Design Patterns

En kort introduktion

Vad är ett designmönster?

- Ett designmönster är en (namngiven) lösning på ett i en viss omgivning ofta återkommande problem.
- I mjukvaru-världen: En uppsättning klasser med givna relationer och operationer som löser ett återkommande problem.

Först: Önskvärt i OO kod

- Klasser ska ha hög kohesion - en och endast en roll
- Låg koppling mellan moduler/klasser – beroendet mellan klasser ska vara minimalt, en ändring i en klass ska ha så liten påverkan som möjligt på omgivande klasser
- Programmera mot gränssnitt, inte mot implementationer

Inledande exempel: Ett enda objekt, tillgängligt överallt

Objekträknare, dålig lösning

```
public class QueueTicketDispenser {
    private static int objectCount = 0;
    private int nr;

    public QueueTicketDispenser() throws Exception {
      if(objectCount > 0) throw new Exception();
      nr = 1;
      objectCount++;
    }

    public int getNextNumber() {
      return nr++;
    }
}
```

Ett enda objekt, tillgängligt överallt

- Problem med lösningen (objekträknare)?

- Om vi försöker skapa fler objekt upptäcks detta först vid exekvering (Exception, run-time)
- Skulle referensen försvinna kan vi inte återskapa objektet, eller inte längre komma åt det objekt som finns

Ett enda objekt, tillgängligt överallt

static data, bättre?

```
public class QueueTicketDispenser {
   private static int nr = 1;

   public static int getNextNumber() {
       return nr++;
   }
}
```

Ett enda objekt, tillgängligt överallt

- Problem med lösningen (static data)?

- static → datat initieras vid programstart, oavsett om det behövs eller ej
- Ej riktigt objekt, d.v.s. ej tillgång till mekanismer för objekt

Ett enda objekt, tillgängligt överallt

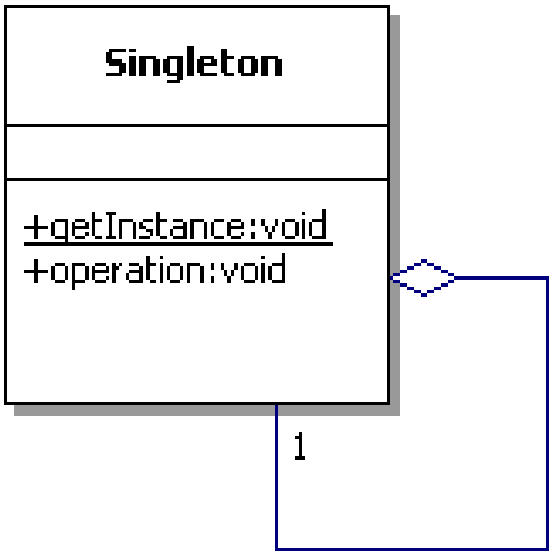
Bästa lösningen (?)

```
public class QueueTicketDispenser {
   private int nr;
   private static QueueTicketDispenser theObject;

   public static QueueTicketDispenser getInstance() {
        if(theObject == null)
                theObject = new QueueTicketDispenser();
        return theObject;
   }
   private QueueTicketDispenser() {
        nr = 1;
   }
   public int getNextNumber() {
        return nr++;
   }
}
```

Ett enda objekt, tillgängligt överallt - Singleton

Designmönstret Singleton

- Endast ett objekt
- Objektet skapas först när det behövs
- Lösningen ej trådsäker



- Hackerns favoritmönster. Risk för sjukdomen ”Singeltonit”.

Typer av designmönster

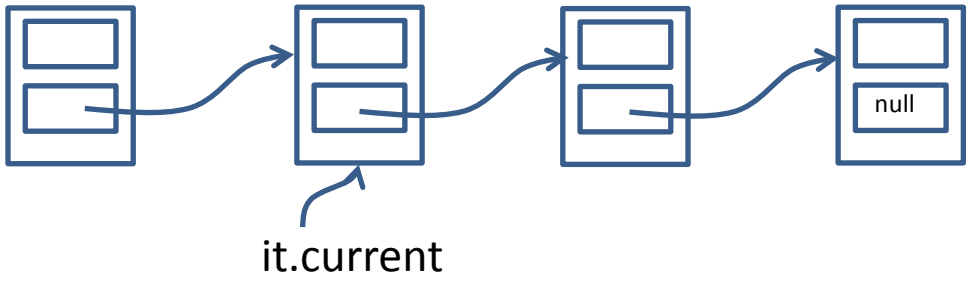
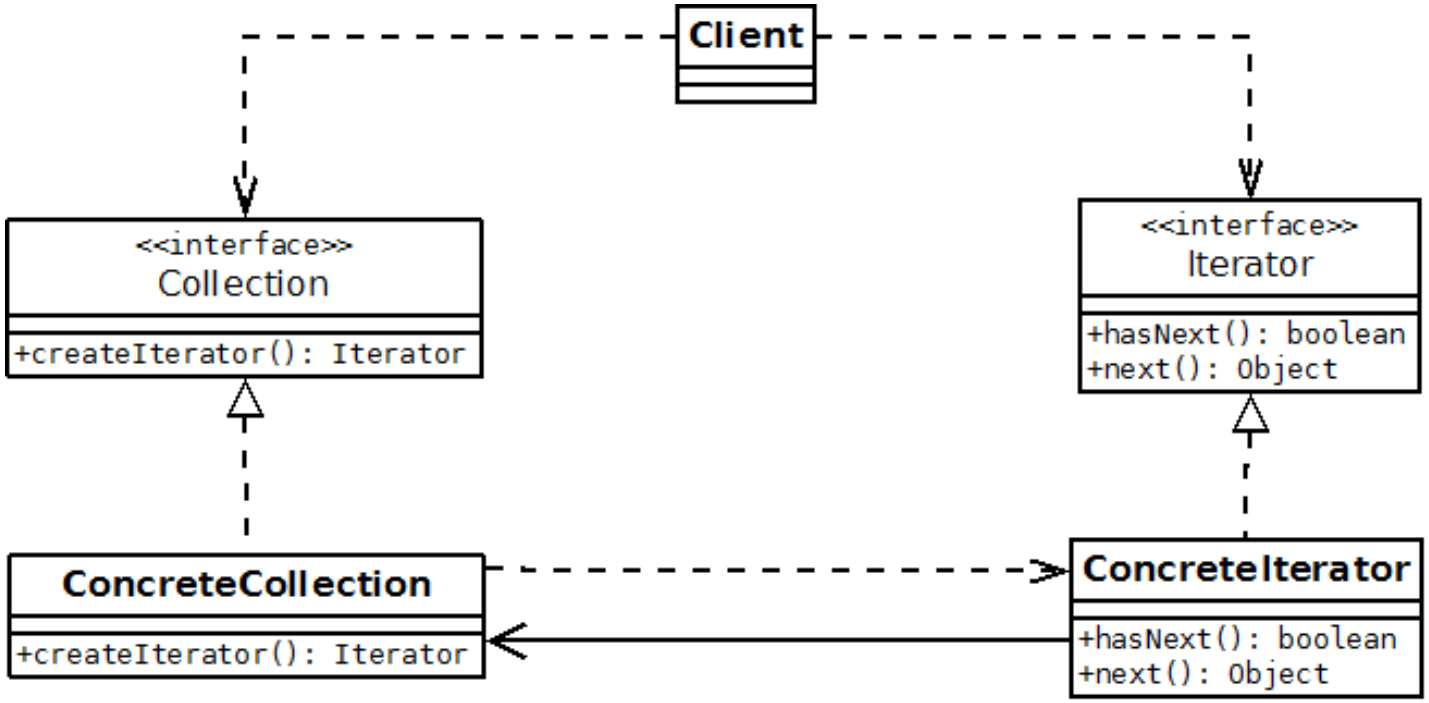
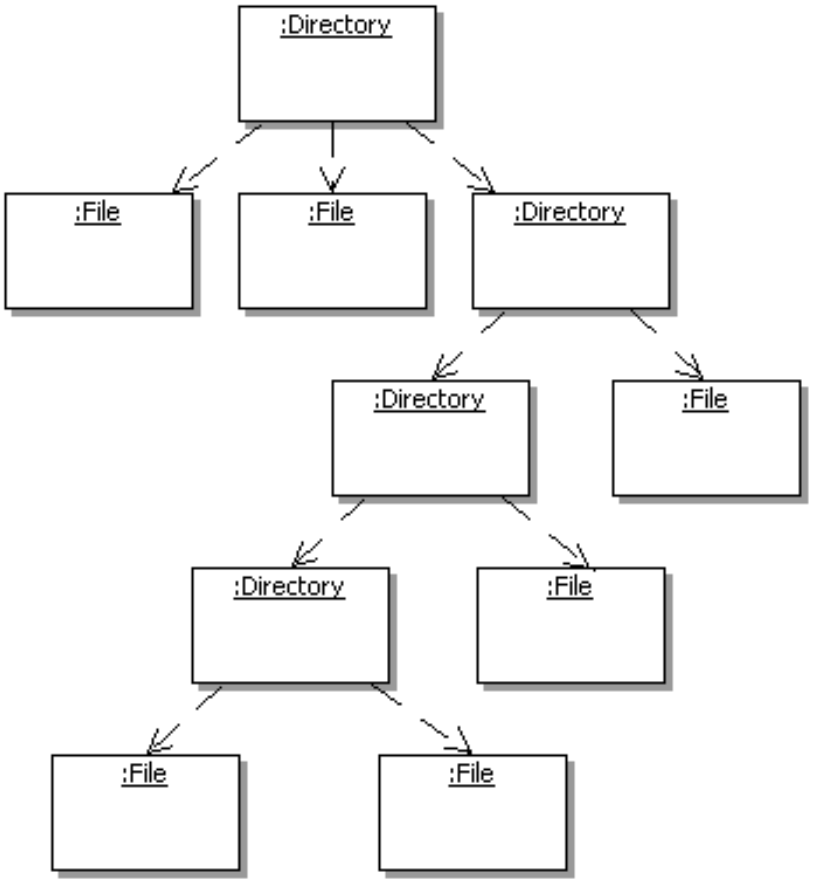
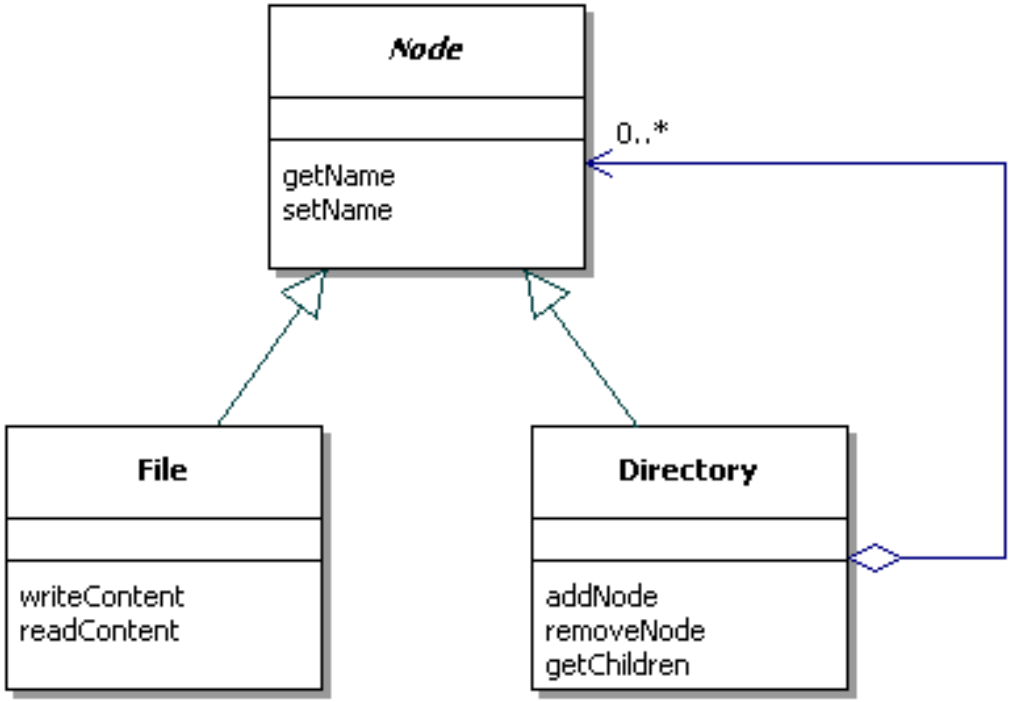
Enligt ”Gang of Four”:

- Skapelsemönster
- Strukturmönster
- Beteendemönster

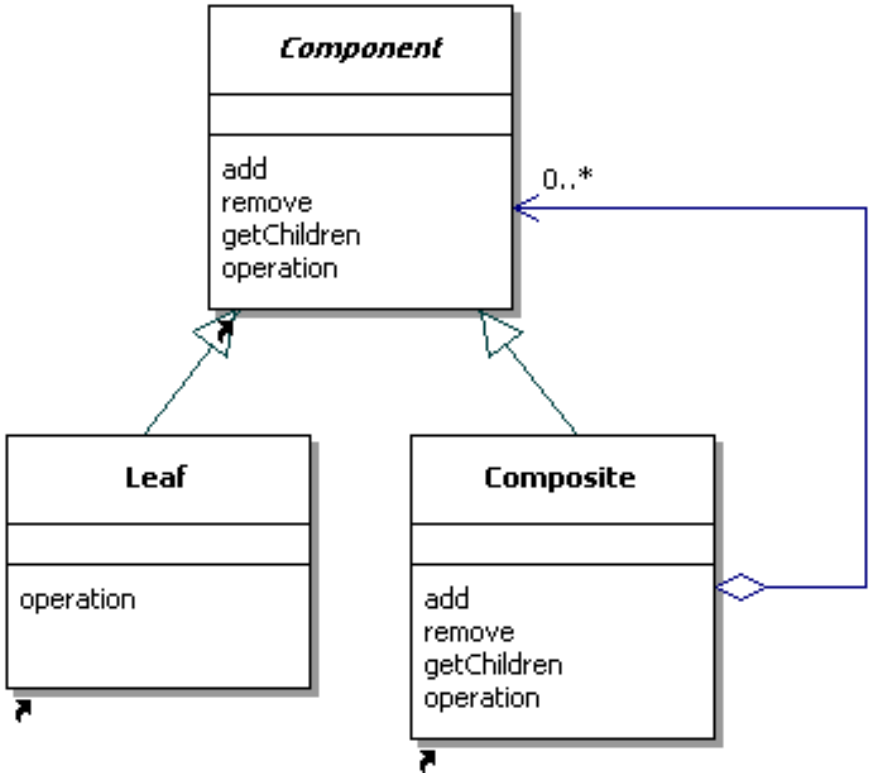
Iterator

- Att löpa igenom objekten i en behållare på ett av behållaren oberoende sätt (?)

- `List list = new ArrayList() / newLinkedList();`
- ```
  Iterator it = list.iterator();
  while(it.hasNext()) {
      Object o = it.next();
      …;
  }
  ```

- Javas ”for each”-loop är syntaktiskt socker för iterator-loop:
  `for(Object o: list) { …`
  ```

Iterator internt

- Inkapslat i iteratorn:
- - i fallet lista med intern array – index för aktuellt element samt referens till listan
- - i fallet länkad lista – en referens till aktuellt element som succesivt flyttas fram



Iterator


Client
<<interface>>
Collection
+createIterator(): Iterator
<<interface>>
Iterator
+hasNext(): boolean
+next(): Object
ConcreteCollection
+createIterator(): Iterator
ConcreteIterator
+hasNext(): boolean
+next(): Object

Composite: Att hantera hierarkier

- Ett filsystem består av noder som kan vara av typen fil eller directory Ett directory kan innehålla noder.



Composite, lösning




- ```
  class Directory extends Node {
    private ArrayList<Node> theNodes;
    ….
  ```
- Ska metoderna addNode, removeNode, getChildren finnas redan i basklasen Node?
  ```

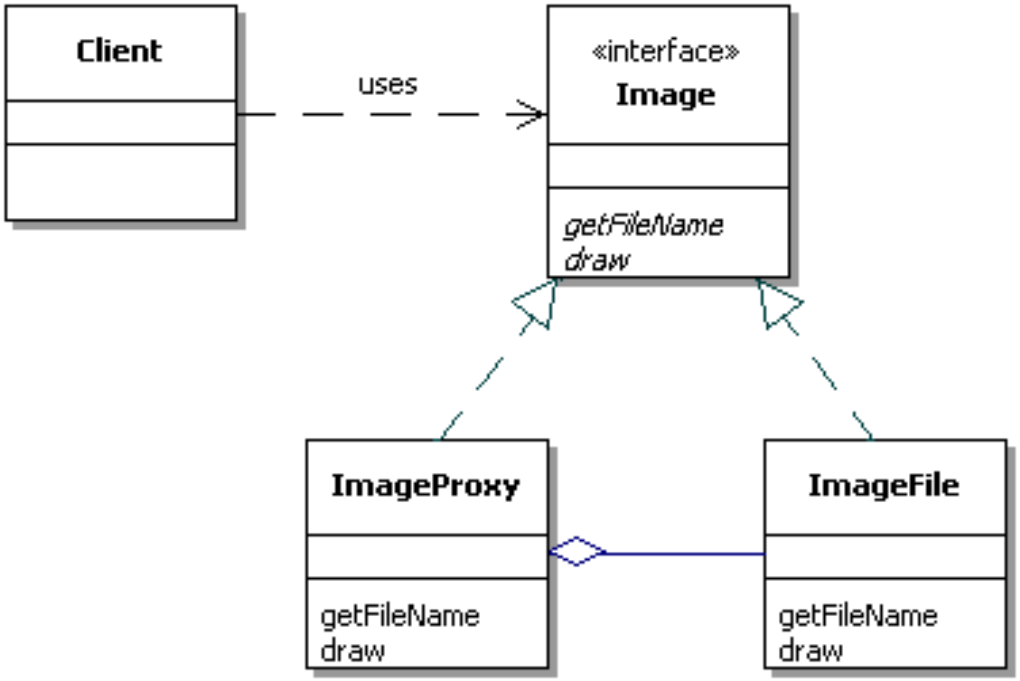
Composite, generell lösning



- Använd Composite när du vill representera objekt och hierarkier av objekt samt använda operationer för både enskilda objekt och hierarkier på samma sätt

Proxy: Ställföreträdare

- Vi vill inte initiera en kostsam resurs förrän den verkligen behövs, ”lazy proxy”.



Proxy: Ställföreträdare

```
public interface Image {
   public abstract String getFileName();
   public abstract void draw(Graphics g);
}

public class ImageFile implements Image {
   private String file;
   public ImageFile(String file) {
        // Open and load the image...
   }
   public String getFileName() {
        return file;
   }
   public void draw(Graphics g) {
        // Draw the image...
   }
}
```

Proxy: Ställföreträdare

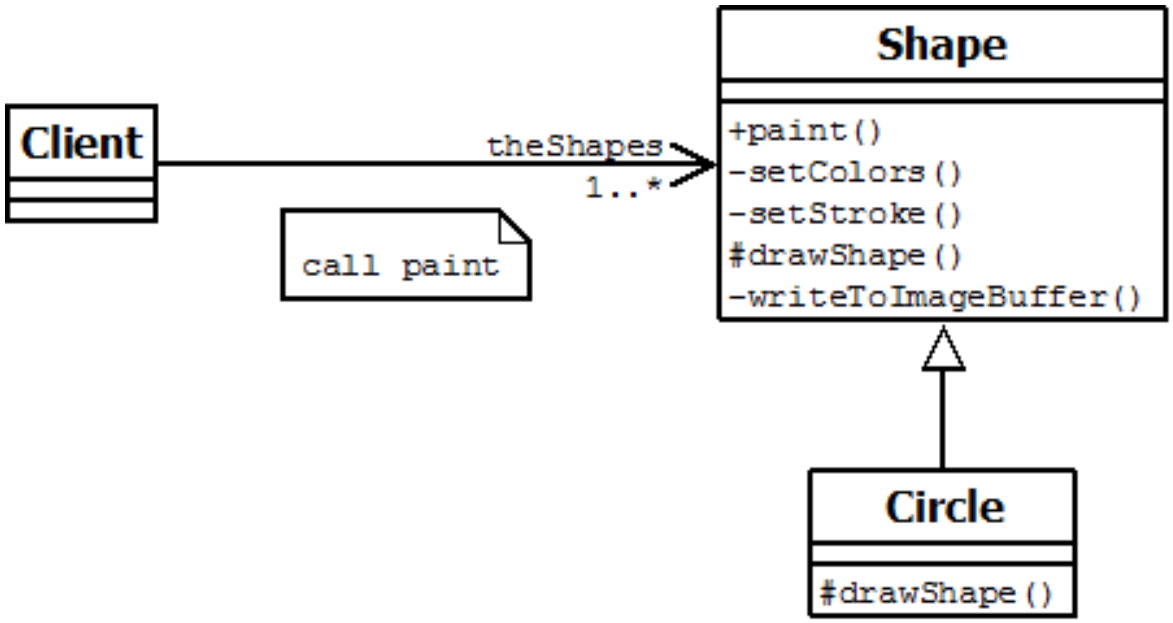
```
public class ImageProxy implements Image {
   private String file;
   private ImageFile realImage;

   public ImageProxy(String file) {
        this.file = file;
        realImage = null;
   }
   public void draw(Graphics g) {
        if(realImage == null)
                realImage = new ImageFile(file);
        realImage.draw(g);
   }
   public String getFileName() {
        return file;
   }
}
```

Proxy: Ställföreträdare

Ibland vill/kan man inte anropa ett objekt direkt utan måste gå via en ställföreträdare, proxy

- Objektet finns på en annan dator
- Objektet är resurskrävande
- Åtkomstskydd behövs
- Smarta referenser

Template method



- Implementera de kodstycken i en metod som inte varierar i superklassen (som separata, privata eller final, metoder)
- Omdefiniera de kodstycken i en metod som varierar i subklasserna (via separata metoder)

Template method

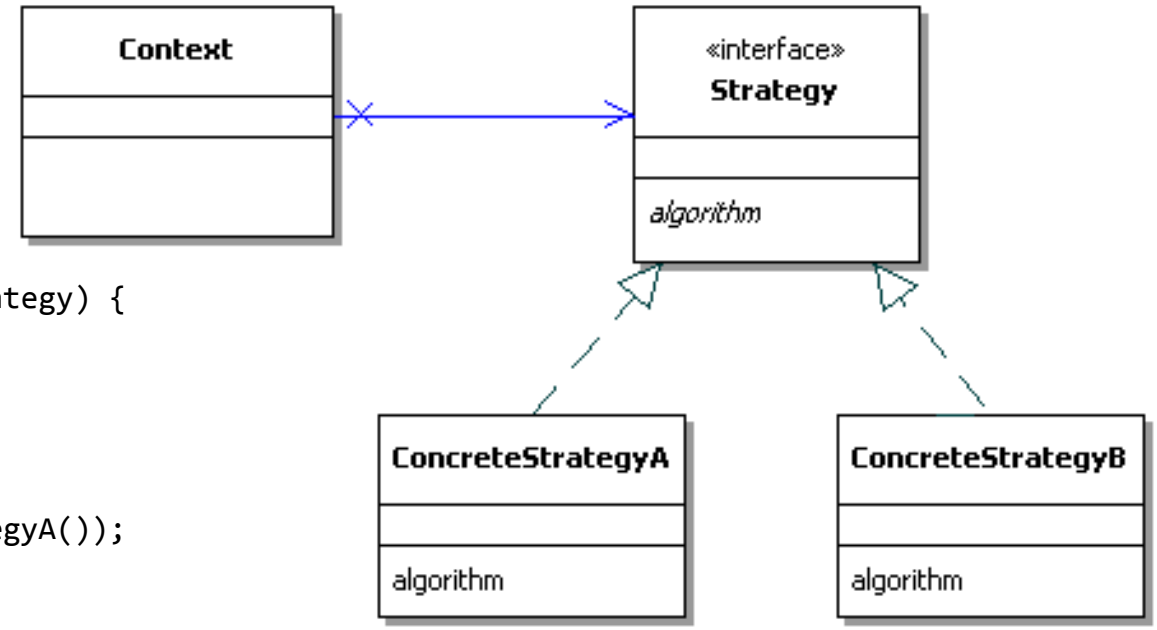
```
public class Shape{

    public final void paint(…) {
        setColors();
        setStroke();
        drawShape(…);
        writeToImageBuffer();
    }

    private void setColors() { // ... }
    private void setStroke() { // ... }

    private void writeToImageBuffer() { // ... }

    protected void drawShape(…) {
         // draw, using g. Override in subklasses.
    }
}
```

En metod deklarerad ”final” kan inte omdefinieras i subklasser

Denna metod ska omdefinieras i subklasser - kan ev. deklareras ”abstract”

Strategy: Att kapsla in beteende

- Syfte: Att kapsla in beteende att användas som strategi/delmoment i andra algoritmer – strategier kan bytas ut dynamiskt
- Ersätter villkorade specialfall




```
void context(Strategy strategy) {
   …;
   strategy.algorithm();
   …;
}
-----
context(new ConcreteStrategyA());
…;
```

Strategy: Att kapsla in beteende

- Vi önskar använda en och samma sorteringsalgoritm, men göra kriterierna vi använder, för att jämföra objekten, utbytbara.
- Ett interface vars skilda implementationer definierar olika sätt att jämföra:

```
public interface Comparator<T> {
   public abstract int compare(T obj1,T obj2);
}
```

Strategy: Att kapsla in beteende

- Sorteringsalgoritm:

```
public static <T> void sort (
                ArrayList<T> list, Comparator<T> c) {
  …
   for( … ) {
       for( … ) {
         if(c.compare(list.get(i),list.get(j)) < 0) {
           // Swap…
         }
     …
   …
```

Strategy: Att kapsla in beteende

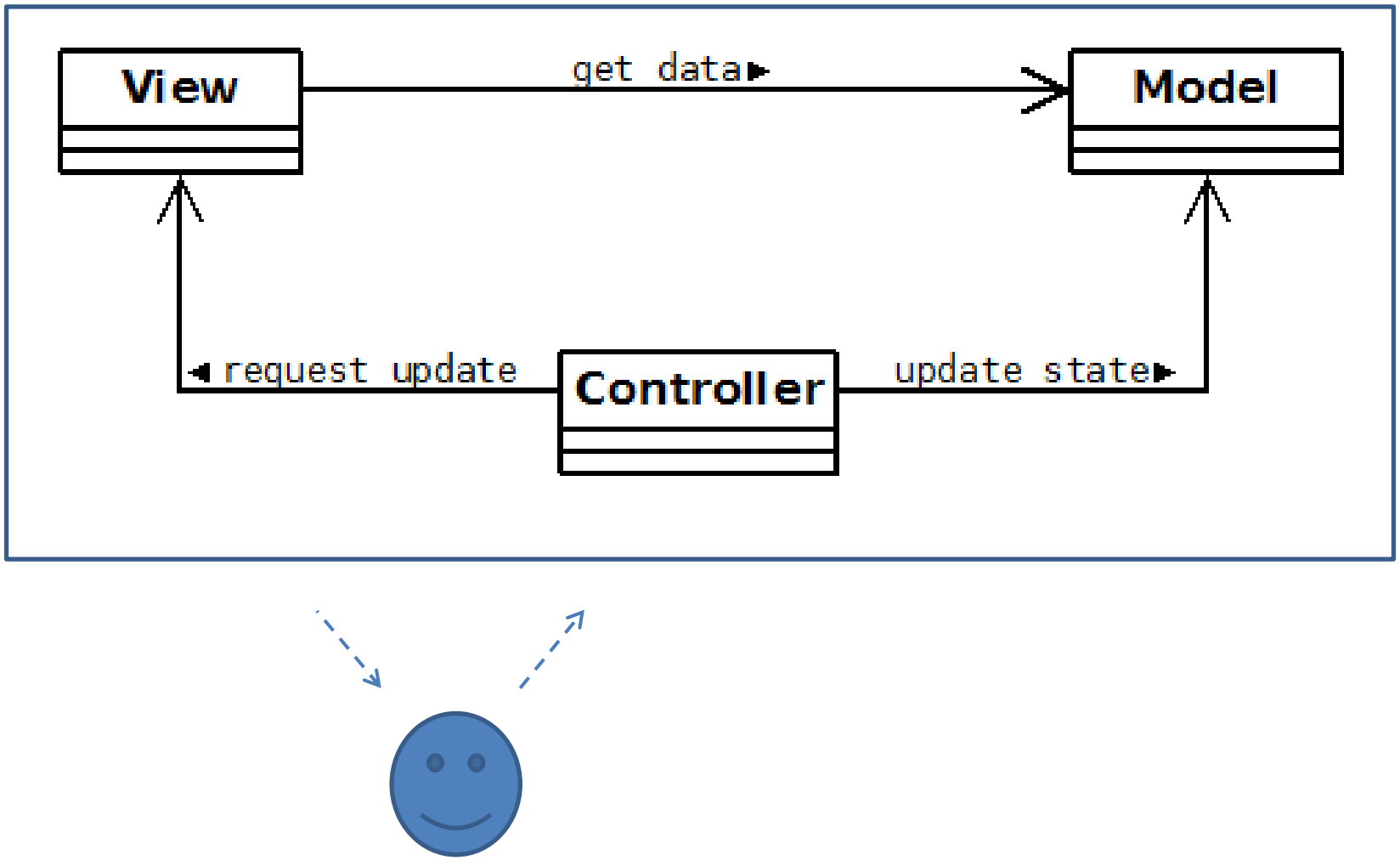
- Ex på Comparator

```
public class LastNameComparator implements
   Comparator<Person> {
   public int compare(Person p1, Person p2) {
       return p1.getLastName().compareTo(
              p2.getLastName());
   }
}
```

Model-View-Controller

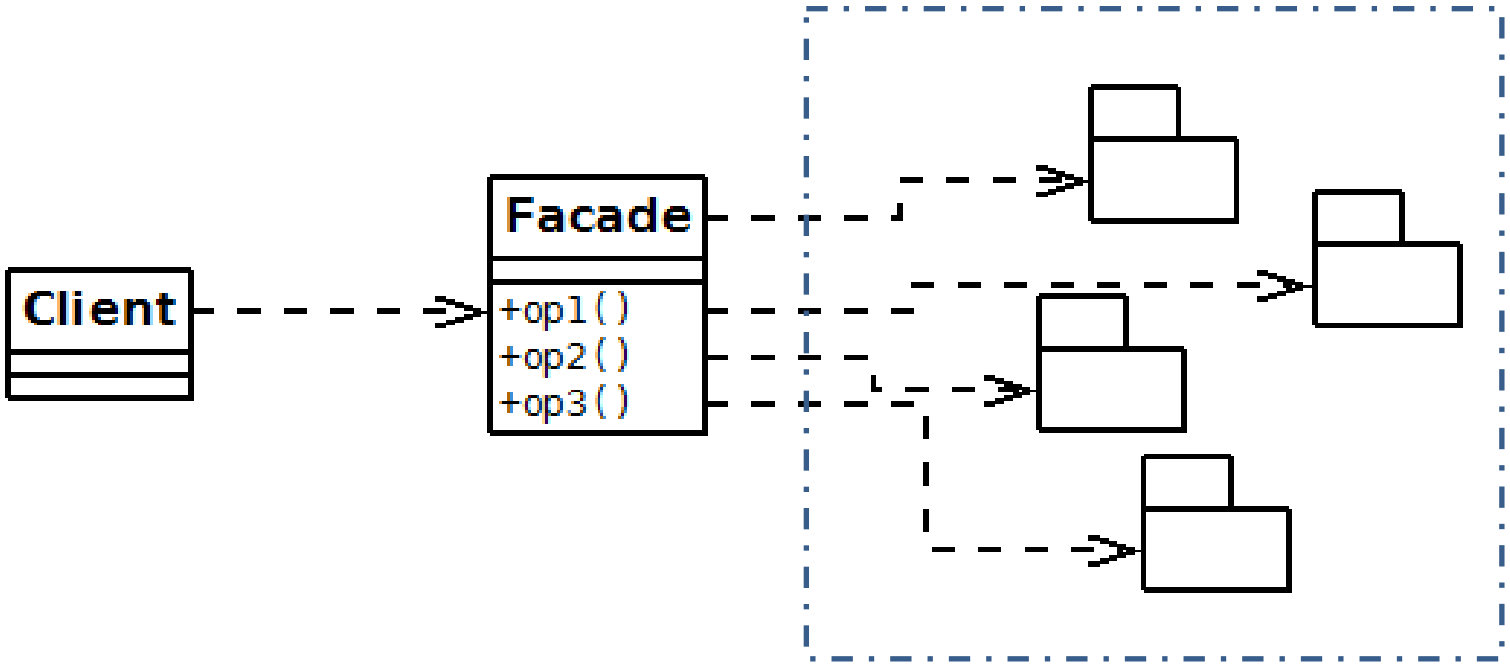
- Applikationer med användargränssnitt
- Separerar roller
  - modellen: de data applikationen ska representera och logiken kring detta
  - vyn: presentationen av data
  - controllern: hanterar input från användaren, initierar uppdateringart av modellen och vyn
- Uppfyller
  - Hög kohesion
  - Låg koppling
  - Möjligt att byta ut vyn, alternativt ha flera vyer till samma modell

Model-View-Controller



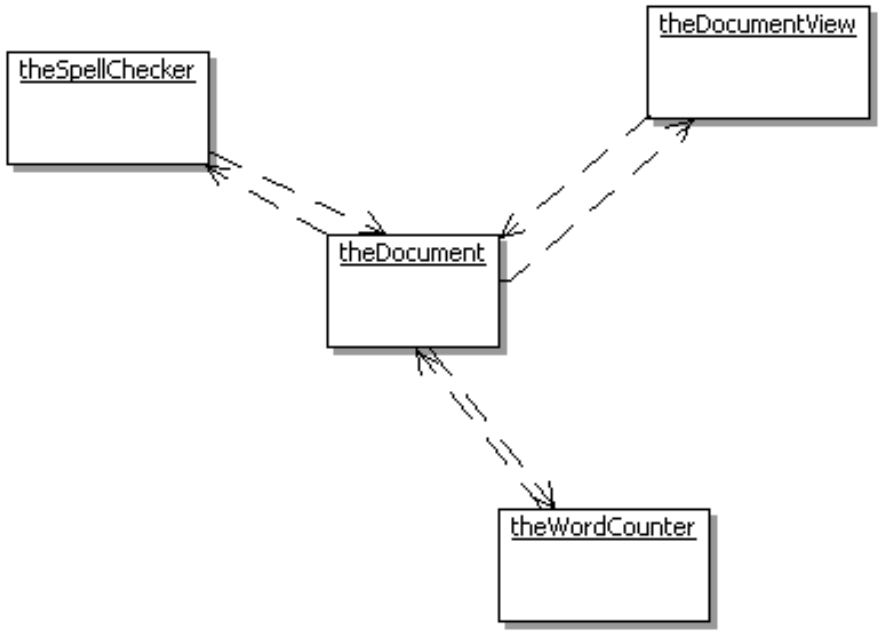
Façade

- Definiera ett förenklat gränssnitt mot ett komplicerat system



Observer-Subject: Enkelrikta beroende

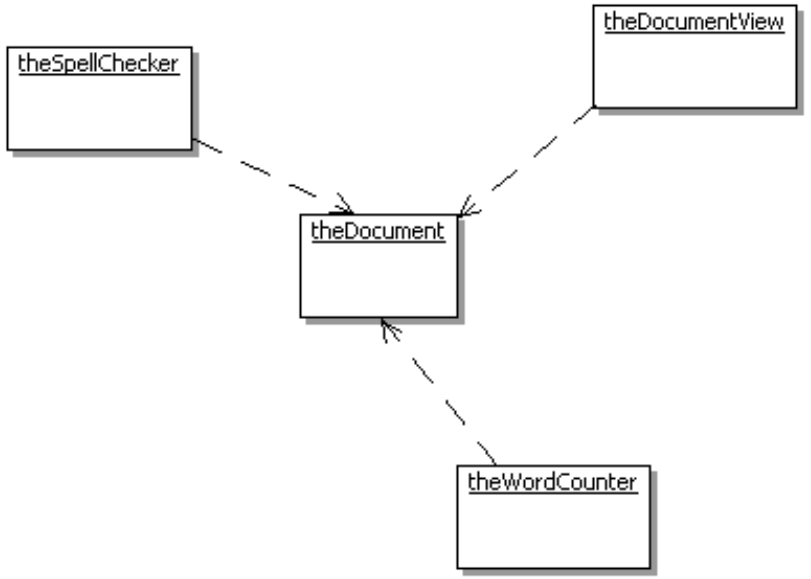
- När ett objekt (subject) uppdaterats behöver objekt (observers) veta detta



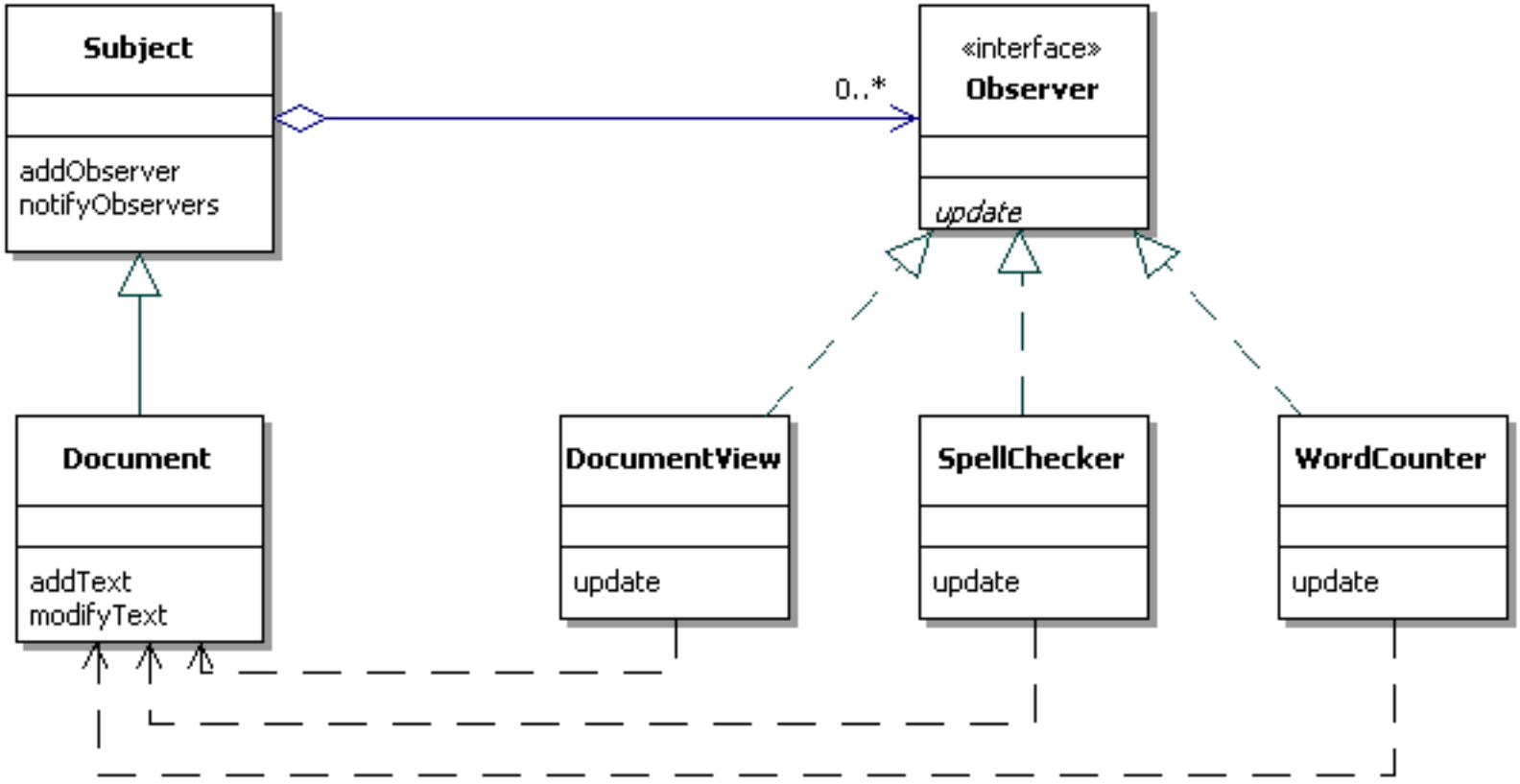
- Model-View: Vyerna (observers) behöver veta när modellen (subject) uppdaterats

Observer-Subject: Enkelrikta beroende

- Vi önskar enkelriktade beroenden:

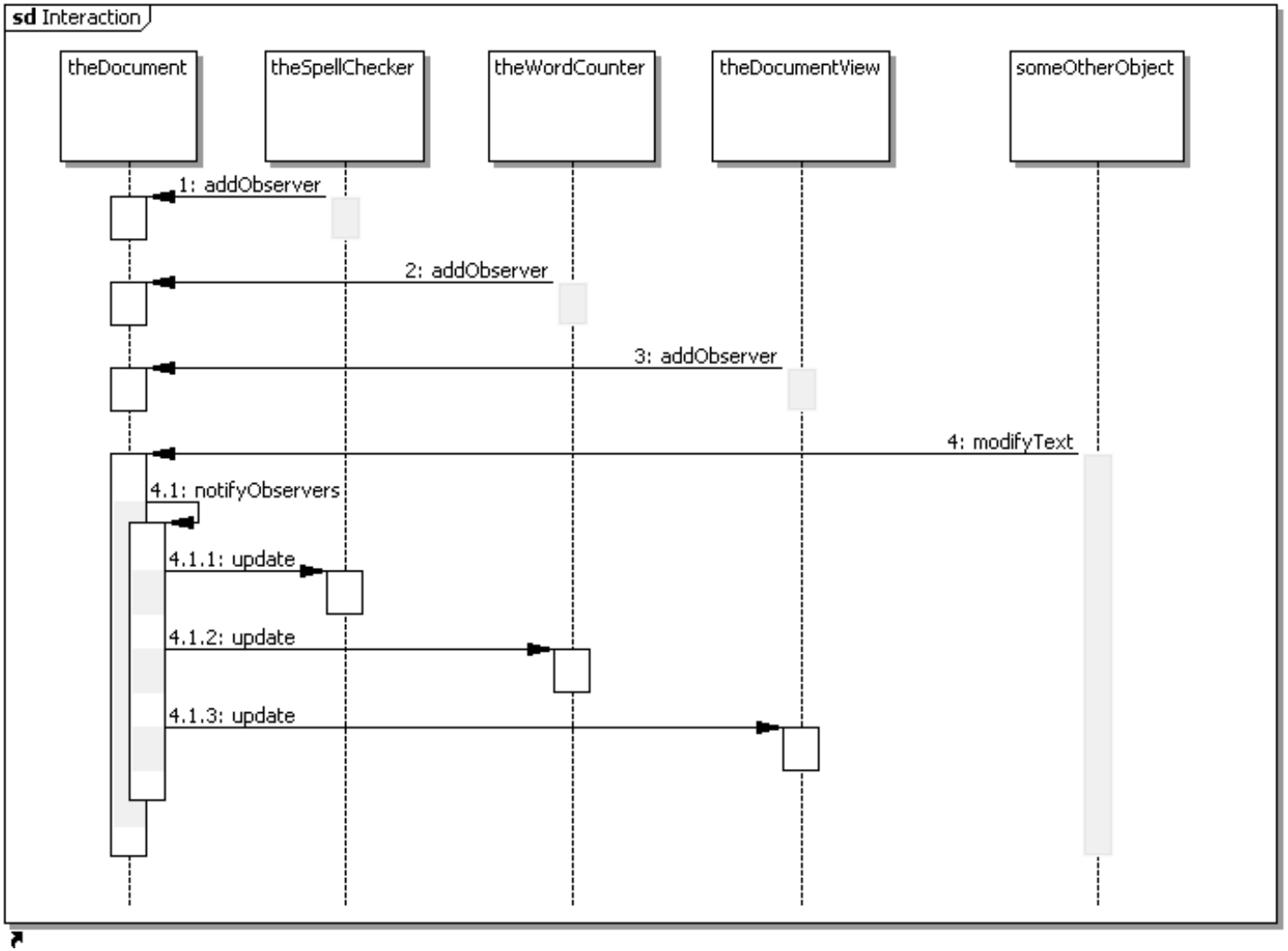


Observer-Subject: Enkelrikta beroende



- Metoder som modifierar dokumentet ska anropa notifyObservers()

Observer-Subject: Enkelrikta beroende



Anti-mönster

- ”Call-super” – ett super-sub förhållande som kräver att en omdefinierad metod, m, anropar super.m
- Rectangle-Square-problem – arv som inte följer Liskovs substitutionsprincip
- ”God object” (inte good object) – ett objekt som har för många roller (> 1)
- ”Sequential coupling” – en klass som kräver att deras metoder anropas i en viss ordning
- ”Singletonitis”
- YAFL, ”Yet Another F* Layer” – onödiga lager eller indirekta anrop. Orsakas av överanvändning av designmönster

Arv vs. Komposition

- Arv – en linje *är en* form (relation mellan klasser)
- Subklassen har (viss) insyn i/beroende av superklassens implementation – kan göra subklassen känslig för förändringar i superklassen
  - undvik att ärva från klasser som inte säkert är färdigutvecklade
  - undvik att ärva klasser från andra paket, API
- Komposition – en bil *består av en* motor (relation mellan objekt)
- Komposition är en strängare form av aggregat, de aggregerade objekten är privata och deras livslängd bestäms av aggregatets livslängd.

Arv vs. Komposition

Exempel: Skapa en kö-klass med hjälp av java.util.LinkedList

- Kontrakt:
  - Element läggs till i slutet av kön (enqueue)
  - Element tas bort från början av kön (dequeue)
  - size, isEmpty
- En kö är en specialisering av en lista?
  …eller…
  En kö består av en lista?

Arv vs. Komposition

Via arv:

```
class Queue<T> extends LinkedList<T> {

        public void enqueue(T element) {
                this.addLast(element);
        }

        public T dequeue() {
                return this.removeFirst();
        }
}
```

Varför är arv en mindre god idé i detta fall?

Svar: Sub-klassen ärver metoder som bryter kontraktet!

Arv vs. Komposition

Via komposition:

```
class Queue<T> {
        private LinkedList<T> list = new LinkedList<T>();

        public void enqueue(T element) {
                list.addLast(element);
        }

        public T dequeue() {
                return list.removeFirst();
        }

        public int size() {
                return list.size();
        }

        public boolean isEmpty() {
                return list.isEmpty();
        }
}
```

Arv vs. komposition

- Vilken av implementationerna uppfyller kontraktet för kö-klassen
- - Arv => hela gränssnittet ärvs. Önskvärt?
  - Arv ger möjlighet till polymorfism
  - Egenskaper från flera olika typer ej möjligt om vi inte har multipelt arv (men möjligt vid komposition)
  - Ändringar i superklassens implementation kan påverka subklasser direkt
- Tumregel: Föredra komposition före arv i situationer där båda verkar tänkbara

Generella designprinciper

Bilting: ”Designa med ändringar i åtanke”

Klasser/paket bör uppfylla:

- Hög kohesion (sammanhållning)
- Låg koppling
- Återanvändning möjlig

Läsa mer…

- http://sourcemaking.com/design_patterns
- Designmönster för programmerare
  Ulf Bilting
  Studentlitteratur
- *Design Patterns: Elements of Reusable Object-Oriented Software*
  Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (Gang of Four)
  Addison-Wesley