају еволуцијом могу бити од помоћи у другим системима, али уколико није до краја јасно и познато како и због чега су те структуре настале, већа је вероватноћа да ново искуство неће бити добро примењено, или да ће доћи до пре-рефакторисања.

Рад проучава и наводи проверена рефакторисања која користе обрасце дизајна као смерницу. Свако рефакторисање има следећу структуру: прво је укратко описан проблем, а затим и како се решава. Потом је дато нешто детаљније објашењење о самим околностима у којима је згодно применити рефакторисање. Следи тачка ,,Практича реализација рефакторизације", у којој се наводе кораци које треба применити како би се адекватно рефакторисало, а за њом и примери који демонстрирају рефакторисање.

2 Основни обрасци дизајна

Обрасци дизајна разликују се по грануларности и степену апстракције. Могу се организовати на више начина: према намени, домену, међусобној повезаности. Критеријум на основу којих су класификовани обрасци у оквиру овог рада је према намени, а она може бити: прављење, структура или понашање.
Обрасци за прављење (*конструкциони*) баве се поступком израде објеката. Структурни обрасци баве се састављањем класа и објеката. Обрасци понашања (*бихејворални)* описују како класе или објекти међусобно утичу једни на друге и како деле одговорности.

У даљем тексту наведен је само део основних образаца дизајна и њихових описа наведених у [3]. То су обрасци на која се обрађена рефакторисања углавном надовезују:

1. Конструкциони обрасци

    (а) **Градитељ** (Builder): Раздваја изградњу сложеног објекта од његове репрезентације да би исти процес прављења могао да произведе различите репрезентације.

    (б) **Производни метод** (Factory Method): Дефинише интерфејс за прављење објекта, али препушта поткласи одлуку о класи примерка. Производни метод дозвољава класи да препусти прављење примерка поткласи.

    (ц) **Уникат** (Singleton): Обезбеђује да класа има само један примерак и даје му глобалну тачку приступа.

2. Структурни обрасци

    (а) **Састав** (Composite): Саставља објекте у структуру стабла да би се представиле хијерархије делова и целина. Састав омогућава клијентима уједначено поступање са појединачним објектима и са саставима објеката.

    (б) **Декоратер** (Decorator): Динамички придружује објекту додатне одговорности. Декоратери су флексибилна алтернатива механизму наслеђивања ради проширивања функционалности.

3. Бихејворални обрасци

   (а) **Посматрач** (Observer): Дефинише зависност један-према-више међу објектима да би приликом промене стања једног објекта сви зависни објекти били обавештени и динамички ажурирани.

   (б) **Стратегија** (Strategy): Дефинише фамилију алгоритама, енкапсулира сваког од њих и омогућава да се могу међусобно замењивати. Стратегија омогућава да се алгоритам мења независно од клијената који га користе.

   (ц) **Шаблонски метод** (Template Method): Дефинише скелет алгоритма неке операције препуштајући имплементацију неких корака поткласи. Шаблонски метод дозвољава да поткласе редефинишу неке кораке алгоритма не мењајући структуру алгоритма.

О свим обрасцима може се детаљно прочитати у ,,Design patterns: elements of reusable object-oriented software” [3].

3 Основи рефакторисања

У литератури о рефакторисању, као основна референца користи се ,,Refactoring: Improving the Design of Existing Code" [2]. Наведена су рефакторисања на која се рад у више наврата позива и надовезује. Нека од обрађених рефакторисања су управо побољшања наредних рефакторисања:

- **Екстракција методе** (Extract Method): Постоји фрагмент кода који се може груписати. Формирати метод од тог фрагмента и назвати га у складу са својом наменом.

- **Екстракција интерфејса** (Extract Interface): Више клијената користи исти подскуп метода интерфејса, односно две класе имплементирају исте делове интерфејса. Екстрактовати тај подскуп метода у јединствен интерфејс.

- **Екстракција класе** (Extract Class): Једна класа обавља посао који би требало да обављају две класе. Креирати нову класу, и преместити релевантна поља и метода из старе у нову класу.

- **Екстракција поткласе** (Extract Subclass): Класа садржи функционалности које имплементирају само неки од примерака. Направити поткласу која имплементира тај подскуп функционалности.

- **Линијски метод** (Inline Method): Тело методе једнако описује намену као и сам назив методе. Уместо методе, позивати директно код из тела, а метод обрисати.

- **Измештање методе** (Move Method): Метод се користи (или ће бити коришћен) од стране друге класе више него од стране класе у којој је дефинисан. Креирати нов метод са сличним телом у класи у којој се најчешће позива. Тело старог метода преправити тако да позива нови метод или га потпуно обрисати.

- **Издизање методе** (Pull-Up Method): Постоје методе које враћају исту вредност када се позивају над различитим поткласама. Померити их у наткласу.

- **Формирање шаблонског метода** (Form Template Method): У поткласама постоје две методе које обављају сличне кораке, у истом редоследу, али се ипак у неком кораку разликују. Кораке претворити у методе са истим потписима, тако да обе позивају исте методе. Затим те методе дефинисати у наткласи.

Детаљније о овим али и осталим рефакторисањима може се прочитати у [2].

4 Рефакторизација код конструкционих образаца дизајна

4.1 Уланчани конструктори

Проблем: *Код садржи конструкторе који иницијализују исте атрибуте, па се треће лице може наћи у недоумици који конструктор да позове*
Решење: *Конструкторе учинити повезаним (тј, уланчати их)*

4.1.1 Објашњење

Исти код који се понавља на више места може изазвати доста проблема. Са минималном изменом попут додавања нове променљиве класи, а при том мењајући само један конструктор али не и остале (на пример, од стране особе која није написала оригиналан код, па не познаје систем до танчина), утемељена је сигурна траса bug-у. Што је већи број конструктора класе, то расте и број места на којима треба начинити измене. Потребно је елиминисати поновљени код где год је то могуће. Ово се углавном постиже уланчавањем конструктора: конструктори специфичније намене позивају конструкторе општије намене, све док се не позове последњи конструктор у ланцу. Такав конструктор назива се ,,свеухватљиви", најопштији је, и покрива све случајеве конструктора који постоје у ланцу пре њега. Он не би требало да буде приватан или заштићен. Уколико у коду, пак, постоји превише конструктора, па трећем лицу није у први мах јасно чему класа служи, након примене овог обрасца прибећи примени обрасца ,,Замене вишеструких конструктора методама креирања", описаног у наредном поглављу.

4.1.2 Потенцијални проблеми

1. **Сврха није јасна**: Када постоји више конструктора који раде сличну ствар, лицу које није имплементирало класу али је користи може бити нејасно који конструктор треба да позове. Треба раздвојити конструкторе са општом сврхом (оне које креирају класу са атрибутима чије вредности морају бити дефинисане) од оних које су специфични.

2. **Понављање кода**: Понављање кода у конструкторима чини класе рањивима на грешке и исто тако отежава руковање тим класама. Треба пронаћи заједничке елементе, сместити их у конструкторе опште намене, а у мање оптшим конструкторима дефинисати специфичне целине.

3. **Недовољна једноставност**: Ако више од једног конструктора садрже исти код, теже је стећи увид у то како се они међу собом разликују. Поједноставити конструкторе правећи од њих ланац, где специфичнији позивају општије.

4.1.3 Практична реализација рефакторизације

1. Пронаћи два конструктора који садрже исти код. Утврдити да ли један може позивати другог тако да поновљени код може бити уклоњен из једног од конструктора.

2. Поновити корак 1 све док се не уклони што је више могуће поновљеног кода

3. Смањити видљивост конструктора који не морају да буду јавни

4.1.4 Пример

```
public class Zajam {
  ...
  public Zajam(float nominalan, float izuzetan, int ocena, Date
      datumIsteka) {
    this.strategija = new UslovZaIzvestajUskladjenosti();
    this.nominalan = nominalan;
    this.izuzetan = izuzetan;
    this.ocena = ocena;
    this.datumIsteka = datumIsteka;
  }
  public Zajam(float nominalan, float izuzetan, int ocena, Date
      datumIsteka, Date datumZrelosti) {
    this.strategija = new RevolvingUslovZaIzvestajUskladjenosti();
    this.nominalan = nominalan;
    this.izuzetan = izuzetan;
    this.ocena = ocena;
```

```
    this.datumIsteka = datumIsteka;
    this.datumZrelosti = datumZrelosti;
  }
  public Zajam(KapitalnaStrategija strategija, float nominalan,
     float izuzetan, int ocena, Date datumIsteka, Date
     datumZrelosti) {
    this.strategija = strategija;
    this.nominalan = nominalan;
    this.izuzetan = izuzetan;
    this.ocena = ocena;
    this.datumIsteka = datumIsteka;
    this.datumZrelosti = datumZrelosti;
  }
}
```

Сва три конструктора иницијализују исте атрибуте. Разматрањем се долази до закључка да би било најједноставније да први конструктор позива трећи, тако да оба конструктора буду што оптималнија. Слично, и други конструктор може позивати трећи. Дакле, трећи конструктор је „свеухватљиви" и последњи у ланцу.
Код би изгледао:

```
public class Zajam {
  ...
  public Zajam(float nominalan, float izuzetan, int ocena, Date
     datumIsteka) {
    this(UslovZaIzvestajUskladjenosti(), nominalan, izuzetan,
       ocena, datumIsteka, null);
  }
  public Zajam(float nominalan, float izuzetan, int ocena, Date
     datumIsteka, Date datumZrelosti) {
    this(new RevolvingUslovZaIzvestajUskladjenosti(), nominalan,
       izuzetan, ocena, datumIsteka, datumZrelosti);
  }
  public Zajam(KapitalnaStrategija strategija, float nominalan,
     float izuzetan, int ocena, Date datumIsteka, Date
     datumZrelosti) {
    this.strategija = strategija;
    this.nominalan = nominalan;
    this.izuzetan = izuzetan;
```

```
        this.ocena = ocena;
        this. datumIsteka = datumIsteka;
        this.datumZrelosti = datumZrelosti;
    }
}
```

У складу са тим како се ови конструктори користе доносе се и закључци о томе каква њихова видљивост треба да буде.

4.2 Метод замене вишеструких конструктора методама креирања

Проблем: *Треће лице је у недоумици који конструктор да позове*
Решење: *Заменити конструктор методама креирања које потписом описују намену конструктора, а враћају новокреирани примерак*

4.2.1 Објашњење

У неким програмским језицима, попут Јаве, сви конструктори имају исти назив као и класа у којој се налазе, само се разликују по параметрима које захтевају. Ако постоји само један конструктор, ово не представља проблем, али већ при постојању два конструктора, програмери морају на основу параметара који се прослеђују закључити који конструктор најбоље одговара њиховим потребама. Проблем са овим приступом је што често није јасно чему који од конструктора служи.
Сазревањем система додају се нови конструктори, и то најчешће без претходног проверавања да ли се стари још увек користе. Ово доводи до гомилања непотребног кода, који ипак троши ресурсе и отежава читљивост и разумевање кода.

4.2.2 Потенцијални проблеми

1. **Сврха није јасна**: Многи конструктори не описују довољно јасну своју намену, односно какав објекат креирају. Овај образац превазилази проблем истоимених конструктора тако што нуди методе који се понашају као конструктори, али и стављају до знања чему треба да служе.

2. **Понављање кода**: Не мора нужно постојати понављање кода, те образац не утиче на оптимизацију ове ставке.

3. **Недовољна једноставност**: Дати образац позитивно утиче на поједностављивање кода због тога што свим учесницима у систему постаје јасније чему који од конструктора служи.

4.2.3 Практична реализација рефакторизације

Пре примене ове методе, није згорег размотрити обрасце ,,Екстракције класе" [2] или ,,Екстракције поткласе" [2]. При примени Екстрак-

ције поткласе, размотрити и образац „Енкапсулације класа методама креирања” (следеће поглавље).

1. Идентификовати класу која има вишеструке конструкторе, али не имплементира превише функционалности.

2. Идентификовати „свеухватљиви“ конструктор или креирати такав применом обрасца „Уланчавања конструктора” (претходно поглавље).

3. Идентификовати конструктор који креира баш жељени тип објекта и написати метод креирања за такав објекат. Дати му име такво да је јасно какав тип објекта креира.

4. Заменити позиве конструктора позивима нове методе.

5. Понављати кораке 3 и 4 све док сваки жељени конструктор није замењен одговарајућим методом креирања.

6. Обрисати конструкторе који се више не позивају.

7. Уколико класа нема поткласа, прогласити преостале конструкторе приватним. Иначе, прогласити их заштићеним.

4.2.4 Пример

```
public class Zajam {
   ...
   public Zajam(double nominalan, int ocena, Date datumZrelosti) {
      this(nominalan, 0.00, ocena, datumZrelosti, null);
   }
   public Zajam(double nominalan, int ocena, Date datumZrelosti,
       Date datumIsteka) {
      this(nominalan, 0.00, ocena, datumZrelosti, datumIsteka);
   }
   public Zajam(double nominalan, double izuzetan, int ocena, Date
       datumZrelosti,
   Date datumIsteka) {
      this(null, nominalan, izuzetan, ocena, datumZrelosti,
         datumIsteka);
   }
```

```
  public Zajam(KapitalnaStrategija strategija, double nominalan,
     int ocena,
         Date datumZrelosti, Date datumIsteka) {
  this(strategija, nominalan, 0.00, ocena, datumZrelosti,
     datumIsteka);
  }
  public Zajam(KapitalnaStrategija strategija, double nominalan,
     double izuzetan,
    int ocena, Date datumZrelosti, Date datumIsteka) {
    this.nominalan = nominalan;
    this.izuzetan = izuzetan;
    this.ocena = ocena;
    this.datumZrelosti = datumZrelosti;
    this.datumIsteka = datumIsteka;
    this.strategija = strategija;
    if (strategija == null) {
    if (datumIsteka == null)
      this.strategija = new UslovKapitalneStrategije();
    else if (datumZrelosti == null)
      this.strategija = new RevolvingKapitalnaStrategija();
    else
      this.strategija = new RCTLKapitalnaStrategija();
  }
}
```

Класа садри пет конструктора. Последњи је „свеухватљиви“. Разматрајући их, није једноставно уочити који креира који тип објекта. Дешава се да програмер који их је написао зна у ком случају је потребно позвати који конструктор, али неко за кога је код нов би имао потешкоћа. Први метод користи се за креирање дугорочног зајма, те би одговарајући метод креирања био:

```
public static Zajam kreierajDugorocniZajam(double nominalan, int
   ocena, Date datumZrelosti) {
  return new Zajam(null, nominalan, 0.00, ocena, datumZrelosti,
     null);
}
```

Стари позиви конструктора били би замењени позивом нове статичке методе:

```
Zajam dugorocniZajam = Zajam.kreierajDugorocniZajam(nominalan,
    ocena, datumZrelosti);
```

Слични кораци понове се и за остале конструкторе. На крају се „свеухватљиви“ конструктор проглашава приватним. Уколико класа садржи много атрибута, постојаће и много различитих могућих конструктора, а самим тим и превише метода креирања, што представља лошу страну овог обрасца.

4.3 Енкапсулација класа методама креирања

Проблем: *Објекти класе једног пакета које имплеменетирају исти интерфејс креирају се директно*
Решење: *Начинити конструкторе приватним и допустити креирање примерака класе коришћењем методе креирања из наткласе*

4.3.1 Објашњење

Могућност директног креирања објеката класа је корисна докле год је потребно бити свестан о постојању тих класа. Коришћењем, на пример, уграђених библиотека за рад са скуповима или мапама, није потребно познавати имплементацију тих структура, већ само крајњи начин употребе. Уколико се ради о таквој ситуацији, то јест овакво стање се не мења (на пример, java.util.Collections библиотека), класе које се директно не користе могу бити невидљиве изван пакета, на уштрб јавних метода креирања које се налазе у наткласи и које враћају тип примерка која имплементира одређен интерфејс. Овакав приступ је користан, првенствено што раздваја интерфејс од имплементације и омогућује да се комуникација између класе и програмера обавља преко одређеног интерфејса. Слично, омогућује редукцију усложњавања пакета, сакривањем класа које није потребно да буду видљиве изван пакета. На крају, појједностављује креирање доступних видова примерака тако што даје на располагање скуп метода креирања са јасном наменом.
Није, пак, опште мишљење да је овакав вид рефакторисања добар. На пример, када наткласа мора да има знање о својим поткласама, то доводи до циклуса зависности. Кокретно, ово значи да се при сваком креирању нове поткласе или измене конструктора постојећих поткласа, мора додати односно променити метод креирања у наткласи.

4.3.2 Потенцијални проблеми

1. **Сврха није јасна**: Када се очекује да се са класама комуницира преко једног интерфејса, ово треба бити јасно из кода. Одати намену проглашавањем конструктора заштићеним, а нове примерке креирати помоћу метода креирања из наткласе.

2. **Понављање кода**: Понављање кода није проблем у овом случају.

3. **Недовољна једноставност**: Начинити класе видљивим а очекивати да се са њима интерагује преко једног интерфејса није једноставан задатак. Појједноставити ово онемогућавањем креирања објеката тих класа тако што ће у натакласи бити понуђени методи креирања за њих.

4.3.3 Практична реализација рефакторизације

Предуслов: Све класе имплементирају заједнички, јавни интерфејс и налазе се у истом пакету.

1. У наткласи написати метод креирања са јасном наменом за тип примерка који жељени конструктор креира. Повратни тип методе треба да буде интерфејс који се наслеђује, а тело методе треба да позива одговарајући конструктор.

2. Све постојеће позиве тог конструктора заменити позивима новонаписане методе креирања.

3. Поновити кораке 1 и 2 за све типове примерака које враћају конструктори класе.

4. Прогласити конструктор невидљивим изван пакета.

5. Понављати кораке 1-4 све док сви жељени конструктори поткласа не буду замењени одговарајућим методама креирања из наткласе.

4.3.4 Пример

Дат је пример хијерархије класа које се налазе у пакету званом opisi, и омогућавају мапирање објеката са ентитетима из базе података:

```
package opisi;

public abstract class OpisAtributa {
  protected OpisAtributa(. . .)

  public class LogickiOpis extends OpisAtributa {
    public LogickiOpis( ) {
    super(  );
    }
```

```
public class PodrazumevaniOpis extends OpisAtributa {
  public PodrazumevaniOpis() {
  super(  );
}

public class OpisReference extends OpisAtributa {
  public OpisReference( ) {
  super(  );
}
```

Апстрактни конструктор OpisAtributa је заштићен, а конструктори поткласа су јавни. Примера ради, нека је посматрана поткласа PodrazumevaniOpis:

```
protected List kreirajOpiseAtributa() {
    List rezultat = new ArrayList();
    rezultat.add(new PodrazumevaniOpis("remoteId", getClass(),
       Integer.TYPE));
    rezultat.add(new PodrazumevaniOpis("createdDate", getClass(),
       Date.class));
    rezultat.add(new PodrazumevaniOpis("lastChangedDate",
       getClass(), Date.class));
    rezultat.add(new OpisReference("createdBy", getClass(),
       Korisnik.class,
    KrajnjiKorisnik.class));
    rezultat.add(new OpisReference("lastChangedBy", getClass(),
       Korisnik.class,
    KrajnjiKorisnik.class));
    rezultat.add(new PodrazumevaniOpis("optimisticLockVersion",
       getClass(), Integer.TYPE));
    return rezultat;
}
```

Из контекста донешен је закључак да се тај конструктор користи када је потребно представити мапирање за типове Integer и Date: Прво се напише метод креирања за тип Integer:

```
public abstract class OpisAtributa {
  public static OpisAtributa zaCeoBroj(...) {
    return new PodrazumevaniOpis(...);
```

```
}
```

Повратни тип је OpisAtributa јер је намера омогућити комуникацију са свим поткласама OpisAtributa наткласе преко ње саме, као и сакривање поткласа изван пакета opisi. Минули позиви PodrazumevaniOpis конструктора замењују се позивима новокреиране методе креирања:

```
protected List kreirajOpiseAtributa() {
   List rezultat = new ArrayList();
   rezultat.add(OpisAtributa.zaCeoBroj("remoteId", getClass()));
   rezultat.add(new PodrazumevaniOpis("createdDate", getClass(),
      Date.class));
   rezultat.add(new PodrazumevaniOpis("lastChangedDate",
      getClass(), Date.class));
   rezultat.add(new OpisReference("createdBy", getClass(),
      Korisnik.class,
   KrajnjiKorisnik.class));
   rezultat.add(new OpisReference("lastChangedBy", getClass(),
      Korisnik.class,
   KrajnjiKorisnik.class));
   rezultat.add(OpisAtributa.zaCeoBroj("optimisticLockVersion",
      getClass()));
   return rezultat;
}
```

Поступак се настави за преостала два типа објекта које враћа конструктор. На крају се PodrazumevaniOpis прогласи видљивим само унутар пакета:

```
public class PodrazumevaniOpis extends OpisAtributa {
   protected PodrazumevaniOpis() {
   super(  );
}
```

Ефекат оваквих измена је:

1. Приступање поткласама OpisAtributa-a преко њега самог.
2. Спречавање директног креирања објеката поткласа.
3. Обавештавање других програмера да поткласе OpisAtributa-a нису

предвиђене да буду јавне. Према конвенцији, то значи да се приступ у том случају нуди преко наткласе и одговарајућег интерфејса.

4.4 Екстракција класа креирања

Проблем: *Због превеликог броја метода креирања унутар класе, може се десити да намена класе није на први поглед јасна*
Решење: *Изместити методе креирања за повезан скуп класа у једну тзв. ,,класу креирања"*

4.4.1 Објашњење

У својој бити, ово рефакторисање је тзв. метод ,,Екстракције класе" [2]. Разлика је у томе што се обавља над методама креирања унутар класе. Појава неколико метода креирања унутар једне класе није спорна, али како им број расте, све је теже уочити праву сврху класе. Уколико се то деси, идентитет класе може се повратити измештањем метода креирања у класу чија је то једина сврха. Такве класе се најчешће имплементирају као класе које садрже статичке методе, при чему сваки од њих креира и враћа објекат.

4.4.2 Потенцијални проблеми

1. **Сврха није јасна**: Када логика креирања објекта доминира у односу на остатак логике унутар класе, тада класа не одаје своју сврху сасвим јасно. Чин креирања енкапсулирати у специјалну класу чија је то једина сврха.
2. **Понављање кода**: Понављање кода није проблем у овом случају.
3. **Недовољна једноставност**: Када се одговорности креирања објекта измешају са главним одговорностима класе, она више не може бити једноставна. Поједностављује се премештањем кода који служи за креирање објекта у класу креирања.

4.4.3 Практична реализација рефакторизације

1. Идентификовати класу (у даљем тексту: класа ,,А") која има много метода креирања.
2. Креирати класу која ће постати класа креирања. Наденути јој име у складу са својом сврхом, која је да креира различите објекте из скупа класа.

3. Преместити све методе креирања из А у нову класу, водећи рачуна о видљивости метода као и правима приступа.

4. Све досадашње позиве конструктора у осталом коду заменити позивима нових метода унутар класе креирања.

4.4.4 Пример

Нека се посматра класа Zajam која поседује доста различитих одговорности:

```
public class Zajam {
   ...
   private double nominalan;
   private double izuzetan;
   private int ocena;
   private Date pocetak;
   private KapitalnaStrategija strategija;
   private Date datumIsteka;
   private Date datumZrelosti;

   protected Zajam(double nominalan, Date pocetak, Date datumIsteka,
   Date datumZrelosti, int ocenaRizika, KapitalnaStrategija
      strategija) {
      this.nominalan = nominalan;
      this.pocetak = pocetak;
      this.datumIsteka = datumIsteka;
      this.datumZrelosti = datumZrelosti;
      this.ocena = ocenaRizika;
      this.strategija = strategija;
   }
   public double izracunajKapital() {
      return strategija.izracunaj(this);
   }
   public void setIzuzetni(double noviIzuzetni) {
      izuzetan = noviIzuzetni;
   }
   ...
   public static Zajam noviSavetnik(double nominalan, Date pocetak,
      Date datumZrelosti, int ocena)
```

```
    return new Zajam(nominalan, pocetak, null, datumZrelosti, ocena,
        new KapitalDugorocnogZajma());
}
public static Zajam novoKreditnoPismo(double nominalan, Date
    pocetak,
Date datumZrelosti, int ocena) {
    return new Zajam(nominalan, pocetak, null, datumZrelosti, ocena,
        new KapitalDugorocnogZajma());
}
public static Zajam noviRCTL(double nominalan, Date pocetak,
Date datumIsteka, Date datumZrelosti, int ocena) {
    return new Zajam(nominalan, pocetak, datumIsteka, datumZrelosti,
        ocena, new RCTLKapital());
}
public static Zajam noviRevolving(double nominalan, Date pocetak,
Date datumIsteka, int ocena) {
    return new Zajam(nominalan, pocetak, datumIsteka, null, ocena,
        new RevolvingKapital());
}
public static Zajam noviSPLC(double nominalan, Date pocetak,
Date datumZrelosti, int ocena) {
    return new Zajam(nominalan, pocetak, null, datumZrelosti, ocena,
        new KapitalDugorocnogZajma());
}
public static Zajam noviDugorocniZajam(double nominalan, Date
    pocetak,
Date datumZrelosti, int ocena) {
    return new Zajam(nominalan, pocetak, null, datumZrelosti, ocena,
        new KapitalDugorocnogZajma());
}
public static Zajam noviPromenljiviZajam(double nominalan, Date
    pocetak,
Date datumIsteka, Date datumZrelosti, int ocena) {
    return new Zajam(nominalan, pocetak, datumIsteka, datumZrelosti,
        ocena, new RCTLKapital());
}
}
```

Први корак је креирати класу KreatorZajma, која ће бити полазиште за креирање примерака типа Zajam:

```
public class KreatorZajma {
}
```

Сада се све методе креирања преместе из Zajam у KreatorZajma, али тако да се обе класе налазе унутар истог пакета како би се права приступа могла прилагодити:

```
public class KreatorZajma {
   public static Zajam noviSavetnik(double nominalan, Date pocetak,
   Date datumZrelosti, int ocena) {
      return new Zajam(nominalan, pocetak, null, datumZrelosti,
          ocena, new KapitalDugorocnogZajma());
   }
   public static Zajam novoKreditnoPismo(double nominalan, Date
       pocetak,
   Date datumZrelosti, int ocena) {
      return new Zajam(nominalan, pocetak, null, datumZrelosti,
          ocena, new KapitalDugorocnogZajma());
   }
   public static Zajam noviRCTL(double nominalan, Date pocetak,
   Date datumIsteka, Date datumZrelosti, int ocena) {
      return new Zajam(nominalan, pocetak, datumIsteka,
          datumZrelosti, ocena, new RCTLKapital());
   }
   public static Zajam noviRevolving(double nominalan, Date pocetak,
   Date datumIsteka, int ocena) {
      return new Zajam(nominalan, pocetak, datumIsteka, null,
          ocena, new RevolvingKapital());
   }
   public static Zajam noviSPLC(double nominalan, Date pocetak,
   Date datumZrelosti, int ocena) {
      return new Zajam(nominalan, pocetak, null, datumZrelosti,
          ocena, new KapitalDugorocnogZajma());
   }
   public static Zajam noviDugorocniZajam(double nominalan, Date
       pocetak,
   Date datumZrelosti, int ocena) {
      return new Zajam(nominalan, pocetak, null, datumZrelosti,
          ocena, new KapitalDugorocnogZajma());
   }
```

```
  public static Zajam noviPromenljiviZajam(double nominalan, Date
      pocetak,
  Date datumIsteka, Date datumZrelosti, int ocena) {
    return new Zajam(nominalan, pocetak, datumIsteka,
        datumZrelosti, ocena, new RCTLKapital());
  }
}
```

За крај променити све позиве конструктора:

```
Zajam dugorocniZajam = Zajam.noviDugorocniZajam()
```

са позивима метода:

```
Zajam dugorocniZajam = KreatorZajma.noviDugorocniZajam()
```

И тиме је ово рефакторисање приведено крају.

4.5 Полиморфно креирање уместо Производне методе

Проблем: *Класе у хијерархији имплементирају метод слично, једина разлика је у типу објекта који креирају*

Решење: *Направити само једну верзију такве методе, сместити је у натакласу, тако да позива производни метод који регулише креирање*

4.5.1 Објашњење

Производна метода [3] је полиморфна метода која креира и враћа тзв. производ (eng. Product), а његова декларација налази се у наткласи или унутар интерфејса. Метода може бити имплементирана у наткласи, а предефинисана у поткласама како би послужила за креирање тј. иницијализацију адекватног типа производа. Користи се, на пример, када се метод налази у наткласи а предефинисан је у једној или више поткласа, готово идентично у односу на оригиналну имплементцију изузев разлике у типу објекта који се креира. Тада се све верзије овог метода могу обухватити једним, који се налази у наткласи, и назива се Шаблонским методом [3].Он преузима позиве и преусмерава их одговарајућим поткласама, како би се оне постарале о креирању правог подтипа.

4.5.2 Потенцијални проблеми

1. **Сврха није јасна**: Добро осмишљен назив производне методе боље одаје сврху него када се директно позива конструктор. Фабрички метод такође наговештава да све класе које креира заправо имплементирају заједнички интерфејс.

2. **Понављање кода**: Понављање кода је углавном последица потребе да се један исти објекат креира на више различитих начина. Елиминисати понављање писањем јединственог метода који креира жељени објекат позивом производне методе.

3. **Недовољна једноставност**: Једноставније је разумети код који садржи позиве ка производној методи, него онај који директно креира објекте. Међутим, овакви позиви могу деловати комплексније у односу на обичне, директне позиве метода које креирају објекте.

4.5.3 Практична реализација рефакторизације

Прва ситуација

Метод је поновљен јер наткласа и поткласа креирају објекат на различите начине

1. У методи наткласе, применити ,,Екстракцију методе" [2] на код који креира објекат, добијајући производни метод. Повратни тип мора бити генерички.

2. На сваку поткласу која предефинише горњи метод, на начин описан под 1, издвојити део који креира објекат.

3. Уклонити верзије метода из поткласа које више нису потребне. Уколико се не очекује да њихове поткласе предефинишу тај метод, прогласити га коначним (кључна реч final).

Друга ситуација

Метод је поновљен унутар више поткласа јер оне креирају објекат на различите начине

1. Креирати производни метод у наткласи. Прогласити га апстрактним уколико није смислено да постоји предефинисана имплементација, иначе га дефинисати тако да враћа објекат дефинисан наткласом.

2. У свакој поткласи која имплементира метод, издвојити део који креира објекат како би се произвео производни метод са потписом који има и производни метод наткласе.

3. Применити рефакторисање ,,Формирање Шаблонског метода" [2].

4.5.4 Примери

Прва ситуација

Дат је пример који демонстрира систем који служи за исписивање текста у XML формату. Поткласе класе ApstraktniPisacStrane саме одлучују о томе који текст ће произвести. ApstraktniPisacStrane садржи шаблонски метод stranaTeksta():

```
public abstract class ApstraktniPisacStrane {
   public String stranaTeksta() {
   OutputBuilder graditeljIzlaza = new XMLBuilder();
   upisiZaglavljeU(graditeljIzlaza);
   upisiTeloU(graditeljIzlaza);
   upisPodnozjeU(graditeljIzlaza);
   return graditeljIzlaza.toString();
}

protected abstract void upisiTeloU(OutputBuilder graditelj);
protected abstract void upisPodnozjeU(OutputBuilder graditelj);
protected abstract void upisiZaglavljeU(OutputBuilder graditelj);
```

Метод stranaTeksta() креира објекат OutputBuilder подтипа XMLBuilder, и тај објекат прослеђује трима методама, након чега враћа објекат типа OutputBuilder. Поткласе предефинишу те методе како би се адекватно понашале при прослеђеном типу објекта. Међутим, понекад је потребно да излаз ипак буде типа DOMBuilder, који даје приступ DOM-у HTML странице. Поткласа класе ApstraktniPisacStrane, звана PrimernoObezbedjeniPisacStrane, имплементира баш овај вид излаза, те је stranaTeksta() метод, у поткласи, предефинисан на следећи начин:

```
public class PrimarnoObezbedjeniPisacStrane extends
    ApstraktniPisacStrane {

   public String stranaTeksta() {
      OutputBuilder graditeljIzlaza = new DOMBuilder();
      upisiZaglavljeU(graditeljIzlaza);
      upisiTeloU(graditeljIzlaza);
      upisPodnozjeU(graditeljIzlaza);
      return graditeljIzlaza.toString();
   }
```

Уочава се да је метод готово идентичан, једина је разлика у типу OutputBuilder подтипа који се креира. На основу прве ситуације, рефакторисање би било изведено тако што се у наткласи, ApstraktniPisacStrane, примени ,,Екстракција метода" [2], како би се произвео производни метод, назван kreirajGraditeljIzlaza():

```
public abstract class ApstraktniPisacStrane {
```

```
  public String stranaTeksta() {
    OutputBuilder graditeljIzlaza = kreirajGraditeljIzlaza();
    upisiZaglavljeU(graditeljIzlaza);
    upisiTeloU(graditeljIzlaza);
    upisPodnozjeU(graditeljIzlaza);
    return graditeljIzlaza.toString();
  }
  protected OutputBuilder kreirajGraditeljIzlaza() {
    return new XMLBuilder();
  }
```

Слично се учини и са другом поткласом:

```
public class PrimarnoObezbedjeniPisacStrane extends
    ApstraktniPisacStrane {
  ...
  public String stranaTeksta() {
    OutputBuilder graditeljIzlaza = kreirajGraditeljIzlaza();
    upisiZaglavljeU(graditeljIzlaza);
    upisiTeloU(graditeljIzlaza);
    upisPodnozjeU(graditeljIzlaza);
    return graditeljIzlaza.toString();
  }
  protected OutputBuilder kreirajGraditeljIzlaza() {
    return new DOMBuilder();
  }
```

Сада се метод stranaTeksta() може обрисати из класе PrimarnoObezbedjeniPisacStrane.

Друга ситуација

На даље, разматран је пример где до понављања долази у двема поткласама. Ова ситуација се може превазићи увођењем производне као и шаблонске методе.

```
abstract class Upit {
  public abstract void izvrsiUpit() throws UpitException;

  class UpitSD51 extends Upit ...
  public void izvrsiUpit() throws UpitException {
```

```
    if (sdUpit != null) sdUpit.izbirisiRezultate();
    sdUpit = sdSesija.createQuery(SDUpit.OTVOREN_ZA_UPIT);
    executeQuery(sdUpit);
  }

  class UpitSD52 extends Upit ...
  public void izvrsiUpit() throws UpitException {
    if (sdUpit != null) sdUpit.izbirisiRezultate();
    sdUpit =
       sdSesijaSaUlaskom.createQuery(SDUpit.OTVOREN_ZA_UPIT);
    executeQuery(sdUpit);
  }
```

Додаје се производни метод наткласи Upit и проглашава се апстрактним како би поткласе могле да га наследе:

```
abstract class Upit ...
  protected abstract SDUpit kreirajUpit() throws UpitException;
```

Потом се додаје производни метод и поткласама, издвајањем дела која креира објекат из њихових верзија методе izvrsiUpit():

```
class UpitSD51 extends Upit ...

  protected SDUpit kreirajUpit() {
    return sdSesija.kreirajUpit(SDUpit.OTVOREN_ZA_UPIT);
  }
  public void izvrsiUpit() throws UpitException ...
    if (sdUpit != null) sdUpit.izbirisiRezultate();
    sdUpit = kreirajUpit();
    executeQuery(sdUpit);
  }

class UpitSD52 extends Upit ...
  protected SDUpit kreirajUpit() {
    return sdSesijaSaUlaskom.kreirajUpit(SDUpit.OTVOREN_ZA_UPIT);
  }
  public void izvrsiUpit() throws UpitException {
    if (sdUpit != null) sdUpit.izbirisiRezultate();
    sdUpit = kreirajUpit();
    executeQuery(sdUpit);
```

```
}
```

Коначно, примењује се рефакторисање ,,Формирање шаблонског метода" [2], како би била произведена јединствена верзије методе izvrsiUpit() и то у наткласи:

```
abstract class Upit ...
protected abstract SDUpit kreirajUpit() throws UpitException;

public void izvrsiUpit() throws UpitException {
   if (sdUpit != null) sdUpit.izbirisiRezultate();
   sdUpit = kreirajUpit();
   executeQuery(sdUpit);
}

class UpitSD51 extends Upit ...
   protected SDUpit kreirajUpit() {
      return sdSesija.kreirajUpit(SDUpit.OTVOREN_ZA_UPIT);
   }
class UpitSD52 extends Upit ...
   protected SDUpit kreirajUpit() {
      return sdSesijaSaUlaskom.kreirajUpit(SDUpit.OTVOREN_ZA_UPIT);
   }
```

5 Рефакторизација код структурних образаца дизајна

5.1 Замена условних рачуница обрасцем Стратегије

Проблем: *Код садржи много условне логике при рачунању*
Решење: *Проследити рачунске задатке објекту Стратегије*

5.1.1 Објашњење

Превише if...else исказа може прилично закомпликовати код и учинити га тешким за разумевање. Стратегија је образац који добро барата кодом који се бави било каквим прорачунавањем. Објекат у ком је потребно нешто израчунати (у даљем тексту: контекст) садржи тзв. објекат Стратегије коме се прослеђује рачунски задатак. Ово поједностављује контекстну класу премештањем рачуна у мале независне објекте чије је баш то рачунање једина одговорност, али на више различитих начина. Такође, потребно је одлучити се о томе како ће се из тог, новог објекта Стратегије, приступати подацима из контекста. Једна од могућности је проследити му цео контекст, а атрибуте тог објекта начинити јавним или обезбедити потребне get методе.

5.1.2 Потенцијални проблеми

1. **Сврха није јасна**: Често се због превише if...else iskaza не види који су тачно кораци при рачунању, а самим тим ни чему оно заиста користи. Овај проблем се превазилази измештањем рачуна у посебне класе Стратегије, где свака има засебну сврху.

2. **Понављање кода**: Готово увек долази до понављања условних провера при оваквом виду рачунања. На начин описан под 1 превазилази се и овај проблем.

3. **Недовољна једноставност**: Класе које садрже превише условне логике не могу бити једноставне. Такође, ако класа садржи превише условне логике како би израчунала нешто на више различитих начина, може се догодити да буде и комплекснија него што

стварно јесте. Поједноставити овакве класе издвајањем свих варијанти рачуна у засебне класе Стратегије, а онда им делегирати контекст како би се обавило рачунање.

5.1.3 Практична реализација рефакторизације

1. У класи (у даљем тексту: класа ,,A”) идентификовати метод који садржи доста рачунања и условне логике. То ће на даље бити тзв. контекст за објекат Стратегије.

2. Креирати класу Стратегије, и наденути јој име у складу са тим шта је њен задатак. На пример, имену контекстне класе додати реч,,Стратегија”.

3. Применити рефакторисање ,,Измештања методе” [2] како би се рачунски код сада налазио у новој класи. Ако се у њему користе информације из класе A, проследити му A као параметар конструктора или параметар методе која је главна у класи Стратегије (на пример, она која позива помоћне функције и враћа резултат). Омогућити да потребни атрибути буду видљиви.

4. У класи A креирати атрибут (у даљем тексту: ,,S”) за класу Стратегије, и инстанцирати објекат.

5. Реажурирати рачунски метод из A тако да делегира рачун објекту S.

6. У класи Стратегије, на првобитни рачунски метод као и све помоћне методе, применити рефакторисање ,,Замене условног поломорфизмом” [2].

7. У класи A додати код који поставља вредност за S, тако да се то врши или интерно или тако да се омогући постављање изван класе A, на пример, додавањем одговарајуће set методе.

5.1.4 Пример

У датом примеру, приказан је горе описан начин рефакторисања. Свакако, оно није једина могућност. Класа Zajam на којој се врши рефакторисање, могла је бити подељена и на три поткласе, при чему свака

предефинише исти метод на себи својствен начин. Ипак, примењено је баш ово рефакторисање, јер је било потребно ad-hoc рачунање, у ком није било места доношење одлуке о адекватном подтипу који одговара тренутном контексту. Нека је дат метод izracunajKapital() са својим помоћним методама:

```
public class Zajam ...
  private double nominalni;
  private double izuzetni;
  private int ocena;
  private double procenatNeiskoriscenosti;
  private Date pocetak;
  private Date datumIsteka;
  private Date datumZrelosti;
  private static final int MILISEKUNDA_PO_DANU = 86400000;

  public double izracunajKapital() {
    return iznosRizika() * trajanje() *
       FaktoriRizika.zaOcenuRizika(ocena);
  }
  private double izracunajIznosRizikaZaNeiskorisceno() {
    return (nominalni - izuzetni) * procenatNeiskoriscenosti;
  }
  private double trajanje() {
  if (datumIsteka == null)
    return ((datumZrelosti.getTime() - pocetak.getTime()) /
       MILISEKUNDA_PO_DANU) / 365;
  else if (datumZrelosti == null)
    return ((datumIsteka.getTime() - pocetak.getTime()) /
       MILISEKUNDA_PO_DANU) / 365;
  else {
    long milisekundaDoIsteka = datumIsteka.getTime() -
       pocetak.getTime();
    long milisekundaOdIstekaDoZrelosti = datumZrelosti.getTime()
       - datumIsteka.getTime();
    double trajanjeRevolvinga = (milisekundaDoIsteka /
       MILISEKUNDA_PO_DANU) / 365;
    double trajanjeZajma = (milisekundaOdIstekaDoZrelosti /
       MILISEKUNDA_PO_DANU) / 365;
    return trajanjeRevolvinga + trajanjeZajma;
```

```
      }
   }
   private double iznosRizika() {
      if (procenatNeiskoriscenosti != 1.00)
         return izuzetni + izracunajIznosRizikaZaNeiskorisceno();
      else
         return izuzetni;
      }
   public void setIzuzetni(double noviIzuzetni) {
      izuzetni = noviIzuzetni;
   }
   private void setProcenatNeiskoriscenosti() {
      if (datumIsteka != null && datumZrelosti != null) {
         if (ocena > 4)
            procenatNeiskoriscenosti = 0.95;
         else
            procenatNeiskoriscenosti = 0.50;
      } else if (datumZrelosti != null) {
         procenatNeiskoriscenosti = 1.00;
      } else if (datumIsteka != null) {
         if (ocena > 4)
            procenatNeiskoriscenosti = 0.75;
         else
            procenatNeiskoriscenosti = 0.25;
   }
```

Објекат Стратегије садржаће код методе izracunajKapital(), те изгледа:

```
public class KapitalnaStrategija {
   public double izracunaj() {
      return iznosRizika() * trajanje() *
          FaktoriRizika.zaOcenuRizika(ocena);
   }
}
```

Наравно, потребно је омогућити приступ помоћним методама, те објекат Zajam бива прослеђен, притом мењајући видљивост одговарајућим пољима:

```
public double izracunaj(Zajam zajam) {
```

```
   return zajam.iznosRizika() * zajam.trajanje() *
      FaktoriRizika.zaOcenuRizika(zajam.ocena);
}
```

Сада је потребно преместити сваки рачунски корак, веома пажљиво, у нову класу. Прво, метод iznosRizika():

```
public double iznosRizika() {
   if (procenatNeiskoriscenosti != 1.00)
      return izuzetni + izracunajIznosRizikaZaNeiskorisceno();
   else
      return izuzetni;
   }
```

Он у себи користи вредности атрибута објекта Zajam. Нека се проучавањем кода долази до закључка да је поље izuzetni веома коришћено унутар читаве класе, док се поље procenatNeiskoriscenosti и методе setProcenatNeiskoriscenosti() и izracunajIznosRizikaZaNeiskorisceno() користе само у izracunajKapital() методи. Дакле, осим атрибута izuzetni, преостали код се може лагодно преместити:

```
public class KapitalnaStrategija {
   private Zajam zajam;
   public double izracunaj(Zajam zajam) {
   this.zajam = zajam;
   return iznosRizika() * zajam.trajanje() *
      FaktoriRizika.zaOcenuRizika(zajam.ocena);
   }
   private double calcProcenatNeiskoriscenosti() {
      if (zajam.datumIsteka != null && zajam.datumZrelosti != null)
         {
         if (zajam.ocena > 4)
            return 0.95;
         else
            return 0.50;
      } else if (zajam.datumZrelosti != null) {
         return 1.00;
      } else if (zajam.datumIsteka != null) {
         if (zajam.ocena > 4)
            return 0.75;
```

```
        else
            return 0.25;
        }
            return 0.0;
    }
    private double izracunajIznosRizikaZaNeiskorisceno() {
        return (zajam.nominalni - zajam.izuzetni) *
            calcProcenatNeiskoriscenosti();
    }
    public double iznosRizika() {
        if (calcProcenatNeiskoriscenosti() != 1.00)
            return zajam.izuzetni +
                izracunajIznosRizikaZaNeiskorisceno();
        else
            return zajam.izuzetni;
    }
}
```

Потребно је још подесити видљивост одговарајућих поља:

```
  public class Zajam ...
  public double nominalni;
  public double izuzetni;
  public int ocena;
  //private double procenatNeiskoriscenosti; <-- zamenjen metodom
     iz KapitalnaStrategija
  public Date pocetak;
  public Date datumIsteka;
  public Date datumZrelosti;
```

Наравно, друга могућност је написати get методе, а самим тим ускладити и код у KapitalnaStrategija класи. На крају, додати поље KapitalnaStrategija у Zajam класи:

```
public class Zajam ...
  private KapitalnaStrategija strategija = new
     KapitalnaStrategija();
```

Затим позвати izracunajKapital() метод:

```
public double izracunajKapital() {
```

```
  return strategija.izracunaj(this);
}
```

Као што је већспоменуто, ово рефакторисање се може обавити и другачије. У овом случају, досадашње рефакторисање може бити даље побољшано управо тим „Заменом условног поломорфизмом” [2]. Даљи кораци демонстрирају ово рефакторисање.
Прво се идентификују сви могући начини обављања овог рачуна, што су у овом случају три различита начина која одговарају различитим типовима позајмице.
Креира се прва поткласа за дугорочне позајмице:

```
public class KapitalDugorocnogZajma extends KapitalnaStrategija {
}
```

Она треба да садржи код специфичан за њу:

```
public class KapitalDugorocnogZajma extends KapitalnaStrategija {
  protected double trajanje() {
  return (
  (zajam.getDatumZrelosti().getTime() -
     zajam.getPocetak().getTime()) / MILISEKUNDA_PO_DANU) / 365;
  }
  protected double iznosRizika() {
    return zajam.getIzuzetan();
  }
}
```

У складу са тим, морају се унети и измене у наткласи:

```
public class Zajam...
  private KapitalnaStrategija strategija;

  protected Zajam(double nominalni, Date pocetak, Date datumIsteka,
  Date datumZrelosti, int ocena, KapitalnaStrategija strategija) {
  this.nominalni = nominalni;
  this.pocetak = pocetak;
  this.datumIsteka = datumIsteka;
  this.datumZrelosti = datumZrelosti;
  this.ocena = ocena;
  this.strategija = strategija;
```

```
}

public static Zajam noviRCTL(double nominalni, Date pocetak, Date
    datumIsteka,
Date datumZrelosti, int ocena) {
   return new Zajam(nominalni, pocetak, datumIsteka, datumZrelosti,
       ocena, new KapitalnaStrategija());
}
public static Zajam noviRevolving(double nominalni, Date pocetak,
    Date datumIsteka,
int ocena) {
   return new Zajam(nominalni, pocetak, datumIsteka, null, ocena,
       new KapitalnaStrategija());
}
public static Zajam noviDugorocniZajam(double nominalni, Date
    pocetak, Date datumZrelosti,
int ocena) {
   return new Zajam(nominalni, pocetak, null, datumZrelosti, ocena,
       new KapitalDugorocnogZajma());
}
```

Слични кораци понове се и за преостала две поткласе. У случају да је остало још некаквог понављања кода, може се код рефакторисати и даље. На пример, све горње методе садрже три исте линије кода, које садрже формулу за рачунање временског периода. Како би се елиминисало ово понављање, примењује се рефакторисање „Издизање методе" [2]:

```
public abstract class KapitalnaStrategija
   private static final int DANA_U_GODINI = 365;

   protected double izracunajTrajanje(Date pocetak, Date kraj) {
      return ((kraj.getTime() - pocetak.getTime()) /
          MILISEKUNDA_PO_DANU) / DANA_U_GODINI;
   }

   public class KapitalDugorocnogZajma extends KapitalnaStrategija
   protected double trajanje() {
      return izracunajTrajanje(zajam.getPocetak(),
          zajam.getDatumZrelosti());
```

```
   }

   public class RevolvingKapital extends KapitalnaStrategija {
      protected double trajanje() {
         return izracunajTrajanje(zajam.getPocetak(),
            zajam.getDatumIsteka());
   }

   public class RCTLKapital extends KapitalnaStrategija
      protected double trajanje() {
         double trajanjeRevolvinga =
            izracunajTrajanje(zajam.getPocetak(),
            zajam.getDatumIsteka());
         double trajanjeZajma =
            izracunajTrajanje(zajam.getDatumIsteka(),
            zajam.getDatumZrelosti());
         return trajanjeRevolvinga + trajanjeZajma;
      }
}
```

Тиме је ово рефакторисање завршено.

5.2 Замена имплицитног формирања дрвета Саставом

Проблем: *Имплицитно се формира структура дрвета, користећи примитиван тип репрезентације, на пример надовезивањем елемента на један објекат типа String*
Решење: *Заменити примитивну репрезентацију сложеном*

5.2.1 Објашњење

Један од проблема са оваквом конструкцијом дрвета је у томе што је код веома спрегнут са самом репрезентацијом. Можда овакво прављење дрвета делује једноставније, али када је потребно обавити било какву измену, поступак може бити веома временски захтеван. Код сложене репрезентације, клијент само треба да кодом саопшти шта и где жели да дода у дрво. Овакав приступ је много мање осетљив на грешке.
Ипак, у случају да у систему не постоји превише дрвета, примена Састава може представљати пре-рефакторисање. Уколико се временом, пак, јави потреба за креирањем више дрвета, може се прво код појједноставити раздвајањем логике грађења дрвета од логике приказивања.

5.2.2 Потенцијални проблеми

1. **Сврха није јасна**: Најподеснији код за грађење дрвета одаје структуру дрвета у тој мери да не оптерећује детаљима креирања онога ко тај код чита. Примитивно грађење дрвета излаже превише детаља. Драђење дрвета обрасцем Састава је боље у смислу да сакрива неке детаље везане за сам процес конструкције.

2. **Понављање кода**: Код који негенерички гради дрво често садржи исти скуп корака - прављење чвора, додавање чвора дрвету, балансирање и слично. Сложено грађење минимизује понављање тако што енкапсулира инструкције које се понављају.

3. **Недовољна једноставност**: Већа је вероватноћа да ће се грешка јавити при ручном прављењу дрвета него ако се то ради помоћу обрасца Састава. Овако изграђена дрвета су далеко једноставнија.

5.2.3 Практична реализација рефакторизације

1. Лоцирати код са примитивним грађењем дрвета који треба бити рефакторисан.
2. Идентификовати типове чворова за нову сложену репрезентацију.
3. Омогућити чворовима да имају потомке. Онемогућити брисање потомака уколико систем има само могућност додавања, али не и брисања чворова.
4. Уколико је потребно, омогућити постављање својстава чворовима.
5. Заменити првобитни код који креира дрво позивима нове сложене репрезентације.

5.2.4 Пример

Нека је дат пример генерисања XML документа у Јави:

```
String porudzbine = "<porudzbine>";
    porudzbine += "<porudzbina broj='123'>";
      porudzbine += "<stavka broj='x1786'>";
        porudzbine += "Vrata od kola";
      porudzbine += "</stavka>";
    porudzbine += "</porudzbina>";
  porudzbine += "</porudzbine>";
```

У овом случају, сваки чвор у дрвету има отворену етикету « porudzbine» и затворену етикету «/ porudzbine». Неки чворови имају и својства и вредности. На основу кода, идентификује се само један тип чвора који је потребан за сложену верзију дрвета. Како се ради о „пипавом" поступку, примењује се test-first приступ, тако да се прво прави чвор типа XMLEtiketa као и метод који креира дрво са једним чвором:

```
public void testJedanCvor() {
  String ocekivaniRezultat =
    "<porudzbine>" +
    "</porudzbine>";
  XMLEtiketa porudzbine = new XMLEtiketa("porudzbine");
  assertXMLEquals("drvo sa jednim cvorom", ocekivaniRezultat,
     porudzbine.toString());
```

```
}
public class XMLEtiketa {
   private String imeEtikete;
   public XMLEtiketa(String ime) {
      imeEtikete = ime;
   }
   public String toString() {
      String rezultat = new String();
         rezultat += "<" + imeEtikete + ">";
         rezultat += "</" + imeEtikete + ">";
      return rezultat;
   }
}
```

Потребно је омогућити чворовима да имају потомке:

```
public void testDodavanjaPotomakaDrvetu() {
   String ocekivaniRezultat =
      "<porudzbine>" +
         "<porudzbina>" +
            "<stavka>" +
            "</stavka>" +
         "</porudzbina>" +
      "</porudzbine>";
   XMLEtiketa porudzbine = new XMLEtiketa("porudzbine");
   XMLEtiketa porudzbina = new XMLEtiketa("porudzbina");
   XMLEtiketa stavka = new XMLEtiketa("stavka");
   porudzbine.add(porudzbina);
   porudzbina.add(stavka);
   assertXMLEquals("dodavanje potomaka", ocekivaniRezultat,
       porudzbine.toString());
}
public class XMLEtiketa {
   private String imeEtikete;
   private List potomci = new ArrayList();
   public XMLEtiketa(String ime) {
      imeEtikete = ime;
   }
   public void add(XMLEtiketa potomak) {
      potomci.add(potomak);
```

```
    }
    public String toString() {
        String rezultat = new String();
        rezultat += "<" + imeEtikete + ">";
        Iterator it = potomci.iterator();
        while (it.hasNext()) {
            XMLEtiketa cvor = (XMLEtiketa) it.next();
            rezultat += cvor.toString();
        }
        rezultat += "</" + imeEtikete + ">";
        return rezultat;
    }
}
```

Такође, потребно је проширити сложену репрезентацију тако да подржава и XML атрибуте и њихове вредности:

```
public void testDrvoSaAtributimaIVrednostima() {
    String ocekivaniRezultat =
        "<porudzbine>" +
            "<porudzbina>" +
                "<stavka broj='12660' kolicina='1'>" +
                    "Kuca za psa" +
                "</stavka>" +
                "<stavka broj='54678' kolicina='1'>" +
                    "Hranilica za ptice" +
                "</stavka>" +
            "</porudzbina>" +
        "</porudzbine>";
    XMLEtiketa porudzbine = new XMLEtiketa("porudzbine");
    XMLEtiketa porudzbina = new XMLEtiketa("porudzbina");
    XMLEtiketa stavka1 = new XMLEtiketa("stavka");
    stavka1.dodajAtribut("broj", "12660");
    stavka1.dodajAtribut("kolicina", "1");
    stavka1.setValue("Kuca za psa");
    XMLEtiketa stavka2 = new XMLEtiketa("stavka");
    stavka2.dodajAtribut("broj", "54678");
    stavka2.dodajAtribut("kolicina", "1");
    stavka2.setValue("Hranilica za ptice");
    porudzbine.add(porudzbina);
```

```
   porudzbina.add(stavka1);
   porudzbina.add(stavka2);
   assertXMLEquals("atributi i njihove vrednosti",
       ocekivaniRezultat, porudzbine.toString());
}
public class XMLEtiketa {
   private String imeEtikete;
   private String vrednostEtikete = "";
   private String atributi = "";
   private List potomci = new ArrayList();
   public XMLEtiketa(String ime) {
      imeEtikete = ime;
   }
   public void add(XMLEtiketa potomak) {
      potomci.add(potomak);
   }
   public void dodajAtribut(String ime, String vrednost) {
      atributi += (" " + ime + "='" + vrednost + "'");
   }
   public void dodajVrednost(String vrednost) {
      vrednostEtikete = vrednost;
   }
   public String toString() {
      String rezultat = new String();
      rezultat += "<" + imeEtikete + atributi + ">";
      Iterator it = potomci.iterator();
      while (it.hasNext()) {
         XMLEtiketa cvor = (XMLEtiketa) it.next();
         rezultat += cvor.toString();
      }
      if (!vrednostEtikete.equals(""))
         rezultat += vrednostEtikete;

      rezultat += "</" + imeEtikete + ">";
      return rezultat;
   }
}
```

На крају, првобитни код који се користио за прављење дрвета замени се новим:

```
XMLEtiketa porudzbine = new XMLEtiketa("porudzbine");
XMLEtiketa porudzbina = new XMLEtiketa("porudzbina");
porudzbina.dodajAtribut("broj", "123");
porudzbine.add(porudzbina);
XMLEtiketa stavka = new XMLEtiketa("stavka");
stavka.dodajAtribut("broj", "x1786");
stavka.dodajVrednost("Vrata od kola");
porudzbina.add(stavka);
```

5.3 Енкапсулација сложеног кода помоћу Градитељ обрасца

Проблем: *Сложен код садржи превише детаља имплементације, те је при грађењу дрвета потребно обављати много послова - креирање, форматирање, додавање и уклањање чворова, водећи све време рачуна о томе да ли се користи добра логика*

Решење: *Енкапсулирати сложен код у једноставнију Градитељ класу*

5.3.1 Објашњење

Градитељска класа пружа једноставан и практичан интерфејс за баратање дрветима, сакривајући детаље о томе како се чворови повезују и који међукораци се морају извршити током грађења. Оба обрасца, Градитељ и Састав, могу се применити када се ради са, на пример, XML форматом.

Уколико се ради само Састав обрасцем, биће изложено превише детаља. Насупрот њима, XML Градитељи нуде пријатан интерфејс за креирање дрвета, док се сама класа ослања на образац Састав како би исто дрво приказала.

5.3.2 Потенцијални проблеми

1. **Сврха није јасна**: Када се гради дрво, важно је знати шта се додаје дрвету и на ком месту. Уколико постоји превише других, техничких детаља, може се десити да основни кораци не буду најјаснији. Градитељи, ‘пак, пружају много јаснији интерфејс.

2. **Понављање кода**: Сложен код за грађење дрвета је пун позива који служе за креирање нових чворова и додавања тих чворова дрвету. Градитељски код уклања ова понављања тако што сам барата прављењем и поједностављује чин додавања чворова.

3. **Недовољна једноставност**: Приступом Састав обрасца, неопходно је познавати шта, где и како се додаје дрвету. Након примене обрасца Градитељ, потребно је знати само шта и где се додаје, а сама класа се брине о остатку посла.

5.3.3 Практична реализација рефакторизације

1. Лоцирати сложен код који је потребно енкапсулирати.

2. Креирати Градитељ класу:

   - Дати јој приватан примерак променљиве коју креира изабрани сложени код.
   - Иницијализовати примерак у конструктору.
   - Креирати метод који враћа резултат - изграђено дрво

3. Креирати методе са јасном наменом унутар Градитељ класе, за сваки тип чвора који се додаје унутар сложеног кода.

4. Заменити сложени код позивима метода Градитељ класе.

5.3.4 Пример

Нека је дат сложени код из претходног поглавља:

```
XMLEtiketa porudzbine = new XMLEtiketa("porudzbine");
XMLEtiketa porudzbina = new XMLEtiketa("porudzbina");
porudzbina.dodajAtribut("broj", "123");
porudzbine.add(porudzbina);
XMLEtiketa stavka = new XMLEtiketa("stavka");
stavka.dodajAtribut("broj", "x1786");
stavka.dodajVrednost("Vrata od kola");
porudzbina.add(stavka);
```

Дефинише се XMLGraditelj класа, енкапсулира се првобитни слозхени код, иницијализује објекат, и имплементира toString() метод који приказује резултат изграђивања. Све се ово обавља из тест методе, обзиром да је посао пипав и подложан прављењу грешака:

```
public void testJedanCvor() {
   String ocekivaniRezultat =
      "<porudzbine>" +
      "</porudzbine>";
   XMLGraditelj graditelj = new XMLGraditelj("porudzbine");
   assertXMLEquals("drvo sa jednim cvorom", ocekivaniRezultat,
       graditelj.toString());
```

```
}
```

Сада Градитељ изгледа:

```
public class XMLGraditelj {
   private XMLEtiketa koren;
   public XMLGraditelj(String imeKorena) {
      koren = new XMLEtiketa(imeKorena);
   }
   public String toString() {
      return koren.toString();
   }
}
```

Следеће што је потребно урадити је креирати метод за сваки могући тип чвора који се додаје у дрво. У овом случају је то само објекат типа XMLEtiketa. Међутим, ови чворови се могу додавати на различите нацхине. За почетак, случај када се додају као непосредни потомци:

```
public void testDodajIspod() {
   String ocekivaniRezultat =
      "<porudzbine>" +
         "<porudzbina>" +
            "<stavka>" +
            "</stavka>" +
         "</porudzbina>" +
      "</porudzbine>";
   XMLGraditelj graditelj = new XMLGraditelj("porudzbine");
   graditelj.dodajIspod("porudzbina");
   graditelj.dodajIspod("stavka");
   assertXMLEquals("dodavanje ispod", ocekivaniRezultat,
       graditelj.toString());
}
```

Те се мора имплементирати метод dodajIspod(). Потребно је начининицијализују и ситне измене унутар XMLGraditelj класе:

```
public class XMLGraditelj {
   private XMLEtiketa koren;
   private XMLEtiketa tekuci;
   public XMLGraditelj(String imeKorena) {
```

```
      koren = new XMLEtiketa(imeKorena);
      tekuci = koren;
   }
   public void dodajIspod(String imePotomka) {
      XMLEtiketa potomak = new XMLEtiketa(imePotomka);
      tekuci.add(potomak);
      tekuci = potomak;
   }
   public String toString() {
      return koren.toString();
   }
}
```

Даље је потребно омогућити додавање чвора на истом нивоу (тзв. додавање рођака):

```
public void testDodajUIstomNivou() {
   String ocekivaniRezultat =
      "<porudzbine>" +
         "<porudzbina>" +
         "<stavka>" +
            "</stavka>" +
            "<stavka>" +
         "</stavka>" +
         "</porudzbina>" +
      "</porudzbine>";
   XMLGraditelj graditelj = new XMLGraditelj("porudzbine");
   graditelj.dodajIspod("porudzbina");
   graditelj.dodajIspod("stavka");
   graditelj.dodajPored("stavka");
   assertXMLEquals("dodavanje u istom nivou", ocekivaniRezultat,
       graditelj.toString());
}
public class XMLGraditelj {
   private XMLEtiketa koren;
   private XMLEtiketa tekuci;
   private XMLEtiketa roditelj;
   public XMLGraditelj(String imeKorena) {
      koren = new XMLEtiketa(imeKorena);
      tekuci = koren;
```

```
        roditelj = koren;
    }
    public void dodajIspod(String imePotomka) {
        XMLEtiketa potomak = new XMLEtiketa(imePotomka);
        tekuci.add(potomak);
        roditelj = tekuci;
        tekuci = potomak;
    }
    public void dodajPored(String imeRodjaka) {
        XMLEtiketa rodjak = new XMLEtiketa(imeRodjaka);
        roditelj.add(rodjak);
        tekuci = rodjak;
    }
    public String toString() {
        return koren.toString();
    }
}
```

Потребно је омогућити и додавање атрибута:

```
public void testDodajIspodSaAtributom() {
    String ocekivaniRezultat =
        "<porudzbine>" +
            "<porudzbina broj='12345' kolicina='2'>" +
            "</porudzbina>" +
        "</porudzbine>";
    graditelj = createBuilder("porudzbine");
    graditelj.dodajIspod("porudzbina");
    graditelj.dodajAtribut("broj", "12345");
    graditelj.dodajAtribut("kolicina", "2");
    assertXMLEquals("izgradjeni xml", ocekivaniRezultat,
        graditelj.toString());
}
public class XMLGraditelj...
public void dodajAtribut(String ime, String vrednost) {
    tekuci.dodajAtribut(ime, vrednost);
    }
}
```

И на крају се првобитни код замени XMLGraditelj класом.

```
XMLGraditelj porudzbine = new XMLGraditelj("porudzbine");
porudzbine.dodajIspod("porudzbina");
porudzbine.dodajAtribut("broj", "123");
porudzbine.dodajIspod("stavka");
porudzbine.dodajAtribut("broj", "x1786");
porudzbine.dodajVrednost("Vrata od kola");
```

Коначни код не садржи ни једну референцу ка, сада енкапсулираној, класи XMLEtiketa.

5.3.5 Оптимизација

Још један пример који демонстрира префињеност и једноставност обрасца Градитељ је оптимизација која се може извршити на наведеном коду. Његовим разматрањем, као и разматрањем контекста у којем се позива, нека је уочено да класа StringBuffer коју користи XMLGraditelj није нарочито оптимална из аспекта перформанси. Она се користи како би се могло приказати цело дрво, то јест на њу се, за сваки чвор, додаје његова репрезентација, да би на крају садржала целовиту репрезентацију. StringBuffer-у коришћеном у овом случају нису задата никаква ограничења по питању величине, што значи да се додавањем XML-а по аутоматизму увећава, све док не може садржати више података. Ово значи да се у позадини извршавају два задатка: проширивање објекта и реалокација меморије.
Решење је предвидети величину StringBuffer-а пре инстанцирања, како би одмах био инстанциран у тој величини, без потребе да се накнадно увећава.
Нека је први корак, опет, писање теста. Нека он генерише XML дрво позивајући XMLGraditelj, а онда пореди дужину резултата са дужином ниске коју враћа Градитељска како би се могле израчунати потребне дужине (параметри) за StringBuffer:

```
public void testVelicinaBafera() {
   String ocekivaniRezultat =
      "<porudzbine>" +
         "<porudzbina broj='123'>" +
         "</porudzbina>" +
      "</porudzbine>";
   graditelj = kreirajGraditelja("porudzbine");
```

```
   graditelj.dodajIspod("porudzbina");
   graditelj.dodajAtribut("broj", "123");
   int unetaVelicina = graditelj.toString().length();
   int izracunataVelicina = ((XMLGraditelj)
      graditelj).velicinaBafera();
   assertEquals("velicina bafera", unetaVelicina,
      izracunataVelicina);
}
```

Да би прошао овај тест, потребно је додати/ажурирати следеће атрибуте и методе:

```
public class XMLGraditelj {
   private int velicinaIzlaznogBafera;
   private static int DUZINA_NAZIVA_ETIKETE = 5;
   private static int DUZINA_NAZIVA_ATRIBUTA = 4;
   public void dodajAtribut(String ime, String vrednost) {
      // logika dodavanja atributa etiketi
      uvecajVelicinuBaferaDuzinomAtributa(ime, vrednost);
   }
   public void dodajIspod(String imePotomka) {
      // logika dodavanja etikete ispod druge etikete
      uvecajVelicinuBaferaDuzinomEtikete(imePotomka);
   }
   public void dodajPored(String imeRodjaka) {
      // logika dodavanja etikete u istom nivou gde je i tekuca
         etiketa
      uvecajVelicinuBaferaDuzinomEtikete(imeRodjaka);
   }
   public void dodajPoredRoditelja(String imeUjaka) {
      // logika dodavanja etikete u istom nivou gde se nalazi i
         roditelj tekuce etikete
      uvecajVelicinuBaferaDuzinomEtikete(imeUjaka);
   }
   public void dodajVrednost(String vrednost) {
      // logika dodavanja vrednosti etiketi
      uvecajVelicinuBaferaDuzinomVrednosti(vrednost);
   }
   public int velicinaBafera() {
      return velicinaIzlaznogBafera;
```

```
    }
    private void uvecajVelicinuBaferaDuzinomAtributa(String ime,
        String vrednost) {
        velicinaIzlaznogBafera += (ime.length() + vrednost.length() +
            DUZINA_NAZIVA_ATRIBUTA);
    }
    private void uvecajVelicinuBaferaDuzinomEtikete(String etiketa) {
        int velicinaOtvorenihIZatvorenihEtiketa = etiketa.length() *
            2;
        velicinaIzlaznogBafera +=
            (velicinaOtvorenihIZatvorenihEtiketa +
            DUZINA_NAZIVA_ETIKETE);
    }
    private void uvecajVelicinuBaferaDuzinomVrednosti(String
        vrednost) {
        velicinaIzlaznogBafera += vrednost.length();
    }
    protected void inicijalizuj(String imeKorena) {
        // logika inicijalizacije graditelja i korenskog cvora
        velicinaIzlaznogBafera = 0;
        uvecajVelicinuBaferaDuzinomEtikete(imeKorena);
    }
}
```

Начињене промене унутар XMLGraditelj класе нису видљиве изван ње. Једина додатна промена коју је потребно извршити је промена над toString() методом, како би се StringBuffer инстанцирао са жељеном величином, те био прослеђен корену дрвета, тако да покупи садржај XML дрвета.
Да би се ово урадило, потребно је метод:

```
public String toString() {
    return koren.toString();
}
```

написати као:

```
public String toString() {
    return koren.toStringHelper(new
        StringBuffer(velicinaIzlaznogBafera));
}
```

Тиме је ово рефакторисање завршено. Сада је XMLGraditelj знатно бржи.

5.4 Формирање над-интерфејса

Проблем: *Потребно је да наткласа имплементира исти интерфејс који имплементира поткласа*
Решење: *Покупити јавне методе који су специфични за поткласу и сместити их у наткласу, тако да постоје у њој, али не имплементирају никакву логику*

5.4.1 Објашњење

Ово рефакторисање је применљиво приликом прављења тзв. невидљивих омотача (енг. transparent enclosure). Декоратери су типични примери невидљивих омотача: они имплементирају исти интерфејс као и примерци које им се додају (поткласе). На пример, у Јави, UnmodifiableList је невидљиви омотач, јер он имплементира интерефејс List, а креира га објекат који такође имплементира интерфејс List. Тај објекат помоћу ког се креира UnmodifiableList је објекат који је обмотан. UnmodifiableList обмотава List објекат јер и он сам имплементира List.
Дакле, уколико метод очекује да му буде прослеђен аргумент типа List, може се проследити ArrayList (који имплементира List) или UnmodifiableList који обмотава ArrayList.
Ово рефакторисање је неопходно уколико се формира невидљиви омотач за поткласу, али не постоји адекватни интерефејс за то. У том интерфејсу све методе које позива поткласа морају бити јавне, укључујући и оне које су наслеђене од наткласе. Након креирања над-интерфејса унутар наткласе, може се извршити ,,Екстракција интерфејса" [2] како би се добио интерфејс потребан за фомирање невидљивог омотача.
Ово рефакторисање је саставни део рефакторисања ,,Пребацивање украшавања у Декоратер", наведеног у наредном поглављу.

5.4.2 Практична реализација рефакторизације

1. Уочити који метод недостаје. То је јавни метод поткласе који није декларисан у наткласи.

2. Додати копију тог метода у наткласи, такву да је тело метода празно (енг. null-behaviour).

3. Понављати кораке 1 и 2 све док наткласа и поткласа немају исти интерфејс.

5.4.3 Пример

Рефакторисање је приказано на коду библиотеке за HTML парсер, намењем употреби у програмском језику Јава (више о самој структури и начину рада у наредном поглављу). Нека је потребно написати невидљиви омотач за класу звану HTMLStringNode, која имплементира следеће јавне методе:

```
public class HTMLStringNode extends HTMLNode...
   public String toPlainTextString()...
   public String toHTML()...
   public String toString()...
   public void collectInto(Vector collectionVector, String
      filter)...
   public void accept(HTMLVisitor visitor)...
   public void decodeContents(boolean shouldDecodeContents)...
   public void removeEscapeCharacters(boolean
      shouldRemoveEscapeCharacters)...
```

HTMLStringNode наслеђује већину јавних метода из класе HTML-Node, осим:

```
public void decodeContents(boolean shouldDecodeContents)...
public void removeEscapeCharacters(boolean
    shouldRemoveEscapeCharacters)...
```

Примера ради, метод decodeContents() унутар HTMLStringNode имплементиран је на следећи начин:

```
public class HTMLStringNode extends HTMLNode...
   public void decodeContents(boolean shouldDecodeContents) {
      this.shouldDecodeContents = shouldDecodeContents;
}
```

Копира се метод decodeContents() у HTMLNode, тако да тело метода буде празно:

```
public class HTMLNode...
   public void decodeContents(boolean shouldDecodeContents) {
   }
```

Слично се уради и за removeEscapeCharacters.

Следећи корак је екстракција интерфејса из класе HTMLNode. Овај интерфејс се онда користи како би се направило невидљиви омотач за HTMLStringNode. У следећем рефакторисању је овај поступак објашњен до краја.

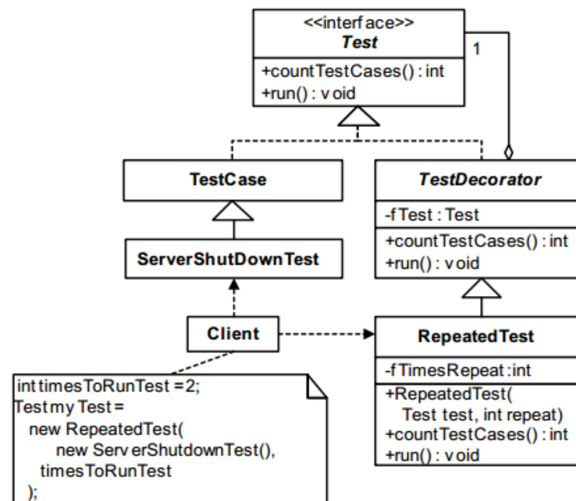
6 Рефакторизација код бихејворалних образаца дизајна

6.1 Пребацивања украшавања у Декоратер

Проблем: *Класа садржи код који динамички придружује објекту додатне одговорности (односно, украшава га)*
Решење: *Пребацити украшавање у Декоратер*

6.1.1 Објашњење

Обично се додавање нових функционалности систему врши тако што се постојећи код надогради новим. Каже се да нови код ,,украшава" првобитну класу. Међутим, овакав код усложњава класу - додају јој се нови атрибути и нове методе, при чему се они не користе увек, већ с времена на време, у неким могућим, новонасталим случајевима коришћења. Декоратор образац нуди добар компромис када настане овај случај. Свако украшавање смешта се у засебне класе, а онај основни објекат се ,,украси" новим својствима тј. функционалностима онда када се за тим јави потреба, при самом извршавању.
Примена Декоратер обрасца може се уочити у тзв. JUnit-у [5]. Овај алат омогућава једноставно писање и покретање тестова. Сваки тест је објекат типа TestCase. Уколико је потребно покренути исти тест више пута, сама класа TestCase не нуди такву могућност. За ову додатну функционалност потребно је украсити TestCase објекат RepeatedTest Декоратерем:



Ипак, уколико класа садржи неколико десетина јавних метода, не треба применити Декоратер образац при рефакторисању. Декоратер је сам по себи „невидљиви омотач” (видети претходно поглавље, енг. transparent enclosure), што значи да имплементира апсолутно све јавне методе које имплементира и објекат који украшава. Из таквог рефакторисања произишло би превише бескорисног кода. Невидљиви омотачи обмотавају класе на начин да изван њих самих не постоји свест о томе да су обмотане, с обзиром да имају идентичан интерфејс.

6.1.2 Потенцијални проблеми

1. **Сврха није јасна**: Неке делове кода напросто није потребно често извршавати. Међутим, уколико се ти делови нађу међу кодом који се нужно извршава, постаје нејасно који код је неопходан а који представља проширења могућности. Декоратер образац пружа могућност да се јасно представи шта је суштина објекта, а шта његове додатне функционалности и својства.

2. **Понављање кода**: Декоратери нуде добар начин да се манипулише са различитим украшавањима објекта, без понављања кода.

3. **Недовољна једноставност**: Код који комбинује неопходно и опционо није ни издалека једноставан као код који садржи само оно што је есенцијално за класу. С друге стране, ни сами Декоратери

нису увек једноставни за употребу, поготово ако је потребно бринути и о редоследу у ком се додају.

6.1.3 Практична реализација рефакторизације

1. Идентификовати украшену класу, тј. класу која поред основних функционалности садржи и нека опциона својства и методе.
   Водити рачуна о томе она не имплементира превише јавних метода. Уколико је то случај, како је Декоратер невидљиви омотач, и он ће сам имплементирати све те методе, те покушати са редукцијом метода: брисањем, премештањем, променом видљивости - или уопште не вршити ово рефакторисање.

2. Идентификовати или креирати класу за обмотавање. То је класа односно интерфејс у коме су декларисани сви јавни методи који су декларисани и унутар класе која се обмотава.
   Уколико класа за обмотавање не постоји, направити је применом рефакторисања наведеног у претходном поглављу, ,,Формирање над-интерфејса", након чега следи рефакторисање ,,Екстракције Интерфејса" [2].

3. Креирати Декоратер: класу која обмотава (проширује, украшава) класу за обмотавање. Назвати је у складу са тим на који начин украшава класу (на пример RepeatedTest је Декоратер који омогућава покретање тестова одређен број пута).
   Уколико је ово први Декоратер који постоји у систему, он не треба да буде апстрактан, већ конкретан. Декоратери треба да буду апстрактни уколико их има више, при чему имају заједничке методе које имплементирају.

4. Додати приватно или заштићено поље Декоратеру, звано делегат, које ће бити инстанца класе за обмотавање. Додати конструктор Декоратеру, који као параметар прихвата објекат за обмотавање, и поставља да делегат показује на овај објекат.
   Уколико је потребно, могу се проследити и додатни параметри који се користе при украшавању (на пример, repeat променљива у public RepeatedTest(Test test, int repeat)).

5. Позиве ка методама Декоратера преусмерити ка позивима метода делегата.

6. Следећи корак је додавање логике украшавања Декоратеру. Ово се може обавити применом „Екстракције методе" [2] како би се украшавање нашло у једној методи, копирањем тог метода у Декоратер класу, па позивањем копираног метода из метода Декоратера које морају да додају украшавање пре односно после позива делегату.

7. Ажурирати остатак кода тако да се сада односи на „украшени" објекат.

8. Уклонити украшавање из полазне класе, као и остатак кода који има везе са њим.

6.1.4 Пример

HTML парсер је слободна библиотека који омогућава прегледање садржаја HTML страница као скупа специјалних HTML објеката. Парсер функционише тако што етикете односно ниске угњеждене унутар етикета преведе у одговарајуће објекте, попут HTMLTag, HTMLStringNode, HTMLEndTag, HTMLImageTag итд.
Примена Декоратер обрасца у овом парсеру види се, на пример, у структури HTMLStringNode класе. Објекти ове класе инстанцирају се у току ижршавања кода, када парсер наиђе на текст угњежден између две етикете.
Садржај HTML објекта може се испитати на два начина. Први начин је коришћењем toPlainTextString() метода, који враћа чист текст. Други начин употреба методе toHTML(), која враћа HTML репрезентацију објекта. У оба случаја, за класу HTMLStringNode резултат ће бити исти.
Једно корисно украшавање класе HTMLStringNode је омогућавање декодирања нумеричких и словних референци. На пример, & у „ &", < у „«", > у „»" и сл.
Да би омогућио ову функционалност, парсер садржи класу Translate која у себи има метод decode().
Примера ради, нека је дат следећи код, који пролази кроз колекцију HTMLNodes објеката, и декодира & знак у чворовима који су типа HTMLStringNode:

```
public void testDekodiranjeAmpersenda() throws Exception {
```

```
  String NASLOV_RADIONICE = "<H1>The Testing &amp; Refactoring
     Workshop</H1>";

  String DEKODIRANI_NASLOV_RADIONICE = "<H1>The Testing &
     Refactoring Workshop</H1>";

  StringBuffer dekodiraniSadrzaj = new StringBuffer();
  HTMLParser parser = HTMLParser.createParser(NASLOV_RADIONICE);
  HTMLEnumeration cvorovi = parser.elements();
  while (cvorovi.hasMoreNodes()) {
    HTMLNode cvor = cvorovi.nextNode();
    if (cvor instanceof HTMLStringNode) {
      HTMLStringNode stringCvor = (HTMLStringNode) cvor;
      decodedContent.append(Translate.decode(stringCvor.toPlainTextString()));
         // dekodiranje
    }
    if (cvor instanceof HTMLTag)
       dekodiraniSadrzaj.append(cvor.toHTML());
  }
  assertEquals("dekodirani sadrzaj", DEKODIRANI_NASLOV_RADIONICE,
     dekodiraniSadrzaj.toString());
}
```

Пручавањем кода у којем се парсер користи, долази се до закључка да је декодирање чворова функционалност која је потребна с времена на време, али и да тада програмер сваког пута, цео поступак мора обавити сам: пролазак кроз чворове, провера да ли је чвор баш типа HTMLStringNode, декодирање, креирање ниске итд. Погодност у овом случају би била да се сам парсер бави овим пословима, тако да све што програмер треба да уради је да позове одговарајући метод парсера.
Први корак био би приступ без икаквог рефакторисања, већ додавањем кода који обавља декодирање - директно у HTMLStringNode класу. Важно је направити одговарајуће измене и у осталим класама које су повезане са HTMLStringNode, и ове промене су такође наведене.

Класа HTMLParser у себи садржи индикатор, да ли треба или не треба декодирати чворове:

```
public class HTMLParser...
```

```
    private boolean shouldDecodeStringNodes = false;
    public void setStringNodeDecoding(boolean
        shouldDecodeStringNodes) {
      this.shouldDecodeStringNodes = shouldDecodeStringNodes;
    }
    public boolean shouldDecodeStringNodes() {
      return shouldDecodeStringNodes;
    }
```

StringParser се прилагођава тако да провери вредност shouldDecode-StringNodes индикатора из HTMLParser-а:

```
public class StringParser...
    public HTMLNode find(HTMLReader reader, String input, int
        position)...
        HTMLStringNode stringNode = new HTMLStringNode(textBuffer,
            textBegin, textEnd);
        if (reader.getParser().shouldDecodeStringNodes())
            stringNode.decodeContents(true);
        return stringNode;
```

Коначно, HTMLStringNode класа се прилагођава тако да може да подржи украшавање:

```
public class HTMLStringNode...
    protected StringBuffer textBuffer;

    private boolean shouldDecodeContents = false;

    public void decodeContents(boolean shouldDecodeContents) {
      this.shouldDecodeContents = shouldDecodeContents;
    }
    public String toPlainTextString() {
      return nodeContents();
    }
    public String toHTML() {
      return nodeContents();
    }
    private String nodeContents() {
      String result = textBuffer.toString();
      if (shouldDecodeContents) result = Translate.decode(result);
```

```
    return result;
  }
```

Код са почетка примера, који пролази кроз чворове и декодира их, сада изгледа једноставније:

```
public void testDekodiranjeAmpersenda() throws Exception {

   String NASLOV_RADIONICE = "<H1>The Testing &amp; Refactoring
       Workshop</H1>";

   String DEKODIRANI_NASLOV_RADIONICE = "<H1>The Testing &
       Refactoring Workshop</H1>";

   StringBuffer dekodiraniSadrzaj = new StringBuffer();
   HTMLParser parser = HTMLParser.createParser(NASLOV_RADIONICE);
   parser.setStringNodeDecoding(true);
   HTMLEnumeration cvorovi = parser.elements();

   // dekodiranje
   while (corovi.hasMoreNodes())
     dekodiraniSadrzaj.append(cvorovi.nextNode().toHTML());

   assertEquals("dekodirani sadrzaj", DEKODIRANI_NASLOV_RADIONICE,
       dekodiraniSadrzaj.toString());
}
```

Како ово није једино украшавање парсера, добра пракса је поновити исти поступак и за остала украшавања (у овом случају је то декодирање симбола за нови ред и табулатор, \n и \t). И за ово декодирање додаје се адекватан индикатор HTMLParser класи:

```
public class HTMLStringNode...
  private boolean shouldRemoveEscapeCharacters = false;

  public String toPlainTextString() {
    return nodeContents();
  }

  public String toHTML() {
    return nodeContents();
```

```
}

private String nodeContents() {
  String result = textBuffer.toString();

  if (shouldDecodeContents)
     result = Translate.decode(result);
  if (shouldRemoveEscapeCharacters)
  result = HTMLParserUtils.removeEscapeCharacters(result);
  return result;
}
```

Друга страна медаље оваквог приступа је та што је за свако ново украшавање потребно додати нови индикатор, нове методе, конфигурацију итд, уопште нову ,, special-case" логику. У оваквој ситуацији је практичније поједноставити дизајн применом Декоратер обрасца. У остатку примера наведено је рефакторисање праћењем корака наведених под тачком ,,Практична реализација рефакторизације".
Прво је потребно уочити украшену класу, што је у овом случају HTMLStringNode. Важно је да та класа није ,,пренатрпана" атрибутима, индикаторима, методама, иначе би Декоратер образац само одмогао (што је образложено раније).
Следећи корак је креирање класе за обмотавање, тј. класе или интерфејса који декларише све јавне методе класе HTMLStringNode, као и остале јавне методе које та класа евентуално наслеђује. Та класа не треба да има атрибуте, већ само методе. Натткласа HTMLNode није добра класа за обмотавање, обзиром да садржи два атрибута, nodeBegin и nodeEnd. Разлог због којег класа за обмотавање не треба да садржи атрибуте је тај што она додаје додатне функционалности, дакле додаје неко понашање објекту који украшава, те она не треба да изнова дефинише већдефинисане атрибуте. У кораку 2 наведено је да, уколико класа за обмотавање не постоји, треба је направити. Она се може добити применом претходног рефакторисања, Формирањем над-интерфејса, тако што ће класу HTMLNode проширити јавним методама специфичним за класу HTMLStringNode:

```
public abstract class HTMLNode...

   public int elementBegin()...
```

```
    public int elementEnd()...
    public String toPlainTextString()...
    public String toHTML()...
    public String toString()...
    public void collectInto(Vector collectionVector, String
       filter)...
    public void accept(HTMLVisitor visitor)...

    public void decodeContents(boolean shouldDecodeContents)... //
       iz HTMLStringNode
    public void removeEscapeCharacters(boolean
       shouldRemoveEscapeCharacters)... // iz HTMLStringNode
```

Примени се метод ,,Екстракције интерфејса" [2] на HTMLNode како би се добила класа (односно интерфејс) за обмотавање. Адекватније име за ову класу би било управо HTMLNode, те се HTMLNode преименује у AbstractNode, а Декоратер класа назива се у складу са својом функционалношћу, DecodingStringNode:

```
public class DecodingStringNode implements HTMLNode {
   public String toPlainTextString() {
      return null;
   }
   public String toHTML() {
      return null;
   }
   public void collectInto(Vector collectionVector, String filter)
       {}
   public int elementBegin() {
      return 0;
   }
   public int elementEnd() {
      return 0;
   }
   public void accept(HTMLVisitor visitor) {}
   public void decodeContents(boolean shouldDecodeContents) {}
   public void removeEscapeCharacters(boolean
       shouldRemoveEscapeCharacters) {}
}
```

Потребно је DecodingStringNode класи додати поље у коме ће се налазити објекат који украшава (делегат), и које се поставља у конструктору:

```
public class DecodingStringNode implements HTMLNode...
   private HTMLNode delegate;

   public DecodingStringNode(HTMLNode delegate) {
      this.delegate = delegate;
   }
```

Сада је потребно позиве ка методама Декоратера преусмерити ка позивима метода делегата:

```
public class DecodingStringNode implements HTMLNode {

   private HTMLNode delegate;

   public DecodingStringNode(HTMLNode delegate) {
      this.delegate = delegate;
   }
   public String toPlainTextString() {
      return delegate.toPlainTextString();
   }
   public String toHTML() {
      return delegate.toHTML();
   }
   public void collectInto(Vector collectionVector, String filter) {
      delegate.collectInto(collectionVector, filter);
   }
   public int elementBegin() {
      return delegate.elementBegin();
   }
   public int elementEnd() {
      return delegate.elementEnd();
   }
   public void accept(HTMLVisitor visitor) {
      delegate.accept(visitor);
   }
   public void decodeContents(boolean shouldDecodeContents) {
      delegate.decodeContents(shouldDecodeContents);
```

```
  }
  public void removeEscapeCharacters(boolean
      shouldRemoveEscapeCharacters) {
    delegate.removeEscapeCharacters(shouldRemoveEscapeCharacters);
  }
}
```

Следећи корак је додавање логике украшавања HTMLStringNode класе Декоратер класи, DecodingStringNode.
Прво, применом „Екстракције методе” [2], код за украшавање смешта се у један метод:

```
public class HTMLStringNode extends AbstractNode...
public String toPlainTextString() {
  return nodeContents();
}
public String toHTML() {
  return nodeContents();
}
private String nodeContents() {
  String result = textBuffer.toString();
  if (shouldDecodeContents) result = decode(result); // poziv
      ekstraktovanog metoda
  if (shouldRemoveEscapeCharacters) result =
      HTMLParserUtils.removeEscapeCharacters(result);
  return result;
}
private String decode(String result) { // ekstraktovani metod
  return Translate.decode(result);
}
```

Онда се тај метод копира у DecodingStringNode класу:

```
public class DecodingStringNode implements HTMLNode...
  private String decode(String result) {
    return Translate.decode(result);
  }
```

Коначно, позива се метод decode() DecodingStringNode класе из метода које додавају украшавање након позива делегату: toPlainTextString() и toHTML():

```
public class DecodingStringNode implements HTMLNode...
   public String toPlainTextString() {
      return decode(delegate.toPlainTextString());
   }
   public String toHTML() {
      return decode(delegate.toHTML());
   }
```

Потребно је још ажурирати код који користи украшени објекат, тако што ће се сада односити на DecodingStringNode објекат. На пример, у StringParser класи постоји такав код:

```
public HTMLNode find(HTMLReader reader, String input, int
    position, boolean ignoreStateMode)......
   HTMLStringNode stringNode = new HTMLStringNode(textBuffer,
       textBegin, textEnd);
   if (reader.getParser().shouldDecodeStringNodes())
       stringNode.decodeContents(true);
   if (reader.getParser().shouldRemoveEscapeCharacters())
      stringNode.removeEscapeCharacters(true);
return stringNode;
```

Треба променити тип променљивој stringNode тако да је типа HTMLNode (инструкција 7), па у складу са тим и остале фрагменте кода који су се ослањали на претходни тип:

```
public HTMLNode find(HTMLReader reader, String input, int
    position, boolean ignoreStateMode)......
   HTMLNode stringNode = new HTMLStringNode(textBuffer, textBegin,
       textEnd);
   if (reader.getParser().shouldDecodeStringNodes()) {
      stringNode = new DecodingStringNode(stringNode);
      stringNode.decodeContents(true);
   }
   if (reader.getParser().shouldRemoveEscapeCharacters())
      stringNode.removeEscapeCharacters(true);
return stringNode;
```

На крају је још потребно уклонити код везан за декодирање који се пре користио, као и све што је имало везе са њим, у смислу конфигура-

ције:

```
public class HTMLStringNode extends AbstractNode...
    //private boolean shouldDecodeContents = false;
    private boolean shouldRemoveEscapeCharacters = false;
    public String toPlainTextString() {
      return nodeContents();
    }
    public String toHTML() {
      return nodeContents();
    }
    private String nodeContents() {
      String result = textBuffer.toString();
      //if (shouldDecodeContents) result = decode(result);
      if (shouldRemoveEscapeCharacters) result =
          HTMLParserUtils.removeEscapeCharacters(result);
      return result;
    }
   /*
    private String decode(String result) {
      return Translate.decode(result);
    }
    */
  /*
   public void decodeContents(boolean shouldDecodeContents) {
      this.shouldDecodeContents = shouldDecodeContents;
    }
    */
    public void removeEscapeCharacters(boolean
        shouldRemoveEscapeCharacters) {
      this.shouldRemoveEscapeCharacters =
          shouldRemoveEscapeCharacters;
    }
```

Наставити брисање све док се не изгуби сваки траг претходном дизајну. Овим је рефакторисање завршено.

6.2 Замена чврсто-кодираног слања обавештења Посматрач обрасцем

Проблем: *Класа или више њених поткласа шаљу обавештења другим објектима у одређеним тренуцима*
Решење: *Заменити код који служи за слање обавештења Посматрачем*

6.2.1 Објашњење

Као и за већину других рефакторисања, дато је корисно уколико се примени на правом месту, када је његова примена заиста и потребна. Аутори књиге ,, Design Patterns" [3] сугеришу употребу овог обрасца у следећим ситуацијама:

- Када апстракција има два аспекта од којих један зависи од другога. Енкапсулирање тих аспеката у засебне објекте омогућава њихово независно мењање и вишекратну употребу.
- Када промена у једном објекту захтева промене у другим објектима, а није познато колико других објеката треба променити.
- Када објекту треба омогућити да обавештава друге објекте без икакве претпоставке о томе који су то објекти. Другим речима, када се жели избећи да објекти буду тесно везани.

На пример, уколико у систему постоји нека класа А, која треба да обавести класу В уколико дође до неког догађаја, легитимно је имати код унутар класе А који то чини праволинијски. Међутим, уколико треба обавестити и класу С о неком догађају, ситуација се компликује и постаје могуће да је смислено применити Посматрач образац.
Позитиван одговор на било које од следећих питања, даје сигнал да треба применити рефакторисање наведено у даљем тексту:

- Да ли постоји понављање кода који служи за слање обавештења?
- Да ли су креиране релативно бесмислене поткласе како би се систем прилагодио новим обавештењима?
- Да ли се логика за слање обавештења превише усложњава?

- Да ли је смислено проследити цео објекат класи А само да би се прослеђеном објекту послало обавештење?

6.2.2 Потенцијални проблеми

1. **Сврха није јасна**: Чврсто-кодирана обавештења обављају свој посао, али се то у коду не види јасно, јер обавештења шаљу у фази извршавања кода. Уколико се обавештења шаљу уз помоћ Посматрач обрасца, информација о томе ко шаље односно ко прима обавештења је видљива у самој декларацији класа.

2. **Понављање кода**: Уколико се за сваку класу која треба да прима обавештења пише изнова код, велика је вероватноћа да ће исти код бити непотребно понављан.

3. **Недовољна једноставност**: Слање неколицине различитих типова обавештења може бити чврсто кодирано. Међутим, уколико број различитих догађаја односно објеката које треба обавештавати расте, настају нове поткласе како би се подржали нови захтеви. Поједноставити подршку новим захтевима увођењем Посматрач обрасца.

6.2.3 Практична реализација рефакторизације

1. Идентификовати субјекат: класу која прихвата објекат и садржи чврсто-кодиране инструкције за слање обавештења.

2. Дефинисати Посматрача: интерфејс у коме је декларисан скуп јавних метода које позива субјекат када шаље обавештења објекту.

3. Додати субјект-класи листу Посматрача и могућност додавања елемената тој листи одговарајућим методом. Додати и метод за уклањање Посматрача само уколико је могуће обуставити слање обавештења.

4. Код из субјект-класе који је прихватао објекат и директно му слао обавештења заменити кодом који пролази кроз листу Посматрача, ажурирајући сваки елемент те листе.

5. Свакој класи којој треба слати обавештења додати имплементацију интерфејса Посматрача.

6. Заменити код који је прослеђивао објекат субјекту са кодом који додаје новог Посматрача листи која се налази у субјекту.

6.2.4 Пример

Код који је у даљем тексту разматран потиче из JUnit [5] оквира за тестирање. У верзији JUnit 3.x, аутори су дефинисали специфичне поткласе класе TestResult (попут UITestResult, SwingTestResult, TextTestResult) чија је одговорност да скупљају информације добијане током тестирања и прослеђују их TestRunner објектима. Свака TestResult поткласа била је спренуга са одговарајућим TestRunner објектом (попут AWT TestRunner, Swing TestRunner, Text-based TestRunner). Током извршавања кода, након креирања поткласе класе TestResult, TestRunner прослеђује сам себе том TestResult објекту, и чека да му TestResult објекат пошаље обавештење. Свака поткласа TestResult-а је на овај начин чврсто кодирана да комуницира са одговарајућим TestRunner-ом. Одговор на прво питање (Да ли постоји понављање кода који служи за слање обавештења?) је позитиван, те је прави тренутак за примену Посматрач обрасца.
Прво је потребно пронаћи субјекат. У овом случају нека то буде UITestResult класа, али како ће она бити елеминисана (јер је њена једина улога да директно комуницира са AWT TestRunner класом), касније ће то постати TestResult класа.

```
package junit.framework;

public class TestResult extends Object {

   protected Vector fFailures;

   public TestResult() {
      fFailures = new Vector(10);
   }
   public synchronized void addFailure(Test test, Throwable t) {
      fFailures.addElement(new TestFailure(test, t));
   }
   public synchronized Enumeration failures() {
      return fFailures.elements();
```

```
  }
  protected void run(TestCase test) {
    startTest(test);
    try {
      test.runBare();
    } catch (AssertionFailedError e) {
      addFailure(test, e);
    }
    endTest(test);
  }
}
```

UITestResult класа (истакнуте су линије у којима се види спрегнутост између UITestResult и AWT TestRunner):

```
package junit.ui;

class UITestResult extends TestResult {

  private TestRunner fRunner; // ovde

  UITestResult(TestRunner runner) {
    fRunner = runner; // ovde
  }
  public synchronized void addFailure(Test test, Throwable t) {
    super.addFailure(test, t);
    fRunner.addFailure(this, test, t); // ovde
  }
  ...
}
```

TestRunner класа:

```
package junit.ui;

public class TestRunner extends Frame {

  private TestResult fTestResult;
  ...
  protected TestResult createTestResult(TestRunner runner) {
    return new UITestResult(TestRunner.this); // ovde
```

```
   }
   synchronized public void runSuite() {...
      fTestResult = createTestResult(TestRunner.this);
      testSuite.run(fTestResult);
   }
   public void addFailure(TestResult result, Test test, Throwable
       t) {
      fNumberOfFailures.setText(Integer.toString(result.testFailures()));
      appendFailure("Failure", test, t);
   }
}
```

Дефинисање интерфејса Посматрача (званог TestListener):

```
package junit.framework;

   public interface TestListener {
      public void addError(Test test, Throwable t);
      public void addFailure(Test test, Throwable t);
      public void endTest(Test test);
      public void startTest(Test test);
   }
```

Додавање листе Посматрача субјекту и омогућавање класама које имплементирају интерфејс Посматрача да себе додају у листу. Ово се обавља над TestResult класом како би било лакше касније елиминисати UITestResult поткласу:

```
public class TestResult extends Object {
   protected Vector fFailures;
   protected Vector fListeners;
   public TestResult() {
      fFailures = new Vector();
      fListeners = new Vector();
   }
   public synchronized void addListener(TestListener listener) {
      fListeners.addElement(listener);
   }
}
```

Сада треба ажурирати листу када се деси одређен догађај, што пред-

стављa рефакторисање логике метода из интерфејса Посматрача, addFailure(), addError() итд. Због једноставности, приказано је само рефакторисање addFailure() методе.
Метод addFailure() првобитно је изгледао на следећи начин:

```
class UITestResult. . .
   public synchronized void addFailure(Test test, Throwable t) {
      super.addFailure(test, t);
      fRunner.addFailure(this, test, t);
   }
```

addFailure() метод из TestResult класе неће мењати своју функционалност, али ће сада пролазити кроз листу Посматрча и позивати њихове addFailure() методе:

```
class TestResult. . .
   public synchronized void addFailure(Test test,
       AssertionFailedError t) {
      fFailures.addElement(new TestFailure(test, t));
      for (Enumeration e= cloneListeners().elements();
          e.hasMoreElements(); ) {
         ((TestListener)e.nextElement()).addFailure(test, t);
      }
   }
```

Коначно, да би AWT TestRunner могао да себе региструје као Посматрача TestResult класе и дода у поменуту листу, мора имплементирати интерфејс Посматрача:

```
package junit.ui;

public class TestRunner extends Object implements TestListener . .
    .
```

Коначно, још треба регистровати објекат који жели да прима обавештења:

```
package junit.ui;

public class TestRunner extends Object implements TestListener {
   private Vector fFailedTests;
```

```
    private TestResult fTestResult;
    protected TestResult createTestResult() {
        return new TestResult();
    }
    synchronized public void runSuite() {
        ...
        fTestResult = createTestResult();
        fTestResult.addListener(TestRunner.this); // registracija kod
            Posmatracha
        ...
    }
}
```

Сличан поступак примени се и на остале поткласе.

6.3 Замена условних претрага Спецификацијом

Проблем: *Користи се условна логика за претрагу објеката*
Решење: *Претраживати користећи образац Спецификације*

6.3.1 Објашњење

Спецификатор је објекат који садржи правило о другом објекту, који ће на даље бити зван *ентитет*. Када је потребно одабрати специфичан објекат из неког скупа објеката, креира се Спецификатор ком се наведу жељени критеријуми које треба да задовољи извучени објекат.
Образац Спецификације је згодно користити када расте број могућих комбинација по којима се објекти претражују. На пример, нека постоји класа PronalazacProizvoda који претражује објекте типа Proizvod на различите начине:

```
PronalazacProizvoda
  public List premaID(String proizvodID)
  public List premaBoji(Color bojaProizvodaZaPretrazivanje)
  public List premaCeni(float zeljenaCena)
  public List premaVelicini(int velicinaProizvodaZaPretrazivanje)
  public List ispodCene(float cena)
```

Временом број комбинација се увећава:

```
PronalazacProizvoda
  public List premaBojiIIspodCene(Color boja, float cena)
  public List premaBojiVeliciniIIspodCene(Color boja, int
     velicina, float cena)
  public List premaBojiIIznadCene(Color boja, float cena)
```

Образац Спецификације је направљен како би управо овакав проблем био решен. Идеја је да се услови претраге сами претворе у објекте како би могли бити комбиновати на произвољан начин. Овај приступ прилично подсећа на SQL упите, али он то није. Спецификација је концепт независан од било каквих база података.

6.3.2 Потенцијални проблеми

1. **Сврха није јасна**: Уколико је логика одабира ентита енкапсулирана у методама које јасно одају своју намену, и сам код постаје јаснији.

2. **Понављање кода**: Када је потребно одабрати ентитет из скупа ентитета на различите начине, постоји велика могућност да многе од тих претрага наликују једна другој, те да има и доста поновљеног кода. Образац Спецификације омогућава елиминацију понављања тако што даје могућност да критеријуми постоје независно и могу се комбиновати.

3. **Недовољна једноставност**: Уколико не постоји могућност комбиновања различитих критеријума при одабиру ентитета, овај образац може непотребно закомпликовати ствар, те је у том случају боље користити тривијалну логику одабира ентитета.

6.3.3 Практична реализација рефакторизације

1. Креирати апстрактну класу која ће се користити као Спецификатор. Назвати је тако да у себи садржи назив ентитета који одабира, као и реч „ Specifikacija” (на пример, ProizvodSpecifikacija).

2. Декларисати zadovoljavaNaOsnovu() метод у Спецификаторској класи, такав да враћа вредност типа boolean и прихвата један аргумент: тип ентитета који се извлачи из неког скупа.

3. Креирати odaberiPrema(...) метод који узима Спецификаторску класу као аргумент, и враћа колекцију ентитета. Њега сместити унутар класе која се понаша као репозиторијум тих ентитета.

4. Пронаћи логику одабира ентитета (што је обично итерирање кроз неку колекцију), такво да се ослања на један критеријум (попут proizvod.getCena() == unetiIznos) и за тај критеријум конкретизовати Спецификаторску класу, и назвати је по њему (на пример, ProizvodCenaSpecifikacija). Имплементирати метод zadovoljavaNaOsnovu() тако да враћа true или false, у складу са вредношћу прослеђеног ентитета.

5. Заменити критеријум и одговарајућу логику одабира ентитета позивом odaberiPrema(...) метода, прослеђујући му новокреирани Спецификатор.

6. Понављати кораке 4 и 5 при свакој појави логике одабира ентитета, за одговарајући критеријум.

7. Уколико је могуће, применити образац ,,Замене имплицитног формирања дрвета сложеним”. Ово омогућава комбиновање више критеријума унутар једне сложене Спецификације, употребом логичких операција као што су and(), or() и not().

8. Поновити корак 7 за све преостале критеријуме.

9. Применити образац ,,Енкапсулације класа методама креирања” имплементирањем метода креирања унутар Спецификаторске класе за сваку конкретну Спецификацију.

6.3.4 Пример

Следећи пример расветљује начин на који се ово рефакторисање примењује.
Нека је дата класа SkladisteProizvoda која садржи различите објекте типа Proizvod, и класа PronalazacProizvoda која познаје SkladisteProizvoda:

```
public class PronalazacProizvodaTest extends TestCase...

    private PronalazacProizvoda pronalazac;

    private Proizvod vatrogasniKamion = new Proizvod("f1234",
        "Vatrogasni kamion",
    Color.red, 8.95f, VelicinaProizvoda.SREDNJE);

    private Proizvod barbika = new Proizvod("b7654", "Barbika",
    Color.yellow, 15.95f, VelicinaProizvoda.MALO);

    private Proizvod frizbi = new Proizvod("f4321", "Frizbi",
    Color.pink, 9.99f, VelicinaProizvoda.VELIKO);
```

```
  private Proizvod palicaZaBejzbol = new Proizvod("b2343",
      "Palica za bejzbol",
  Color.white, 8.95f, VelicinaProizvoda.NEPRIMENLJIVO);

  private Proizvod igrackaKabriolet = new Proizvod("p1112",
      "Porshe kabriolet igracka",
  Color.red, 230.00f, VelicinaProizvoda.NEPRIMENLJIVO);

  protected void kreirajIPopuni() {
    pronalazac = new
        PronalazacProizvoda(napraviSkladisteProizvoda());
  }
 ...
  private SkladisteProizvoda napraviSkladisteProizvoda() {
    SkladisteProizvoda skladiste = new SkladisteProizvoda();
    skladiste.add(vatrogasniKamion);
    skladiste.add(barbika);
    skladiste.add(frizbi);
    skladiste.add(palicaZaBejzbol);
    skladiste.add(igrackaKabriolet);
    return skladiste;
  }
```

Прво је потребно добро разумети како се код користи, те се разматрају тест примери. testPretrazivanjePremaBoji проверава да ли метод PronalazacProizvoda.premaBoji проналази црвене играчке, док testPretrazivanjePremaCeni проверава да ли PronalazacProizvoda.premaCeni проналази играчке са наведеном ценом:

```
public class PronalazacProizvodaTest extends TestCase...

public void testPretrazivanjePremaBoji() {
   List pronadjeniProizvodi = pronalazac.premaBoji(Color.red);
   assertEquals("pronadjena 2 crvena proizvoda", 2,
       pronadjeniProizvodi.velicina());
   assertTrue("pronadjen vatrogasni kamion",
       pronadjeniProizvodi.contains(vatrogasniKamion));
   assertTrue(
      "pronadjen porshe kabriolet igracka",
   pronadjeniProizvodi.contains(igrackaKabriolet));
```

```
}
public void testPretrazivanjePremaCeni() {
   List pronadjeniProizvodi = pronalazac.premaCeni(8.95f);
   assertEquals("pronadjen proizvod koji kosta $8.95", 2,
       pronadjeniProizvodi.velicina());
   for (Iterator i = pronadjeniProizvodi.iterator(); i.hasNext();) {
      Proizvod p = (Proizvod) i.next();
      assertTrue(p.getCena() == 8.95f);
   }
}
```

И код који задовољава ове тестове:

```
public class PronalazacProizvoda...

private SkladisteProizvoda skladiste;
public PronalazacProizvoda(SkladisteProizvoda skladiste) {
   this.skladiste = skladiste;
}
public List premaBoji(Color bojaProizvodaZaPretrazivanje) {
   List pronadjeniProizvodi = new ArrayList();
   Iterator proizvodi = skladiste.iterator();
   while (proizvodi.hasNext()) {
      Proizvod proizvod = (Proizvod) proizvodi.next();
      if (proizvod.getBoja().equals(bojaProizvodaZaPretrazivanje))
          pronadjeniProizvodi.add(proizvod);
   }
   return pronadjeniProizvodi;
}
public List premaCeni(float zeljenaCena) {
   List pronadjeniProizvodi = new ArrayList();
   Iterator proizvodi = skladiste.iterator();
   while (proizvodi.hasNext()) {
      Proizvod proizvod = (Proizvod) proizvodi.next();
      if (proizvod.getCena() == zeljenaCena)
          pronadjeniProizvodi.add(proizvod);
   }
   return pronadjeniProizvodi;
}
```

Као што се може приметити, постоји пуно поновљеног кода у два горе наведена метода. Током овог рефакторисања, она бивају елиминисана. Како би се оно до краја извело и испратило, потребно је размотрити још тестова. Један проналази Proizvod објекте на основу боје, величине са ценом испод задате, док други претражује на основу боје и тако да им је цена изнад задате:

```
public class PronalazacProizvodaTest extends TestCase...

public void testPretrazivanjePremaBojiVeliciniIIspodCene() {
   List pronadjeniProizvodi =
      pronalazac.premaBojiVeliciniIIspodCene(
   Color.red,
   VelicinaProizvoda.MALO, 10.00f);
   assertEquals(
     "nije pronadjen mali crveni proizvod cene ispod $10.00",
   0,
   pronadjeniProizvodi.velicina());
   pronadjeniProizvodi = pronalazac.premaBojiVeliciniIIspodCene(
   Color.red,
   VelicinaProizvoda.SREDNJE,
   10.00f);
   assertEquals(
     "pronadjen vatrogasni kamion pri pretrazi jeftinih crvenih
        igracaka srednje velicine",
   vatrogasniKamion,
   pronadjeniProizvodi.get(0));
}
public void testPretrazivanjeIspodCeneBojeOsim() {
   List pronadjeniProizvodi = pronalazac.ispodCeneBojeOsim(9.00f,
      Color.white);
   assertEquals(
     "pronadjen 1 proizvod koji nije beo i kosta < $9.00",
   1,
   pronadjeniProizvodi.velicina());
   assertTrue("pronadjen vatrogasni kamion",
      pronadjeniProizvodi.contains(vatrogasniKamion));
   pronadjeniProizvodi = pronalazac.ispodCeneBojeOsim(9.00f,
      Color.red);
   assertEquals(
```

```
      "pronadjen 1 proizvod koji nije crven i kosta < $9.00",
   1,
   pronadjeniProizvodi.velicina());
   assertTrue("pronadjena palica za bejzbol",
       pronadjeniProizvodi.contains(palicaZaBejzbol));
}
```

И код који задовољава ове тестове:

```
public class PronalazacProizvoda...

public List premaBojiVeliciniIIspodCene(Color boja, int velicina,
    float cena) {
   List pronadjeniProizvodi = new ArrayList();
   Iterator proizvodi = skladiste.iterator();
   while (proizvodi.hasNext()) {
      Proizvod proizvod = (Proizvod) proizvodi.next();
      if (proizvod.getBoja() == boja && proizvod.getVelicina() ==
          velicina && proizvod.getCena() < cena)
          pronadjeniProizvodi.add(proizvod);
   }
   return pronadjeniProizvodi;
}
public List ispodCeneBojeOsim(float cena, Color boja) {
   List pronadjeniProizvodi = new ArrayList();
   Iterator proizvodi = skladiste.iterator();
   while (proizvodi.hasNext()) {
      Proizvod proizvod = (Proizvod) proizvodi.next();
      if (proizvod.getCena() < cena && proizvod.getBoja() != boja)
          pronadjeniProizvodi.add(proizvod);
   }
   return pronadjeniProizvodi;
}
```

Опет, дати код садржи доста понављања, с обзиром да сваки од метода итерира кроз исти скуп објеката и одабира само оне који задовољавају дати критеријум. Сада је могуће почети са рефакторисањем.
Прво је потребно креирати апстрактну класу ProizvodSpecifikacija:

```
public abstract class ProizvodSpecifikacija {
}
```

Додавање метода zadovoljavaNaOsnovu() класи ProizvodSpecifikacija:

```
public abstract class ProizvodSpecifikacija {
   public abstract boolean zadovoljavaNaOsnovu(Proizvod proizvod);
}
```

Креирање odaberiPrema(...) метода, који узима објекат типа ProizvodSpecifikacija и враћа колекцију објеката типа Proizvod који задовољавају дату Спецификацију (Јавин тип List). Овај метод се додаје класи SkladisteProizvoda. Друга могућност би била додавање класи PronalazacProizvoda, али како се из подсекције „Практична реализација рефакторизације” види, она на крају неће ни постојати.

```
public class SkladisteProizvoda...

public Iterator iterator() {
   return proizvodi.iterator();
}
public List odaberiPrema(ProizvodSpecifikacija
    proizvodSpecifikacija) {
   List pronadjeniProizvodi = new ArrayList();
   Iterator proizvodi = iterator();
   while (proizvodi.hasNext()) {
      Proizvod proizvod = (Proizvod) proizvodi.next();
      if (proizvodSpecifikacija.zadovoljavaNaOsnovu(proizvod))
          pronadjeniProizvodi.add(proizvod);
   }
   return pronadjeniProizvodi;
}
```

Сада је могуће конкретизовати ProizvodSpecifikacija, на пример, за производе са датом бојом:

```
public class ProizvodBojaSpecifikacija extends
    ProizvodSpecifikacija {
   private Color bojaProizvodaZaPretrazivanje;
   public ProizvodBojaSpecifikacija(Color
       bojaProizvodaZaPretrazivanje) {
      this.bojaProizvodaZaPretrazivanje =
```

```
        bojaProizvodaZaPretrazivanje;
  }
  public boolean zadovoljavaNaOsnovu(Proizvod proizvod) {
    return
        proizvod.getBoja().equals(bojaProizvodaZaPretrazivanje);
  }
}
```

Тело PronalazacProizvoda.premaBoji(...) методе сада се може заменити позивом новог PronalazacProizvoda.odaberiPrema(...) метода, прослеђујући му новокреирани објекат типа ProizvodBojaSpecifikacija:

```
public class PronalazacProizvoda...
public List premaBoji(Color bojaProizvodaZaPretrazivanje) {
  return odaberiPrema(new
      ProizvodBojaSpecifikacija(bojaProizvodaZaPretrazivanje));
/*
  List pronadjeniProizvodi = new ArrayList();
  Iterator proizvodi = skladiste.iterator();
  while (proizvodi.hasNext()) {
    Proizvod proizvod = (Proizvod) proizvodi.next();
    if (proizvod.getBoja().equals(bojaProizvodaZaPretrazivanje))
        pronadjeniProizvodi.add(proizvod);
  }
  return pronadjeniProizvodi;
*/
}
```

Како метод premaBoji(...) садржи само једну линију кода, он не мора постојати, већ се његово тело може директно извршавати, што је тзв. „Линијски метод” [2]:

```
public class PronalazacProizvoda...
/*
public List premaBoji(Color bojaProizvodaZaPretrazivanje) {
  return odaberiPrema(new
      ProizvodBojaSpecifikacija(bojaProizvodaZaPretrazivanje));
}
*/

public class PronalazacProizvodaTest extends TestCase...
```

```
    public void testPretrazivanjePremaBoji()...
//  List pronadjeniProizvodi = pronalazac.premaBoji(Color.red);
    List pronadjeniProizvodi = skladiste.odaberiPrema(
        new ProizvodBojaSpecifikacija(Color.red));
```

Понавља се претходни поступак за сваку појаву логике одабира ентитета која се заснива на претрази по неком критеријуму, што проузрокује конкретизацију следећих класа за одговарајуће критеријуме:

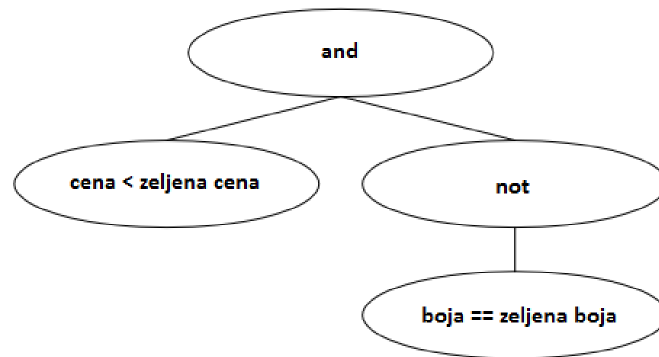
id	ProizvodIdSpecifikacija
velicina	VelicinaProizvodaSpecifikacija
cena	ProizvodCenaSpecifikacija
ispod cene	ProizvodIznadCeneSpecifikacija
iznad cene	ProizvodIznadCeneSpecifikacija

Још увек се не имплементирају комбинације критеријума.
Пре рефакторисања, претрага на основу више критеријума обавља се на следећи начин:

```
public class PronalazacProizvoda...
public List ispodCeneBojeOsim(float cena, Color boja) {
   List pronadjeniProizvodi = new ArrayList();
   Iterator proizvodi = skladiste.iterator();
   while (proizvodi.hasNext()) {
      Proizvod proizvod = (Proizvod) proizvodi.next();
      if (proizvod.getCena() < cena && proizvod.getBoja() != boja)
         pronadjeniProizvodi.add(proizvod);
   }
   return pronadjeniProizvodi;
}
```

Дакле, постоје два услова: proizvod.getCena $<$ cena и proizvod.getBoja() != boja. Класа ProizvodIznadCeneSpecifikacija је конкретизација типа ProizvodSpecifikacija која помаже при проналажењу објеката са ценом испод дате. Слично, ProizvodBojaSpecifikacija омогућава претрагу на основу дате боје. У овом тренутку потребно је имати сложену Спецификацију која ће омогућити комбиновање две конкретне Спецификације, негирајући услов боје. Ово рефакторисање је већописано (Замена имплицитног формирања дрвета Саставом).
Поставља се питање шта је тачно имплицитно дрво у овом случају. Кри-

теријуми цене и боје формирају следећу структуру дрвета:



Како би се ово рефакторисање извело, потребно је креирати две слозхене Спецификаторске класе, AndProizvodSpecifikacija и NotProizvodSpecifikacija:

```
public class AndProizvodSpecifikacija extends
    ProizvodSpecifikacija {
   private ProizvodSpecifikacija prvaSpecifikacija,
       drugaSpecifikacija;
   public AndProizvodSpecifikacija(
   ProizvodSpecifikacija prvaSpecifikacija, ProizvodSpecifikacija
       drugaSpecifikacija) {
      this.prvaSpecifikacija = prvaSpecifikacija;
      this.drugaSpecifikacija = drugaSpecifikacija;
   }
   public boolean zadovoljavaNaOsnovu(Proizvod proizvod) {
      return prvaSpecifikacija.zadovoljavaNaOsnovu(proizvod) &&
         drugaSpecifikacija.zadovoljavaNaOsnovu(proizvod);
   }
}
public class NotProizvodSpecifikacija extends
    ProizvodSpecifikacija {
   private ProizvodSpecifikacija uslovZaNegiranje;
   public NotProizvodSpecifikacija(
   ProizvodSpecifikacija uslovZaNegiranje) {
      this.uslovZaNegiranje = uslovZaNegiranje;
   }
```

```
    public boolean zadovoljavaNaOsnovu(Proizvod proizvod) {
        return !uslovZaNegiranje.zadovoljavaNaOsnovu(proizvod);
    }
}
```

Са постојањем ових класа, могуће је применити корак 5, који подразумева измену имплементације PronalazacProizvoda.ispodCeneBojeOsim(...) тако да користи сложену Спецификацију:

```
public class PronalazacProizvoda...
public List ispodCeneBojeOsim(float cena, Color boja) {
    ProizvodSpecifikacija kompletnaSpecifikacija = new
        AndProizvodSpecifikacija(
    new ProizvodIznadCeneSpecifikacija(cena),
    new NotProizvodSpecifikacija(
    new ProizvodBojaSpecifikacija(boja)));
    return skladiste.odaberiPrema(kompletnaSpecifikacija);
}
```

Наведени код демонстрира још примењених измена, као и примену „Линијског Метода" [2]:

```
public class PronalazacProizvodaTest extends TestCase...
public void testPretrazivanjeIspodCeneBojeOsim() {
//  List pronadjeniProizvodi = pronalazac.ispodCeneBojeOsim(9.00f,
    Color.white);
    ProizvodSpecifikacija specifikacija = new
        AndProizvodSpecifikacija(
    new ProizvodIznadCeneSpecifikacija(9.00f),
    new NotProizvodSpecifikacija(
    new ProizvodBojaSpecifikacija(Color.white)));
    List pronadjeniProizvodi = skladiste.odaberiPrema(specifikacija);
    assertEquals(
        "pronadjen 1 proizvod koji nije beo i kosta < $9.00",
    1,
    pronadjeniProizvodi.velicina());
    assertTrue("pronadjen vatrogasni kamion",
        pronadjeniProizvodi.contains(vatrogasniKamion));
// pronadjeniProizvodi = pronalazac.ispodCeneBojeOsim(9.00f,
    Color.red);
    specifikacija = new AndProizvodSpecifikacija(
```

```
    new ProizvodIznadCeneSpecifikacija(9.00f),
    new NotProizvodSpecifikacija(
    new ProizvodBojaSpecifikacija(Color.red)));
    pronadjeniProizvodi = skladiste.odaberiPrema(specifikacija);
    assertEquals(
        "pronadjen 1 proizvod koji nije crven i kosta < $9.00",
    1,
    pronadjeniProizvodi.velicina());
    assertTrue("pronadjena palica za bejzbol",
        pronadjeniProizvodi.contains(palicaZaBejzbol));
}
```

Понавља се корак 7 за све преостале критеријуме на основу којих се претражују производи. На крају, класа PronalazacProizvoda не садржи ниједан метод, те може бити уклоњена.
Коначно, да би се рефакторисање привело крају, потребно је применити образац „Енкапсулације класа методама креирања". Класе које је потребно енкапсулирати су јавни, конкретизовани Спецификатори, такви да сви наслеђују ProizvodSpecifikacija и имплементирају њен јавни метод, zadovoljavaNaOsnovu().
На пример, пре рефакторисања, класа ProizvodBojaSpecifikacija изгледа:

```
public class ProizvodBojaSpecifikacija extends
    ProizvodSpecifikacija {
    private Color bojaProizvodaZaPretrazivanje;
    public ProizvodBojaSpecifikacija(Color
        bojaProizvodaZaPretrazivanje) {
        this.bojaProizvodaZaPretrazivanje =
            bojaProizvodaZaPretrazivanje;
    }
    public boolean zadovoljavaNaOsnovu(Proizvod proizvod) {
        return
            proizvod.getBoja().equals(bojaProizvodaZaPretrazivanje);
    }
}
```

Њу је могуће енкапсулирати на следећи начин:

```
public abstract class ProizvodSpecifikacija...
```

```
public static ProizvodSpecifikacija premaBoji(final Color boja) {
   return new ProizvodSpecifikacija() {
      public boolean zadovoljavaNaOsnovu(Proizvod proizvod) {
         return proizvod.getBoja().equals(boja);
      }
   };
}
```

Више није потребно имати јавни конструктор, с обзиром да јој се прослеђују сви потребни подаци да би обавила претрагу.
Код који је директно инстанцирао ProizvodColorSpecitication сада је измењен:

```
public void testPretrazivanjePremaBoji()...
   List pronadjeniProizvodi = skladiste.odaberiPrema(
   // new ProizvodBojaSpecifikacija(Color.red));
      ProizvodSpecifikacija.premaBoji(Color.red));
```

Исто рефакторисање примени се за енкапсулацију два сложена Спецификатора из корака 7, and() и not():

```
public abstract class ProizvodSpecifikacija...
public ProizvodSpecifikacija and(
final ProizvodSpecifikacija drugaSpecifikacija) {
   return new AndProizvodSpecifikacija(this, drugaSpecifikacija);
}
private class AndProizvodSpecifikacija extends
    ProizvodSpecifikacija {
   private ProizvodSpecifikacija prvaSpecifikacija,
       drugaSpecifikacija;
   public AndProizvodSpecifikacija(
   ProizvodSpecifikacija prvaSpecifikacija, ProizvodSpecifikacija
       drugaSpecifikacija) {
      this.prvaSpecifikacija = prvaSpecifikacija;
      this.drugaSpecifikacija = drugaSpecifikacija;
   }
   public boolean zadovoljavaNaOsnovu(Proizvod proizvod) {
      return prvaSpecifikacija.zadovoljavaNaOsnovu(proizvod) &&
         drugaSpecifikacija.zadovoljavaNaOsnovu(proizvod);
   }
}
```

```
public static ProizvodSpecifikacija not(final
    ProizvodSpecifikacija uslovZaNegiranje) {
   return new ProizvodSpecifikacija() {
      public boolean zadovoljavaNaOsnovu(Proizvod proizvod) {
         return !uslovZaNegiranje.zadovoljavaNaOsnovu(proizvod);
      }
   };
}
```

Те се и код који их позива измени на следећи начин:

```
public class PronalazacProizvodaTest extends TestCase...
public void testPretrazivanjeIspodCeneBojeOsim()...
/*
ProizvodSpecifikacija specifikacija = new AndProizvodSpecifikacija(
new ProizvodIznadCeneSpecifikacija(9.00f),
new NotProizvodSpecifikacija(
ProizvodSpecifikacija.premaBoji(Color.white)));
*/
ProizvodSpecifikacija specifikacija =
    ProizvodSpecifikacija.ispodCene(9.00f).and(
ProizvodSpecifikacija.not(
ProizvodSpecifikacija.premaBoji(Color.white)));
```

Тиме је ово рефакторисање завршено.

6.4 Сједињавање кода за случај једног и кода за случај обраде више премерака применом Састав обрасца

Проблем: *Класа се не понаша исто при раду са једним објектом и раду са колекцијом објеката*

Решење: *Обједините два понашања коришћењем Састав обрасца*

6.4.1 Објашњење

У случају да у систему постоји код који различито манипулише једним објектом и колекцијом објеката, пригодно је применити образац Састав и елиминисати ову недоследност. Најчешће не постоји довољно добар разлог да се таква два приступа раздвајају. Уколико то није случај, онај ко користи такву класу бива засипан ирелевантним информацијама о томе како је класа имплементирана, а требало би му само пружити униформан интерфејс за рад и поштедети га сувишних информација. Слично, овакво раздвајање компликује употребу класе, а шта је њена функционалност може пасти у други план.

6.4.2 Потенцијални проблеми

1. **Сврха није јасна**: Важно је да на први мах буде јасно чему класа служи и које су њене могућности. Како она заиста те могућности и обезбеђује је мање важно, и при понуди интерфејса, клијента треба растеретити сувишних детаља имплементације. Уколико се раздваја понашање у случају једног објекта од понашања према више њих, управо долази до непотребног оптерећивања клијента неважним информацијама за њега. Одати сврху јасно применом Састав обрасца, који омогућава униформан интерфејс за рад са различитим бројем објеката.

2. **Понављање кода**: Понављање кода је можда баш главни проблем у овом случају. Применом Састав обрасца објединити различита понашања у једно.

3. **Недовољна једноставност**: Чим се на један начин барата једним а на други начин са више објеката, то је сигнал да је класа

непотребно компликованија него што треба да буде. Поједноставити код применом Састав обрасца како би се сваки број објеката третирао на исти начин.

6.4.3 Практична реализација рефакторизације

1. Идентификовати код који испитује да ли је реч о једном или више објеката, па креирати сложену класу која ће их једнако третирати.
2. Пронаћи код који позива код уочен под 1, те га заменити кодом који се ослања на сложену класу.
3. Поновити корак 2 где год се прави провера да ли се располаже једним или више објеката.
4. Понављати претходне кораке за сваки одломак кода попут тог описаног под 1, све док се сви ти одломци не могу безбедно обрисати.

6.4.4 Пример

Следећи пример надовезује се на пример наведен у претходном поглављу. Дакле, тиче се класе ProizvodSpecifikacija и њених подтипова и тога како они одабирају објекте типа Proizvod из репозиторијума објеката, SkladisteProizvoda.
Да би се добро шватио циљ рефакторисања, потребно је разумети како се те класе користе, што се најбоље постиже посматрањем неколико тест примера:

```
public class SkladisteProizvodaTest extends TestCase...
    private SkladisteProizvoda skladiste;

    private Proizvod vatrogasniKamion = new Proizvod("f1234",
        "Vatrogasni kamion",
    Color.red, 8.95f, VelicinaProizvoda.SREDNJE);
    private Proizvod barbika = new Proizvod("b7654", "Barbika",
    Color.yellow, 15.95f, VelicinaProizvoda.MALO);
    private Proizvod frizbi = new Proizvod("f4321", "Frizbi",
    Color.pink, 9.99f, VelicinaProizvoda.VELIKO);
    private Proizvod palicaZaBejzbol = new Proizvod("b2343",
        "Palica za bejzbol",
    Color.white, 8.95f, VelicinaProizvoda.NEPRIMENLJIVO);
```

```
    private Proizvod igrackaKabriolet = new Proizvod("p1112",
        "Porshe kabriolet igracka",
    Color.red, 230.00f, VelicinaProizvoda.NEPRIMENLJIVO);
    protected void kreirajIPopuni() {
      skladiste = new SkladisteProizvoda();
      skladiste.add(vatrogasniKamion);
      skladiste.add(barbika);
      skladiste.add(frizbi);
      skladiste.add(palicaZaBejzbol);
      skladiste.add(igrackaKabriolet);
    }
```

Први тест пример претражује објекте из репозиторијума на основу боје:

```
public class SkladisteProizvodaTest extends TestCase...
public void testPretrazivanjePremaBoji() {
   List pronadjeniProizvodi =
       skladiste.odaberiPrema(ProizvodSpecifikacija.premaBoji(Color.red));
   assertEquals("pronadjena 2 crvena proizvoda", 2,
       pronadjeniProizvodi.velicina());
   assertTrue("pronadjen vatrogasni kamion",
       pronadjeniProizvodi.contains(vatrogasniKamion));
   assertTrue(
      "pronadjen porshe kabriolet igracka",
   pronadjeniProizvodi.contains(igrackaKabriolet));
}
```

Метод skladiste.odaberiPrema(...) изгледа:

```
public class SkladisteProizvoda...
    private List proizvodi = new ArrayList();
    public Iterator iterator() {
      return proizvodi.iterator();
    }
    public List odaberiPrema(ProizvodSpecifikacija
        proizvodSpecifikacija) {
      List pronadjeniProizvodi = new ArrayList();
      Iterator proizvodi = iterator();
      while (proizvodi.hasNext()) {
         Proizvod proizvod = (Proizvod) proizvodi.next();
```

```
        if (proizvodSpecifikacija.zadovoljavaNaOsnovu(proizvod))
            pronadjeniProizvodi.add(proizvod);
      }
      return pronadjeniProizvodi;
    }
```

Наредни тест позива други skladiste.odaberiPrema(...) метод. Тај метод прима колекцију ProizvodSpecifikacija објеката, како би изабрао различите врсте производа из репозиторијума (подржава више критеријума претраге):

```
public class SkladisteProizvodaTest extends TestCase...
    public void testPretrazivanjePremaBojiVeliciniIIspodCene() {
      List specifikacije = new ArrayList();
      specifikacije.add(ProizvodSpecifikacija.premaBoji(Color.red));
      specifikacije.add(ProizvodSpecifikacija.premaVelicini(VelicinaProizvoda.MALO));
      specifikacije.add(ProizvodSpecifikacija.premaMinimalnojCeni(10.00f));
      List pronadjeniProizvodi =
          skladiste.odaberiPrema(specifikacije);
      assertEquals(
        "nije pronadjen mali crveni proizvod cene ispod $10.00",
      0,
      pronadjeniProizvodi.velicina());
      specifikacije.clear();
      specifikacije.add(ProizvodSpecifikacija.premaBoji(Color.red));
      specifikacije.add(ProizvodSpecifikacija.premaVelicini(VelicinaProizvoda.SREDNJE));
      specifikacije.add(ProizvodSpecifikacija.premaMinimalnojCeni(10.00f));
      pronadjeniProizvodi = skladiste.odaberiPrema(specifikacije);
      assertEquals(
        "pronadjen vatrogasni kamion pri pretrazi jeftinih crvenih
            igracaka srednje velicine",
      vatrogasniKamion,
      pronadjeniProizvodi.get(0));
    }
```

Метод skladiste.odaberiPrema(...) који прихвата листу (List) изгледа:

```
public class SkladisteProizvoda {
   public List odaberiPrema(List specifikacijeProizvoda) {
      List pronadjeniProizvodi = new ArrayList();
```

```
    Iterator proizvodi = iterator();
    while (proizvodi.hasNext()) {
      Proizvod proizvod = (Proizvod) proizvodi.next();
      Iterator specifikacije = specifikacijeProizvoda.iterator();
      boolean zadovoljavaSveSpecifikacije = true;
      while (specifikacije.hasNext()) {
        ProizvodSpecifikacija proizvodSpecifikacija =
           ((ProizvodSpecifikacija) specifikacije.next());
        zadovoljavaSveSpecifikacije &=
           proizvodSpecifikacija.zadovoljavaNaOsnovu(proizvod);
      }
      if (zadovoljavaSveSpecifikacije)
          pronadjeniProizvodi.add(proizvod);
    }
    return pronadjeniProizvodi;
  }
```

Уколико се упореде ова два skladiste.odaberiPrema(...) метода, уочава се доста понављања. Добар део може бити уклоњен уколико се један метод прилагоди тако да имплементира уједно и други:

```
public class SkladisteProizvoda...
   public List odaberiPrema(ProizvodSpecifikacija
      proizvodSpecifikacija) {
     ProizvodSpecifikacija[] specifikacije = {
       proizvodSpecifikacija
     };
     return odaberiPrema(Arrays.asList(specifikacije));
   }
   public List odaberiPrema(List specifikacijeProizvoda)...
   // ista implementacija kao i ranije
```

Међутим, ово оставља и даље два odaberiPrema(...) метода. Уколико ће се увек позивати онај који узима List као аргумент, може се применити „Линијски метод” [2] на odaberiPrema(ProizvodSpecifikacija proizvodSpecifikacija):

```
public class SkladisteProizvodaTest extends TestCase...
   public void testPretrazivanjePremaBoji()...
//   List pronadjeniProizvodi =
   skladiste.odaberiPrema(ProizvodSpecifikacija.premaBoji(Color.red));
```

```
ProizvodSpecifikacija[] specifikacije = {
  ProizvodSpecifikacija.premaBoji(Color.red)
};
List pronadjeniProizvodi =
    skladiste.odaberiPrema(Arrays.asList(specifikacije));
```

Ипак, ово није допринело побољшању имплементације, с обзиром да је сада неопходно креирати листу ProizvodSpecifikacija објеката чак и када се при одабиру користи само један критеријум. Закључак је да је боље имати две odaberiPrema(...) методе него непотребно креирати листу и када је потребно проследити само један објекат.
Проблем са почетка и даље постоји: из угла трећег лица, интерфејс претраге на основу једног и на основу више критеријума није исти. Уједно, постоји још један, додатни проблем: тренутна имплементација захтева да производ задовољи све критеријуме задате спецификацијом, те услов попут:

proizvod.getBoja() != zeljenaBoja ||
proizvod.getCena() < zeljenaCena

не би могао да се примени.

Следећи корак при рефакторисању сам се намеће, а то је имплементација ,,or" услова. Ово се најбоље може одрадити применом Састав обрасца.
Чак и уколико су тренутне потребе такве да се јављају само ,,and" и ,,or" услови, и даље је паметно имплементирати и ,,or", због тога што:

- Није тешко.
- Постојаће само један odaberiPrema(...) метод, једноставнији од постојећег.
- То чини систем флексибилним да подржи и евентуалне будуће захтеве који се могу јавити.

Прво је потребно направити сложену класу. Њена улога је да омогући спајање два услова (спецификације) тако да се третирају као један, без потребе да се креира листа спецификација. Нека се зове AndProizvodSpecifikacija и изгледа:

```
public class AndProizvodSpecifikacija extends
    ProizvodSpecifikacija {
   private ProizvodSpecifikacija prvaSpecifikacija,
       drugaSpecifikacija;
   public AndProizvodSpecifikacija(
   ProizvodSpecifikacija prvaSpecifikacija,
   ProizvodSpecifikacija drugaSpecifikacija) {
      this.prvaSpecifikacija = prvaSpecifikacija;
      this.drugaSpecifikacija = drugaSpecifikacija;
   }
   public boolean zadovoljavaNaOsnovu(Proizvod proizvod) {
      return prvaSpecifikacija.zadovoljavaNaOsnovu(proizvod) &&
         drugaSpecifikacija.zadovoljavaNaOsnovu(proizvod);
   }
}
```

Ово омогућава позивање odaberiPrema(...) методе која прихвата један објекат типа ProizvodSpecifikacija:

```
public class SkladisteProizvoda...
    public List odaberiPrema(ProizvodSpecifikacija
        proizvodSpecifikacija) {
      List pronadjeniProizvodi = new ArrayList();
      Iterator proizvodi = iterator();
      while (proizvodi.hasNext()) {
         Proizvod proizvod = (Proizvod) proizvodi.next();
         if (proizvodSpecifikacija.zadovoljavaNaOsnovu(proizvod))
             pronadjeniProizvodi.add(proizvod);
      }
      return pronadjeniProizvodi;
    }
```

Сада је потребно заменити код треће стране који је прослеђивао листу ProizvodSpecifikacija објеката, кодом који користи AndProizvodSpecifikacija и позива odaberiPrema(...) који узима један ProizvodSpecifikacija објекат:

```
public void testPretrazivanjePremaBojiVeliciniIIspodCene() {
// List specifikacije = new ArrayList();
// specifikacije.add(ProizvodSpecifikacija.premaBoji(Color.red));
```

```
//
   specifikacije.add(ProizvodSpecifikacija.premaVelicini(VelicinaProizvoda.MALO));
//
   specifikacije.add(ProizvodSpecifikacija.premaMinimalnojCeni(10.00f));
// List pronadjeniProizvodi =
   skladiste.odaberiPrema(specifikacije);

  ProizvodSpecifikacija specifikacija = new
     AndProizvodSpecifikacija(
  ProizvodSpecifikacija.premaBoji(Color.red),
     new AndProizvodSpecifikacija(
  ProizvodSpecifikacija.premaVelicini(VelicinaProizvoda.MALO),
  ProizvodSpecifikacija.premaMinimalnojCeni(10.00f)));
  List pronadjeniProizvodi = skladiste.odaberiPrema(specifikacija);
```

Уколико више нема кода који позива odaberiPrema(...) метод који прихвата листу, он се може обрисати. Тиме је ово рефакторисање завршено, с обзиром да SkladisteProizvoda више не садржи различите методе који праве разлику да ли је прослеђен један или је прослеђено више објеката.
Као што је већ било случаја, и ово рефакторисање се може поправити. Једна од таквих корекција је енкапсулирање сложене класе AndProizvodSpecifikacija, применом рефакторисања Енкапсулација класа методама креирања, са почетка овог рада. Да би се то учинило, AndProizvodSpecifikacija мора постати приватна, угњеждена класа класе ProizvodSpecifikacija и за њу се мора обезбедити метод креирања:

```
public abstract class ProizvodSpecifikacija...
   private class AndProizvodSpecifikacija extends
      ProizvodSpecifikacija {
     private ProizvodSpecifikacija prvaSpecifikacija,
        drugaSpecifikacija;
     public AndProizvodSpecifikacija(
     ProizvodSpecifikacija prvaSpecifikacija,
     ProizvodSpecifikacija drugaSpecifikacija) {
        this.prvaSpecifikacija = prvaSpecifikacija;
        this.drugaSpecifikacija = drugaSpecifikacija;
     }
     public boolean zadovoljavaNaOsnovu(Proizvod proizvod) {
        return prvaSpecifikacija.zadovoljavaNaOsnovu(proizvod) &&
```

```
            drugaSpecifikacija.zadovoljavaNaOsnovu(proizvod);
      }
    }
    public ProizvodSpecifikacija and(
    final ProizvodSpecifikacija drugaSpecifikacija) {
      return new AndProizvodSpecifikacija(this, drugaSpecifikacija);
    }
```

У складу са тим мора се изменити и код који је позивао ранију имплементацију. На пример, код:

```
public void testPretrazivanjePremaBojiVeliciniIIspodCene()...
    ProizvodSpecifikacija specifikacija = new
        AndProizvodSpecifikacija(
    ProizvodSpecifikacija.premaBoji(Color.red),
    new AndProizvodSpecifikacija(
    ProizvodSpecifikacija.premaVelicini(VelicinaProizvoda.MALO),
    ProizvodSpecifikacija.premaMinimalnojCeni(10.00f)));
    List pronadjeniProizvodi =
        skladiste.odaberiPrema(specifikacija);
```

постаје:

```
public void testPretrazivanjePremaBojiVeliciniIIspodCene()...
    ProizvodSpecifikacija specifikacija =
        ProizvodSpecifikacija.premaBoji(Color.red).and(
    ProizvodSpecifikacija.premaVelicini(VelicinaProizvoda.MALO).and(
    ProizvodSpecifikacija.premaMinimalnojCeni(10.00f)));
    List pronadjeniProizvodi =
        skladiste.odaberiPrema(specifikacija);
```

Тиме је омогућена једноставна композиција услова у један, употребљавана од стране јединственог odaberiPrema(...) метода.

6.5 Замена акумулирајућег кода Сакупљајућим параметром

Проблем: *Постоји кабасти метод који акумулира информације у једној променљивој*

Решење: *Прослеђивати тзв. Сакупљајући параметар (енг. Collecting Parameter) методама, тако да се резултати метода чувају у том параметру*

6.5.1 Објашњење

Сакупљајући параметар је објекат који се прослеђује методама како би прикупио информације од свих њих. На пример, овај образац је користан ако се жели уситнити кабаст метод на мање методе („Екстракције методе", [2]), али је потребно сакупити информације из сваког од добијених, мањих метода. Уместо да свака од њих враћа резултате који се на крају укомбинују у излаз, може се један објекат прослеђивати свим методама, при чему они надовезују информације на његову вредност.
Сакупљајући параметар добро функционише у пару са композитним обрасцем због тога што се може искористити да рекурзивно прикупи информације из композитне структуре.

6.5.2 Потенцијални проблеми

1. **Сврха није јасна**: Кабасти код најчешће није у први мах јасан. Разјаснити које информације се акумулирају у излазну променљиву разбијањем метода на мање методе са јасном наменом, који уписују своје резултате у један исти параметар.

2. **Понављање кода**: Овај образац не утиче знахајно на редукцију понављања кода.

3. **Недовољна једноставност**: Код поједноставити разбијањем на мање целине, као што је објашњено у првој ставци.

6.5.3 Практична реализација рефакторизације

1. Идентификовати код који прикупља информације за излазну променљиву (на даље: резултат). Тај резултат ће бити Сакупљајући параметар. Уколико тип променљиве није такав да може памтити

информације по излазу из методе, променити му тип. На пример, ово је случај са Јавином класом String, те би у овом случају резултат требало да буде типа StringBuffer.

2. Пронаћи део који акумулира податке и издвојити га у засебан приватни метод (коришћењем „Екстракције методе", [2]). Тај метод не треба да враћа било шта (повратни тип void), а прослеђује му се резултат. Унутар метода, информације се уписују у прослеђени резултат.

3. Поновити претходни корак за сваки случај у ком се акумулирају неке информације, све док првобитни код није у потпуности замењем методама описаним у тој ставци.

6.5.4 Пример

Пример демонстрира како се резултујући композитан код из секције „Замена имплицитног дрвета композитним" рефакторише да би се у њему применио овај образац. Конкретно, ради се о предефинисању toString() метода, који рекурзивно посећује чворове XML дрвета и враћа његову XML репрезентацију.

Метод toString() изгледа на следећи начин:

```
class XMLEtiketa...
public String toString() {
   String rezultat = new String();
   rezultat += "<" + imeEtikete + " " + atributi + ">";
   Iterator it = potomci.iterator();
   while (it.hasNext()) {
      XMLEtiketa cvor = (XMLEtiketa) it.next();
      rezultat += cvor.toString();
   }
   if (!vrednostEtikete.equals("")) rezultat += vrednostEtikete;
   rezultat += "</" + imeEtikete + ">";
   return rezultat;
}
```

Потребно је променити повратни тип из String у StringBuffer, како би било могућа имплементација овог обрасца:

```
StringBuffer rezultat = new StringBuffer("");
```

Први корак који акумулира информације је код који на отворен XML таг надовезује његове атрибуте резултујућој промењљивој:

```
rezultat += "<" + imeEtikete + " " + atributi + ">";
```

Одговорајући метод изгледа:

```
private void pisiOtvarajucuEtiketuU(StringBuffer rezultat) {
   rezultat.append("<");
   rezultat.append(ime);
   rezultat.append(atributi.toString());
   rezultat.append(">");
}
```

Првобитан код замењује се са:

```
StringBuffer rezultat = new StringBuffer("");
pisiOtvarajucuEtiketuU(rezultat);
```

Претходни поступак се понавља за свако наредно надовезивање информација на излазну промењљиву. У овом случају, то је моменат у ком се додају потомци XML чвору, и ту се појављује рекурзија:

```
class XMLEtiketa...
   public String toString()...
      Iterator it = potomci.iterator();
      while (it.hasNext()) {
         XMLEtiketa cvor = (XMLEtiketa) it.next();
         rezultat += cvor.toString(); // rekurzija
      }
   if (!vrednostEtikete.equals("")) rezultat += vrednostEtikete;
   ...
   }
```

Управо ова рекурзија отежава екстракцију кода у један метод, а следећи код то и демонстрира:

```
private void pisiPotomkeU(StringBuffer rezultat) {
   Iterator it = potomci.iterator();
```

```
   while (it.hasNext()) {
      XMLEtiketa cvor = (XMLEtiketa) it.next();
      cvor.toString(rezultat); // ne moze jer toString() ne
          prihvata argumente
   }
   ...
}
```

Како метод toString() не прихвата аргументе, није могуће праволинијски применити екстракцију методе, те је потребно наћи други начин. Једна могућност је користити помоћни метод, који ради исту ствар као и toString(), али прихвата StringBuffer као Сакупљајући параметар:

```
public String toString() {
   return toStringPomocni(new StringBuffer(""));
}
private String toStringPomocni(StringBuffer rezultat) {
   pisiOtvarajucuEtiketuU(rezultat);
   ...
   return rezultat.toString();
}
```

Сада је могуће наставити издвајање у метод:

```
private String toStringPomocni(StringBuffer rezultat) {
   pisiOtvarajucuEtiketuU(rezultat);
   pisiPotomkeU(rezultat);
   ...
   return rezultat.toString();
}
private void pisiPotomkeU(StringBuffer rezultat) {
   Iterator it = potomci.iterator();
   while (it.hasNext()) {
      XMLEtiketa cvor = (XMLEtiketa) it.next();
      cvor.toStringPomocni(rezultat); // sada radi rekurzivni poziv
   }
   if (!value.equals(""))
      rezultat.append(value);
}
```

Метод pisiPotomkeU ради две ствари: додаје потомке рекурзивно, а

онда уписује вредност тагу, уколико бар једна већ постоји. Добра пракса је да метод има тачно једну одговорност, те се уписивање вредности издваја у посебан метод:

```
private void pisiVrednostU(StringBuffer rezultat) {
   if (!value.equals(""))
     rezultat.append(value);
}
```

На крају, потребно је издвојити у посебан метод и део кода који надовезује затварајући XML таг. Крајњи код изгледа:

```
public class XMLEtiketa...
    public String toString() {
      return toStringPomocni(new StringBuffer(""));
    }
    private String toStringPomocni(StringBuffer rezultat) {
      pisiOtvarajucuEtiketuU(rezultat);
      pisiPotomkeU(rezultat);
      pisiVrednostU(rezultat);
      pisiZatvarajucuEtiketuU(rezultat);
      return rezultat.toString();
    }
    private void pisiOtvarajucuEtiketuU(StringBuffer rezultat) {
      rezultat.append("<");
      rezultat.append(ime);
      rezultat.append(atributi.toString());
      rezultat.append(">");
    }
    private void pisiPotomkeU(StringBuffer rezultat) {
      Iterator it = potomci.iterator();
      while (it.hasNext()) {
         XMLEtiketa cvor = (XMLEtiketa) it.next();
         cvor.toStringPomocni(rezultat);
      }
    }
    private void pisiVrednostU(StringBuffer rezultat) {
      if (!value.equals("")) rezultat.append(value);
    }
    private void pisiZatvarajucuEtiketuU(StringBuffer rezultat) {
      rezultat.append("</");
```

```
      rezultat.append(ime);
      rezultat.append(">");
    }
}
```

Мала модификација која чини претходни код ефикаснијим је следеће:

```
public String toString() {
   StringBuffer rezultat = new StringBuffer("");
   toStringPomocni(rezultat);
   return rezultat.toString();
}
public void toStringPomocni(StringBuffer rezultat) {
   pisiOtvarajucuEtiketuU(rezultat);
   pisiPotomkeU(rezultat);
   pisiVrednostU(rezultat);
   pisiZatvarajucuEtiketuU(rezultat);
}
```

Ово је могуће јер се повратна вредност toStringPomocni() методе користи само када се позива из toString() методе, а иначе се игнорише.

6.6 Декомпоновање методе

Проблем: *Није једноставно разумети логику кода унутар методе*
Решење: *Декомпоновати метод у мање логичке целине, приближно истог обима одговорности, са јасним описом посла који свака од целина обавља*

6.6.1 Објашњење

Реч је о потпуно једноставном обрасцу. Идеја је да се метод који обавља неки посао декомпонује на мање целине - методе, који обављају делове посла, назване у складу са одговорношћу коју имају. Тиме код постаје читљивији, јаснији, а самим тим се и лакше уочавају bug-ови (уколико постоје).

6.6.2 Потенцијални проблеми

1. **Сврха није јасна**: Може на први поглед бити јасно *шта* метод ради, али не мора бити јасно и *како*. Уколико се разбије на мање логичке целине постаје лагоднији за тумачење и разумевање.

2. **Понављање кода**: Уколико се неки код понавља на више места у оквиру метода, овим обрасцем би тај део кода постао једна мања целина, позивана одакле је потребно. Тиме се елимише и понављање.

3. **Недовољна једноставност**: Згодан тест једноставности је прочитати код метода. Уколико он ,,не звучи” као текст енглеског језика, такав да има смисла и може се разумети, већ је прекомпликован или конфузан, постоји простор за рефакторисање овим обрасцем.

6.6.3 Практична реализација рефакторизације

Не постоје конкретни кораци које треба предузети приликом овог рефакторисања, али се таксативно може навести на шта треба обратити пажњу:

- Декомпоноване методе имају између 5 и 10 линија кода.

- Уколико постоји, уклонити понављање, смештањем кода који се понавља у засебан метод.

- Разјаснити функције мањих целина давањем једноставних и јасних имена методама и промењивама.
- Одложити оптимизацију. Нека на првом месту буде једноставност. Евентуална даља рефакторисања обавити тек кад се метод декомпонује.
- Нека мање целине буду на сличном нивоу комплексности и одговорности.

6.6.4 Пример

Нека је дат метод contains(), који проверава да ли се објекат налази унутар другог објекта тако што испитује да ли је садржан потпуно или барем делимично:

```
public boolean contains(Component c) {
   Point p = c.getLocation();
   int koordX = new Double(p.getX()).intValue();
   int koordY = new Double(p.getY()).intValue();
   boolean potpunoUnutar =
      (koordX >= koordinate[0] && koordY >= koordinate[1]
      && (koordX + KartaKomponenta.SIRINA) <= koordinate[2])
      && (koordY + KartaKomponenta.VISINA) <= koordinate[3];
   if (potpunoUnutar) return true;

   koordX = koordX + KartaKomponenta.SIRINA;
   koordY = koordY + KartaKomponenta.VISINA;
   boolean delimicnoUnutar =
      (koordX > koordinate[0]
      && koordY > koordinate[1]
      && (koordX < koordinate[2])
      && (koordY < koordinate[3]));
   return delimicnoUnutar;
}
```

Један од примера где се овај метод користи је када се проверава да ли је карта унутар поља првог играча, те уколико јесте, карта се одатле премешта:

```
public void testKartaIzvanPoljaPrvogIgraca() {
```

```
   Ruka ruka = (Ruka) igra.getIgracNaRedu().getRuka();
   Karta karta = (Karta) ruka.elements().nextElement();
   KartaKomponenta c = new KartaKomponenta(karta, igra);
   PoljeIgraca polje = igra.getPoljeIgraca(0);
   igra.pomeriKartu(c, polje.gornjiLevi());
   assertEquals("polje sadrzi kartu", true, polje.contains(c));

   igra.pomeriKartu(c, KartaKomponenta.SIRINA + 10,
      KartaKomponenta.VISINA + 10);
   assertEquals("polje ne sadrzi kartu", false, polje.contains(c));
}
```

Прва идеја је „смањити” метод contains(). На пример, променљива potpunoUnutar:

```
boolean potpunoUnutar=
    (koordX>=koordinate[0]
    && koordY>=koordinate[1]
    && (koordX+KartaKomponenta.SIRINA)<=koordinate[2])
    && (koordY+KartaKomponenta.VISINA)<=koordinate[3];
```

Упркос томе што променљива јасно описује чему служи, contains() би био једноставнији када би њен код био замењен позивом методе која ради исту ствар:

```
public boolean contains(Component c) {
   Point p = c.getLocation();
   int koordX = new Double(p.getX()).intValue();
   int koordY = new Double(p.getY()).intValue();
   if (potpunoUnutar(koordX, koordY)) return true;

   koordX = koordX + KartaKomponenta.SIRINA;
   koordY = koordY + KartaKomponenta.VISINA;

   boolean delimicnoUnutar =
      (koordX > koordinate[0]
      && koordY > koordinate[1]
      && (koordX < koordinate[2])
      && (koordY < koordinate[3]));
   return delimicnoUnutar;
}
```

```
private boolean potpunoUnutar(int koordX, int koordY) {
   return (koordX >= koordinate[0]
   && koordY >= koordinate[1]
   && (koordX + KartaKomponenta.SIRINA) <= koordinate[2])
   && (koordY + KartaKomponenta.VISINA) <= koordinate[3];
}
```

Слично је и са delimicnoUnutar:

```
public boolean contains(Component c) {
   Point p = c.getLocation();

   int koordX = new Double(p.getX()).intValue();
   int koordY = new Double(p.getY()).intValue();
   if (potpunoUnutar(koordX, koordY)) return true;

   koordX = koordX + KartaKomponenta.SIRINA;
   koordY = koordY + KartaKomponenta.VISINA;
   return delimicnoUnutar(koordX, koordY);
}

private boolean delimicnoUnutar(int koordX, int koordY) {
    return (koordX > koordinate[0]
    && koordY > koordinate[1]
    && (koordX < koordinate[2])
    && (koordY < koordinate[3]));
}
```

Иако много једноставнији, метод contains() и даље у себи садржи декларацију промењљивих koordX и koordY које се користе само за интерну употребу метода delimicnoUnutar() и potpunoUnutar(). Најлакше је оптимизовати их прослеђујући им тацху Point p, тако да саме извлаче потребне информације:

```
public boolean contains(Component c) {
   Point p = c.getLocation();
   if (potpunoUnutar(p)) return true;
   return delimicnoUnutar(p);
}
```

Ипак, овде није крај. Линија:

```
Point p = c.getLocation();
```

је детаљ, док наредна два реда представљају логику методе. Све подцелине декомпоноване методе треба да буду на сличном нивоу комплексности и одговорности, а на овај начин то својство није остварено. Може се и обема методама проследити компонента Component c, па да свака од њих унутар свог кода позива Point p = c.getLocation(), али на тај начин се једна ствар ради на два места, што је изузетно лоша пракса. У овој занимљивој ситуацији потребно је донети одлуку о томе шта је приоритет: избалансирати одговорности подцелина или избећи понављање кода? Циљ овог рефакторисања је да метод буде читљив и једноставан, те метод contains() ипак изгледа:

```
public boolean contains(Component c) {
   return potpunoUnutar(c) || delimicnoUnutar(c);
}

private boolean potpunoUnutar(Component c) {
   Point p = c.getLocation();
   int koordX = new Double(p.x).intValue();
   int koordY = new Double(p.y).intValue();
   return (koordX >= koordinate[0]
   && koordY >= koordinate[1]
   && (koordX + KartaKomponenta.SIRINA) <= koordinate[2])
   && (koordY + KartaKomponenta.VISINA) <= koordinate[3];
}

private boolean delimicnoUnutar(Component c) {
   Point p = c.getLocation();
   int koordX = new Double(p.x).intValue() + KartaKomponenta.SIRINA;
   int koordY = new Double(p.y).intValue() + KartaKomponenta.VISINA;
      return (koordX > koordinate[0]
      && koordY > koordinate[1]
      && (koordX < koordinate[2])
      && (koordY < koordinate[3]));
}
```

Само један позив Point p = c.getLocation() на почетку contains()

методе нарушавао је својство читљивости и једноставности методе, стога се ипак прибегло овом компромису.

7 Закључак

Рефакторисање представља процес измене постојећег кода, али не и његовог понашања. Оно за последицу даје читљивији и јаснији код, више пута употребљив, лакши за употребу. Обично се састоји од серија ситних, стандардизованих тзв. ,,микро-рефакторисања" и различита развојна окружења углавном имају подршку да основна рефакторисања изводе и самостално.

Рефакторисање се може сматрати путовањем. Оно почиње од тачке ,,А" која се назива ,,сумњивим кодом" (енг. code smell). Пример сумњивог кода је предугачак метод или два метода која садрже готово идентичан код. Када се такав код препозна, почиње путовање у знаку трансформације старог кода у нови, који се понаша исто као и стари али више није ,,сумњив". На пример, предугачак метод се може разделити на више метода које изводе мање порције послова, док се поновљени код може сместити у један посебан метод кога старе методе позивају, уместо да исти код наводе у свом телу.
Након добро обављеног рефакторисања, код је пре свега лакши за употребу. Постојеће багове је лакше уочити и отклонити, јер је сам код читљивији и намена му је јаснија. Такође, код постаје проширивији. Једноставније је обогатити постојеће могућности система уколико се користе препознатљиви обрасци дизајна. Оно за последицу даје чак и већу флексибилност.

Пре сваког рефакторисања потребно је имати одређене тестове, који показују да почетни код ради на жељени начин. Уколико тестови не пролазе изворни код, неопходно је прво отклонити проблем у њему, јер је касније теже одредити да ли је грешка настала као последица рефакторисања или је постојала и пре него што се почело са рефакторисањем. Након тога, уколико је рефакторисање добро изведено, тестови пролазе на исти начин као и кад су били позивани у контексту изворног кода. То чини поступак рефакторисања у солидној мери безбедним.
Након написаних тестова, процес рефакторисања се своди на итеративни циклус извођења малих трансформација над кодом, а затим тестирања тих фрагмената. Уколико тест у неком тренутку не прође, то је сигнал да је последња измена лоше изведена и да је треба поновити на другачији начин. Мали кораци и стално тестирање играју важну улогу при

поступку рефакторисања.

Као обрасци дизајна, и рефакторисање је безвременски концепт. Универзалност образаца дизајна оставља за последицу да се различити програмери, односно пројектанти, иако временски и просторно удаљени, нађу у истим дилемама у вези са пројектовањем система. Уколико постоји квалитетно решење, више пута примењено и истестирано, оно се може поделити са осталим члановима огромне програмерске заједнице. Овај рад је управо пример таквог доприноса. У њему су наведени само најзаступљенији, чак ни не нарочито компликовани примери сумњивог кода. Решења не могу бити до краја прецизна, већ само описна, написана с намером да се уз њих једноставније дође до тачке ,,Б", а то је јасан, читљив код, више пута употребљив, скалабилан и флексибилан.

Може се рећи да су резултати побољшавања кода и додавање нових функционалности у директној зависности. О томе сведочи следећи контраст: Усмеривањем искључиво на додавање нових функционалности, без осврта на дизајн, увођење нових функционалности постаје све спорије и теже. Ипак, са уредном навиком конзистентног рефакторисања, упркос незанемарљивој временској компоненти коју оно подразумева, додавање нових функционалности постаје једноставније и изводи се брже.

Литература

[1] Kerievsky, Joshua. *Refactoring to patterns.* Pearson Deutschland GmbH. 2005.

[2] Fowler, Martin. *Refactoring: improving the design of existing code.* Pearson Education India, 2002.

[3] Gamma, Erich. et al. *Design patterns: elements of reusable object-oriented software.* Pearson Education, 1994.

[4] Gamma, Erich. et al. *Gotova rešenja: elementi objektno orijentisanog softvera.* CET Computer Equipment and Trade, 2002.

[5] Kent Beck i Erich Gamma. *JUnit Testing Framework.* Доступно на вебу (http://www.junit.org).