

# Laboration 4

Casper Beckman

December 31, 2025

## Contents

### 1 Uppgift 1 (och extrauppgiften)

Vi har fått i uppgift att implementera Composite-mönstret. Vad detta är kan ses i uppgiftslydelsen. Vi skriver först ett interface för `Component`, vilken kommer bli implementerad av både `Leaf` och `Composite`.

```
<Component.java>≡
public interface Component {
    String toString();
    double getWeight();
    String getName();
}
```

Här är `getName` endast relevant för extrauppgiften. Notera att vi ej har deklarerat funktionerna som endast skulle användas i `Component`. Vi implementerar dessa i `AbstractComponent.java`

```
(AbstractComponent.java)≡
import java.util.ArrayList;
public class AbstractComponent implements Component {
    protected double weight;
    protected String name;
    protected ArrayList <Component> children; // this is always empty for leaf nodes

    public double getWeight() {
        (AbstractComponent getWeight)
    }

    public String toString() {
        (AbstractComponent toString)
    }
    public String getName() {
        return name;
    }
}
```

Vi gör nu implementationen av `Leaf.java`. Det enda vi behöver från denna är en konstruktor som tar vikt och namn av vad vi nu packar:

```
(Leaf.java)≡
public class Leaf extends AbstractComponent {
    public Leaf(double weight, String name) {
        this.weight=weight;
        this.name=name;
    }
}
```

Composite-klassen blir något mer avancerad. Här behöver vi även initiera våra children ty dessa kan ha barn. Dessutom behöver vi implementera `add` och `remove`. Ignorera implementationen av `Iterable` tillsvidare, vi återkommer till den i extrauppgiften.

```
⟨Composite.java⟩≡
import java.util.ArrayList;
import java.util.Iterator;

public class Composite extends AbstractComponent implements Iterable<Component>{
    public Composite(double weight, String name) {
        this.weight = weight;
        this.name = name;
        children = new ArrayList<>();
    }

    public void add(Component C) {
        children.add(C);
    }

    public void remove(Component c) {
        children.remove(c);
        for (Component t : children) {
            if (t instanceof Composite) {
                ((Composite)t).remove(c);
            }
        }
    }

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

    public Iterator<Component> iterator() {
        ⟨Composite iterator⟩
    }
}
```

Vi noterar att vi vill att alla komponenter som innehålls i en `Composite` ska vara med i `getWeight` respektive `toString`, därmed är det naturligt att implementera dessa rekursivt.

```
<AbstractComponent getWeight>≡
    double sum = weight;
    if (children != null) {
        for (Component c : children) {
            sum+=c.getWeight();
        }
    }
    return sum;
```

Notera att denna implementation ger en ”trailing newline”, dock påverkar detta inte oss i vår implementation därmed bryr vi oss inte om att göra en mer korrekt implementation.

```
<AbstractComponent toString>≡
    StringBuilder sb = new StringBuilder();
    sb.append(name).append("\n");
    if (children != null) {
        for (Component c : children) {
            sb.append(c.toString());
        }
    }
    return sb.toString();
```

Vi återkommer till klienten (dvs `Main.java`) senare. Vi går nu igenom de olika iteratorna. I vår `Composite` har vi:

```
<Composite iterator>≡
    return new BFIIterator(this);
    //return new DFIIterator(this);
```

där `BFIterator` är breadth-first-iterator och `DFIterator` är depth-first-iterator. Vi har implementationer för dessa i `BFIterator.java` respektive `DFIterator.java`. Vi börjar med implementationen av `BFIterator`. Vi vill använda oss av en kö-struktur för att enklast kunna implementera detta. För detta används Javas egna länkade lista. Vi bygger upp kön vid initiering så att elementen ges i breadth-first ordning när vi sedan använder `pop` på listan. Som kan ses i koden görs detta rekursivt.

```
<BFIterator.java>≡
import java.util.Iterator;
import java.util.LinkedList;
public class BFIterator implements Iterator<Component>{
    private LinkedList<Component> queue;
    public BFIterator(Composite composite) {
        queue = new LinkedList<Component>();
        queue.add(composite);
        queueBuilder(composite);
    }

    private void queueBuilder(Composite currentNode) {
        for (int i = 0; i < currentNode.children.size(); i++) {
            queue.add(currentNode.getChildAt(i));
        }

        for (int i = 0; i < currentNode.children.size(); i++) {
            Component maybeComposite = currentNode.getChildAt(i);
            if (maybeComposite instanceof Composite) {
                queueBuilder((Composite) maybeComposite);
            }
        }
    }

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

    public Component next() {
        return queue.pop();
    }
}
```

Metodiken i `DFIterator.java` är analog, fast konstruktionen är för en depth-first-iterering:

```
<DFIterator.java>≡
import java.util.Iterator;
import java.util.LinkedList;
public class DFIterator implements Iterator<Component>{
    private LinkedList<Component> queue;

    // build the queue
    public DFIterator(Composite composite) {
        queue = new LinkedList<Component>();
        queue.add(composite);
        queueBuilder(composite);
    }

    private void queueBuilder(Composite currentNode) {
        for (Component c : currentNode.children) {
            queue.add(c);
            if (c instanceof Composite) {
                queueBuilder((Composite) c);
            }
        }
    }

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

    public Component next() {
        return queue.pop();
    }
}
```

Vi redovisar då klientkoden även för extrauppgiften samtidigt. Vi vill skapa ett antal lådor vi lägger i varandra, vi har som krav djup tre. Vi ska ha minst 10 objekt totalt och vi ska kontrollera att `remove` fungerar som den ska. Detta görs genom följande implementation:

```
<Main.java>≡
public class Main { // first assignment (and X4), see Factory for assignment two
    public static void main(String[] args) {
        Composite suitcase = new Composite(3000, "suitcase");
        Composite mediumBag = new Composite(1000,"medium bag");
        Composite smallBag = new Composite(500, "small bag");
        Leaf superImportantThing = new Leaf(50, "super important thing");
        suitcase.add(mediumBag);
        mediumBag.add(smallBag);
        smallBag.add(superImportantThing);

        Composite[] bags = {suitcase, mediumBag, smallBag};

        for (int i=0; i < 10; i++) { // expensive to bring weights on a flight
            bags[i%3].add(new Leaf((double) (i+1)*100, (i+1)+" kg weight"));
        }
        System.out.println("weight of suitcase: "+suitcase.getWeight());
        System.out.println("things:\n"+suitcase);
        suitcase.remove(smallBag); // check that lower depth stuff gets removed
        System.out.println();
        System.out.println("weight of suitcase: "+suitcase.getWeight());
        System.out.println("things:\n"+suitcase);

        System.out.println("\n\n");
        for (Component c : suitcase) {
            System.out.println(c.getName());
        }
    }
}
```

## 2 Uppgift 2

Vi ska här använda ett Factory-designmönster. Vi vill gömma alla konstruktors, därmed skapar vi alla filer som vår fil med `main`-metoden beror på i ett paket som vi sedan importerar. Notera att om vi inte skriver `public`, `private` eller `protected` så har vår metod (i detta fall `Human`) endast scope i paketet. Först skapar vi människan:

```
(human/Human.java)≡
package human;

public abstract class Human {
    String pnr;
    Human(){}
    public static Human create(String pnr) {
        if (pnr.charAt(9)=='0') {
            return new NonBinary(pnr);
        }
        else if (pnr.charAt(9)%2 == 0) {
            return new Man(pnr);
        }
        else {
            return new Woman(pnr);
        }
    }
}
```

Sedan skapar vi alla klasser som ärver denna den abstrakta människan.

```
(human/Man.java)≡
package human;
public class Man extends Human {
    protected Man(String pnr) {
        this.pnr=pnr;
    }
}
```

```
(human/NonBinary.java)≡
package human;
public class NonBinary extends Human {
    protected NonBinary(String pnr) {
        this.pnr=pnr;
    }
}
```

```
{human/Woman.java}≡
package human;
public class Woman extends Human {
    protected Woman(String pnr) {
        this.pnr=pnr;
    }
}
```

Vi testar att detta fungerar i `Factory.java`, notera att vi får compile-time-errors om vi avkommenterar kommentarerna. Om assertsen avklaras så skapas våra människor korrekt.

```
{Factory.java}≡
import static org.junit.Assert.*;
import human.*;

public class Factory { // second assignment
    public static void main(String[] args) {
        // Human h = new Human();{}; // this throws compilation error
        // Human m = new Man("this is not visible");

        Human billie = Human.create("1234567890");
        Human bob = Human.create("1234567892");
        Human bella = Human.create("1234567891");

        assertTrue(billie instanceof NonBinary);
        assertTrue(bob instanceof Man);
        assertTrue(bella instanceof Woman);

        System.out.println("all are correct instances");
    }
}
```