Design Pattern – analiza primera

1. Usklađivanje mape sajta sa izmenom sadržaja Web sajta
Zahvaljujući Observer DP, prilikom svakog dodavanja, brisanja ili menjanja stranice generiše se automatski novi sitemap fajl i šalje obaveštenje web pretraživačima da je došlo do promene na sajtu.
Svaka stranica nasleđuje apstraktnu klasu koja obaveštava SiteMap klasu da je došlo da izmene i da treba ažurirati mapu sajta.

Observer - definiše interfejs za ažuriranje objekta koje treba obavestiti o promeni subjekta

```
public interface Observer
{void Update(object subject);}
```

Subject - Njega može da posmatra bilo koji broj posmatrača. On zna za svoje posmatrače i obezbeđuje interfejs za povezivanje i raskidanje veze sa objektima Observera.

```
public abstract class Subject
{
   private ArrayList observers = new ArrayList();

   public void AddObserver(Observer observer)
   {observers.Add(observer);}

   public void RemoveObserver(Observer observer)
   {observers.Remove(observer);}

   public void Notify()
   {
       foreach(Observer observer in observers)
        {observer.Update(this);}
   }
}
```

ConcreteSubject (Page) - čuva stanje od interesa za objekte *ConcreteObserver* i šalje obaveštenje posmatračima kada se stanje promeni.

```
public class Page:Subject
{
  public AddPage(DataRow r)
  {
    //kod za dodavanje stranice
     Notify();
   }
}
```

ConcreteObserver (SiteMap**)**
Održava referencu na objekat *ConcreteSubject*.
Čuva stanje koje treba da ostane konzistentno sa stanjem subjekta.
Implementira *Observer* interfejs za ažuriranje da bi stanje ostalo konzistentno sa stanjem subjekta.

```
public class SiteMaph : Observer
{
   public void Update(object subject)
   { if(subject is Page) GenerateXML((Page)subject);}

   private void GenerateXML(Page page)
   { //kod za generisanje siteMapa}
}
```

2. Ilustracija Observera: sledeći strukturni kôd prikazuje upotrebu obrasca Observer gde se registrovani objekti označavaju, a zatim i ažuriraju kada se promeni stanje.

```
import java.util.*;

// "Subject"
abstract class Subject
{
  private ArrayList observers = new
ArrayList();
  // Metodi
  public void Attach( Observer observer )
  {   observers.add( observer );}

  public void Detach( Observer observer )
  {    observers.remove( observer );}

  public void Notify()
  {
    for( int i=0; i<observers.size(); i++ )
    {
    Observer o = (Observer) observers.get(i);
      o.Update();
    }
  }
}

// "ConcreteSubject"
class ConcreteSubject extends Subject
{
  private String subjectState;

  public String getSubjectState()
  {    return subjectState; }

public void setSubjectState( String value )
       {subjectState = value; }
}

// "Observer"
abstract class Observer
{
  // Metodi
  abstract public void Update();
}

// "ConcreteObserver"
class ConcreteObserver extends Observer
{
  private String name;
  private String observerState;
  private ConcreteSubject subject;

  // Konstruktor
  public ConcreteObserver( ConcreteSubject
subject,  String name )
  {this.subject = subject; this.name = name;}

  // Metodi
  public void Update()
  {
    observerState =subject.getSubjectState();
    System.out.println( " Novo stanje
Observera " + name + " je " +
                        observerState );
  }

  public ConcreteSubject getSubject()
  {    return subject; }
```

```
using System;
using System.Collections;

namespace GOF.Observer
{
  // MainApp test aplikacija
  class MainApp
  {
    static void Main()
    {
      // konfigurisanje Observer DP
 ConcreteSubject s = new ConcreteSubject();

     s.Attach(new ConcreteObserver(s,"X"));
     s.Attach(new ConcreteObserver(s,"Y"));
     s.Attach(new ConcreteObserver(s,"Z"));

 //promeni subject i obavesti observere
      s.SubjectState = "ABC";
      s.Notify();

      Console.Read();
    }
  }

  // "Subject"

  abstract class Subject
  {
    private ArrayList observers = new
ArrayList();
   // Metodi
    public void Attach(Observer observer)
    {
      observers.Add(observer);
    }

    public void Detach(Observer observer)
    {
      observers.Remove(observer);
    }

    public void Notify()
    {
      foreach (Observer o in observers)
      {
        o.Update();
      }
    }
  }

  // "ConcreteSubject"

  class ConcreteSubject : Subject
  {
    private string subjectState;

    // Svojstvo
    public string SubjectState
    {
      get{ return subjectState; }
      set{ subjectState = value; }
    }
  }
```

```
public void setSubject(ConcreteSubject value)
       {      subject = value;}
}


public class Ilustracija
{
  public static void main( String[] args )
  {
    // konfigurisanje Observer DP
   ConcreteSubject s = new ConcreteSubject();
   s.Attach( new ConcreteObserver( s,"X" ) );
   s.Attach( new ConcreteObserver( s,"Y" ) );
   s.Attach( new ConcreteObserver( s,"Z" ) );

    //promeni subject i obavesti observere
    s.setSubjectState( "ABC" );
    s.Notify();

   /* promeni subject i obavesti observere
              s.setSubjectState( "DEF" );
              s.Notify();
     */
  }
}
```

```
  // "Observer"

  abstract class Observer
  {
    public abstract void Update();
  }

  // "ConcreteObserver"

  class ConcreteObserver : Observer
  {
    private string name;
    private string observerState;
    private ConcreteSubject subject;

    // Konstruktor
    public ConcreteObserver(
      ConcreteSubject subject, string name)
    {
      this.subject = subject;
      this.name = name;
    }

    public override void Update()
    {
     observerState = subject.SubjectState;
   Console.WriteLine("Novo stanje Observera
{0} je {1}", name, observerState);
    }

    // Svojstva
    public ConcreteSubject Subject
    {
      get { return subject; }
      set { subject = value; }
    }
  }
}
```

Izlaz

```
Novo stanje Observera X je ABC
Novo stanje Observera Y je ABC
Novo stanje Observera Z je ABC
```

3. Primer iz stvarnog sveta koji prikazuje obrazac Observer u kome se registrovani investitori obaveštavaju svaki put kada akcije promene svoje vrednosti.

4. Simulator kotla za grejanje se, između ostalog, sastoji od klase `MeracPritiska` sa metodama `int uzmiPritisak()` koja vraća tekući pritisak u kotlu i metodom `podesiPritisk(int)` sa kojom se postavlja tekući pritisak. Za prikaz pritiska zadužena je GUI klasa `PritisakPrikaz`. Takođe, sigurnosni ventil, predstavljen klasom `SigurnosniVentil` treba da automatski reaguje kada pritisak u kotlu pređe unapred zadatu vrednost i da se otvori. Korišćenjem Observer DP simulirati rad kotla pri čemu će se iz test klase postavljati vrednost pritiska u kotlu dok će se pritisak prikazivati iz metoda klase `PritisakPrikaz`, a otvaranje sigurnosnog ventila će biti prikazano iz metoda klase `SigurnosniVentil`. Svi prikazi treba da budu na konzoli.

5. Novinska agencija sakuplja vesti i objavljuje ih svojim pretplatnicima na različite načine: email, SMS, RSS, ... Koristeći Observer DP simulirati rad novinske agencije, pri čemu čim se desi vest, odmah se obavešavaju pretplatnici. Rešenja mora biti proširivo u smislu da se mora podržati nova forma pretplatnika (može da se pojavi nova komunikaciona tehnologija).

6. Korišćenjem *Composite* DP, kreirati predstavu proizvoljnog HTML dokumenta kao stablo instanci dole navedenih klasa i zatim na konzoli ispisati HTML dokument. Parser HTML fajlova kreira stablo apstraktne

sintakse sa sledećim elementima:

- HTMLDocument
  - HTMLHead
    - HTMLTitle
    - HTMLMeta
  - HTMLBody
    - HTMLHeading1
    - HTMLParagraph
      - HTMLAnchor

Svi elementi HTML dokumenta, kao i sam *HTMLDocument* imaju metodu print.

7. Ilustracija Strategije: sledeći strukturni kôd prikazuje upotrebu Strategy DP koji enkapsulira funkcionalnosti u formi objekta. Ovo klijentu omogućava da dinamički menja strategiju primene algoritama.

```
// "Strategija"
abstract class Strategija
{
abstract public void
InterfejsAlgoritma();
}

// "KonkretnaStrategijaA"
class KonkretnaStrategijaA extends
Strategija
{
public void InterfejsAlgoritma()
      {
      System.out.println("Pozvana
KonkretnaStrategijaA.InterfejsAlgorit
ma()");
      }
}

// "KonkretnaStrategijaB"
class KonkretnaStrategijaB extends
Strategija
{
public void InterfejsAlgoritma()
      {
      System.out.println("Pozvana
KonkretnaStrategijaB.InterfejsAlgorit
ma()");
      }
}

// "KonkretnaStrategijaC"
class KonkretnaStrategijaC extends
Strategija
{
public void InterfejsAlgoritma()
      {
      System.out.println("Pozvana
KonkretnaStrategijaC.InterfejsAlgorit
ma()");
      }
}

// "Kontekst"
class Kontekst
{
```

```
using System;

namespace GOF.Strategy
{

  class MainApp
  {
    static void Main()
    {
      Context context;

// Tri konteksta prate razlicite strategije
context = new Context(new ConcreteStrategyA());
context.ContextInterface();

context = new Context(new ConcreteStrategyB());
context.ContextInterface();

context = new Context(new ConcreteStrategyC());
context.ContextInterface();
      Console.Read();
    }
  }

  // "Strategy"

  abstract class Strategy
 {public abstract void AlgorithmInterface();}

  // "ConcreteStrategyA"

  class ConcreteStrategyA : Strategy
  {
    public override void AlgorithmInterface()
    {
     Console.WriteLine(
        "Pozvano
ConcreteStrategyA.AlgorithmInterface()");
    }
  }

  // "ConcreteStrategyB"

  class ConcreteStrategyB : Strategy
  {
    public override void AlgorithmInterface()
    {
      Console.WriteLine(
        "Pozvano
ConcreteStrategyB.AlgorithmInterface()");
    }
```

```
        Strategija strategija;

        // Konstruktor
public Kontekst(Strategija
strategija)
{this.strategija = strategija;}

        // Metodi
public void InterfejsKonteksta()
{strategija.InterfejsAlgoritma();}
}


public class Ilustracija
{
public static void main(String[]
args)
{
//Tri konteksta prate razlicite
//strategije
Kontekst c = new Kontekst(new
KonkretnaStrategijaA());
c.InterfejsKonteksta();

Kontekst d = new Kontekst(new
KonkretnaStrategijaB());
d.InterfejsKonteksta();

Kontekst e = new Kontekst(new
KonkretnaStrategijaC());
e.InterfejsKonteksta();
}
}
```

```
  }

  // "ConcreteStrategyC"

  class ConcreteStrategyC : Strategy
  {
    public override void AlgorithmInterface()
    {
      Console.WriteLine(
        "Pozvano
ConcreteStrategyC.AlgorithmInterface()");
    }
  }

  // "Context"

  class Context
  {
    Strategy strategy;

    // Konstruktor
    public Context(Strategy strategy)
    {
      this.strategy = strategy;
    }

    public void ContextInterface()
    {
      strategy.AlgorithmInterface();
    }
  }
}
```

```
Izlaz
Pozvano ConcreteStrategyA.AlgorithmInterface()
Pozvano ConcreteStrategyB.AlgorithmInterface()
Pozvano ConcreteStrategyC.AlgorithmInterface()
```

8. Primer realnog slučaja prikazuje upotrebu Strategy DP koji enkapsulira različite algoritme sortiranja u formi objekata. Ovo klijentu omogućava da dinamički menja strategije sortiranja (ovde su podržani Quicksort, Shellsort i Mergesort).

```
import java.util.*;

// "Strategy"
abstract class SortStrategy
{
        // Metodi
        abstract public void Sort(ArrayList list);
}

// "ConcreteStrategy"
class QuickSort extends SortStrategy
{
        // Metodi
        public void Sort(ArrayList list)
        {
                //list.QuickSort();  nije implementirano
                System.out.println("QuickSortirana lista ");
```

```
        }
}

// "ConcreteStrategy"
class ShellSort extends SortStrategy
{
        // Metodi
        public void Sort(ArrayList list)
        {
                //list.ShellSort(); nije implementirano
                System.out.println("ShellSortirana lista ");
        }
}

// "ConcreteStrategy"
class MergeSort extends SortStrategy
{
        // Metodi
        public void Sort(ArrayList list)
        {
                Collections.sort(list); // MergeSort je default sort algoritam
                System.out.println("MergeSortirana lista ");
        }
}

// "Context"
class SortedList
{
        // polja
        private ArrayList list = new ArrayList();
        private SortStrategy sortstrategy;

        // Konstruktor
        public void SetSortStrategy(SortStrategy sortstrategy)
        {
                this.sortstrategy = sortstrategy;
        }

        // Metodi
        public void Sort()
        {
                sortstrategy.Sort(list);
        }

        public void Add(String name)
        {
                list.add(name);
        }

        public void Display() //prikaz rezultata
        {
                for (int i = 0; i < list.size(); i++)
                {
                        String name = list.get(i).toString();
                        System.out.println(" " + name);
                }
        }
```

```
}

public class RealanSlucaj
{
	public static void main(String[] args)
	{
		// Dva konteksta prate različite  strategije
		SortedList studentRecords = new SortedList();
		studentRecords.Add("Sima");
		studentRecords.Add("Jovan");
		studentRecords.Add("Sanja");
		studentRecords.Add("Ana");
		studentRecords.Add("Vesna");
		studentRecords.Display();

		studentRecords.SetSortStrategy(new MergeSort());
		studentRecords.Sort();
		studentRecords.Display();
	}
}
```

IZLAZ

```
 Sima
 Jovan
 Sanja
 Ana
 Vesna
MergeSortirana lista
 Ana
 Jovan
 Sanja
 Sima
 Vesna
```

9. Upotrebom Strategy DP realizujte aplikaciju koja implementira različita ponašanja robota. Robot je primerak context klase koja pamti i beleži kontekst informacije kao što su pozicija, biske prepreke,... i prosleđuje ih klasi Strategy.

Konkretne Strategije (svaka definiše specifično ponašanje): AgresivnoPonašanje, OdbrambenoPonašanje, NormalnoPonašanje. Ova klasa određuje akciju na osnovu informacija dobijenih od senzora robota (kao što su pozicija, biske prepreke,...).

U main sekciji aplikacije potrebno je kreirati nekoliko robota i nekoliko različitih ponašanja. Svaki robot ima svoje ponašanje. Na primer: 'LjutiRobot' je agresivan i napada druge robote u areni , 'Robot v.8.1' je plašljiv i beži u suprotnom smeru kada susretne drugog robota u areni, 'R2' je staložen i ignoriše druge robote. U nekom trenutku rada, menjaju se i ponašanja svakog robota.

10. Klasa `StudentskaSluzba` čuva studente sortirane po broju indeksa. Potrebno je prikazati studente sortirane po prezimenu i imenu. Upotrebom Strategy DP napraviti metodu za prikaz sortirane liste pri čemu će algoritam za sortiranje biti izmenjiv. Napraviti dve konkretne implementacije algoritma za sortiranje: `BubbleSort` i `QuickSort`.

Napomena: Nije potrebno zaista implementirati algoritme za sortiranje već je dovoljno samo na konzoli ispisati koje sortiranje se obavlja.