

Laboration 2: *Designmönster*

Bakgrund

Det har visat sig väldigt svårt att beskriva hur ett system, eller en delösning, skall konstrueras på ett bra sätt. Det har överhuvud taget varit svårt att veta om en viss typ av design är bra eller inte. Designmönster verkar vara en lösning på dessa problem!

Med designmönster försöker man beskriva återkommande bra lösningar som (helst) återfinns i flera olika system. Man försöker fånga kärnan i lösningen så att den går att återanvända i andra (inte nödvändigtvis lika) sammanhang. Man försöker också sätta namn på mönstren så att de som använder dem skall få kraftfullare metoder för att konceptuellt hantera problem men inte minst för att ge ett kraftfullare språk vid designdiskussioner med andra.

Mål

Efter laborationen skall deltagaren känna till, behärska och ha erfarenhet från att ha använt eller implementerat program som använder några grundläggande designmönster. Du ska också öva på att skriva tester innan du skriver riktig applikationskod genom att bland annat använda JUnit.

Uppskattad tidsåtgång

13 timmar per gruppmedlem.

Förberedelse

Läs kapitel 23, 33 (sidor 525-528), 34 (sidor 546-551) i Larman, föreläsning 9 och 10 om designmönster.

Tips

Se <http://hillside.net/patterns/> med massor av länkar till olika beskrivningar av designmönster.

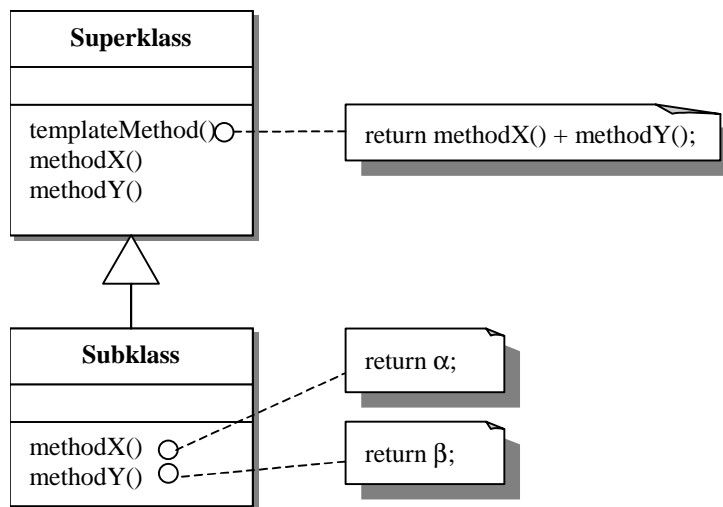
Se <http://http.mindview.net/Books/TIPatterns/> online boken "Thinking in Patterns with JAVA" (Bruce Eckel) om designmönster.

Uppgifter

Du skall studera några designmönster i mer detalj och försöka applicera dem på några små minimala (stilsierade) problem.

a) Template method

I mönstret *Template method* definierar en metod i en superklass ett algoritmskelett där detaljer "sparas" till subclasserna. Som i följande figur där en klass implementerar en template method som använder sig av andra metoder. Tips: Se Larman kapitel 34.12 och "Thinking in Patterns" kapitel 3.



Figur 1 Template method

- Det är välkänt att en engelsk gallon är 4,546 liter och att en amerikansk gallon är 3,785 liter.

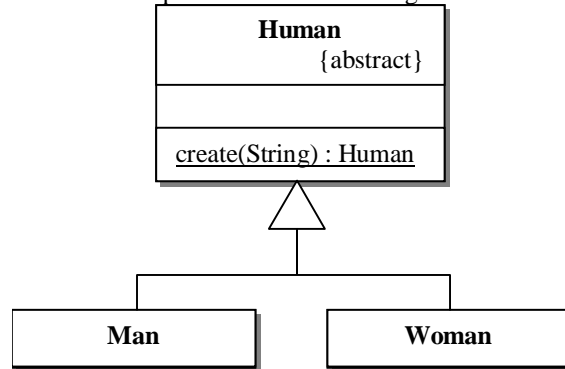
En *pint* definieras (i båda systemen) som 1/8 gallon.

Konstruera en abstrakt klass *SIConverter* som superklass till dom två konkreta klasserna *ImperialConverter* och *USConverter*.

Implementera en metod `pint()` som en template method i den abstrakta klassen. Metoden skall returnera hur många liter en pint är genom att dividera resultatet av meddelandet `gallon()` med 8. Metoden `gallon()` skall i sin tur implementeras i respektive subclass och svara med hur många liter en gallon är i det aktuella systemet. Skriv test kod först.

b) Abstract Factory

Implementera en abstrakt klass `Human` med två subklasser `Man` respektive `Woman`. Klasserna behöver inte ha några attribut eller instansmetoder, om ni inte tycker att det är nödvändigt. Tips: Se Larman kapitel 33.6 och "Thinking in Patterns" kapitel 5.

**Figur 2** Abstract Factory exempel

Använd en *abstract factory-teknik* och implementera en klassmetod enligt följande i `Human`:

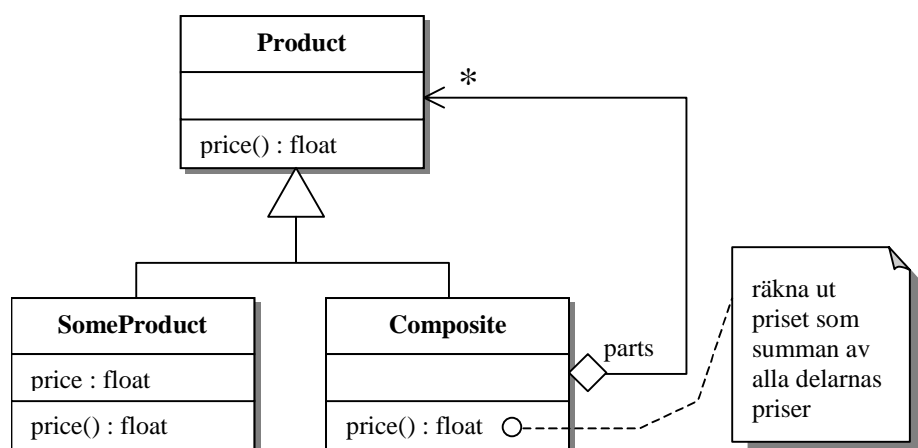
```
public static Human create(String socSecNo)
    { /* kodkropp */ }
```

Där `socSecNo` skall vara ett personnummer. Om näst sista siffran i personnumret är udda så skall en instans av `Man` returneras och annars en instans av `Woman`. Se också till att det enda sättet att instansera någon av dem konkreta klasserna är via den ovan angivna metoden. Tips: använd modifieraren `protected` (eller möjligen `package`) för konstruktören. Skriv test kod först

c) Composition

Mönstret *Composition* är också känt under namnet *Composite*. Tips: Se Larman kapitel 23.7.

Lite konkretiserat och exemplifierat ingår en *composite* i en klasshierarki som i följande illustration.

**Figur 3** Composition exempel

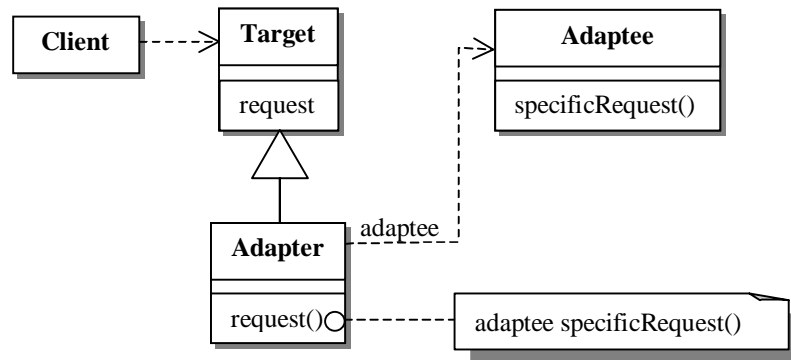
Skriv test kod med JUnit enligt denna illustration. Använd några "vanliga" produkter och en eller flera composites. Räkna ut priset på en composite enligt föreslagen "algoritm".

d) Observer

Implementera en klass `Publisher` som publicerar trycksaker och en `Subscriber` som prenumererar på dessa trycksaker. Det räcker om `Publisher` har en vektor till vilken man kan lägga till "emulerade" trycksaker i form av strängar. Använd mönstret *Observer* för att låta flera prenumeranter prenumerera på att nya trycksaker läggs in bland publicistens trycksaker. Dvs i metoden som lägger till en trycksak skall eventuella beroende objekt uppmärksammas. Tips: se Larman kapitel 23.9 och "Thinking in Patterns" kapitel 10. Vid Java-implementationen skall gränssnittet `Observer` och klassen `Observable` användas på lämpligt sätt.

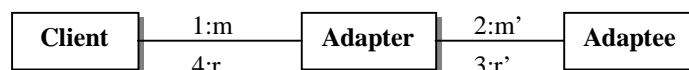
e) Adapter

En adapter är ett objekt som anpassar gränssnittet mellan två objekt till ett som passar båda. Som i Figur 4 där `client` kommunicerar med `adaptee` med hjälp av en `adapter` som anpassar `adaptee:s` gränssnitt till `target:s`. Tips: Se Larman kapitel 23.1 och "Thinking in Patterns" kapitel 7.



Figur 4 Adapter klassdiagram

Vi kan betrakta en adapter som ett filter mellan en klient och ett adapterat objekt, där adaptern översätter meddelanden och svar så att de passar båda (se Figur 5).



Figur 5 Adapter samarbetsdiagram

- Antag att vi har följande interface:

```
public interface ClientInterface {
    public int value(); // Läs värde
    public void value(int v); // Ändra värde
}
```

och

```
public interface AdapteeInterface {
    public int getValue(); // Läs värde
    public void setValue(int v); // Ändra värde
}
```

- Definiera gränssnitten.
- Konstruera två klasser som implementerar respektive gränssnitt.
- Konstruera en adapter som gör det möjligt för klienten att kommunicera med servern (dvs adaptee) med hjälp av det egna gränssnittet.

- Skriv kod som testar det hela!

f) Strategy

Använd *mönstret strategy* för att konstruera en enkel räknare till vilken man kan "skicka" ett objekt som definierar startvärde, stoppvärde och steg. Tips: Se Larman kapitel 23.6 och "Thinking in Patterns" kapitel 6. Det hela skulle sedan kunna se ut enligt följande:

```
up = new CounterUp(1, 10, 2);
down = new CounterDown(10, 0, 1);
counterUp = new MyCounter(up);
counterDown = new MyCounter(down);
counterUp.start();
counterDown.start();
```

I `MyCounter` skall den variabel som bundits till det aktuella strategiobjektet, här med namnet `counter`, kunna användas på i stil följande sätt :

```
while(!counter.atEnd())
    {System.out.println(counter.value());
    counter.next();}
```

Om ni vill får ni göra andra detaljlösningar, bara ni använder *mönstret strategy*. Använd också gärna *interface* för att beskriva hur argumentet till konstruktören `MyCounter` skall se ut. Låt sedan respektive "Counter-klass" implementera detta gränssnitt. Skriv test kod som testar det hela.

g) Personregister med Observer, Strategy och Adapter

Konstruera ett *personregister* i vilket objekt som implementerar följande gränssnitt kan läggas in:

```
interface PersonInterface {
    public String name();
    public String address();
    public String email();
}
```

- Till listan skall man kunna *lägga till* eller *ta bort* en person eller få en ut *en vektor* av alla personer som för närvarande finns i listan.
- Använd mönstret *Observer* så att objekt som är intresserade av förändringar av listan meddelas om en person läggs till eller tas bort från listan.
- Använd mönstret *Strategy* för att byta mellan olika sorteringsfunktioner som skall användas för att generera den ovan angivna vektorn av personer. Det skall åtminstone finnas tre olika sorters sorteringar (tex sortera växande efter namn, sortera fallande efter namn eller sortera växande med avseende på email).
- Slutligen använd mönstret *Adapter* för att se till att även objekt med följande gränssnitt kan läggas in i registret:

```
interface SvensktPersonInterface {
    public String namn();
    public String adress();
    public String epost();
}
```