Design Patterns

Programare Orientată pe Obiecte

Proiectarea aplicațiilor

- Realizarea unui program presupune e*tapa inițială de proiectare*
- Este posibil ca după implementarea proiectului inițial să se revină la această etapă
  - Îmbunătățire – refactorizare
  - Modificare esențială
- Faza de proiectare
  - Se stabilesc clasele, inerfețele necesare şi relațiile dintre ele
    - Clase noi specifice aplicației
    - Clase predefinite de uz general
  - Pentru un proiect mai complex: crearea de modele ce reprezintă relațiile dintre clase şi obiecte, rolul fiecărui obiect etc. - UML (Unified Modeling Language)

Proiectarea aplicațiilor

- Proiectare ce are în vedere posibilitatea de a fi extins şi adaptat ulterior într-un mod cât mai simplu şi cât mai sigur – *design for change*

- Extindere simplă şi sigură
  - Nu se va recurge la modificarea unor clase existente
  - Se vor defini clase noi care
    - Înlocuiesc clasele din programul existent
    - Se vor adăuga claselor existente

- Posibilitatea reutilizării unor clase din aplicație şi în alte aplicații

Proiectarea aplicațiilor

- Clasele si obiectele necesare într-o aplicatie pot rezulta din:
  - Analiza aplicației concrete şi modelarea obiectelor şi acțiunilor din aplicație;
  - Analiza altor aplicații şi a bibliotecilor standard Java =>
    - Clase cu caracter mai abstract pentru:
      - Grupare de obiecte
      - Intermediere între obiecte
      - Alt rol care nu este evident din descrierea aplicației
  - Cunoaşterea unor soluții de proiectare deja folosite cu succes în alte aplicații – *design patterns*

Proiectarea aplicațiilor

- Mărirea flexibilitătii în extinderea şi adaptarea unei aplicații:
  - prin mărirea numărului de clase si obiecte din aplicație!
- Un proiect ce conține numai clasele rezultate din analiza aplicației poate fi mai compact,
  - dar nu este scalabil şi adaptabil la alte cerințe apărute după realizarea prototipului aplicației!
- Sunt importante:
  - separarea părților susceptibile de schimbare de părțile care nu se mai modifică
  - evitarea cuplajelor "strânse" dintre clase

Design Patterns

- Recunoaşterea şi inventarierea unor scheme (şabloane) de proiectare “Design Patterns”:
  - grupuri de clase şi obiecte care cooperează pentru realizarea unor funcții.
- În cadrul acestor scheme există clase care au un anumit rol în raport cu alte clase şi care au primit un nume ce descrie acest rol;
- Exemple: clase iterator, clase observator, clase “fabrică” de obiecte, clase adaptor ş.a.
- Dincolo de detaliile de implementare se pot identifica clase cu aceleaşi responsabilități în diferite aplicații.

Design Patterns. Definiții

- Definitii posibile pentru schemele de proiectare folosite în aplicații cu clase:
- Soluții optime generale şi reutilizabile ale unei probleme comune în design-ul software.
- Descrieri ale soluțiilor sau template-uri ce pot fi aplicate pentru rezolvarea problemei.
- Reguli pentru realizarea anumitor sarcini în proiectarea programelor cu obiecte.
- Abstractizări la un nivel superior claselor, obiectelor sau componentelor.
- Scheme de comunicare (de interacțiune) între obiecte.

Avantaje şi recomandări

- Argumentul principal în favoarea studierii si aplicării schemelor de clase:
  - aplicarea acestor scheme conduce la programe mai usor de modificat.
    - în general prin clase (obiecte) “slab” cuplate, care stiu cât mai putin unele despre altele.

- Principalele recomandări care rezultă din analiza schemelor de proiectare şi a aplicațiilor:
  - Proiectarea cu interfețe şi clase abstracte este preferată față de proiectarea cu clase concrete
    - permite separarea utilizării de implementare.
  - Este recomandată crearea de clase si obiecte suplimentare, cu rol de intermediari
    - pentru decuplarea unor clase cu roluri diferite.

Clasificare

Se disting trei categorii:

1. Creational patterns: scheme “creaționale“
   - generează obiectele necesare,
   - mecanisme de creare a obiectelor
   - Ex. *Singleton*, *Factory* etc

2. Structural patterns: scheme structurale
   - definesc relații între entități
   - grupează mai multe obiecte în structuri mai mari.
   - Decorator, *Adapter*, Composite, Proxy etc.

3. Behavioral patterns: scheme de interacțiune
   - definesc comunicarea între entități
   - *Iterator*, *Visitor*, *Observer*, *Command*, Mediator etc.

Design pattern-ul *Command*

- *Behavioral pattern*: legat de interacțiunea dintre componente, de felul în care se efectuează apelurile.
- Este utilizat un obiect pentru a reprezenta şi încapsula toate informațiile necesare pentru a apela mai târziu o anumită metodă:
  - Numele metodei, obiectul de care ține metoda şi valorile parametrilor metodei.
- Permite:
  - actiuni (comenzi) multiple
  - decuplarea alegerii operatiei executate de locul unde este emisă comanda.

- foloseste o interfată generală, cu o singură functie, de felul următor:

```
public interface Command {
   public void execute(); // execută o actiune nedefinită încă
}
```

Command pattern – descriere

- Entități:
  - Command
  - Receiver
  - Invoker
  - Client
- Un obiect *command* are un obiect *receiver* şi invocă o metodă a *receiver*-ului într-un mod specific specific clasei acelui *receiver*.
- *Receiver*-ul realizează acțiunea.
- Un obiect *command* este transmis separat unui obiect *invoker*, care invocă comanda, şi opțional, realizează arhivarea execuției comenzii.
- Obiectul *invoker* poate primi orice obiect *command*.
- Atât un obiect *invoker* cât şi obiectele *command* sunt păstrate de un obiect *client*.
- *Clientul* conține decizia legată de ce comandă să se execute şi la ce moment. Pentru a executa o comandă, el transmite obiectul *command* obiectului *invoker*.

Tipuri de componente (roluri):

- **Invoker** – cere execuția unor comenzi
  - apelează acțiuni - metode oferite de obiectele de tip *Command*)
  - poate menține, o *listă a tutoror comenzilor aplicate* pe obiectul (obiectele) comandate.
  - primeşte clase *Command* pe care să le invoce
- **Receiver**
  - clasa asupra căreia se face apelul
  - conține implementarea efectivă a ceea ce se doreşte executat
- **Command** - obiectele pentru reprezentarea comenzilor implementează această interfață sau o extind (dacă este clasă abstractă)
  - conțin metode cu nume sugestiv pentru executarea acțiunii comenzii - execute()
  - conțin referințe către *receivers* pentru a realiza comanda
  - *concrete command* - implementări/subclase
- **Client** – crează comenzile concrete şi setează receiver-ul lor

Exemplu

- Un “switch” cu două comenzi: aprinderea/stingerea luminii.
- Acest switch poate fi folosit cu orice alt gen de acțiune pentru un alt aparat, nu doar pentru aprinderea-stingerea luminii, metoda Switch-ului poate primi orice subclasă Command ca parametru.
- Exemplu: îl putem configura pentru pornirea unui motor.

```
import java.util.List;
import java.util.ArrayList;

 /* interfata Command */
public interface Command {
   void execute();
}
```

Exemplu

```
/* Invoker */
public class Switch {
   private List<Command> history = new ArrayList<Command>();
    public void storeAndExecute(Command cmd) {
            history.add(cmd);           // optional
            cmd.execute();
   }
}

/* Receiver */
public class Light {
   public void turnOn() {
      System.out.println("The light is on");
   }
   public void turnOff() {
      System.out.println("The light is off");
   }
}
```

Exemplu

```
 /* Comanda pentru aprinderea luminii – comanda concreta 1 */
public class FlipUpCommand implements Command {
    private Light theLight;
    public FlipUpCommand(Light light) {
      theLight = light;
   }
    public void execute(){
      theLight.turnOn();
   }
}
 /* Comanda pentru aprinderea luminii – comanda concreta 2 */
public class FlipDownCommand implements Command {
   private Light theLight;
   public FlipDownCommand(Light light) {
      theLight = light;
   }
    public void execute() {
      theLight.turnOff();
   }
}
```

Exemplu

```
/* Clasa test sau client */
public class PressSwitch {
   public static void main(String[] args){
      Light lamp = new Light();
      Command switchUp = new FlipUpCommand(lamp);
      Command switchDown = new FlipDownCommand(lamp);
      Switch mySwitch = new Switch();
       try {
         if ("ON".equalsIgnoreCase(args[0]))
               mySwitch.storeAndExecute(switchUp);
         else if ("OFF".equalsIgnoreCase(args[0]))
                mySwitch.storeAndExecute(switchDown);
             else
                  System.out.println("\"ON\" sau \"OFF\" !!!");
      } catch (Exception e) {
               System.out.println("Introduceti argumentele!!");
      }
   }
}
```

Design pattern-ul *Command*

- In Swing există mai multe interfețe "ascultător" corespunzătoare interfeței “Command”. Exemplu:

```
public interface ActionListener {
    public void actionPerformed( ActionEvent ev);
}
```

- Un obiect dintr-o clasă ascultător este un obiect "comandă" ce realizează o acțiune.
  - singura legătură dintre partea de interfață grafică a aplicației şi partea de logică specifică aplicației (tratarea evenimentelor).
  - Se realizează astfel decuplarea celor două părți, ceea ce permite modificarea lor separată şi chiar selectarea unei anumite acțiuni în cursul execuției.

Exemplu fără command pattern

```
 …
JMenuBar mbar = new JMenuBar();
setJMenuBar (mbar);
JMenu mFile = new JMenu ("File");
mbar.add (mFile);
JMenuItem open = new JMenuItem ("Open");
    // optiune meniu vertical
JMenuItem exit = new JMenuItem ("Exit");
    // optiune meniu vertical
mFile.add (open);
mFile.addSeparator( );
mFile.add (exit);
open.addActionListener (this);
exit.addActionListener (this);
…..
```

```
// Tratarea evenimentelor
public void actionPerformed (ActionEvent e) {
  Object source = e.getSource(); // sursa evenimentului
  if ( source == open)  fileOpen( );// actiune asociata lui "Open"
  else if (source == exit) System.exit(0); // actiune asociata lui "Exit"
}
```

Exemplul cu *command pattern*

- *o interfață "Command”*

```
public interface Command {
    public void execute();
}
```

- *definirea mai multor clase care implementează această interfață:*

```
class FileExitCmd  extends JMenuItem implements Command {
    public FileExitCmd (String optname) {
            super(optname);  // nume optiune (afisat in meniu)
    }
    public void execute () {
            System.exit(0);          // actiune asociata optiunii exit
    }
}

class FileOpenCmd  extends JMenuItem implements Command
{
    public FileOpenCmd (String optname) {
            super(optname);  // nume optiune (afisat in meniu)
    }
```

Exemplul cu *command pattern*

```
    public void execute () {
            fileOpen();      // actiune asociata optiunii open
    }
}

class CmdListener implements ActionListener {
      public void actionPerformed (ActionEvent e) {
            Command c = (Command) e.getSource();
             c.execute();
      }
} // Invoker
    . . .
    ActionListener cmd = new CmdListener();
    open.addActionListener (cmd);
    exit.addActionListener (cmd);
```

- Metoda "execute" : metodă polimorfică
  - selectarea unei implementări sau alta se face în funcție de tipul variabilei ***c***, deci de tipul obiectului care este sursa evenimentului!

Exemplu utilizare Command pattern

- în programarea meniurilor si barelor de instrumente ("toolbar"), prin utilizarea obiectelor "actiune” - Action
- implementează interfata *Action*.
- O bară de instrumente contine mai multe butoane cu imagini (obiecte *JButton*), dar are acelasi rol cu o bară meniu: selectarea de actiuni de către operator.
- In loc să se adauge celor două bare obiecte diferite (dar care produc aceeasi actiune) se adaugă un singur obiect, de tip *Action*, care contine o metodă *actionPerformed*; câte un obiect pentru fiecare actiune selectată.
- Interfata *Action* ⇔ interfața *Command*
- metoda *actionPerformed* ⇔ metoda *execute*.
- Interfata *Action* extinde interfata *ActionPerformed* cu două metode *setValue* şi *getValue*:
  - stabilirea si obtinerea proprietătilor unui obiect actiune (text afisat în optiune meniu, imagine afisată pe buton din bara de instrumente, o scurtă descriere ("tooltip") a "instrumentului" afisat)

Utilizare

- Facilitează construirea de componente generale care au nevoie să delege, sau să execute apeluri de metode fără să fie necesar să cunoască clasa metodei sau parametrii metodei în momentul în care realizează aceasta!

- Utilizarea unui obiect invoker permite
  - păstrarea evidenței execuției comenzilor
  - implementarea mai multor tipuri de comenzi care sunt gestionate de invoker
- fără să fie necesar ca clientul să ştie că există această arhivare sau aceste tipuri!