Objektorienteret design

Plan

- Introduktion til designmønstre

- Design af generiske komponenter
  - faktorisering
  - generalisering
  - abstrakt kobling

- Design case: animering af algoritmer til sortering

Genbrug

Muligheden for genbrug af eksisterende programdele er en afgørende motivationsfaktor for at benytte objektorienteret programmering.

Forudsætninger for genbrug:

- at *finde* det, man har brug for
- at *forstå* det, man har fundet
- at *modificere* det fundne, så det passer til det aktuelle behov
- at *sammensætte* det modificerede med de øvrige programkomponenter

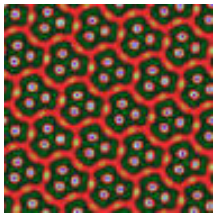
Designmønstre

Genbrug af enkeltklasser ved hjælp af nedarvning er relativt simpelt.

Derimod er det svært at genbruge et *objektorienteret design* (en implementerings *arkitektur*, udtrykt ved samspillet imellem objekter og klasser).

En **designmønster** er en navngiven beskrivelse af en løsning af et tilbagevendende designproblem.

Formål med designmønstre

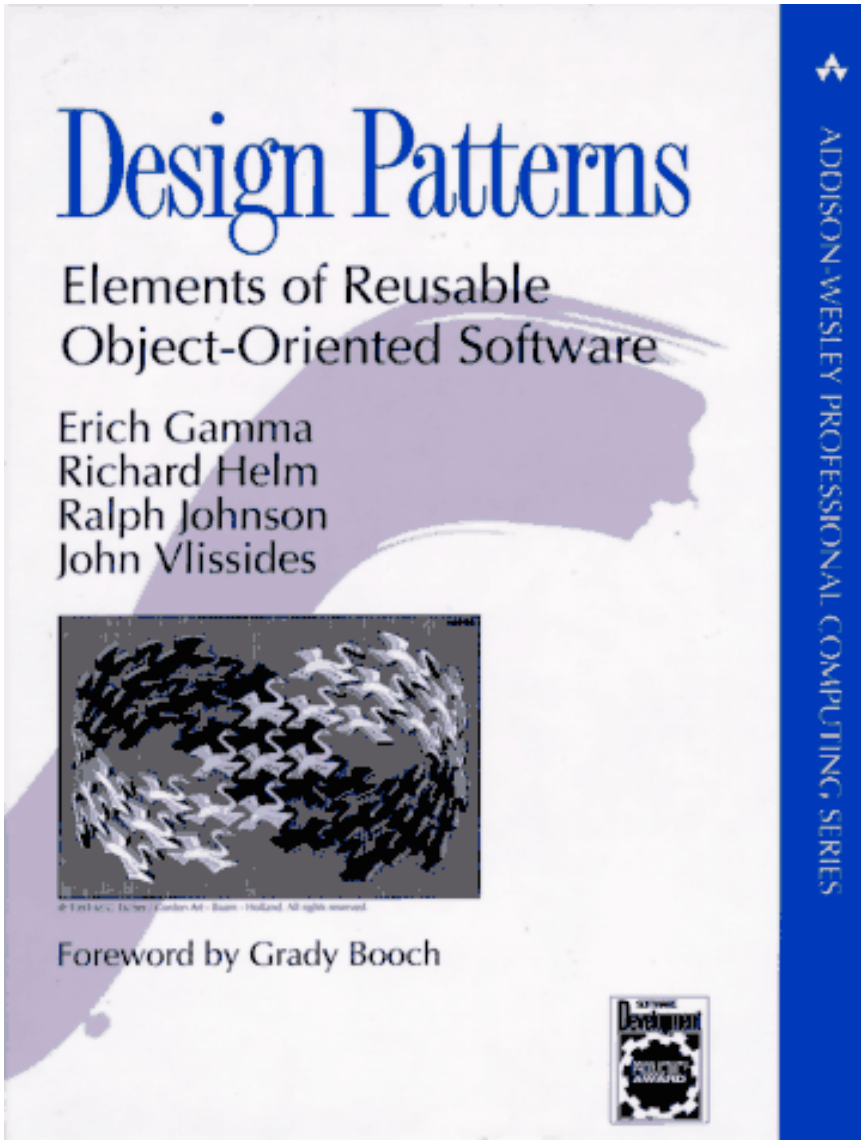


Designmønstre sammenfatter erfaringer om programdesign i nogle få mønstre, der har vist deres værdi i praksis.

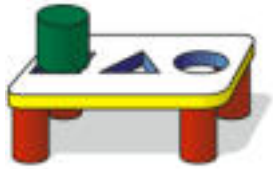
Formål:

- at understøtte genbrug af design
- at øge tilliden til det udviklede programmel
- at give designere et fælles sprog

Gamma et al.: *Design Patterns* [1995] udgør et katalog over de 23 mest almindelige designmønstre.


Design Patterns
Elements of Reusable
Object-Oriented Software
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides
Foreword by Grady Booch
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

Kategorisering af designmønstre

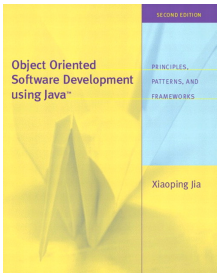


Konstruerende mønstre (**creational** patterns)
omhandler skabelse af objekter

Strukturelle mønstre (**structural** patterns)
omhandler den *statiske* sammensætning og struktur af klasser og objekter (deres relationer)

Adfærdsmæssige mønstre (**behavioral** patterns)
omhandler det *dynamiske* samspil imellem klasser og objekter (deres indbyrdes kommunikation)

Designmønstre i lærebogen



Konstruerende mønstre:

Singleton, **Factory**
Factory Method
Abstract Factory, Builder, Prototype

Strukturelle mønstre:

Composite, **Decorator**, **Proxy**
Adapter

Adfærdsmæssige mønstre:

Template Method, **Strategy**, **Iterator**
State, **Command**
Observer

Behandles

- Nu
- Senere
- Ikke (læs selv kapitel 10)
- Senere, men er ikke med i lærebogen

Beskrivelse af designmønstre

Navn
Kategori
Hensigt
Motivation
Andre navne
Anvendelse
Struktur
Deltagere
Konsekvenser
Implementering

Designmønsteret Singleton

Navn:

Singleton

Kategori:

Konstruerende designmønster

Hensigt:

At sikre, at der kun skabes én instans af en klasse

Motivation:

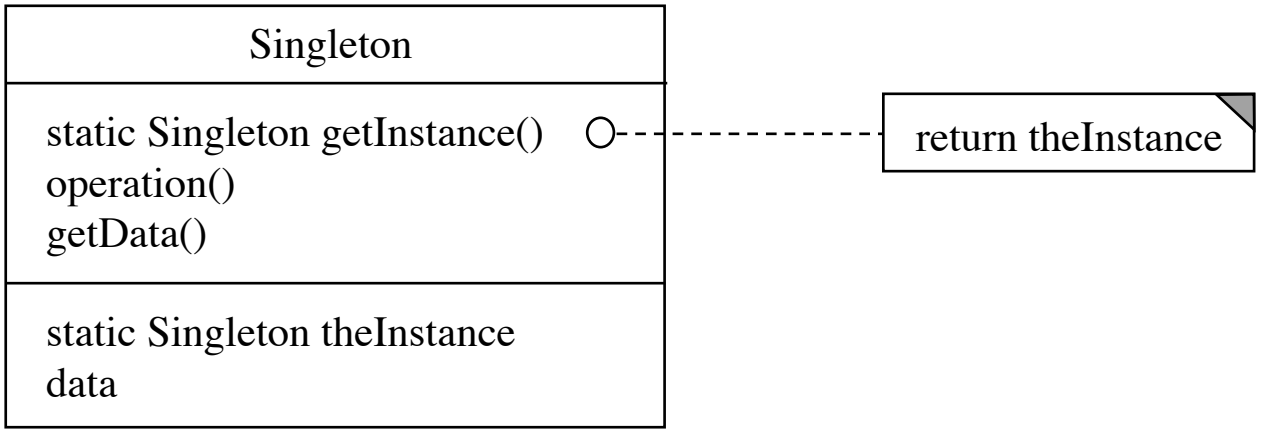
For nogle klasser er det vigtigt, at der kun er en instans af klassen. Der skal f.eks. kun være ét filsystem og én vinduesmanager. En god løsning er, at klassen selv sikrer, at der kun skabes én instans af klassen

Designmønsteret Singleton
(fortsat)

Anvendelse:

Anvend mønsteret, når der skal være eksakt én instans af en klasse

Struktur:



Deltagere:

Der er kun én deltager: klassen Singleton

Designmønsteret Singleton
(fortsat)

Implementering:

```
public class Singleton {
    public static Singleton getInstance() {
        if (theInstance == null)
            theInstance = new Singleton();
        return theInstance;
    }

    private Singleton() {
        «Initialize instance fields»
    }

    «Instance fields and methods»

    private static Singleton theInstance = null;
}
```

Forhindring af samtidig skabelse

```
public static synchronized Singleton getInstance()
```

Herved forhindres to (eller flere) tråde i samtidigt at generere en instans.

Alternative løsninger (1)

(brug af undtagelse)

```
public class Singleton {
    static boolean instanceCreated = false;

    public Singleton() throws SingletonException {
        if (instanceCreated)
            throw new SingletonException();
        instanceCreated = true;
        «Initialize instance fields»
    }

    «Instance fields and methods»
}
```

Alternative løsninger (2)

(brug af klasse med statiske metoder)

```
public final class Singleton {
    private Singleton() {}

   «Static instance fields and methods»
}
```

Design af generiske komponenter

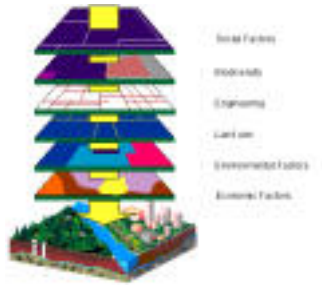
Ved design af **generiske** (genbrugelige uden kodemodifikation) komponenter benyttes følgende to teknikker:

(1) **Faktorisering**
sammenfatning af fælles træk ved komponenter

(2) **Generalisering**
behandling af særtilfælde i en generel kontekst

Midler:
nedarvning, delegering, abstrakte klasser og grænseflader

(1) Faktorisering



En metode til at skabe generiske komponenter er at identificere kodesegmenter, der er ens, eller næsten ens.

(1) Identificer kodesegmenter med samme logik
(2) Sammenfat denne logik i en generisk komponent
(3) Omorganiser programmet, således at alle forekomster af de identificerede kodesegmenter erstattes med brug af den generiske komponent

Faktorisering reducerer mængden af kode, hvilket forenkler vedligeholdelse.

Faktorisering med metoder

```
class Computation {
    void method1(...) {
        //...
        computeStep1();
        computeStep2();
        computeStep3();
        //...
    }

    void method2(...) {
        //...
        computeStep1();
        computeStep2();
        computeStep3();
        //...
    }
}
```

```
class FactorizedComputation {
    void computeAll() {
        computeStep1();
        computeStep2();
        computeStep3();
    }

    void method1(...) {
        //...
        computeAll();
        //...
    }

    void method2(...) {
        //...
        computeAll();
        //...
    }
}
```

Faktorisering ved nedarvning

```
class ComputationA {
    void method1(...) {
        //...
        computeStep1();
        computeStep2();
        computeStep3();
        //...
    }
}


class ComputationB {
    void method2(...) {
        //...
        computeStep1();
        computeStep2();
        computeStep3();
        //...
    }
}
```

```
class Common {
    void computeAll() {
        computeStep1();
        computeStep2();
        computeStep3();
    }
}
```

```
class ComputationA extends Common {
    void method1(...) {
        //...
        computeAll();
        //...
    }
}

class ComputationB extends Common {
    void method2(...) {
        //...
        computeAll();
        //...
    }
}
```

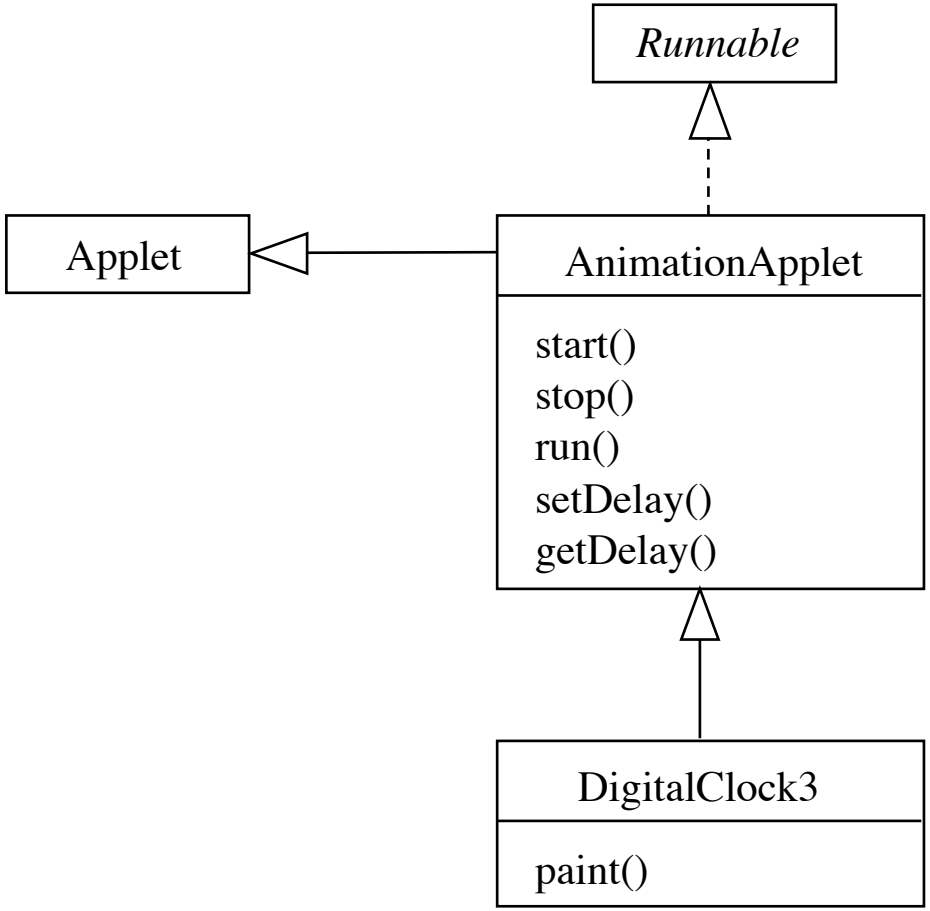
Faktorisering ved delegering

```
class Helper {
    void computeAll() {
        computeStep1();
        computeStep2();
        computeStep3();
    }
}
```

```
class ComputationA {
    void method1(...) {
        //...
        helper.computeAll();
        //...
    }
    Helper helper;
}
```

```
class ComputationB {
    void method2(...) {
        //...
        helper.computeAll();
        //...
    }
    Helper helper;
}
```

En animeringsapplet
(eksempel på faktorisering ved nedarvning)



```
public class AnimationApplet extends java.applet.Applet
                             implements Runnable {
    public void start() {
        if (animationThread == null)
            animationThread = new Thread(this);
        animationThread.start();
    }

    public void stop() { animationThread = null; }

    public void run() {
        while (animationThread != null) {
            repaint();
            try {
                Thread.sleep(delay);
            } catch (InterruptedException e) {}
        }
    }

    public final void setDelay(int delay) { this.delay = delay; }

    public final int getDelay() { return delay; }

    protected Thread animationThread;
    protected int delay = 100;
}
```

```
import java.awt.*;
import java.util.Calendar;

public class DigitalClock3 extends AnimationApplet {
    public DigitalClock3() {
        setDelay(1000);
    }

    public void paint(Graphics g) {
        Calendar calendar = Calendar.getInstance();
        int hour = calendar.get(Calendar.HOUR_OF_DAY);
        int minute = calendar.get(Calendar.MINUTE);
        int second = calendar.get(Calendar.SECOND);
        g.setFont(font);
        g.setColor(color);
        g.drawString(hour +
                     ":" + minute / 10 + minute % 10 +
                     ":" + second / 10 + second % 10, 10, 60);
    }

    protected Font font = new Font("Monospaced", Font.BOLD, 48);
    protected Color color = Color.green;
}
```

Hvorledes undgås flimmer?

Flimmer skyldes kaldet af `repaint`.

`repaint` kalder `update`, der som standard

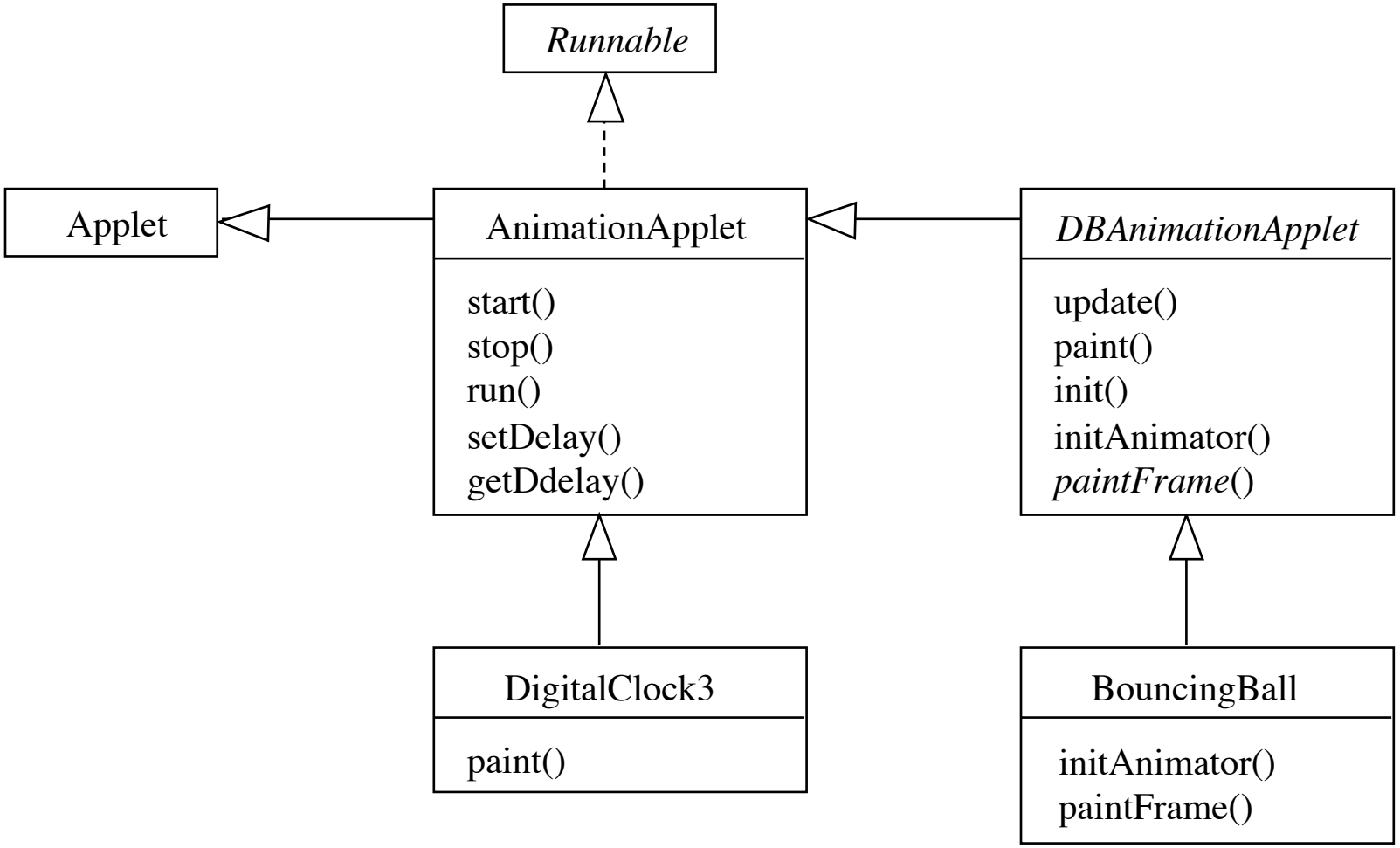
1. maler hele området med baggrundsfarven,
2. sætter forgrundsfarven, og
3. kalder `paint`

Løsning:

Overskriv `update`

Benyt et `Image`-objekt som buffer (mellemlager)

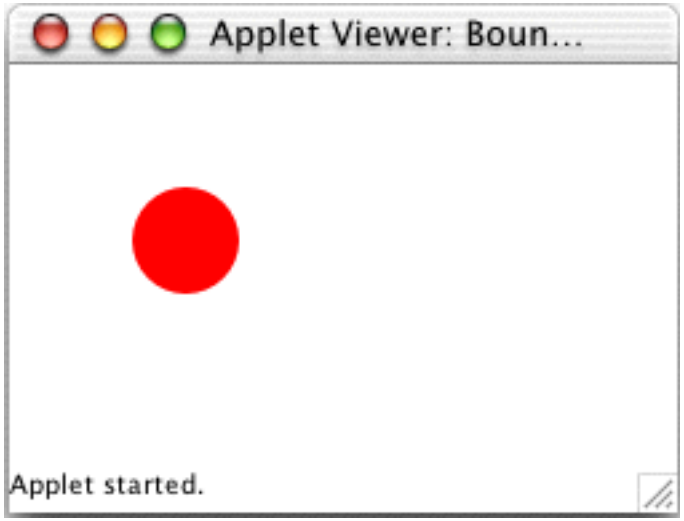
En animeringsapplet, der benytter dobbeltbuffer



```
import java.awt.*;

public abstract class DBAnimationApplet extends AnimationApplet {
    public final void init() {
        dim = getSize();
        im = createImage(dim.width, dim.height);
        offscreen = im.getGraphics();
        initAnimator();
    }

    public final void update(Graphics g) {
        paintFrame(offscreen);
        g.drawImage(im, 0, 0, this);
    }

    public final void paint(Graphics g) { update(g); }

    protected void initAnimator() {}

    protected abstract void paintFrame(Graphics g);

    protected Dimension dim;
    protected Image im;
    protected Graphics offscreen;
}
```

Bemærk `final`-specifikationerne

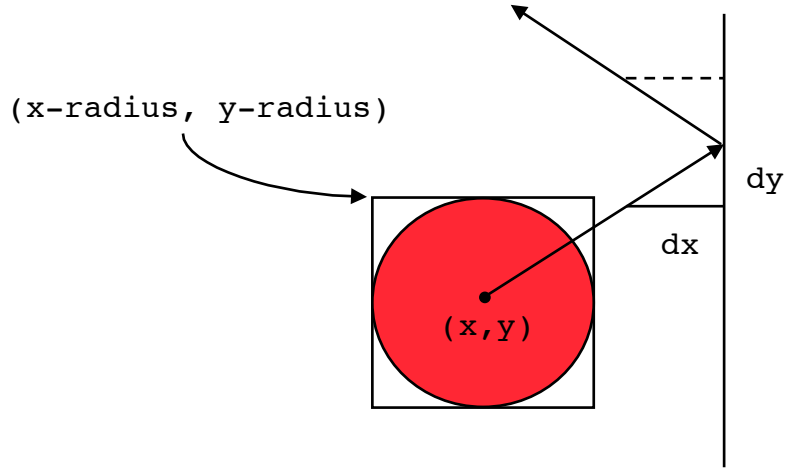
Animering af en hoppende bold



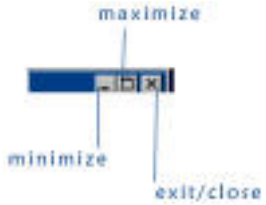
```
import java.awt.*;

public class BouncingBall extends DBAnimationApplet {
    protected void initAnimator() {
        String att = getParameter("delay");
        if (att != null)
            setDelay(Integer.parseInt(att));
        x = dim.width * 2 / 3 ;
        y = dim.height - radius;
    }

    protected void paintFrame(Graphics g) { ... }

    protected int x, y;
    protected int dx = -2, dy = -4;
    protected int radius = 20;
    protected Color color = Color.red;
}
```

fortsættes

```
protected void paintFrame(Graphics g) {
    g.setColor(Color.white)
    g.fillRect(0, 0, dim.width, dim.height);
    if (x < radius || x > dim.width - radius)
        dx = -dx;
    if (y < radius || y > dim.height - radius)
        dy = -dy;
    x += dx; y += dy;
    g.setColor(color);
    g.fillOval(x - radius, y - radius, 2 * radius, 2 * radius);
}
```



Retningslinjer for design



Maksimer fleksibilitet
Sørg for at komponenterne er fleksible. Jo større fleksibilitet, desto større er chancen for genbrug.

Minimer chancerne for misbrug
Veldesignede klasser skal minimere mulighederne for forkert brug, og så vidt muligt afsløre eventuelle fejl på oversættelsestidspunktet.

Faktorisering af fælles kodesegmenter

```
class ContextA {
    void method(...) {
        computeStep1();
        «context specific code A»
        computeStep2();
    }
}
```

```
class ContextB {
    void method(...) {
        computeStep1();
        «context specific code B»
        computeStep2();
    }
}
```

```
abstract class Common {
    void method() {
        computeStep1();
        contextSpecificCode();
        computeStep2();
    }
    abstract void contextSpecificCode();
}
```

```
class ContextA extends Common {
    void contextSpecificCode() {
        «context specific code A»
    }
}

class ContextB extends Common {
    void contextSpecificCode() {
        «context specific code B»
    }
}
```

Designmønsteret Template Method

Kategori:

Adfærdsmæssigt designmønster

Hensigt:

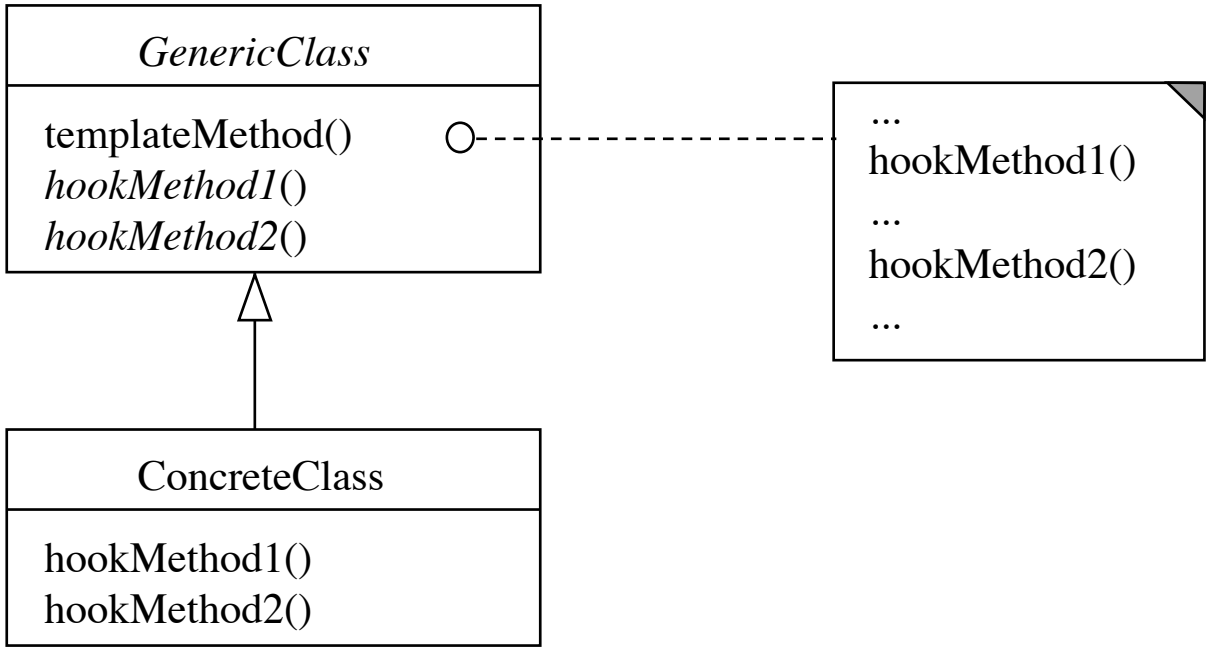
Definer et skelet af en algoritme i en klasses metode og overlad definitionen af nogle af skridtene til metoder i underklasser

Anvendelse:

- For at implementere de invariante dele af en algoritme én og kun én gang og overlade det til underklasser at implementere den adfærd, der kan variere
- For at undgå kodeduplikering

Designmønsteret Template Method
(fortsat)

Struktur:



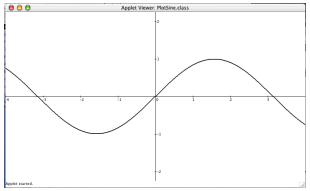
Designmønsteret Template Method
(fortsat)

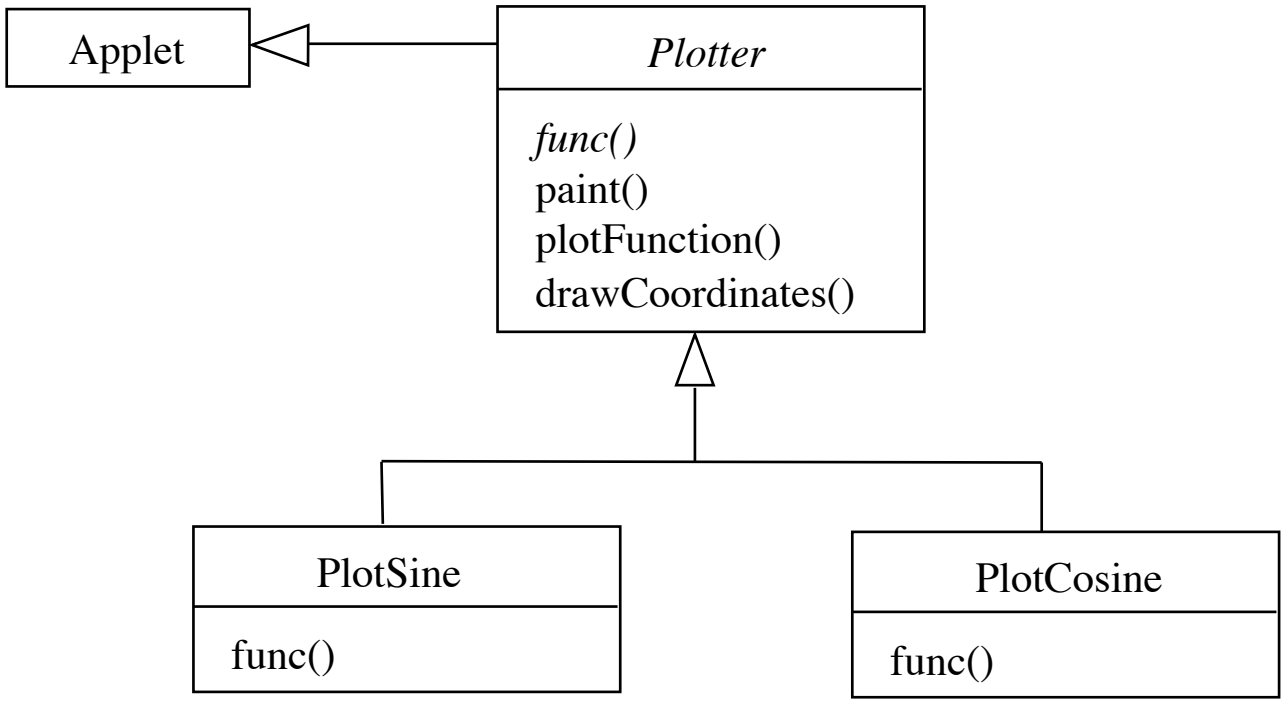
Deltagere:

GenericClass (f.eks. `DBAnimationApplet`), der definerer abstrakte hook-metoder (f.eks. `paintFrame`), som de konkrete underklasser (f.eks. `BouncingBall`) skal implementere, og en template-metode (f.eks. `update`), som kalder hook-metoderne (f.eks. `paintFrame`)

ConcreteClass (f.eks. `BouncingBall`), der implementer hook-metoderne

En generisk funktionsplotter
(eksempel på brug af Template Method)



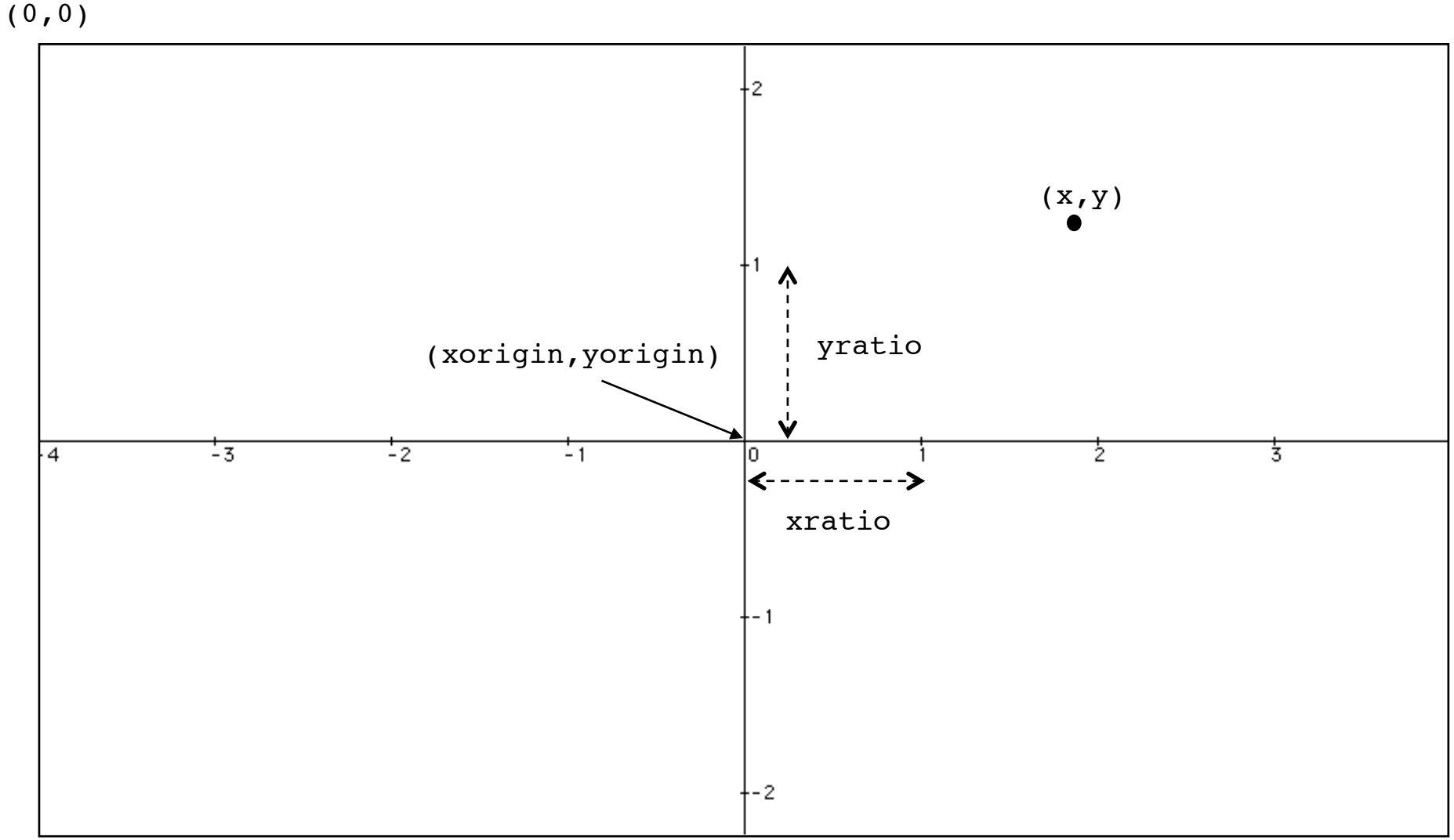


Den generiske klasse

```
import java.awt.*;

public abstract class Plotter extends java.applet.Applet {
    public abstract double func(double x);

    public void init() {
        «Get the parameters and initialize the fields»
    }

    public void paint(Graphics g) {
        drawCoordinates(g);
        plotFunction(g);
    }


    protected void plotFunction(Graphics g) {
        «Plot the function func»
    }

    protected void drawCoordinates(Graphics g) {
        «Draw the x and y axes and tick marks»
    }
}
```

En konkret klasse

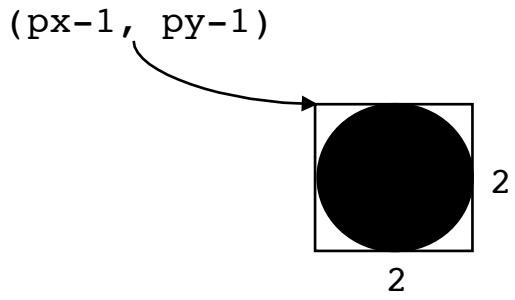
```
public class PlotSine extends Plotter {
    public double func(double x) {
        return Math.sin(x);
    }
}
```

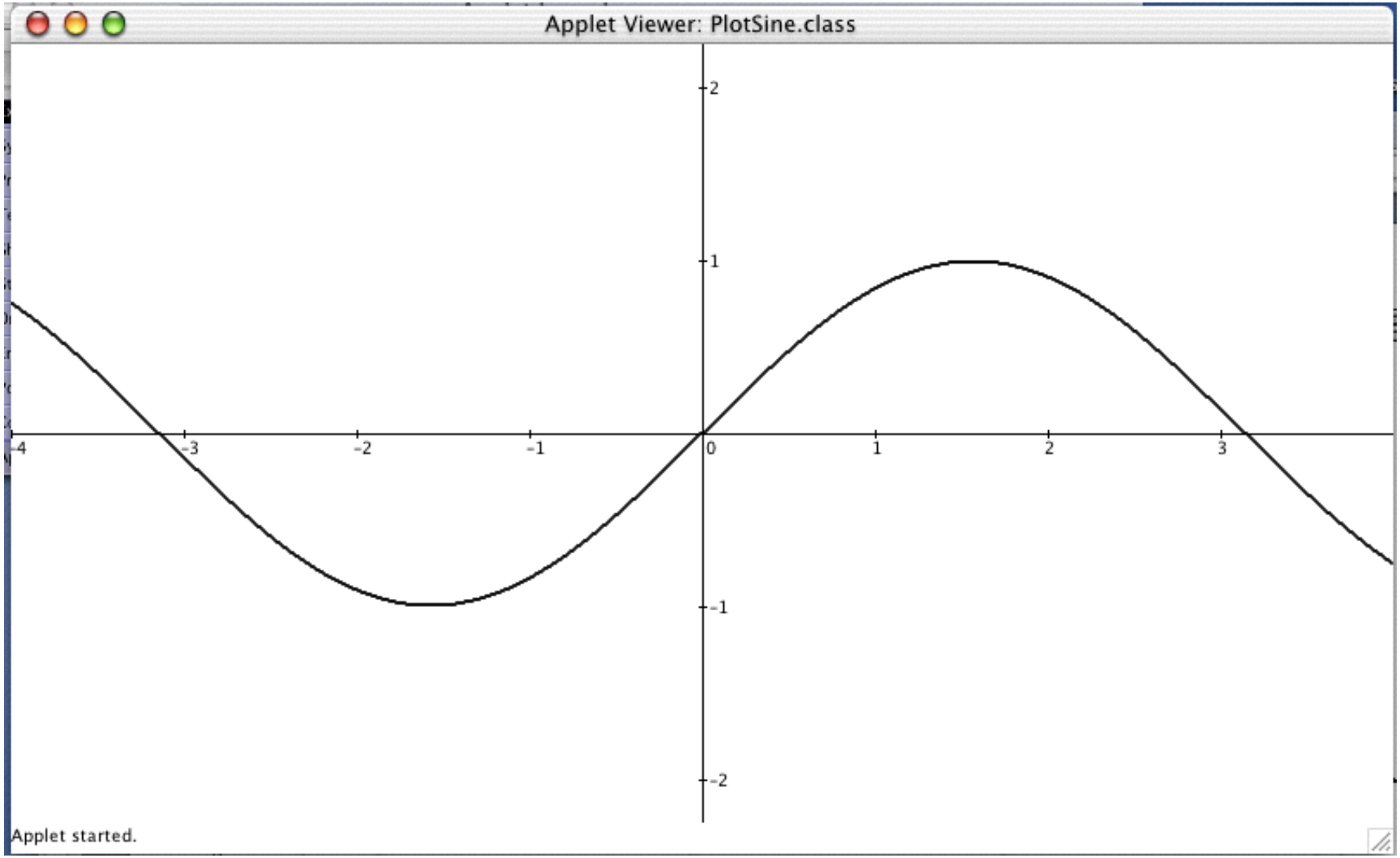


`xorigin`: pixel x-koordinaten for (0, 0)
`yorigin`: pixel y-koordinaten for (0, 0)
`xratio`: antal pixels for en enhed på x-aksen
`yratio`: antal pixels for en enhed på y-aksen

Metoden `plotFunction`

```
protected void plotFunction(Graphics g) {
    for (int px = 0; px < d.width; px++) {
        double x = (double) (px - xorigin) / xratio;
        double y = func(x);
        int py = yorigin - (int) (y * yratio);
        g.fillOval(px - 1, py - 1, 2, 2);
    }
}
```

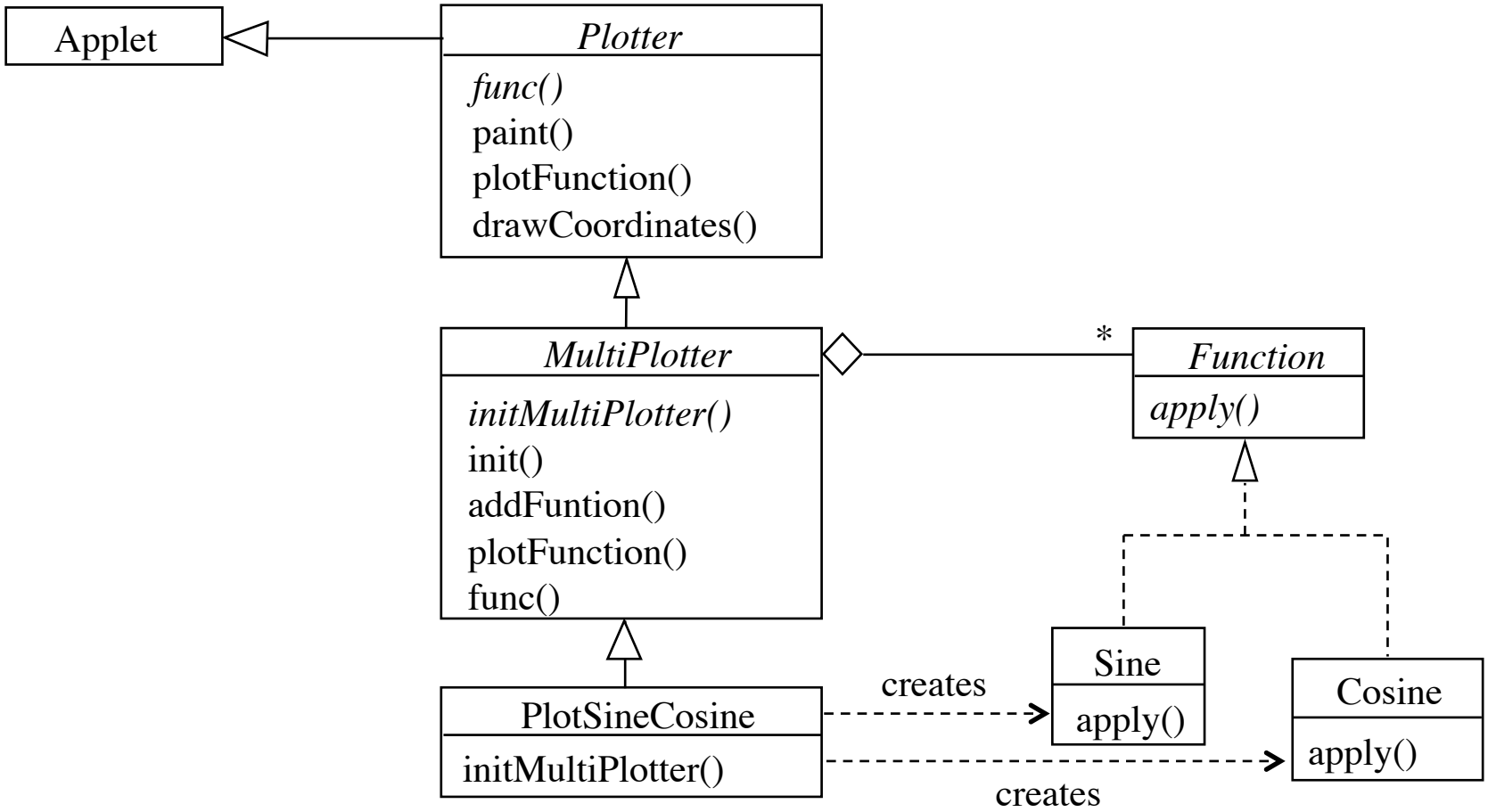



Applet Viewer: PlotSine.class
2
1
-1
-2
-4
-3
-2
-1
0
1
2
3
Applet started.

(2) Generalisering

Generalisering er en teknik, som tager en løsning af et specifikt problem og omformer den således, at den ikke blot kan bruges til løse dette problem, men en mængde af problemer, hvoraf det oprindelige problem blot er et særtilfælde.

En generisk multiplotter
(eksempel på generalisering)



```
public abstract class MultiPlotter extends Plotter {
    abstract public void initMultiPlotter();

    public final void init() {
        super.init();
        initMultiPlotter();
    }

    public final void addFunction(Function f, Color c) {
        if (f != null && numOfFunctions < MAX_FUNCTIONS) {
            functions[numOfFunctions] = f;
            colors[numOfFunctions++] = c;
        }
    }

    protected void plotFunction(Graphics g) {
        «Plot the functions (calling their apply method)»
    }

    public double func(double x) { return 0.0; }

    protected static final int MAX_FUNCTIONS = 5;
    protected int numOfFunctions = 0;
    protected Function[] functions = new Function[MAX_FUNCTIONS];
    protected Color[] colors = new Color[MAX_FUNCTIONS];
}
```

Metoden `plotFunction`

```
protected void plotFunction(Graphics g) {
    for (int i = 0; i < numOfFunctions; i++) {
        g.setColor(colors[i] != null ? colors[i] : Color.black);
        for (int px = 0; px < d.width; px++) {
            double x = (double) (px - xorigin) / xratio;
            double y = functions[i].apply(x);
            int py = yorigin - (int) (y * yratio);
            g.fillOval(px - 1, py - 1, 2, 2);
        }
    }
}
```

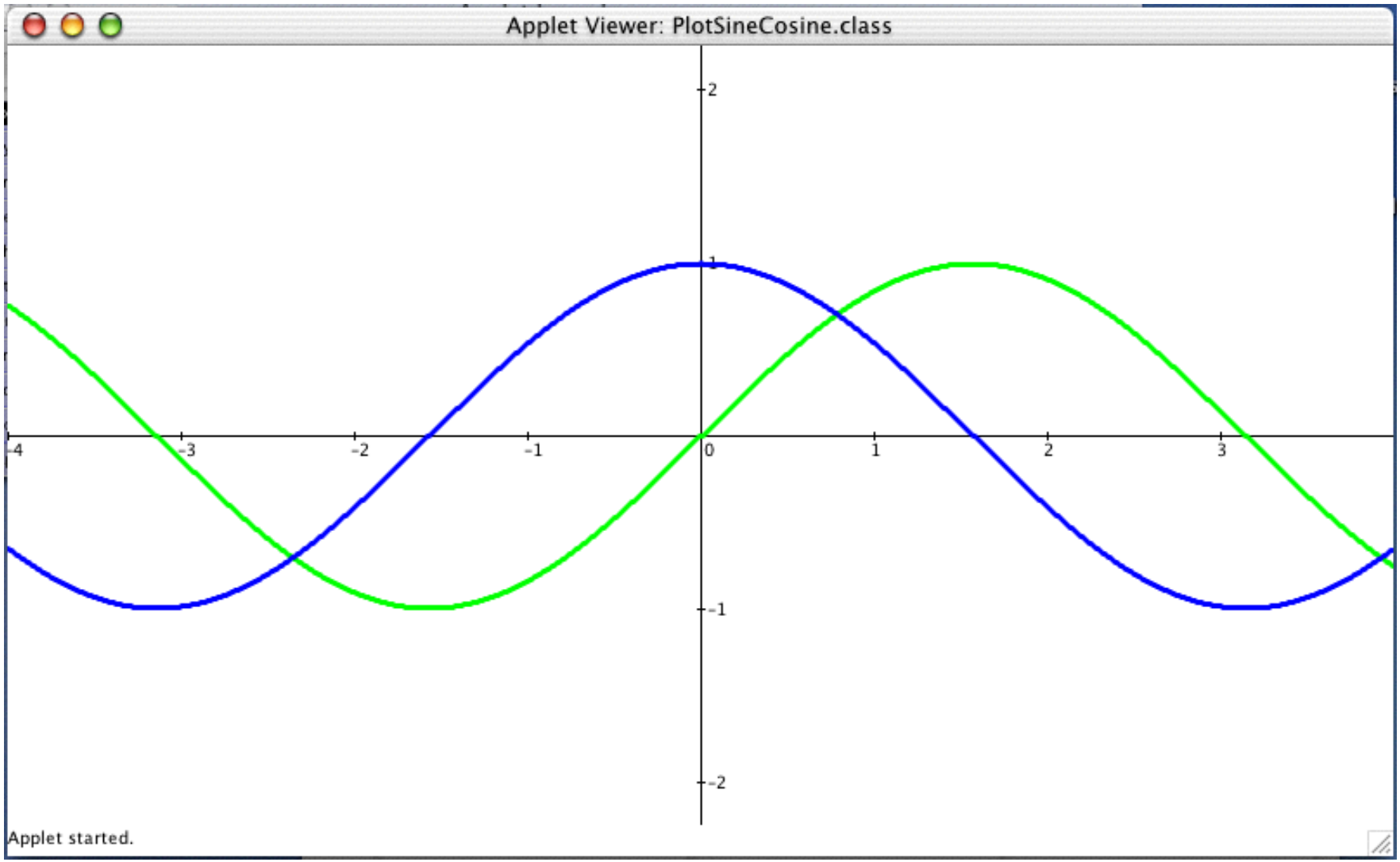
Funktioner

```
interface Function {
    double apply(double x);
}
```

```
public class Sine implements Function {
    public double apply(double x) {
        return Math.sin(x);
    }
}

public class Cosine implements Function {
    public double apply(double x) {
        return Math.cos(x);
    }
}
```

En konkret Multiplotter

```
public class PlotSineCosine extends MultiPlotter {
    public void initMultiPlotter() {
        addFunction(new Sine(), Color.green);
        addFunction(new Cosine(), Color.blue);
    }
}
```


Applet Viewer: PlotSineCosine.class
2
1
-1
-2
-4
-3
-2
-1
0
1
2
3
Applet started.

Designmønsteret Strategy

Kategori:

Adfærdsmæssigt designmønster

Hensigt:

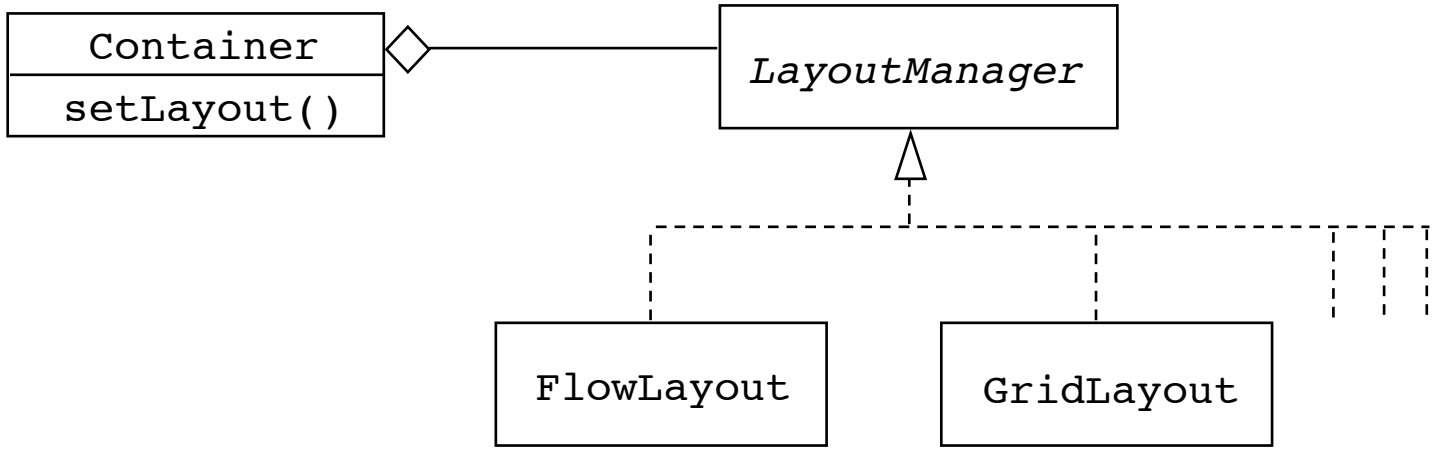
Definer en familie af algoritmer, indkapsl hver af dem, og gør dem udskiftelige

Anvendelse:

- når mange beslægtede klasser kun adskiller sig ved deres adfærd (f.eks. plotter forskellige funktioner)
- når der bruges forskellige udgaver af en algoritme
- når en klasse vælger adfærd ved hjælp af betingede sætninger
- når en algoritme bruger data, som klienter ikke må kende (f.eks. `LayOutManager`)

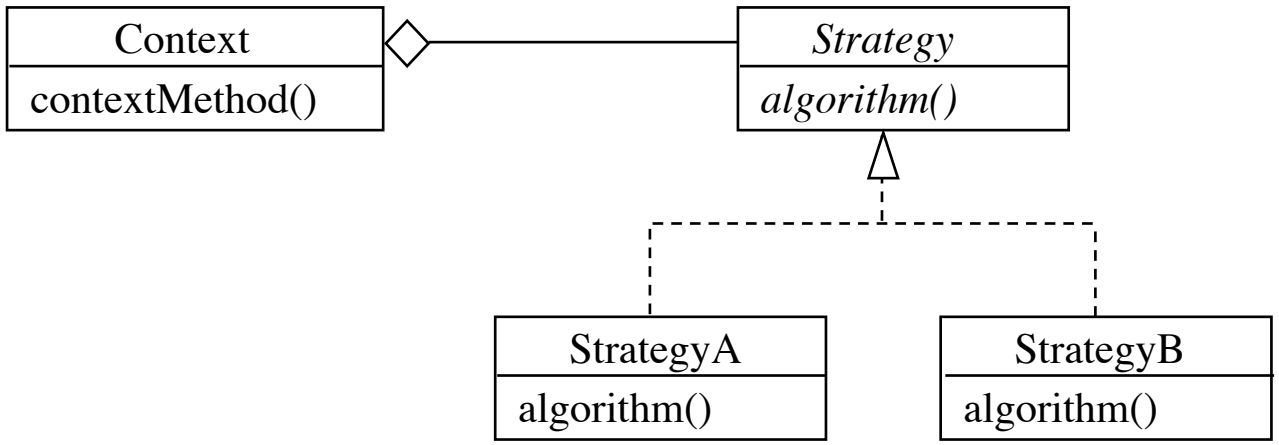
Ekesempel på anvendelse

(LayoutManager)



Designmønsteret Strategy
(fortsat)

Struktur:



Strategy er en variant af *Template Method*:
Ved *Strategy* er hook-metoderne placeret i separate klasser og kaldes ved delegering. Ved *Template Method* er hook-metoderne placeret i underklasser.

Designmønsteret Strategy
(fortsat)

Deltagere:

Strategy (f.eks. `Function`), der definerer en grænseflade, der er fælles for alle algoritmerne

ConcreteStrategy (f.eks. `Sine`), der implementer en algoritme, der bruger Strategy-grænsefladen

Context (f.eks. `MultiPlotter`), der har referencer til Strategy-objekter

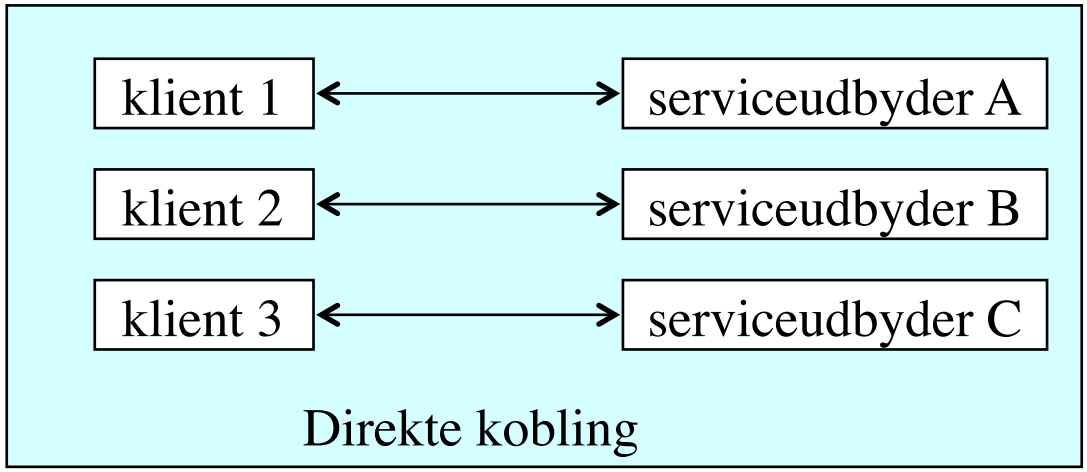
Abstrakt kobling

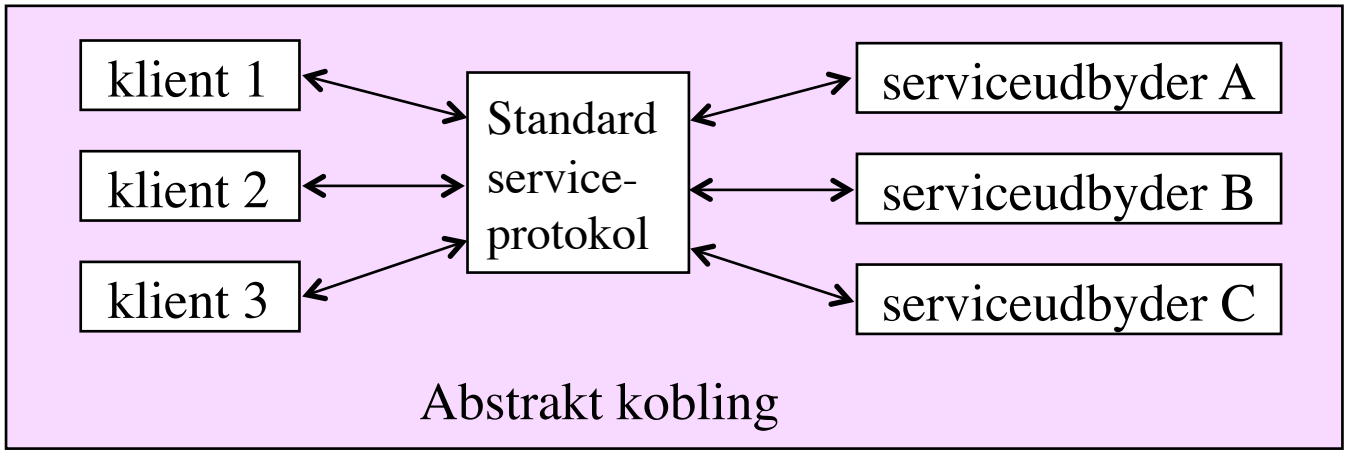
Abstrakt kobling: Klienter modtager service ved hjælp abstrakte klasser eller grænseflader uden at kende de aktuelle konkrete klasser.

Retningslinje for design:
Programmér til en grænseflade, ikke til en implementation.

Herved opnås fleksible og genbrugelige løsninger.

Direkte kobling versus abstrakt kobling





Abstrakte iteratorer

Et eksempel på brug af abstrakt kobling

En abstrakt iterator er en grænseflade, der specificerer faciliteter til sekventielt gennemløb af samlinger af objekter (f.eks. lister, tabeller og træer).

```
interface Iterator {
    Object next();
    boolean hasNext();
    void remove();
}
```

Eksempel på implementering af en iterator

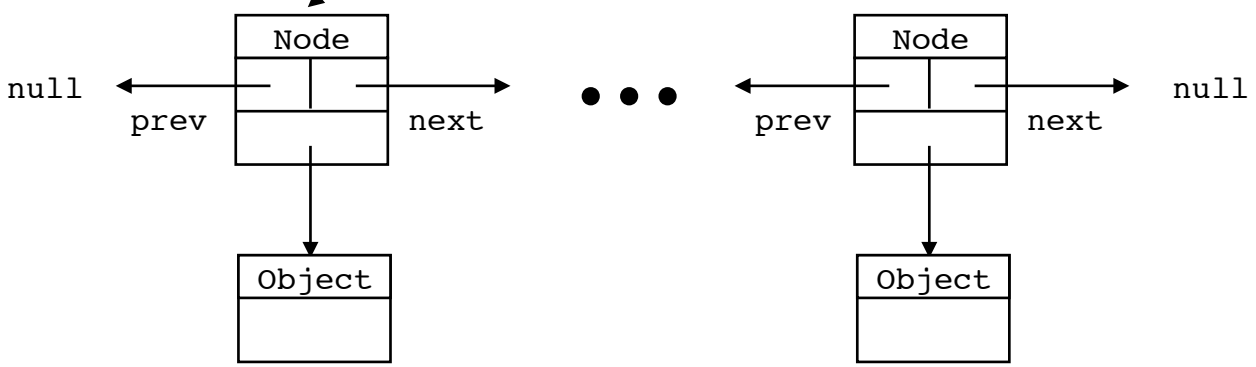
```
interface List {
    Iterator iterator();
    // other methods
}

public class LinkedList implements List {
    public Iterator iterator() {
        return new LinkedListIterator();
    }

    private class LinkedListIterator implements Iterator {
        // concrete methods for the Iterator interface
    }

    // other methods and constructors
}
```

Eksempel på brug

```
List list = new LinkedList();

...

Iterator itr = list.iterator();
while (itr.hasNext())
    System.out.println(itr.next());
```

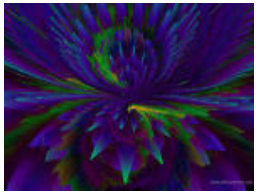
Der er programmeret til grænseflader (`List` og `Iterator`).
Klienten er derfor uberørt af de konkrete implementeringer.

```
public class LinkedList implements List {
    protected static class Node {
        Node prev, next;
        Object element;
    }

    protected Node first;
    //...
}
```



```
private class LinkedListIterator implements Iterator {
    private LinkedList.Node current;

    LinkedListIterator() { current = first; }

    public boolean hasNext() { return current != null; }

    public Object next() {
        Object obj = null;
        if (current != null) {
            obj = current.element;
            current = current.next;
        }
        return obj;
    }

    public void remove() {
        throw new UnsupportedOperationException();
    }
}
```

Designmønsteret Iterator



Kategori:

Adfærdsmæssigt designmønster

Hensigt:

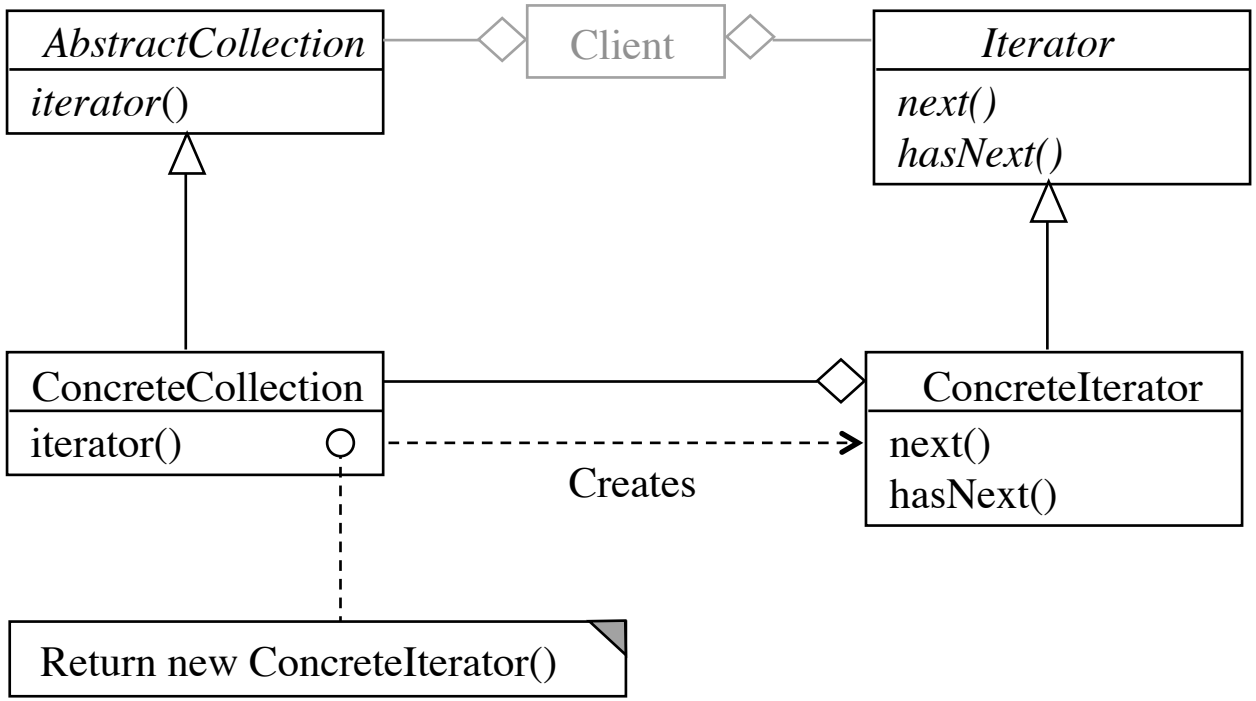
Giver en måde til sekventielt at gennemløbe en samling af objekter

Anvendelse:

- for at tilgå indholdet i en samling objekter uden at afsløre deres interne repræsentation
- for at muliggøre flere samtidige gennemløb af en samling af objekter
- for at understøtte polymorfe gennemløb

Designmønsteret Iterator
(fortsat)

Struktur:



Designmønsteret Iterator
(fortsat)

Deltagere:

Iterator (f.eks. `Iterator`), der definerer en grænseflade for gennemløb af en samling af objekter

ConcreteIterator (f.eks. `LinkedListIterator`), der implementer grænsefladen og holder sig ajour med den aktuelle position i samlingen

AbstractCollection (f.eks. `List`), der definerer en grænseflade for skabelse af en konkret iterator (f.eks. metoden *iterator*)

ConcreteCollection (f.eks. `LinkedList`), der implementerer iterator-metoden, så den returnerer en konkret iterator

Grænsefladen Iterable (Java 5.0)

```
interface Iterable {
    Iterator iterator();
}
```

Hvis en klasse implementerer `Iterable`, kan et objekt af klassen gøres til genstand for et for/each-gennemløb:

```
for (Obj element : list)
    System.out.println(element);
```

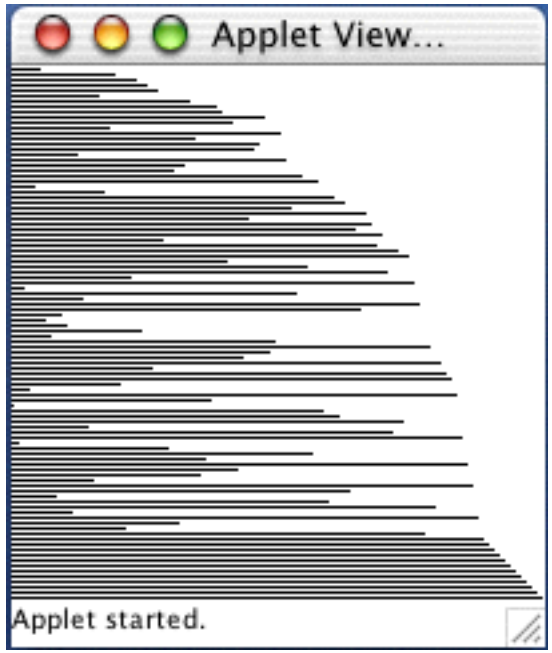
Arrays og Javas kollektionsklasser implementerer `Iterable`.

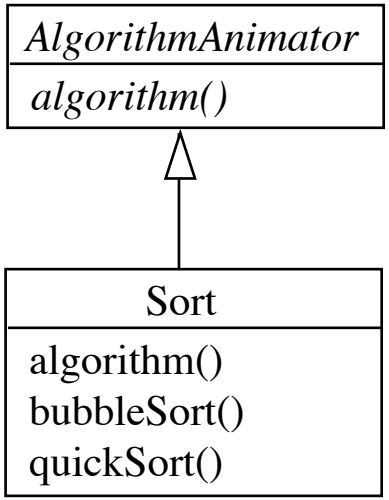
Retningslinjer for design

Adskil urelaterede funktionaliteter
Hvis en klasse indeholder komponenter, der tager sig af urelaterede ting, bør disse komponenter adskilles fra klassen.

Minimer grænseflader
Design den mindst mulige grænseflade for at minimere graden af kobling imellem komponenter.

Animering af sorteringsalgoritmer (1)





Class `AlgorithmAnimator`

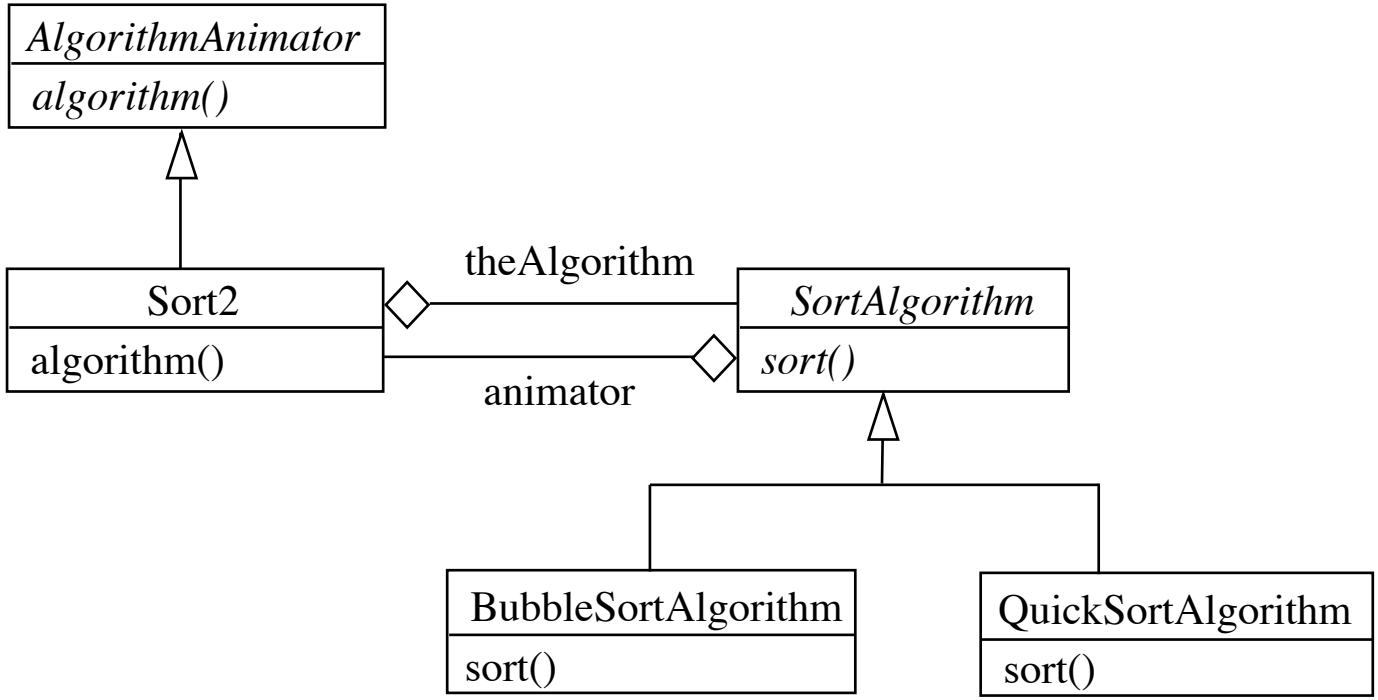
```
public abstract class AlgorithmAnimator
                extends DBAnimationApplet {
    // the hook method
    protected abstract void algorithm();

    // the template method
    public void run() {
        algorithm();
    }

    protected final void pause() {
        if (Thread.currentTread() == animationTread) {
            repaint();
            try {
                Thread.sleep(delay);
            } catch (InterruptedException e) {}
        }
    }
}
```

Class **Sort**

```
public class Sort extends AlgorithmAnimator {
    protected void initAnimator() {
        String at = getParameter("alg");
        algName = at != null ? at : "BubbleSort";
        scramble();
    }

    protected void algorithm() {
        if ("BubbleSort".equals(algName)
            bubbleSort(arr);
        else if ("QuickSort".equals(algName)
            quickSort(arr);
    }

    protected void paintFrame() { ... }

    private void scramble(); { arr = new int[...]; ... }
    private void bubbleSort(int[] arr) { ... }
    private void quickSort(int[] arr) { ... }

    private String algName;
    private int arr[];
}
```

Ufleksibel

Animering af sorteringsalgoritmer (2)



Tillader dynamisk udskiftning af sorteringsalgoritme

Class `SortAlgorithm`

```
public abstract class SortAlgorithm {
    abstract public void sort(int[] a);

    protected SortAlgorithm(AlgorithmAnimator animator) {
        this.animator = animator;
    }

    protected void pause() {
        if (animator != null)
            animator.pause();
    }

    protected void swap(int[] a, int i, int j) {
        int temp = a[i]; a[i] = a[j]; a[j] = temp;
    }

    private AlgorithmAnimator animator;
}
```

Hvor skal den konkrete instans af `SortAlgorithm` skabes?

En mulighed er at skabe instansen i metoden `initAnimator` i klassen `Sort2`.

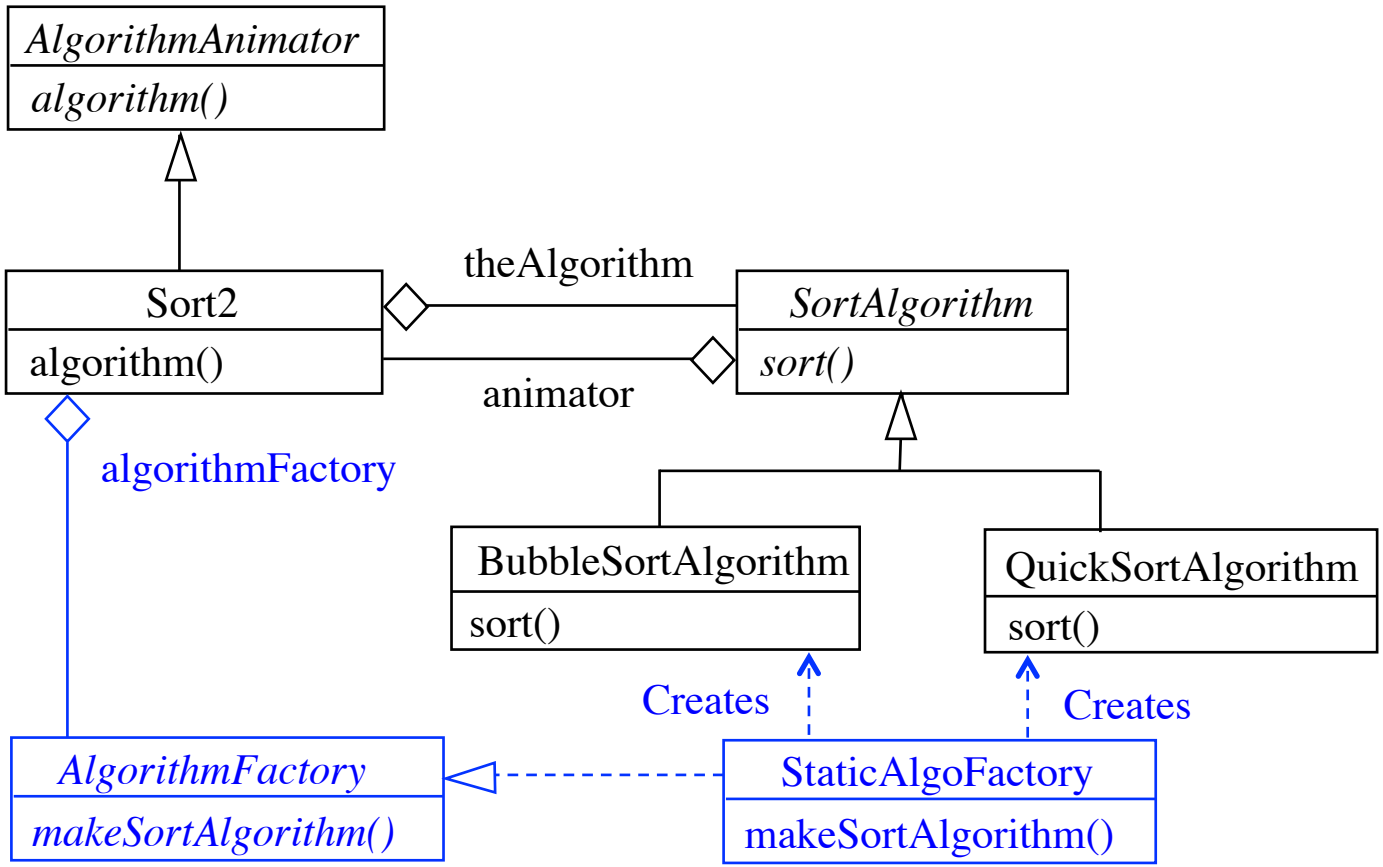
Ulempe:

Klienten (her `Sort2`) skal kende de konkrete `SortAlgorithm`-klasser (de konkrete strategier), hvilket gør det vanskeligt at skifte dem ud.

Løsning:

Indfør en klasse, hvis eneste forpligtelse er skabelse af konkrete `SortAlgorithm`-instanser.

Animering af sorteringsalgoritmer (3)
(med designmønsteret Factory)



Den abstrakte Factory

```
interface AlgorithmFactory {
    SortAlgorithm makeSortAlgorithm(String algName);
}
```

En konkret Factory

```
public class StaticAlgoFactory implements AlgorithmFactory {
    public StaticAlgoFactory(AlgorithmAnimator animator) {
        this.animator = animator;
    }

    public SortAlgorithm makeSortAlgorithm(String algName) {
        if ("BubbleSort”.equals(algName))
            return new BubbleSortAlgorithm(animator);
        else if ("QuickSort".equals(algName))
            return new QuickSortAlgorithm(animator);
        else
            return new BubbleSortAlgorithm(animator);
    }

    protected AlgorithmAnimator animator;
}
```

Designmønsteret Factory

Kategori:

Konstruerende designmønster

Hensigt:

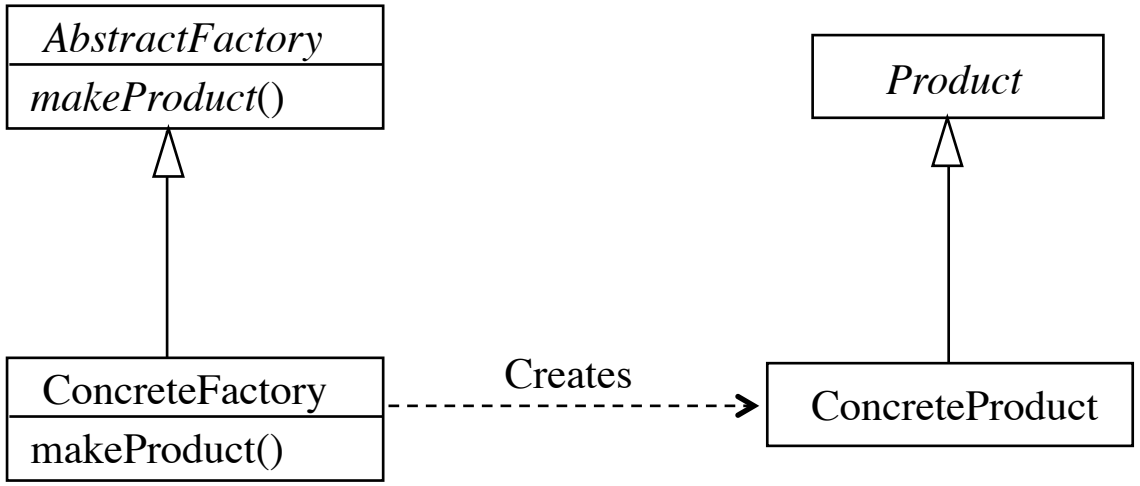
Definer en grænseflade for skabelse af objekter og lad underklasser afgøre, hvilke klasser, der skal instantieres, og hvordan

Anvendelse:

- når et system skal være uafhængigt af, hvorledes dets produkter skabes

Designmønsteret Factory
(fortsat)

Struktur:



Designmønsteret Factory
(fortsat)

Deltagere:

Product (f.eks. `SortAlgorithm`), der definerer en grænseflade for de objekter, som skal skabes

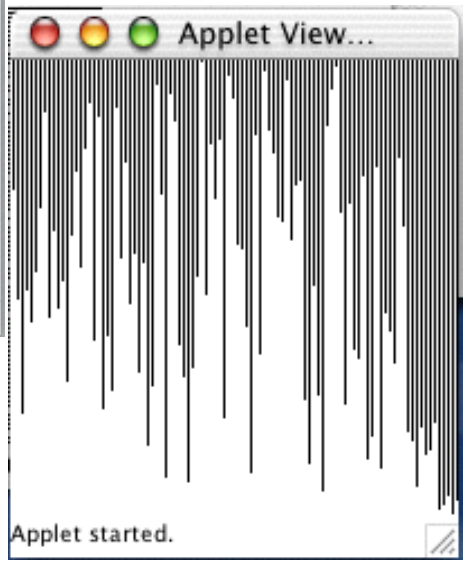
ConcreteProduct (f.eks. `QuickSortAlgorithm`), der implementer *Product*-grænsefladen

AbstractFactory (f.eks. `AlgorithmFactory`), der definerer en metode for skabelse af et *Product*-objekt

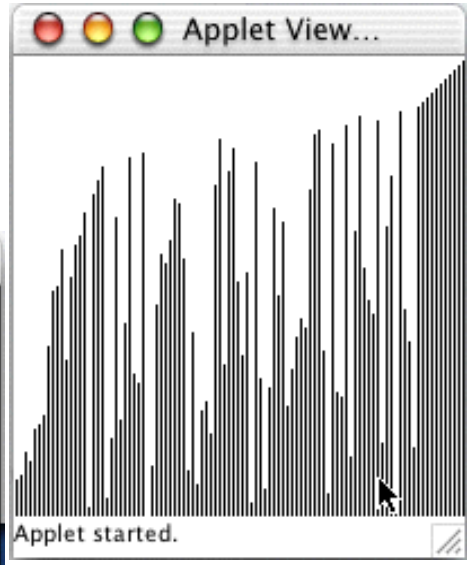
ConcreteFactory (f.eks. `StaticAlgoFactory`), der implementerer metoden til skabelse af *ConcreteProduct*-objekter

Valg af udseende

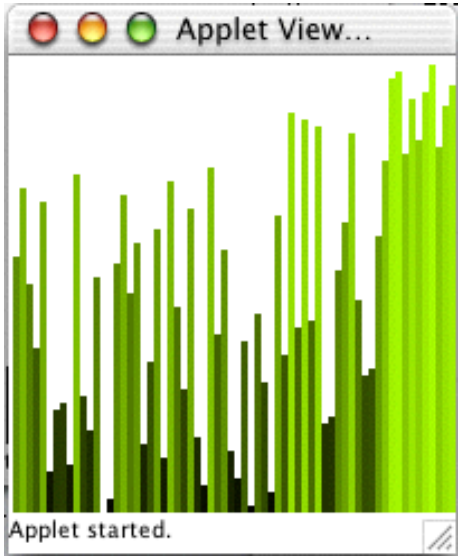
Horizontal



Vertical

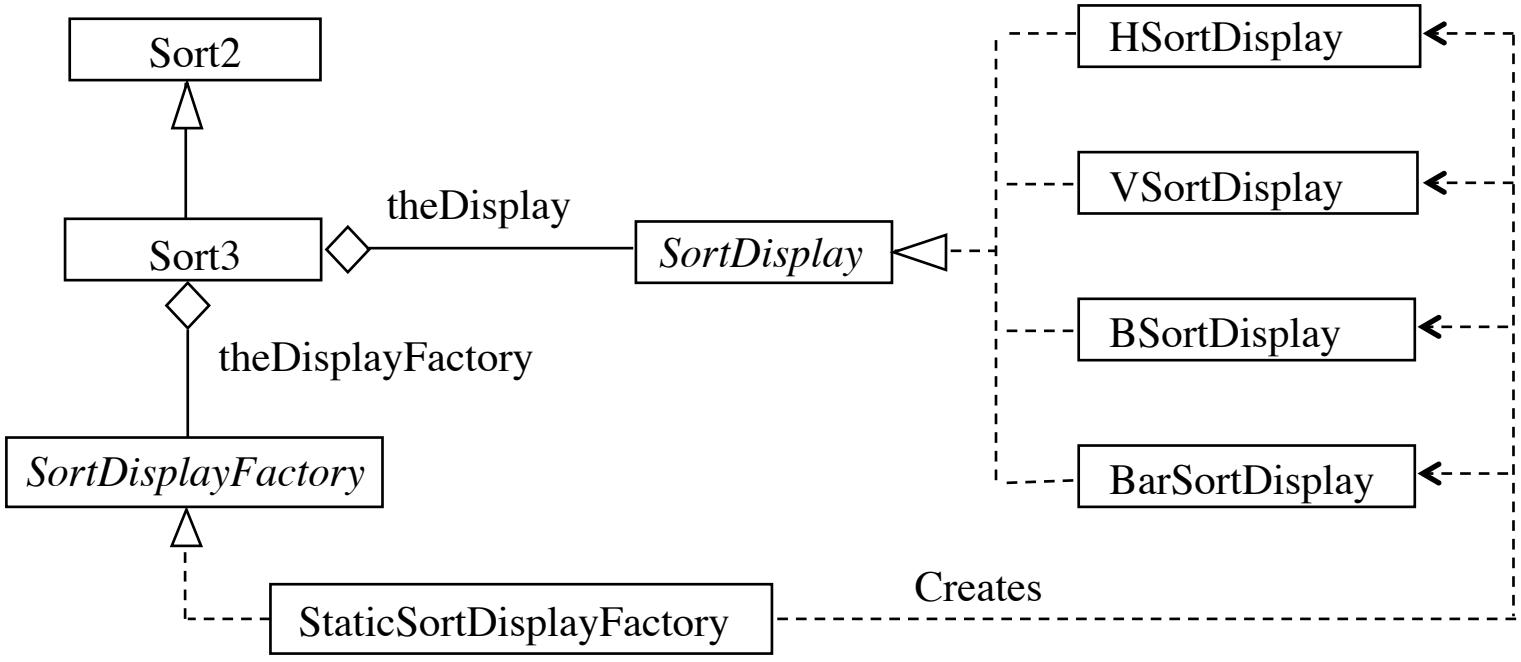


Bottom



Bar

Animering af sorteringsalgoritmer (4)



SortDisplay

```
interface SortDiplay {
    void display(int[] a, Graphics g, Dimension d);
}
```

Anvendelse i `Sort3`:

```
protected void initAnimator() {
    String att = getParameter("dis");
    theDisplayFactory = new DisplayFactory();
    theDisplay = theDisplayFactory.makeSortDisplay(att);
    super.initAnimator();
}


protected void paintFrame(Graphics g) {
    theDisplay.display(arr, g, getSize());
}
```

En dynamisk Factory

```
import java.lang.reflect.*;

public class DynamicAlgoFactory implements SortAlgorithmFactory {
    public SortAlgorithm makeSortAlgorithm(String algName) {
        try {
            Class classForName = Class.forName(algName + "Algorithm");
            Class animatorClass = animator.getClass();
            Constructor constructor = null;
            do {
                try {
                    constructor = classForName.getDeclaredConstructor(
                                          new Class[] { animatorClass });
                } catch (NoSuchMethodException e) {
                    animatorClass = animatorClass.getSuperclass();
                }
            } while (constructor == null);
            return (SortAlgorithm) constructor.newInstance(
                                        new Object[] { animator });
        } catch (Exception e) {
            throw new RuntimeException(algName + ": " + e.toString());
        }
    }
```

En generisk Factory

```
public class GenericFactory {
    public static Object getInstance(String className) {
        try {
            return Class.forName(className).newInstance();
        } catch (Exception e) {
            throw new RuntimeException(className + ": " + e.toString());
        }
    }

    public static Object getInstance(String className, Object parameter) {
        // ...
    }
}
```

fortsættes

```
public static Object getInstance(String className, Object parameter) {
    try {
        Class classForName = Class.forName(className);
        Class parameterClass = parameter.getClass();
        Constructor constructor = null;
        do {
            try {
                constructor = classForName.getDeclaredConstructor(
                                        new Class[] { parameterClass });
            } catch (NoSuchMethodException e) {
                parameterClass = parameterClass.getSuperclass();
            }
        } while (constructor == null);
        return constructor.newInstance(new Object[] { parameter });
    } catch (Exception e) {
        return null;
    }
}
```

Anvendelse af den generiske Factory

```
theAlgorithm = (SortAlgorithm) GenericFactory.
                getInstance(algName + "Algorithm", animator);


theDisplay = (SortDisplay) GenericFactory.
             getInstance(displayName + "Display");
```

Ugeseddel 4

21. september - 28. september

- Læs følgende sider i kapitel 8 i lærebogen:
  s. 305 - 332 og s. 366 – 395 (afsnit 8.3 overspringes).

- Begynd at overveje, hvad afleveringsopgaven skal omhandle.
  Læs vejledningen, der kan hentes via kursets hjemmeside.

- Løs opgaverne 7.3 og 7.5.