Labb4_H20

Laboration 4 - Några designmönster

Designmönster beskriver beprövade lösningar som kan användas i flera olika, men liknande, sammanhang inom objektorienterad programmering. Med designmönster i "verktygslådan" kan designproblem lösas snabbare och säkrare än utan. Vissa mönster är enkla och lätta att lära sig medan andra är krångliga.

När man ska lära sig att använda ett nytt mönster praktiskt så är det en god idé att programmera upp mönstret på ett litet "leksaksexempel" för att verkligen förstå hur det är tänkt att fungera.

I den här labben ska några vanliga och populära mönster studeras genom att små exempel programmeras.

Mål

- Att kunna konstruera objekt enligt mönstret **Composite** och utföra enkla men karakteristiska operationer på sådana objekt.
- Att använda **Factory-teknik**: m.h.a. paket och modifierare åstadkomma en factory-metod som skapar objekt av för användaren oåtkomliga subklasser.

Mönstret Composite

Idén med mönstret Composite är att en grupp av objekt ordnade hierarkiskt i en trädstruktur ska kunna behandlas på samma sätt som enstaka objekt. En operation (*ett metodanrop*) på det sammansatta objektet ska medföra att operationen utförs på objektets alla delar. Samma anrop på ett lövobjekt påverkar endast lövet. I anropet `c.op()` ska `c` kunna referera till löv eller till ett sammansatt objekt.

Här finns en beskrivning av Composite: **www.dofactory.com/Patterns/PatternComposite.asp (https://www.dofactory.com/net/composite-design-pattern)**

Kommentar till mönsterbeskrivningen ovan: Man kan välja att i den gemensamma superklassen Component endast ha de attribut och metoder som är meningsfulla i både Leaf och Composite. Då tas alltså add(), remove() och getChild() bort från Component och implementeras bara i Composite där de går att utföra. Om t.ex. add() finns i Component så finns den också i Leaf p.g.a. arvet men add() är meningslös och t.o.m. otillåten i Leaf. Ni får välja själva hur ni vill hantera detta. Attribut som ska användas i båda subklasserna Leaf och Composite ska ovillkorligen definieras i Component.

Uppgift 1 – Resväska som Composite

En resväska kan packas enligt mönstret **Composite**.

Implementera de klasser som behövs för en resväska enligt Composite. Den klass som kallas **Component** i mönstrets vanliga beskrivning är en abstrakt klass från vilken klasserna för de enskilda prylarna ***och*** klassen eller klasserna för sammansättningarna ärver.

Skriv tre klasser: (1) En abstrakt klass, (2) en klass för löven och (3) en klass som ska vara behållare. Låt ett `String`-attribut ange vilken pryl objektet representerar. Behållarklassens prylnamn blir t.ex. *resväska*, *necessär* eller *påse* och exempel på "lövprylar" är *jeans*, *T-shirt*, *tvål* och *schampo*.

Minst tio saker ska packas och minst tre olika nivåer av sammansättningar ska användas. T.ex. kan hårspännen ligga i en påse som ligger i en necessär tillsammans med tvål och schampoo och necessären kan ligga i resväskan bredvid större klädesplagg.

Se UML-diagrammet!

Utöver namnet, låt alla komponenter (klassen **Component** i mönstret) ha en vikt som instansvariabel. Båda attributen ges värden när objekten skapas genom en vanlig konstruktor.

Låt alla komponenter ha operationerna `getWeight()` och `toString()`. `getWeight()` för en liten pryl (löv) returnerar endast prylens vikt medan `getWeight()` för en behållare returnerar hela behållarens vikt (egenvikten plus summan av vikterna av allt den innehåller). `toString()` för en liten pryl returnerar namnet medan `toString()` för en behållare ska returnera en `String` med behållarens namn följt av namnen på *alla* saker som finns i behållaren.

Skriv ett testprogram som bygger upp en resväska. Resväskans totala vikt ska beräknas i ett metodanrop, t.ex. `suitcase.getWeight()`, och skrivas ut (ingen utskrift i metoden). Tänk på att själva väskan och andra behållare har egen vikt utöver vikten för innehållet! Skriv också ut "hela resväskan". Det ska ge en lista över innehållet. Tag sen bort några komponenter ur innehållet. Skriv ut vikt och innehåll igen och kolla att det är rätt!

`getWeight()` och `toString()` ska implementeras enligt mönstret. Att de ger rätt svar räcker inte.

Det innebär att varje anrop på ett **Composite**-objekt ska gå igenom alla objektets barn.

Factory-teknik

Det finns flera designmönster som innehåller ordet *Factory*. Gemensamt för dem är att man *inte* skapar objekt på det vanliga sättet med **new** utan genom ett metodanrop. Metoderna ses som "fabriker" där objekt skapas. Vanliga namn på metoderna är `getInstance()` eller `create()`. Typen för det skapade objektet bestäms i vissa fall i metoden utom räckhåll för klienten

(användaren). Inuti fabriksmetoderna används förstås **new** men klienten, den som använder klassen, har inte direkt tillgång till **new**. Vanliga skäl för den här tekniken är att användaren ska slippa bry sig om objektets typ eller att det säkrare blir rätt typ om metoden väljer och inte användaren eller att klassen själv skall ha full kontroll över vilka objekt som skapas.

Uppgift 2 – Human factory

Implementera en abstrakt klass, `Human`, med tre konkreta subklasser: `NonBinary`, `Woman` och `Man`. I `Human` ska en factory-metod finnas :

```
public static Human create (String pnr) {
    // metodkropp
}
```

Vi förutsätter att parametern är ett nästan riktigt personnummer om tio siffror och ett streck före de fyra sista. Ni behöver inte kontrollera personnummret. Använd bara dess tionde tecken. Om näst sista tecknet är noll ('0') så returneras ett objekt av `NonBinary`, om det är en udda siffra så ska en instans av `Man` returneras, om den är jämn men inte noll så returneras ett objekt av `Woman`. Observera att det är kursledarens idé att tolka personnumren så. Folkbokföringen använder bara jämn och udda för kvinna respektive man.

Skriv ett testprogram som skapar objekt av `Nonbinary`, `Woman` och `Man` med hjälp av metoden ovan. Se till att *det inte går* att skapa objekt av `Nonbinary`, `Woman` eller `Man` på annat sätt än genom metoden ovan. Det ska alltså inte gå att kompilera `new NonBinary(...)` o.s.v. i testprogrammet. Det får inte heller gå att skapa en anonym subklass till `Human`. Koden

`Human h = new Human(){};` // Skapar ett objekt av subklass till Human, klasskroppen är tom.

ska inte heller kompilera i testprogrammet.

Tips: Gör ett paket `human` som innehåller `Human` och dess subklasser. Paketet läggs på en underkatalog med samma namn. Med hjälp av lämpliga modifierare för synlighet (public, private m.fl.) kan man åstadkomma den begärda strukturen. Lägg testprogrammet på katalognivån ovanför paketet. Se också till att testprogrammet bekräftar att objekt av rätt typ skapats genom att metoden `toString()` skriver ut information om objekten, t.ex. enligt nedan.

```
Human billie = Human.create("Billie", "xxxxxx-560x");
Human anna = Human.create("Anna", "xxxxxx-642x");
Human magnus = Human.create("Magnus","xxxxxx-011x");
System.out.println(billie);
System.out.println(anna);
System.out.println(magnus);
```

kan ge utskriften

```
Jag är icke-binär och heter Billie
Jag är kvinna och heter Anna
Jag är man och heter Magnus
```

Redovisning och krav på programmen

- Implementera **Composite**-mönstret enligt labblydelsen.
- Demonstrera körning av ett testprogram där minst tio saker i minst tre nivåer bygger upp en **Composite**. Skriv ut totalvikt och innehåll. Tag bort några saker ut väskan. Skriv ut totalvikt och innehåll igen.
- UML-klassdiagram behövs *inte* om det följer det som länkas till ovan. Ni får ha andra namn på era klasser utan att behöva rita om. Om ni har färre eller fler klasser så rita eget!
- Rita ett *objektdiagram* över de objekt som testprogrammet skapar. Diagrammet behöver inte följa UML-standard men tydligt visa vilka objekt som finns. Rita diagrammet som ett träd!
- Demonstrera en körning av Human-factory-programmet där objekt av de tre subklasserna skapas.
- Demonstrera en annan version av programmet där man försöker med `new NonBinary(...)`, `o.s.v.` och `new Human(){}` men kompilering misslyckas.

Extrauppgift - Mönstret Iterator

Observera: Alla som vill ha betyg A på labbkursen måste göra denna extrauppgift.

Syftet med mönstret är att iterera över eller gå igenom alla element i en sammansatt struktur utan att behöva befatta sig med den underliggande strukuren. Att använda en iterator på en lista är naturligt och implementationen av en listiterator är ofta enkel.

Även för mer komplicerade strukturer har man god nytta av iteratorer. Composite-sammansättningar och andra trädstrukturer kan mycket väl förses med iteratorer. Iteratorn "levererar" elementen i sekvens utan att man behöver veta hur strukturen är uppbyggd. "Man" är här den som använder iteratorn. Den som programmerar iteratorn måste förstås känna till strukturen. Naturliga val för genomgångar av en trädstruktur är preorder, inorder, postorder eller den ordning som gäller vid bredden-först-sökning.

Mer om iteratormönstret finns här: **www.dofactory.com/Patterns/PatternIterator.asp (https://www.dofactory.com/net/iterator-design-pattern)**
Läs även i Java-API:n om **`Iterable`** (http://docs.oracle.com/javase/7/docs/api/java/lang/Iterable.html), **`Iterator`** (http://docs.oracle.com/javase/7/docs/api/java/util/Iterator.html) samt om *enhanced for statement* i följande tutorial: **The for statement** (http://docs.oracle.com/javase/tutorial/java/nutsandbolts/for.html).

Uppgifter: Skriv två iteratorer som går igenom Composite-trädstrukturen. En iterator ska använda ordningen som ges av bredden-först-sökning och den andra iteratorn ska använda preorder trädgenomgång (ordningen i djupet-först-sökning). När iteratorerna demonstreras ska namnet i varje trädnod skrivas ut så att man kan kontrollera att ordningen är rätt. Använd då ***inte*** den toString()-metod som skrivits i grunduppgiften då den ger hela trädets innehåll. Skriv en metod med annat namn som returnerar nodens namn eller byt ut ***toString()*** för den här uppgiften.

Bredden-först-iterator

Besök först roten, därefter rotens alla barn, sen rotens barnbarn o.s.v.

Låt iteratorn implementera interfacet som länkas till ovan och som finns i paketet `java.util`. Det visas här nedanför också. E står här för typen av element som levereras av iteratorn Här ska det vara `Component` eller motsvarande klass.

```
public interface Iterator<E> {
    E next();           // gå fram ett steg och returnera nästa element
    boolean hasNext();  // returnera true om det finns fler element att besöka
    void remove();      // krävs ej i labben, implementeras tom
}
```

Låt **Composite**-klassen implementera `Iterable` (läs ovan) så att den har metoden `iterator()`. Definiera `iterator()` så att den returnerar ett iterator-objekt. Med detta upplägg blir det möjligt att gå igenom **Composite**-strukturen *med en for-each*-sats såväl som med iteratorns metoder. Visa båda i testprogrammet!

Djupet-först-iterator

Implementera även en djupet-först-iterator.

Hantering av två iteratorer till samma klass

Interfacet `Iterable` förutsätter att det finns en enda iterator och denna ska returneras vid anrop av interfacets metod iterator(). Ni ska demonstrera två iteratorer. Det duger att hantera detta genom att ha två satser som skapar objekt av olika iteratorklasser men alltid ha den ena bortkommenterad. Tänk gärna ut en lösning som inte kräver omkompilering men det krävs inte.

Förenklad iterator

Vi förutsätter att trädstrukturen är intakt under en iterators livslängd. Det gör implementering av iteratorn mycket enklare än om vi skulle tillåta borttagning (remove() i iteratorn) eller tillägg (add() i Composite) under iteratorgenomgången. Naturligtvis är det tillåtet att implementera en mer avancerad iterator som klarar att strukturen ändras under genomgången.

Redovisning

Demonstrering av iteratorerna går bra att göra i samma program som demonstrerar Composite-strukuren då labbens första del redovisas. Visa både genomgång med anrop av iteratorns metoder och genomgång med for-each-satsen.