Laboration 3 – Designmönster

Författad av: *Gabriel Afram, Thomas Astner, Alex Darborg, Andreas Edin, Adam Hjernquist, Joakim Sundqvist* och *Mattias Vängman*

Factory Pattern

Avsikt

Factory-mönstret ”producerar” objekt med ett dynamiskt urval av olika subklasser. Ett interface definierar **hur** ett objekt skapas men subklasser till interfacet avgör vilken klass som ska användas för varje enskild instans.

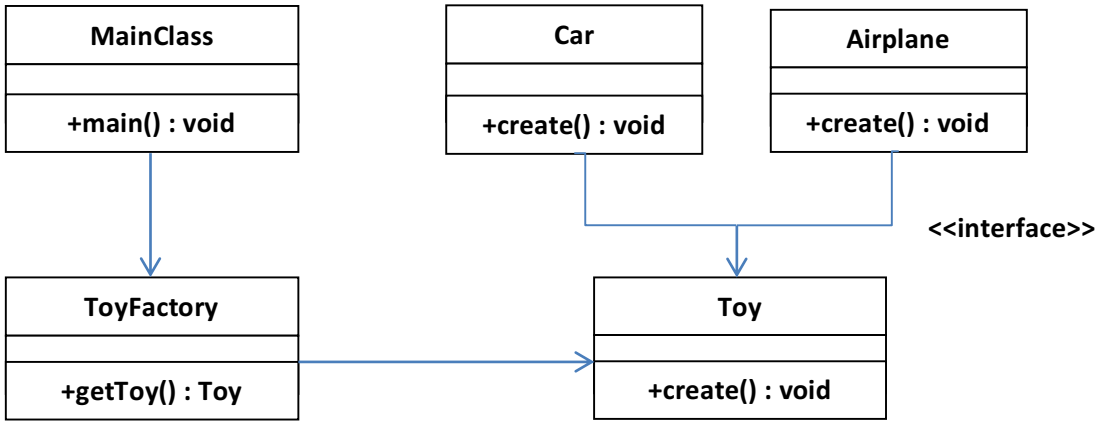
Motivation

Låt oss föreställa oss ett scenario där vi har ett system som använder sig av abstrakta klasser som definierar och upprätthåller relationer mellan objekt. Vi har även en typ av ansvarsområde för att skapa dessa objekt. Detta system vet alltså bara (under exekvering) **när** ett nytt objekt ska skapas men inte exakt av vilken **typ**. Factory-mönstret kan då implementeras via en s.k. Factory-metod där ett interface definieras för att skapa objekt men låter subklasser stå för typ-skillnaderna.

Tillämplighet

Factory-mönstret är användbart när en klass inte kan förutsäga vilken typ av objekt den måste skapa.

Struktur



Deltagare

- **MainClass;** den klass som hanterar alla instanser av Toy.
- **ToyFactory;** fungerar som själva factory-metoden.
- **Toy;** använder ett gemensamt interface med dem olika Toy-typerna Car och Airplane.

Samarbete

ToyFactoryDemo använder ToyFactory-klassen (Factory-metoden) för att skapa och ge tillbaka instanser av Toy-objekt. Den information som ToyFactory får av ToyFactoryDemo

används utav Toy och dem olika Toy-typerna för att ge tillbaka korrekt typ av Toy-objekt, d.v.s. Car eller Airplane.

Konsekvenser

Det kan vara besvärligt att implementera Factory-mönstret i ett redan fungerande system; många fundamentala moduler kan behöva skrivas om. Fabriken måste användas på ett sådant sätt att objekten tillhör samma familj. Om dem inte tillhör samma basklass/interface så kan inte Factory-mönstret användas.

Implementering

ToyFactory kan vara abstrakt och därför kräva att implementationen definieras i subklasser. Om ToyFactory är konkret så kan implementeringen kan vi istället flexibelt åsidosätta default-metoderna och använda den implementation som används i subklasserna.

Kodexempel (Java)

```
public interface Toy {
    void create();
}

public class ToyCar implements Toy {

    @Override
    public void create() {
        //Do stuff
    }
}

public class ToyAirplane implements Toy {

    @Override
    public void create() {
        // Do stuff
    }
}

public class ToyFactory {

    public Toy getToy(String type) {
        if (type == "CAR") {
            return new ToyCar();
        } else if (type == "AIRPLANE") {
            return new ToyAirplane();
        }
    }
}
// A main class to use the factory pattern
```

Relaterade mönster

- **Abstract Factory**, vilket ofta använder implementationer av factory-metoder.

Model-View-Controller

Avsikt

MVC används för att separera ett systems ansvarsområden, framför allt från användargränssnittet. Dessa ansvarsområden delas upp i Model, View och Controller.

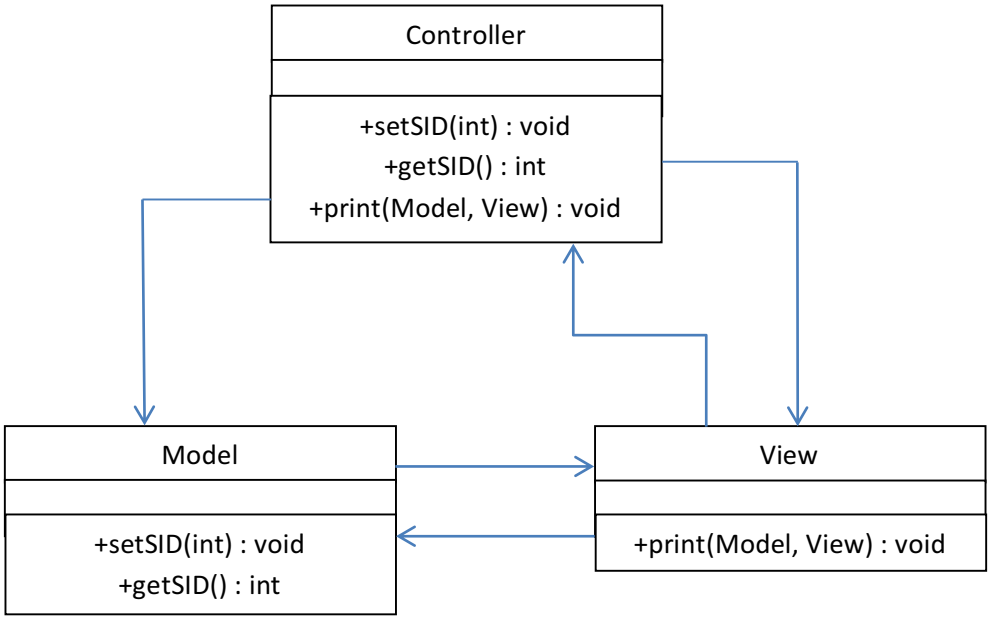
Motivation

I en mjukvaruarkitektur där användare använder ett interface och hanterar en samling data så är det praktiskt att kunna separera dessa ansvarsområden för att underlätta underhåll och utveckling.

Tillämplighet

Ett MVC-system implementeras i tre fundamentala steg; en Model som definierar vad informationen representerar, en View som beskriver hur informationen ska representeras för användaren (exempelvis ett användargränssnitt) och en Controller som bearbetar och besvarar händelser samt hanterar ändringar i Model och View.

Struktur



Deltagare & sammarbete

- **Model**; själva dataobjektet där informationen finns.
- **View**; det gränssnittsmedium som användaren utnyttjar.
- **Controller**; den mekanism som hanterar instanser av Model och view.

Konsekvenser

Istället för att koncentrera sig på att använda den aktuella informationen inuti Model så ligger MVC's fokus främst på att presentera data, vilket inte alltid är optimalt ur ett prestandaperspektiv.

Kodexempel

```
public class Model {
    private int SID;

    public Model(int SID) {
        this.SID = SID;
    }

    public int getSID() {
        return SID;
    }

    public void setSID(int SID) {
        this.SID = SID;
    }
}

public class View {
    public void print(int SID) {
        System.out.println("The SID is " + SID);
    }
}

public class Controller {
    private final Model model;
    private final View view;

    public Controller(Model model, View view) {
        this.model = model;
        this.view = view;
    }

    public void setSID (int SID) { model.setSID(SID); }

    public int getSID () { model.getSID(); }

    public void print() { view.print(model.getSID()); }
}

public class MVCDemo {
    public static void main(String[] args) {
        int SID = 3;

        Model model = new Model(SID);
        View view = new View();
        Controller controller = new Controller(model, view);

        controller.print();   //  -> OUTPUT: "The SID is 3"
        SID++;
        controller.setSID(SID);
        controller.print();   //  -> OUTPUT: "The SID is 4"
    }
}
```

Relaterade mönster

- **Observer Pattern**; används ofta tillsammans med MCV för att hålla View- och Controller-modulerna synkroniserade.

Iterator pattern

Avsikt

Att i en sekvens kunna nå en samling objekt utan att veta någonting om den underliggande representationen.

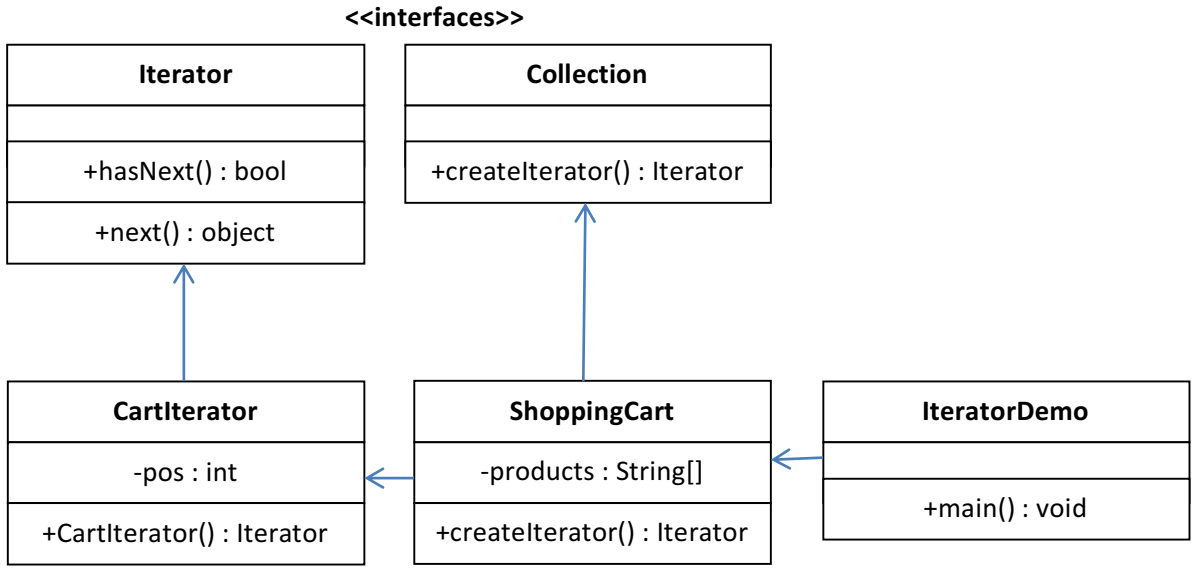
Motivation

I dem fall vi behöver skapa egna typer av listor, arrays, osv, så behöver vi även en metod som kan ge oss tillgång till objekten utan att känna till den interna strukturen och hur dem internt är representerade; vi är bara intresserade av den aktuella informationen. Därför kan vi lägga allt itererande ansvar (åtkomst och placering) i ett iterator-objekt.

Tillämplighet

I situationer där flera olika typer av objektsamlingar håller information och vi behöver ett mönster som ger oss åtkomst parallellt så är det användbart att implementera ett gemensamt interface för iterator-mönstret som sedan kan användas av objekt av liknande typer som ShoppingCart. Det är då enkelt att tillämpa detta mönster på nya typer av samlingar/Collection's.

Struktur



Deltagare & sammarbete

- **Iterator**; det första interfaces för hur vi söker igenom objektstrukturen.
- **CartIterator**; håller position i varje iterering och ger tillbaka Iterator-objektet till ShoppingCart.
- **Collection**; skapar och håller instanser av Iterator.
- **ShoppingCart**; det objekt som kapslar in hela instansen av det som ska sökas igenom.

Konsekvenser

Potentiella problem kan upstå i flertrådade miljöer när Collection hämtar Iterator-objekt från olika trådar om designen inte är trådsäker. Det innebär ytterliggare komplexitet och svårigheter under implementering.

Implementering

Mönstret kan flexibelt implementeras med hjälp av två interface så att flera olika samlingstyper kan användas samtidigt.

Kodexempel

```
public interface Iterator {
   public boolean hasNext();
   public Object Next();
}

public interface Collection {
   public Iterator createIterator();
}

public class ShoppingCart implements Collection {
   private final String products[] = {"Apples", "Cake", "GPU"};

   @Override
   public Iterator createIterator() {
      return new CartIterator();
   }

   private class CartIterator implements Iterator {
      private int pos;

      @Override
      public boolean hasNext() {
         if (pos < products.length) {
            return true;
         }
         return false;
      }

      @Override
      public Object Next() {
         if (this.hasNext()) {
            return products[pos++];
         }
         return null;
      }
   }
}
```

Relaterade mönster

- **Composite Pattern**; iteratorer används ofta rekursivt för att färdas igenom sammansatta strukturer.

Command Pattern

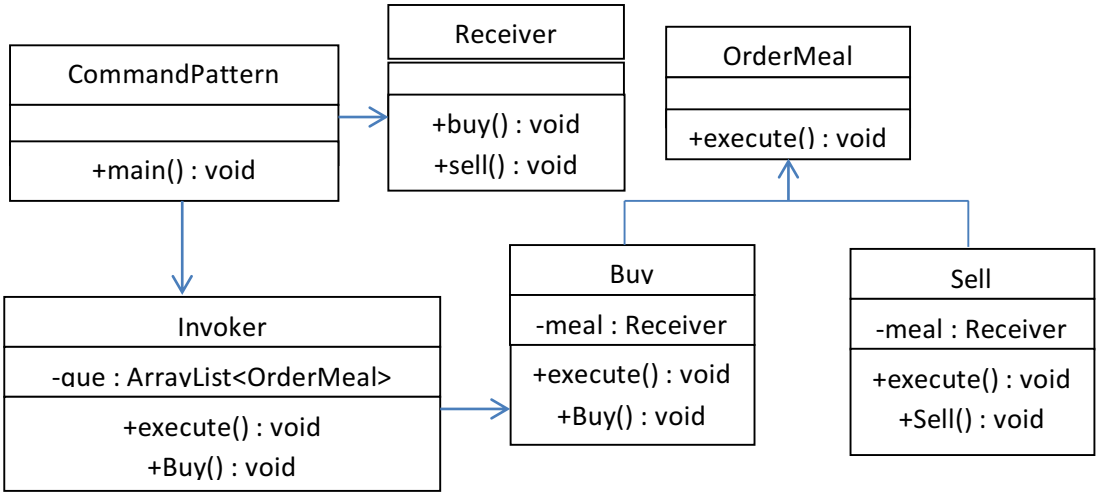
Avsikt

Tanken med Command-mönstret är att kunna skapa kommandon som är insvept/inkapslat i ett objekt för att bl.a. kunna exekvera åtgärden på ett kontrollerat sätt. På så vis kan uppgifter/requests utföras vid ett specifikt tillfälle men också tas bort eller dupliceras.

Motivation

En kund vill på en restaurang kunna för-, varm- och efterrätt på en och samma gång. Kunden ska samtidigt kunna ångra sig innan beställningen läggs. Om kunden exempelvis skulle vilja göra det via en dator-applikation så behöver dessa kommandon kunna utföras på ett strukturerat sätt. Med hjälp av Command-mönstret så kan varje beställning motsvara ett kommando där varje kommando placeras i en lista som senare skall utföras, samtidigt som denna lista kan modifieras under tiden.

Struktur



Deltagare och samarbete

- **OrderMeal**; ett interface för körning av kommandon.
- **Receiver**; definierar hur operationerna utförs.
- **Invoker**; säger åt kommandot att utföra sitt ärende. Detta objekt håller även listan med dem kommandon som ska utföras.
- **Buy & Sell**; implementerar OrderMeal och håller en Reciever.

Kodexempel

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

public class Receiver {
    public void buy() {
        System.out.println("A product was bought");
    }
    public void sell() {
        System.out.println("A product was sold");
    }
}

public class Invoker {
    private final ArrayList<OrderMeal> que = new ArrayList();

    void newOrder(OrderMeal ordermeal) {
        que.add(ordermeal);
    }

    public void executeCommands() {
        que.forEach((orderMeal) -> {
            orderMeal.execute();
        });
    }
}

public class Buy implements OrderMeal{
    private final Receiver meal;

    @Override
    public void execute() { meal.buy(); }

    public Buy(Receiver meal) { this.meal = meal; }
}

public class Sell implements OrderMeal {
    private Receiver meal;

    @Override
    public void execute() { meal.sell(); }

    public Sell(Receiver meal) { this.meal = meal; }
}

public class CommandPatternDemo {
    public static void main(String[] args) {
        Receiver r = new Receiver();

        Buy WhatToBuy = new Buy(r);
        Sell WhatToSell = new Sell(r);

        Invoker i = new Invoker();
        i.newOrder(WhatToBuy);
        i.newOrder(WhatToSell);

        // execute the list of commands
        i.executeCommands();
    }
}
```

Relaterade mönster

- **Composite Pattern**; kan användas tillsammans med Command för att gruppera existerande kommandon för att skapa macros.

Singleton Pattern

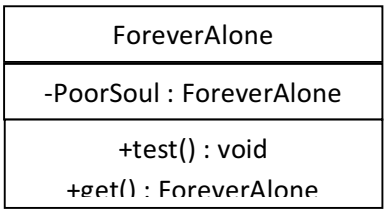
Avsikt

Singleton-mönstret förhindrar att flera än en instans av ett objekt kan skapas.

Motivation

I situationer där flera moduler i ett system vill kunna hantera och modifiera en gemensam resurs eller ett gemensamt objekt så kan det bli problematiskt om nya instanser av ett objekt tillåts att skapas. Därför kan Singleton-mönstret implementeras för att förbjuda skapandet av nya instanser.

Struktur



Deltagare och sammarbete

- **ForeverAlone**; singleton-objektet som instansieras. Klassen definierar hur objektet används för att hämta den specifika instansen (via en get-metod) samt hålla all eventuell inkapslad information.

Konsekvenser

Singleton-mönstret bryter mot SRP (Single Responsibility Principle) eftersom det kontrollerar sitt eget skapande och sin egen livscykel.

Kodexempel

```
public class ForeverAlone {
    private static ForeverAlone PoorSoul = new ForeverAlone();

    private ForeverAlone(){}

    public void test(){ System.out.println("The object exists."); }

    public static ForeverAlone get(){ return PoorSoul; }
}

public class SingletonPattern {
    public static void main(String[] args) {
        ForeverAlone singletonObject = ForeverAlone.get();
        singletonObject.test();
    }
}
```

Relaterade mönster

- **Abstract Factory**; implementeras ofta som singletons för att försäkra om att endast en factory kontrollerar objekt-skapande.

Template Method Pattern

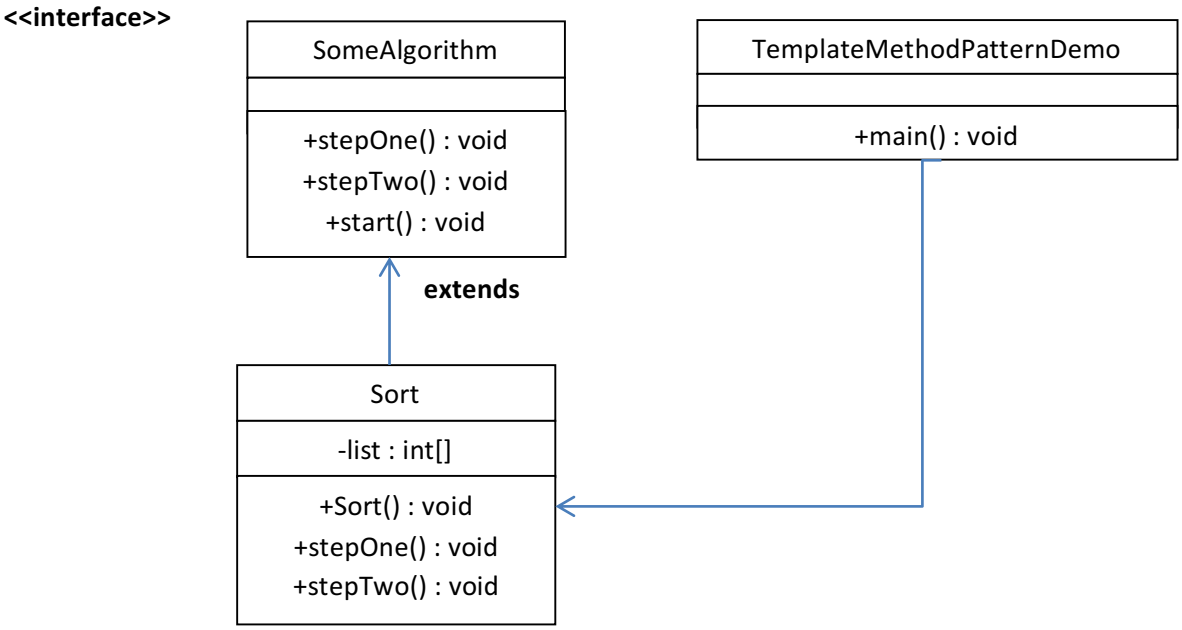
Avsikt

En mall som håller det standardmönster en algoritm använder. På så vis kan subklasser skriva över och modifiera särskilda event i algoritmen.

Motivation

En algoritm kan behöva förändras beroende på vilken data den försöker använda, exempelvis en sorteringsalgoritm. Sorteringsalgoritmen kan behöva sortera en Array<String> liksom en Array<Int>. Därför kan Template Method Pattern standardisera en mall som har respektive subklasser för respektive uppgift som behöver utföras.

Struktur



Deltagare och Sammarbete

- **SomeAlgorithm**; standardklassen som definierar formatet på hur algoritmerna är uppbyggda.
- **Sort**; den specifika subklassen som skriver över SomeAlgoritm, en sorteringsalgoritm i detta fall.

Kodexempel

```
public abstract class SomeAlgorithm {
   abstract void stepOne();
   abstract void stepTwo();

   public final void start(){
      stepOne();

      stepTwo();
   }
}

public class Sort extends SomeAlgorithm {
   private final int[] list;

   public Sort (int[] list) {
      this.list = list;
   }

   @Override
   void stepOne() {
      System.out.println("Original list: " + Arrays.toString(list));
      Arrays.sort(list);
   }

   @Override
   void stepTwo() {
      System.out.println("Sorted list: " + Arrays.toString(list));
   }
}

public class TemplateMethodPatternDemo {
   public static void main(String[] args) {
      int[] list = {3, 1, 7, 4, 2, 8};
      SomeAlgorithm sort = new Sort(list);

      sort.start();
   }
}
```

Relaterade mönster

- **Factory Pattern**; metoderna/funktionerna kallas ofta template-metoder.
- **Strategy**; template-metoder använder arv för att växla vissa delar av en algoritm (se nästa mönster).

Strategy Pattern

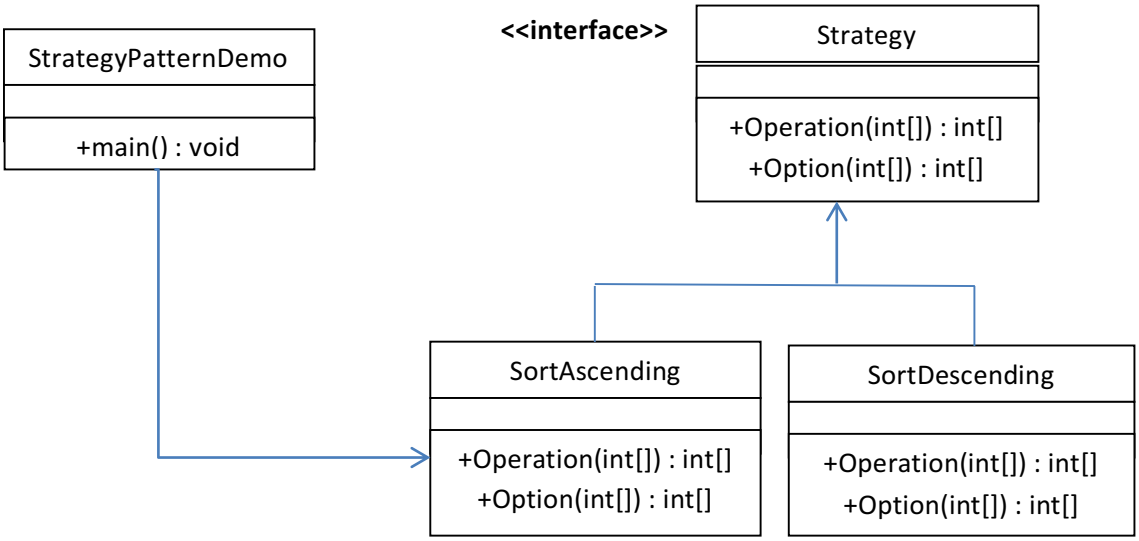
Avsikt

Detta mönster används för att definiera och individuellt inkapsla en familj av algoritmer för att kunna använda olika algoritm-varianter under exekvering, omväxlande.

Motivation

I särskilda situationer kan ett program under exekvering behöva en annan strategi kring hur en algoritm ska utföras. Ett program kan exempelvis behöva använda olika resurser beroende vilken typ av webbläsare som används, vilket underliggande operativsystem som är närvarande, osv.

Struktur



Deltagare & sammarbete

- **Strategy**; det interface som definierar vilka metoder som familje-algoritmerna ska stödja.
- **SortAscending & SortDescending**; subklass och familjeimplementation till Strategy.

Konsekvenser

Klienter måste förstå skillnaden mellan strategierna som finns tillgängliga innan ett lämpligt alternativ kan väljas, för att undvika problem.

Kodexempel

```
public interface Strategy {
   public int[] Operation(int[] list);
   public int[] Option(int[] list);

}

public class SortAscending implements Strategy {

   @Override
   public int[] Operation(int[] list) {
      Arrays.sort(list);
      return list;
   }

   @Override
   public int[] Option(int[] list) {
      return list;
   }

}

public class SortDescending implements Strategy {

   @Override
   public int[] Operation(int[] list) {
      Arrays.sort(list);
      list = Option(list);

      return list;
   }

   @Override
   public int[] Option(int[] list) {
      for(int i = 0; i < list.length / 2; i++) {
         int temp = list[i];
         list[i] = list[list.length - i - 1];
         list[list.length - i - 1] = temp;
      }

      return list;
   }

}

public class StrategyPatternDemo {
   public static void main(String[] args) {
      Strategy Ascendinglist = new SortAscending();
      Strategy Descendinglist = new SortDescending();
      int[] list = {4,2,7,1,3,2};

      list = Ascendinglist.Operation(list);
      System.out.println("Ascending sort: \t" + Arrays.toString(list));

      list = Descendinglist.Operation(list);
      System.out.println("Descending sort: \t" + Arrays.toString(list));
   }
}
```

Relaterade mönster

- **Template Method Pattern**; dessa två mönster kan användas i konjunktion för att kombinera familje-metoder och strategier med templates (mallar).

Facade Pattern

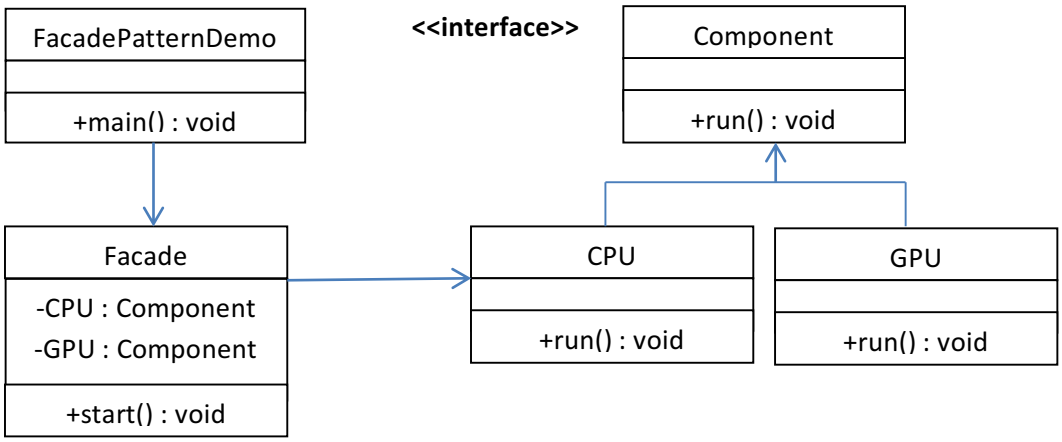
Avsikt

Facade används för att skapa en ”fasad” som gömmer ett mer komplicerat interface. Detta är användbart för att göra koden enklare att använda, förstå, läsa och för att gömma komplexitet.

Motivation

Vid tillfällen där användaren av ett program inte behöver/får veta hur programmet fungerar så är det användbart att skapa en s.k. fasad. Ett exempel på en fasad är gränssnittet i ett program eller ett operativsystem; allt användaren behöver veta är **vad** resultatet blir av interaktionen, inte **hur**. På så vis så är det samtidigt ett mönster som är svårt att låta bli att implementera om systemet har ett användargränssnitt.

Struktur



Deltagare & Sammarbete

- **Component**; ett interface som definierar hur varje komponent ser ut.
- **CPU & GPU**; implementationer av Component, utför olika uppgifter relaterade till deras funktionalitet.
- **Facade**; den centrala klassen i fasad-mönstret. Döljer underliggande komplexitet genom att låta Facade-instanser använda en start-metod.

Kodexempel

```
public interface Component {
   public void run();
}

public class CPU implements Component{
   @Override
   public void run() {
      System.out.println("CPU is running...");
   }
}

public class GPU implements Component {
   @Override
   public void run() {
      System.out.println("GPU is running...");
   }
}

public class Facade {
   private final Component CPU;
   private final Component GPU;

   public Facade() {
      CPU = new CPU();
      GPU = new GPU();
   }

   public void Start(){
      CPU.run();
      GPU.run();
   }
}

public class FacadePattern {
   public static void main(String[] args) {
      Facade System = new Facade();
      System.Start();
   }
}
```

Real uses

Dem flesta system använder någon form utav Facade-liknande mönster. Ur en användares perspektiv i ett operativsystem eller en webbläsare så fungerar gränssnittet som en Facade eftersom vi inte vet eller får någon information om hur dem underliggande processerna till vår interaktion med interfacet faktiskt fungerar.