Resväska som *Composite* i Java

Lisa Dahl och Mostafa Shihadeh

November 19, 2025

Contents

1 Introduktion

Vi vill implementera uppgift 1 (resväska enligt *Composite*-mönstret) med litterär programmering. Vi bygger tre klasser: `Component` (abstrakt), `Leaf` (pryl) och `Composite` (behållare), samt ett testprogram `Client`.

I extrauppgiften X4 lägger vi även till två iteratorer för vår Composite-struktur: en som går *bredden-först* (BFS) och en som går *preorder* (djupet-först). Då kan vi gå igenom alla noder med en for-each-sats eller med iteratorns `hasNext`/`next`-metoder utan att bry oss om hur trädet är uppbyggt.

1.1 Designval

- `Component` bär gemensamma attribut: `name` och `weight` (egen vikt). Dessa lagras som *private final*-fält och kan läsas via getters.
- `Leaf` representerar en enskild pryl. `getWeight()` returnerar bara dess egen vikt.
- `Composite` representerar en behållare med barn. `getWeight()` summerar behållarens egen vikt och alla barns vikter. `toString()` traverserar rekursivt.
- Metoderna `add` och `remove` finns bara i `Composite` (inte i `Component`).
- I X4 låter vi `Composite` implementera `Iterable<Component>` och skriver två iterator-klasser: `BreadthFirstIterator` (BFS) och `PreorderIterator` (DFS).

1.2 Bygga och kompilera

Vi vill skriva en byggfil för GNU Make. De maskin-genererade reglerna läggs i `Suitcase.mk` som tanglas ur denna `.nw`-fil och sedan *inkluderas* från toppens `Makefile`.

Först lägger vi grundmålen och PDF-reglerna:

⟨*Suitcase.mk*⟩≡

```
TARGETS= Suitcase.pdf Suitcase.mk
all: classes Suitcase.pdf

Suitcase.pdf:  Suitcase.tex
  pdflatex -interaction=nonstopmode -halt-on-error Suitcase.tex
  pdflatex -interaction=nonstopmode -halt-on-error Suitcase.tex

Suitcase.tex: Suitcase.nw
  noweave -latex Suitcase.nw > Suitcase.tex
```

Därefter tanglar vi ut varje `.java`-fil ur den litterära källan (samt bäddar in radmarkörer så att `noerr.pl` kan mappa fel till `.nw`):

⟨*Suitcase.mk*⟩+≡

```
Component.java: Suitcase.nw
  notangle -L'//line %L "%F"%N' -RComponent.java Suitcase.nw > Component.java

Leaf.java: Suitcase.nw
  notangle -L'//line %L "%F"%N' -RLeaf.java Suitcase.nw > Leaf.java

Composite.java: Suitcase.nw
  notangle -L'//line %L "%F"%N' -RComposite.java Suitcase.nw > Composite.java

Client.java: Suitcase.nw
  notangle -L'//line %L "%F"%N' -RClient.java Suitcase.nw > Client.java

BreadthFirstIterator.java: Suitcase.nw
  notangle -L'//line %L "%F"%N' -RBreadthFirstIterator.java Suitcase.nw > BreadthFirstI

PreorderIterator.java: Suitcase.nw
  notangle -L'//line %L "%F"%N' -RPreorderIterator.java Suitcase.nw > PreorderIterator
```

Nu lägger vi sektionen för att kompilera och köra Java:

⟨*Suitcase.mk*⟩+≡

```
.PHONY: classes run clean-Suitcase
classes: Component.java Leaf.java Composite.java Client.java BreadthFirstIterator.java
  @if [ -x ./noerr.pl ]; then ./noerr.pl javac *.java; else javac *.java; fi

run: classes
  java Client
```

Slutligen städreglerna:

⟨*Suitcase.mk*⟩+≡

```
clean: clean-Suitcase
clean-Suitcase:
  rm -f Suitcase.tex Suitcase.aux Suitcase.log Suitcase.toc
  rm -f *.class *.java
```

2 Kod

I det här avsnittet definierar vi Javafilerna som tänglas ut från denna `.nw`-fil.

2.1 `Klassen Component`

Ansvar: bas-klass med namn och egenvikt samt abstrakta metoder.

Filen `Component.java` innehåller definitionen av klassen `Component`. Översiktligt ser den ut så här:

⟨*Component.java*⟩≡
```
public abstract class Component {
  ⟨Component attributes⟩
  ⟨Component constructor⟩
  ⟨Component methods⟩
}
```

2.1.1 Attribut

Vi vill att `Component` ska ha följande attribut:

- `name`: namn på komponenten (sträng)
- `weight`: egen vikt i kg (flyttal)

Dessa attribut ska inte kunna ändras efter att objektet skapats. Vi markerar dem därför som `private final` och ger enkla getters.

⟨*Component attributes*⟩≡
```
private final String name;
private final double weight;
```

2.1.2 Konstruktor

Konstruktorn tar emot namn och vikt och lagrar dem.

⟨*Component constructor*⟩≡
```
Component(String name, double weight) {
  this.name = name;
  this.weight = weight;
}
```

2.1.3 Metoder

I `Component` deklarerar vi getters för namnet och egenvikten, samt de abstrakta metoderna `getWeight` och `toString` som alla subklasser måste implementera:

- `getName`: returnerar komponentens namn.
- `getOwnWeight`: returnerar komponentens egen vikt (utan barn).
- `getWeight`: ska returnera totalvikten (inklusive barns vikt).
- `toString`: ska returnera en strängrepresentation av komponenten.

⟨*Component methods*⟩≡
```
public String getName() {
  return name;
}

public double getOwnWeight() {
  return weight;
}

public abstract double getWeight();
public abstract String toString();
```

2.2 Leaf

Ansvar: en enskild pryl att packa.

Filen `Leaf.java` innehåller definitionen av klassen `Leaf`. Översiktligt ser den ut så här:

⟨*Leaf.java*⟩≡
```
public class Leaf extends Component {
  ⟨Leaf constructor⟩
  ⟨Leaf methods⟩
}
```

2.2.1 Konstruktor

Konstruktorn tar namn och egen vikt och skickar vidare till basklassen `Component`.

⟨*Leaf constructor*⟩≡
```
public Leaf(String name, double weight) {
  super(name, weight);
}
```

2.2.2 Metoder

Totalvikten för ett löv är samma som dess egen vikt. `toString` beskriver prylen.

⟨*Leaf methods*⟩≡
```
@Override
public double getWeight() {
  return getOwnWeight();
}

@Override
public String toString() {
  return getName() + " (" + getOwnWeight() + ")";
}
```

2.3 `Composite`

Ansvar: en behållare som kan innehålla andra `Component`. Egen vikt *plus* alla barns vikter utgör totalvikten. I X4 gör vi den också itererbar så att vi kan gå igenom trädet med iteratorer.

Filen `Composite.java` innehåller definitionen av klassen `Composite`. Översiktligt ser den ut så här:

⟨*Composite.java*⟩≡
```
import java.util.*;

public class Composite extends Component implements Iterable<Component> {
  ⟨Composite fields⟩
  ⟨Composite constructor⟩
  ⟨Composite methods⟩
  ⟨Composite iterator methods⟩
}
```

2.3.1 Fält

Vi lagrar barn i en muterbar lista av `Component`.

⟨*Composite fields*⟩≡
```
private final List<Component> children = new ArrayList<>();
```

2.3.2 Konstruktor

Konstruktorn tar behållarens namn och egen vikt.

⟨*Composite constructor*⟩≡
```
public Composite(String name, double weight) {
  super(name, weight);
}
```

2.3.3 Metoder

Barnhantering (`add/remove/getChild/getChildren`) Vi kan lägga till och ta bort barn, och hämta ett barn via index eller läsa alla barn.

⟨*Composite methods*⟩≡
```
public void add(Component component) {
  children.add(component);
}

public void remove(Component component) {
  children.remove(component);
}

public Component getChild(int index) {
  return children.get(index);
}

public List<Component> getChildren() {
  return Collections.unmodifiableList(children);
}
```

Totalvikt (`getWeight`) Totalvikt = behållarens egen vikt + summan av alla barns totalvikter (rekursivt). Varje anrop på ett `Composite`-objekt går igenom dess barn och anropar deras `getWeight()` enligt mönstret.

⟨*Composite methods*⟩+≡
```
@Override
public double getWeight() {
  double totalWeight = getOwnWeight();
  for (Component child : children) {
    totalWeight += child.getWeight();
  }
  return totalWeight;
}
```

`toString` Vi bygger en rekursiv beskrivning där behållarens namn och egen vikt visas, och alla barn listas inom hakparenteser. Även här anropar vi barnens `toString()` enligt mönstret.

⟨*Composite methods*⟩+≡
```
@Override
public String toString() {
  StringBuilder sb = new StringBuilder();
  sb.append(getName()).append(" (").append(getOwnWeight()).append(") [");
  for (int i = 0; i < children.size(); i++) {
    sb.append(children.get(i).toString());
    if (i < children.size() - 1) {
      sb.append(", ");
    }
  }
  sb.append("]");

  return sb.toString();
}
```

Iterator-metoder (X4) `Composite` implementerar `Iterable<Component>`. Defaultiteratorn är bredden-först. Vi lägger också till metoder för att uttryckligen få BFS respektive preorder.

⟨*Composite iterator methods*⟩≡
```
@Override
public Iterator<Component> iterator() {
  // default: bredden-först
  return new BreadthFirstIterator(this);
}

public Iterator<Component> breadthFirstIterator() {
  return new BreadthFirstIterator(this);
}

public Iterator<Component> preorderIterator() {
  return new PreorderIterator(this);
}
```

2.4 BreadthFirstIterator (X4)

Ansvar: gå igenom trädet i bredden-först-ordning: rot, alla barn, alla barnbarn, osv.

Filen `BreadthFirstIterator.java` innehåller iteratorn:

⟨*BreadthFirstIterator.java*⟩≡

```
import java.util.ArrayDeque;
import java.util.Deque;
import java.util.Iterator;
import java.util.NoSuchElementException;

public class BreadthFirstIterator implements Iterator<Component> {
  private final Deque<Component> queue = new ArrayDeque<>();

  // starta med roten i kön
  public BreadthFirstIterator(Component root) {
    queue.add(root);
  }

  @Override
  public boolean hasNext() {
    return !queue.isEmpty();
  }

  @Override
  public Component next() {
    if (!hasNext()) {
      throw new NoSuchElementException();
    }
    Component current = queue.removeFirst();
    if (current instanceof Composite) {
      for (Component child : ((Composite) current).getChildren()) {
        queue.addLast(child);
      }
    }
    return current;
  }

  @Override
  public void remove() {
    // används inte i den här labben
  }
}
```

2.5 `PreorderIterator` (X4)

Ansvar: gå igenom trädet i preorder (djupet-först): besök noden, sedan rekursivt dess barn från vänster till höger.

Filen `PreorderIterator.java` innehåller iteratorn:

⟨*PreorderIterator.java*⟩≡

```
import java.util.ArrayDeque;
import java.util.Deque;
import java.util.Iterator;
import java.util.List;
import java.util.NoSuchElementException;

public class PreorderIterator implements Iterator<Component> {
  private final Deque<Component> stack = new ArrayDeque<>();

  // börja med roten överst på stacken
  public PreorderIterator(Component root) {
    stack.push(root);
  }

  @Override
  public boolean hasNext() {
    return !stack.isEmpty();
  }

  @Override
  public Component next() {
    if (!hasNext()) {
      throw new NoSuchElementException();
    }
    Component current = stack.pop();
    if (current instanceof Composite) {
      List<Component> children = ((Composite) current).getChildren();
      // lägg barnen i omvänd ordning så att första barnet kommer överst
      for (int i = children.size() - 1; i >= 0; i--) {
        stack.push(children.get(i));
      }
    }
    return current;
  }

  @Override
  public void remove() {
    // används inte i den här labben
  }
}
```

2.6 Client

Ansvar: bygga en resväska med minst tre nivåer och minst tio prylar, skriva ut totalvikt och innehåll, ta bort några objekt och skriva ut igen. I X4 visar vi också hur vi går igenom trädet med både for-each och explicit iterator.

Filen `Client.java` innehåller testprogrammet. Översiktligt ser det ut så här:

⟨*Client.java*⟩≡
```
public class Client {
  public static void main(String[] args) {
    ⟨Client build-structure⟩
    ⟨Client print⟩
    ⟨Client iterate-bfs⟩
    ⟨Client iterate-preorder⟩
  }
}
```

2.6.1 Bygg upp strukturen (suitcase)

Vi skapar roten (resväskan), underbehållare och löv, som är kläder eller accessoarer.

⟨*Client build-structure*⟩≡

```
// Roten: själva resväskan
Composite suitcase = new Composite("Resväska", 2.3);

// Necessär (behållare) med egen vikt 0.12 kg och innehåll
Composite necessar = new Composite("Necessär", 0.12);
necessar.add(new Leaf("Tvål",        0.09));
necessar.add(new Leaf("Schampo",     0.22));
necessar.add(new Leaf("Tandborste",  0.03));
necessar.add(new Leaf("Tandkräm",    0.11));

// Påse i necessären (tredje nivån)
Composite pase = new Composite("Påse", 0.01);
pase.add(new Leaf("Hårspännen (10 st)", 0.02));
necessar.add(pase);

// Mindre väska (nivå två)
Composite techbag = new Composite("Tech-väska", 0.20);
techbag.add(new Leaf("Laddare", 0.15));

// Packa allt i resväskan
suitcase.add(new Leaf("T-shirt vit",   0.18));
suitcase.add(new Leaf("T-shirt svart", 0.19));
suitcase.add(new Leaf("Jeans",         0.75));
suitcase.add(new Leaf("Chinos",        0.55));
suitcase.add(new Leaf("Pocketbok",     0.28));
suitcase.add(necessar);
suitcase.add(techbag);
```

2.6.2 Skriv ut totalvikt och innehåll (före borttagning)

⟨*Client print*⟩≡

```
System.out.printf("Totalvikt före borttagning: %.2f kg%n", suitcase.getWeight());
System.out.println("Innehåll före borttagning:");
System.out.println(suitcase.toString());
```

2.6.3 X4: Traversal med iteratorer

För att kontrollera ordningen skriver vi bara ut nodernas namn med `getName()`.

⟨*Client iterate-bfs*⟩≡
```
System.out.println("Bredden-först (for-each, default iterator):");
for (Component c : suitcase) {
  System.out.print(c.getName() + " ");
}
System.out.println();
```

⟨*Client iterate-preorder*⟩≡
```
System.out.println("Preorder (djupet-först, explicit iterator):");
java.util.Iterator<Component> it = new PreorderIterator(suitcase);
while (it.hasNext()) {
  System.out.print(it.next().getName() + " ");
}
System.out.println();
```