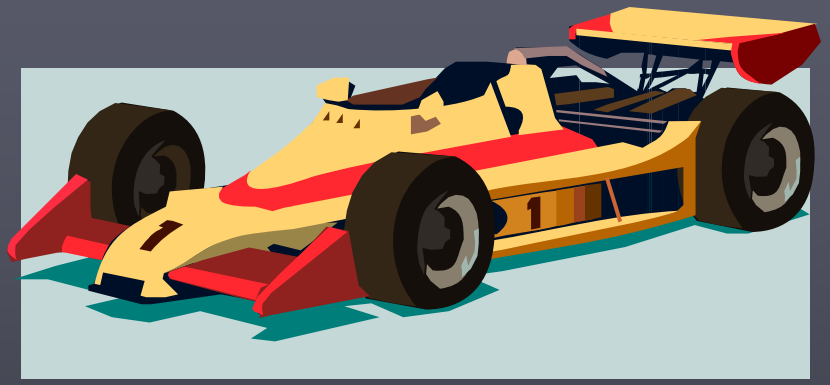
Objektorientering och händelsebaserad programmering

Gustav Taxén
gustavt@csc.kth.se

2D1640 Grafik och Interaktionsprogrammering VT 2007

Racingspel



Vi ska göra ett enkelt racingspel med olika sorters fordon.

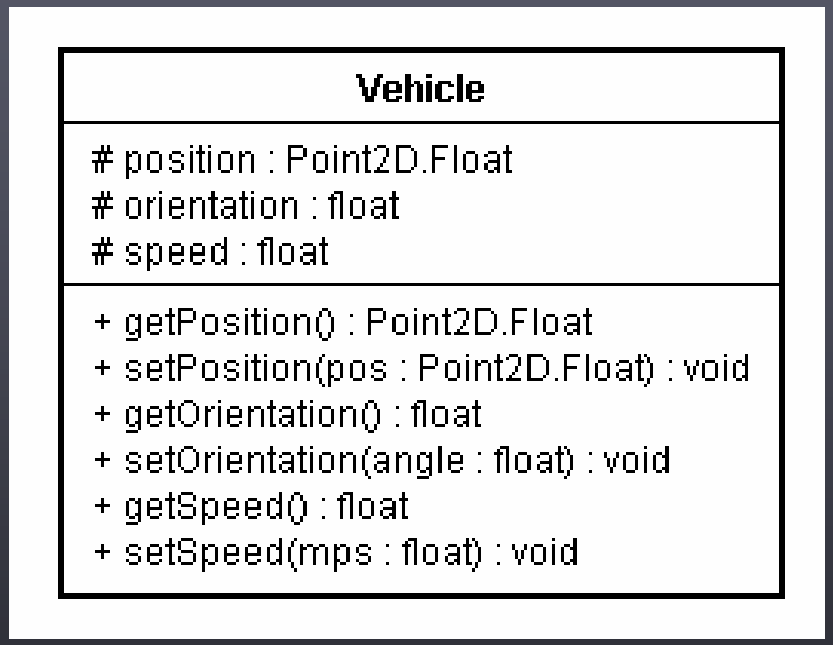
Vilka egenskaper har ett fordon?

- Position
- Orientering
- Hastighet
- Grafik / utseende

Racingspel

```
class Vehicle {
  protected Point2D.Float position;
  protected float orientation;
  protected float speed;

  public Point2D.Float getPosition();
  public void setPosition(Point2D.float p);
  public float getOrientation();
  public void setOrientation(float angle);
  public float getSpeed();
  public void setSpeed(float mps);
}
```

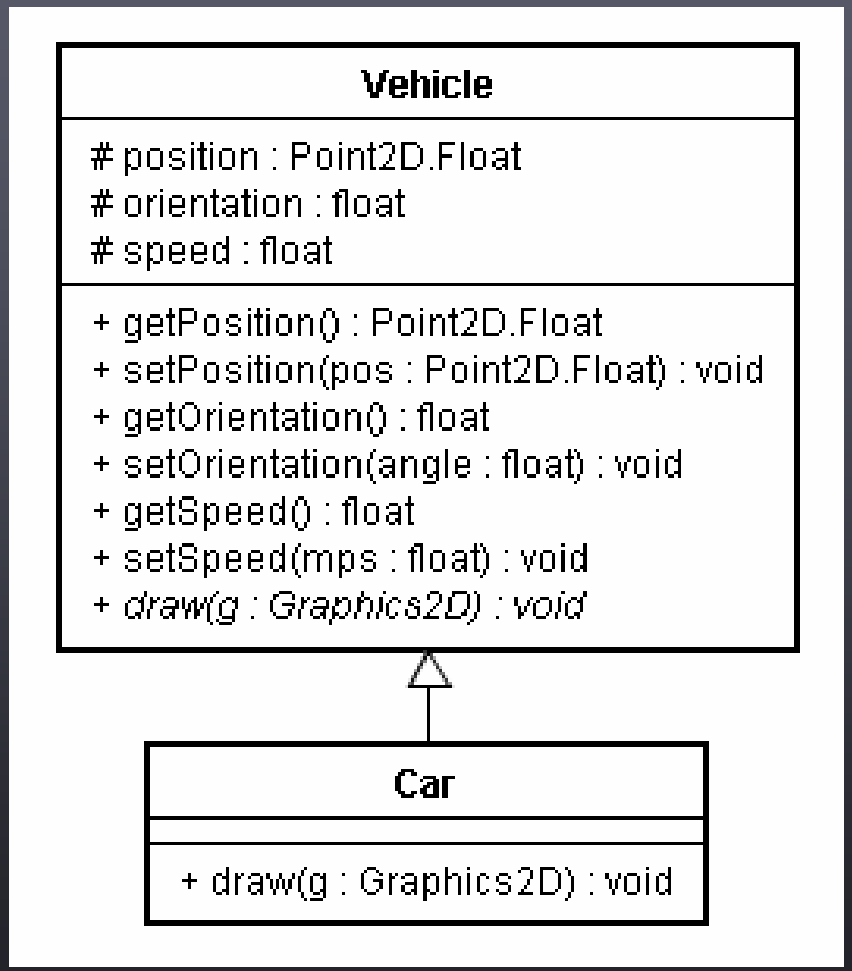
Racingspel



- Klassnamn
- Data
- Metoder

- # = protected
- + = public
- - = private

Racingspel



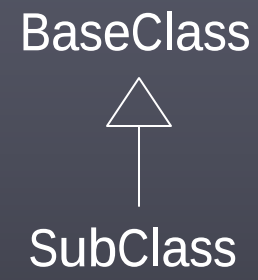
- Varje typ av fordon vet hur det ska rita ut sig självt
- Så vi låter **Vehicle** bli en abstrakt klass med en metod **draw**, som alla fordon måste implementera

Objektorientering i allmänhet

- Ett objekt är en "låda" där man samlar
  - **Data**
  - **Metoder** som (oftast) manipulerar denna data
- Den grundläggande principen för objektorientering är att **data ska vara gömd** för "utomstående" klasser
- Undantaget är subklasser till basklassen, dessa kan eventuellt få accessa data direkt

Objektorientering i allmänhet

```
public class BaseClass {
  private int aNumber;
  protected int anotherNumber;
  public int add(int n) {
    aNumber += n;
    return aNumber;
  }
}

public class SubClass extends BaseClass {
  public void subtract(int n) {
    anotherNumber -= n;
  }
}
```



SubClass kan inte komma åt **aNumber** eftersom den är satt som **private**.

Objektorientering i allmänhet

```
BaseClass b = new BaseClass();
SubClass s = new SubClass();
int n;

n = b.add(10);
n = s.add(10);
s.subtract(3);
```

- Subklassen **ärver** alla basklassens metoder och data.
- En **instans** av subklassen innehåller alltså all data och alla metoder i basklassen plus eventuell data och metoder i subklassen.

Överlagring

- Det är helt OK att skriva olika versioner av samma metod.
- Det som inte får skilja mellan dem är typen på det som returneras.
- Detta brukar kallas **överlagring** (overloading). Man talar ibland också om polymorfism.

```
public class ClassA {
  public void f(int n) {
    ...
  }
  public void f(float f) {
    ...
  }
  public void f(String s) {
    ...
  }
}

ClassA a = new ClassA();
a.f(10);
a.f(3.4f);
a.f("Hello");
```

Överlagring

- En subklass kan ändra beteendet hos en metod.
- Det är också en form av överlagring.

```
public class ClassA {
  public int f() {
    return 10;
  }
}

public class ClassB extends ClassA {
  public int f() {
    return 15;
  }
}

ClassA a = new ClassA();
ClassB b = new ClassB();

int n = a.f();   // 10
int m = b.f();   // 15
```

Konstruktorer och destruktorer

- Klassens **konstruktor** anropas när en instans skapas.
- En default-konstruktor skapas alltid som gör ingenting.
- I t.ex. C++ finns också en **destruktor** som anropas innan instansen tas bort ur minnet.

```
public class MyClass {
  private int n;
  public MyClass(int number) {
    n = number;
  }
  public int getNumber() {
    return n;
  }
}


MyClass c = new MyClass(10);
int n = c.getNumber();
```

Gömda klassmedlemmar

- Man använder ordet **super** för att hänvisa till superklassens uppsättning metoder och variabler.
- Det är det enda sättet att komma åt metoder eller variabler som **gömts** av subklassen.

```
public class Base {
  public boolean aVariable;
  public void aMethod() {
    aVariable = true;
  }
}

public class Sub extends Base {
  public boolean aVariable;
  public void aMethod() {
    aVariable = false;
    super.aMethod();
    aVariable = 10;
    super.aVariable = 15;
  }
}
```

Abstrakta klasser

```
interface MyInterface {
  void add(int n);
}

public class MyClass implements MyInterface {
  void add(int n) {
    ...
  }
}
```

Ibland vill man "tvinga" att en klass implementerar vissa metoder. Det bästa sättet att åstadkomma det är med en s.k. **abstrakt klass**, d.v.s. en klass som talar om vilka metoder som ska finnas i subklassen men som inte bryr sig om hur de är implementerade.

I java kallas abstrakta klasser för **interfaces**.

Abstrakta klasser

```
abstract class MyClass {
  abstract void add(int n);

  void subtract(int n) {
    ...
  }
}
```

Man kan också göra klasser som är delvis abstrakta, d.v.s. har en eller flera abstrakta metoder kombinerat med vanliga metoder och data.

Final-klasser

- En klass som är deklarerad som **final** kan man inte göra subklasser till.
- En metod som är deklarerad som **final** kan inte överlagras.
- En variabel som är deklarerad som **final** kan aldrig ändra värde.

```
final class MyClass {
  ...
}

public class MyClass2 {
  final int f() {
    ...
  }
}

final int c = 10;
```

Meddelanden

- I java-sammanhang brukar man säga att klasser kommunicerar via **meddelanden**.
- Man skickar ett meddelande genom att helt enkelt anropa en metod i en annan klass.

```
public class ClassA {
  public void f(int n) {
    ...
  }
}

public class ClassB {
  private ClassA a;
  public void setA(ClassA a) {
    this.a = a;
  }
  public void sendMsg(int n) {
    a.f(n);
  }
}

ClassB b = new ClassB();
ClassA a = new ClassA();
b.setA(a);
b.sendMsg(10);
```

Instans- och klassvariabler

- Det finns **instansvariabler** och **klassvariabler**.
- Instansvariabler har en egen "kopia" i varje instans.
- Klassvariablerna finns det bara en av.
- Man använder ordet **static** för att ange att en variabel eller metod är en klassvariabel.

```
public class MyClass {
  public int n;
  public static int m;
  public int f() {
  }
  static public int g() {
  }
}

MyClass c = new MyClass();
int n = c.f();
int m = c.g();
int p = MyClass.g();
int q = MyClass.m;
```

Pass by value vs. pass by reference

- I java är alla parametrar till metoder alltid **pass by value**, d.v.s. metoden använder en lokal kopia av de variabler som skickades till den.
- Det gäller även referenser till objekt.
- I andra språk finns också **pass by reference**, d.v.s. man skickar det faktiska objektet.

```
public class MyClass {
  public void f(int n) {
    n = 20; // n är lokal variabel
  }
}

MyClass c = new MyClass();
int v = 10;
c.f(v); // ändrar inte v!
```

Garbage collection

- I Java behöver man inte säga till när objekt ska frigöras.
- Istället finns en s.k. **garbage collector** som rensar minne för objekt som saknar referenser.
- Man kan "släppa" en referens explicit genom att sätta den till **null**.

```
public class MyClass {
  ...
}

MyClass c = new MyClass();
c = null;
```

Type casting

- Ibland måste man **omtolka** en variabel (type casting).
- Vissa omtolkningar är inte tillåtna - i så fall varnar kompilatorn.
- Men ibland kan den inte avgöra om man gjort rätt, så var försiktig.

```
public class Base {
  ...
}
public class Sub1 extends Base {
  ...
}
public class Sub2 extends Base {
  ...
}

int a = 10;
float b = (float)a;
Base[] c = new Base[10];
c[0] = new Sub1();
c[1] = new Sub2();
if (c[0] instanceOf Sub1) {
  Sub1 s = (Sub1)c[0];
}
Sub2 t = (Sub2)c[0]; // VARNING!
```

Inre klasser

- Man kan skapa klasser inuti en klass.
- Dessa kan bara instansieras inuti den omslutande klassen.
- En inre klass har access till alla variabler och metoder i den omslutande klassen.

```
public class MyClass {
  private int n;

  public class MyInnerClass {
    public int f() {
      n = 10;
    }
  }

  MyInnerClass c;

  public void g() {
    c.f();
  }
}
```

Arrays

```
int[] num = new int[10];
for (int j = 0; j < 10; j++) {
  num[j] = j;
}

int[] num2 = { 0, 1, 2 };

System.out.println(num.size);  // Ger 10
System.out.println(num2.size); // Ger 3

int[][] twoDim = new int[2][3];
twoDim[0][0] = 0;

int[][] twoDimDyn = new int[2][];
twoDimDyn[0] = new int[10];
twoDimDyn[1] = new int[3];
```

Strängar

```
char c1 = 'a';

String s1 = new String("Hello");
char c2 = s1.charAt(0); // Ger 'H'

char[] c3 = { 'H', 'e', 'l', 'l', 'o' };
String s2 = new String(c3);
System.out.println(s2.length()); // Ger 5

String s3 = s1 + s2;
System.out.println(s3); // Ger "HelloHello"
```

Namespaces

- I java kan man låta klasser tillhöra ett s.k. **package**.
- Det gör att det är enklare att skilja på klasser som råkar ha samma namn.
- Ta för vana att lägga dina klasser i ett eget package!

```
package myPackage;

import java.swing.*;

public class Base {
   ...
}
```

UML

- Både en mjukvarudesignprocess och en serie diagramtyper.
- Utvecklades av företaget Rational i mitten av 90-talet.
- Är idag standardiserat och används överallt i industrin.
- Ger dock mycket begränsad info om hur användargränssnittet ska se ut!

UML: Use case diagrams

- En abstraktion av en eller flera processer, d.v.s. ett **scenario**.
- Visar **aktörer** och **use cases**, d.v.s. vad aktörerna gör.

http://pigseye.kennesaw.edu/~dbraun/csis4650/A&D/UML_tutorial/index.htm

UML: Klassdiagram

http://pigseye.kennesaw.edu/~dbraun/csis4650/A&D/UML_tutorial/index.htm

UML: Interaktionsdiagram

- Visar hur meddealanden skickas mellan klassinstanser i systemet.
- Ger en översikt över processer i systemet.

http://pigseye.kennesaw.edu/~dbraun/csis4650/A&D/UML_tutorial/index.htm

UML: Tillståndsdiagram

- Visar hur systemet går mellan olika tillstånd.

http://pigseye.kennesaw.edu/~dbraun/csis4650/A&D/UML_tutorial/index.htm

UML: Aktivitetsdiagram

- Liknar tillståndsdiagrammet men fokuserar på aktivitet istället.
- En aktivitet behöver inte vara detsamma som ett tillstånd.
- Kan motsvara funktioner som användaren aktiverar.

http://pigseye.kennesaw.edu/~dbraun/csis4650/A&D/UML_tutorial/index.htm

UML: Fysikdiagram

- Används för att visa relationen mellan hård- och mjukvara i systemet.
- Kan också användas för att visa hur systemet är distribuerat över olika maskiner, t.ex. i ett nätverk.

http://pigseye.kennesaw.edu/~dbraun/csis4650/A&D/UML_tutorial/index.htm

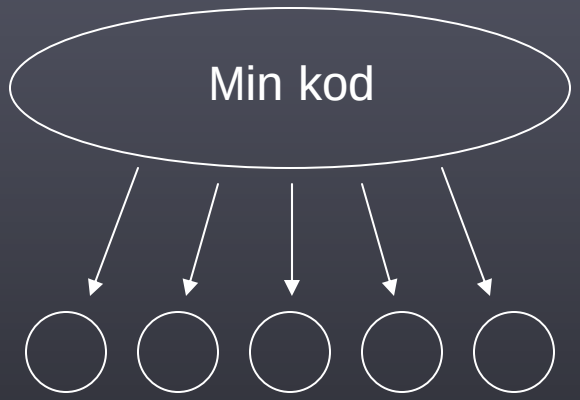
Fönstersystem

Frameworks

- Erfarenheten visar att det kan vara knepigt att hantera toolkits.
- Om allt är "tillåtet" är det lätt att applikationer blir inkonsekventa eller ser "fula" ut.
- Ett **framework** är en uppsättning klasser (abstrakta och vanliga) som "påtvingar" ett visst sätt att arbeta.
- Javas framework för fönsterhantering heter Swing.
- Microsofts motsvarighet heter MFC (som nu är på väg att ersättas av .NET).

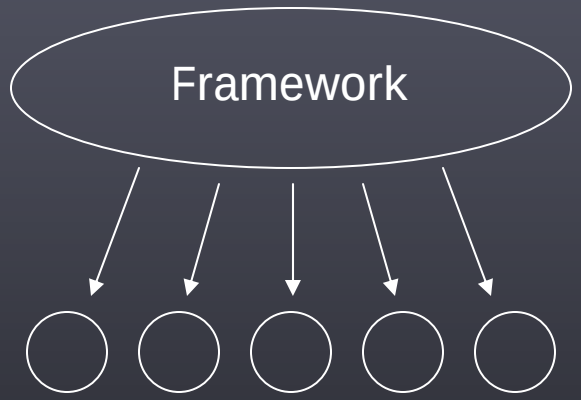
Frameworks

Koda med klassbibliotek



Kod i klassbibliotek

Koda med framework



Min kod,
brukar kallas för **callbacks**.

Frameworks

Ett vanligt sätt att arbeta med frameworks är att man subklassar någon form av basklass med grundfunktionalitet. Sedan överlagrar man de metoder som man intresserar sig för.

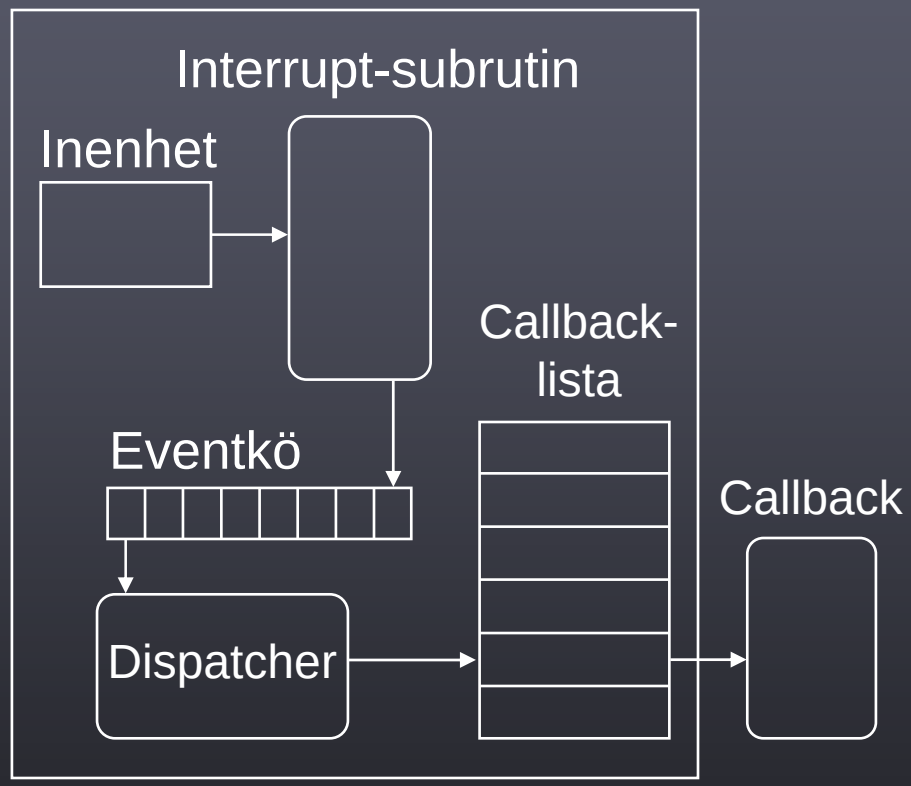
```
import java.awt.*;
import javax.swing.*;

public class MyFrame extends JFrame {
  public void paint(Graphics g) {
    g.drawString("Hello", 10, 50);
  }

  public static void main(String[] args) {
    MyFrame f = new MyFrame();
  }
}
```

Inmatningstyper

- Request
  - Lämna över kontrollen till användaren och vänta på att något sker.
  - Exempel: terminal i Unix.
- Sampling / polling
  - Läsa av värdet hos en enhet så ofta som möjligt.
  - Exempel: Spela in ljud från ljudkort

Inmatningstyper: Events

- Systemet placerar förändringar i en kö.
- Beta av kön vid lämpliga tillfällen.
- Skicka en s.k. **event-instans** med information om varje förändring till alla callbacks som är "intresserade".



Design patterns

- Utgår från arkitekten Alexanders arbete på 70-talet.
- Alexander försökte se mönster i hur man löst återkommande problem inom arkitektur och skrev en bok där han beskriver lösningarna.
- Idén togs upp på allvar inom systemdesign i mitten av 90-talet.
- Den mest kända boken är "Design Patterns: Elements of Reusable Object-Oriented Software" av Gamma, Helm, Johnson, och Vlissides.

Design patterns

- Namn och generell typ
- Intent (vad den gör)
- Also Known As
- Motivation
- Applicability
- Structure (ett UML-diagram)

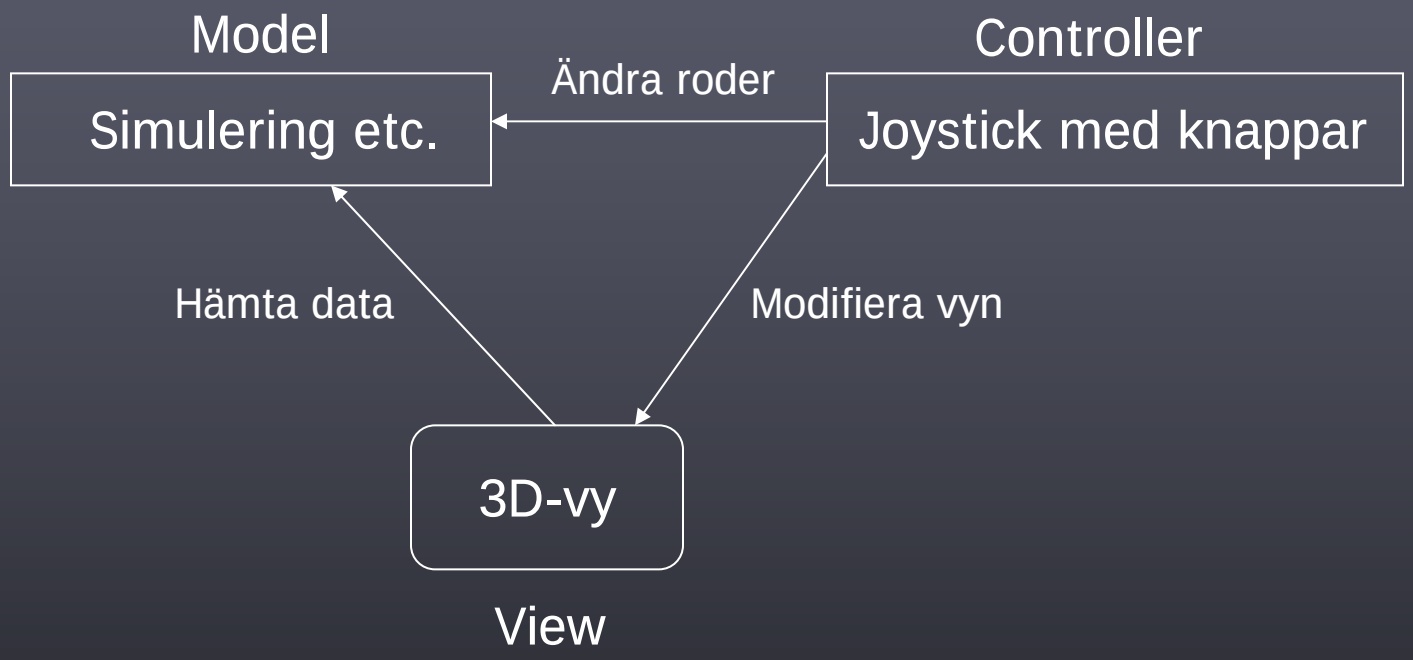
Design patterns

- Participants (beskr. av klasser som ingår)
- Collaborations (hur klasserna arbetar tillsammans)
- Consequences (av att använda mönstret)
- Implementation
- Sample code
- Known uses
- Related patterns

Model View Controller

- En design pattern för användargränssnitt.
- Utvecklades under sent 70-tal, bl.a. för NeXT-datorn.
- Exempel: Flygsimulator
  - **Model** - simuleringsrutiner, etc.
  - **Controller** - joystick kopplad till datorn
  - **View** - vy genom cockpitfönstret

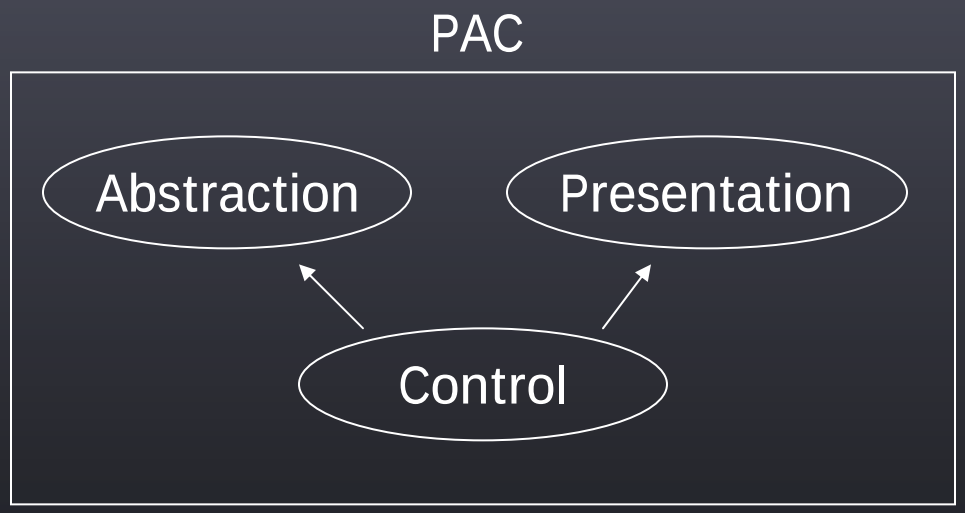
Model View Controller

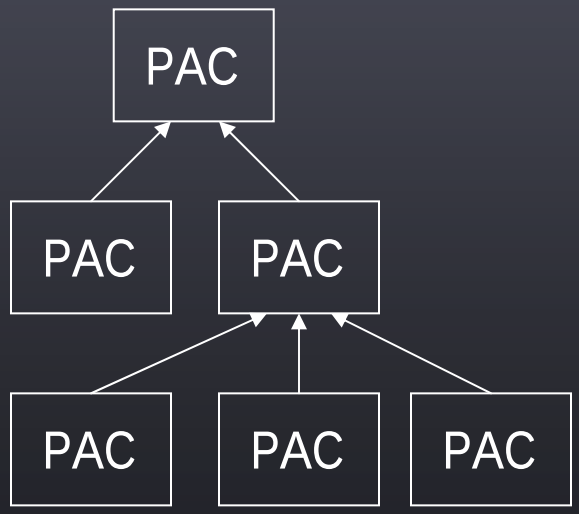


Om gränssnitten mellan de tre komponenterna inte ändras kan vilken som helst av dem bytas ut!

Presentation Abstraction Control

- Variant av MVC som hanterar mer komplexa gränssnitt.
- Utvecklades under mitten av 80-talet.



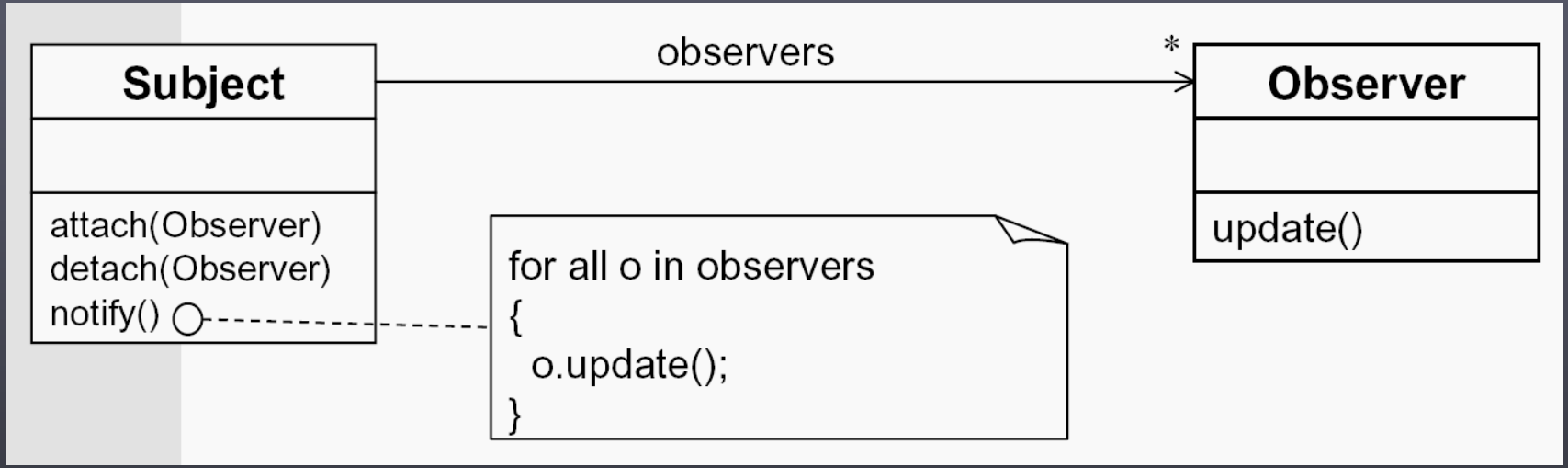


Control-komponenten skickar förändringar uppåt i hierarkin.

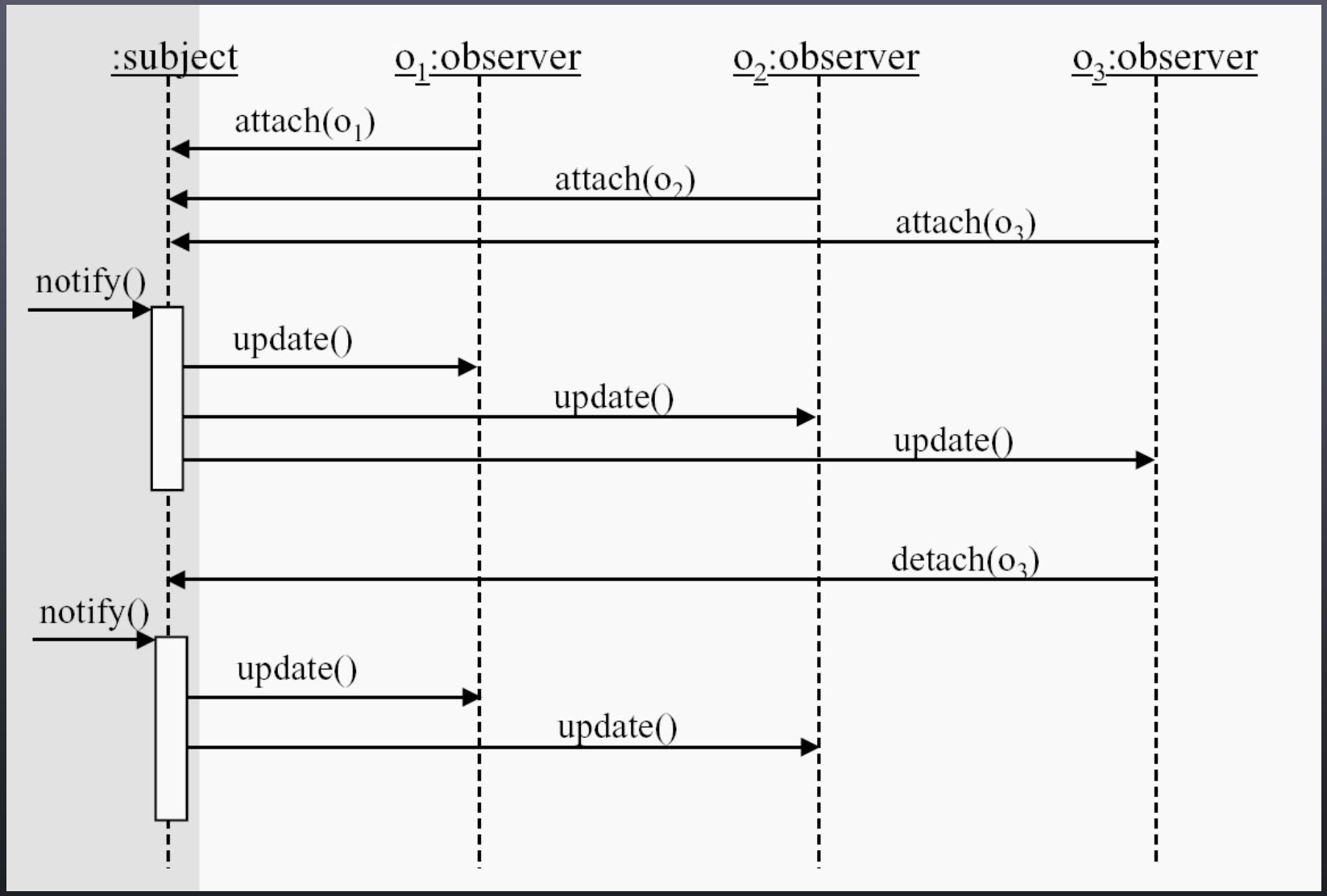
Observer

- Ett av de vanligaste designmönstren.
- Tillåter att ett objekt meddelas om det gjorts en förändring i ett annat objekt, utan att objekten görs starkt knutna till varandra.

Observer


Subject
observers
*
Observer
attach(Observer)
detach(Observer)
notify()
update()
for all o in observers
{
o.update();
}

Observer


:subject
o1:observer
o2:observer
o3:observer
attach(o1)
attach(o2)
attach(o3)
notify()
update()
update()
update()
detach(o3)
notify()
update()
update()

Observer i Java

```
package java.util;

public interface Observer {
  void update(Observable o, Object arg);
}

public class Observable {
  private Observer[] arr;

  public void addObserver(Observer o) {
    arr.addElement(o);
  }

  public void notify(Object arg) {
    for (int i = 0; i < arr.size; i++) {
      arr[i].update(this, arg);
    }
  }
}
```

Det här är inte hela sanningen, men principen är densamma i den riktiga java-implementationen!

Implementering av MVC

- Bygger på observer-mönstret.
- Vy-instanserna registrerar sig hos modellen.
- När modellen ändras anropar den update() hos vyerna.
- I exemplet ni fått utdelat har modellen en särskild metod (addPoint) som controllern anropar.
- Controller är här ihopbakad med vyerna.

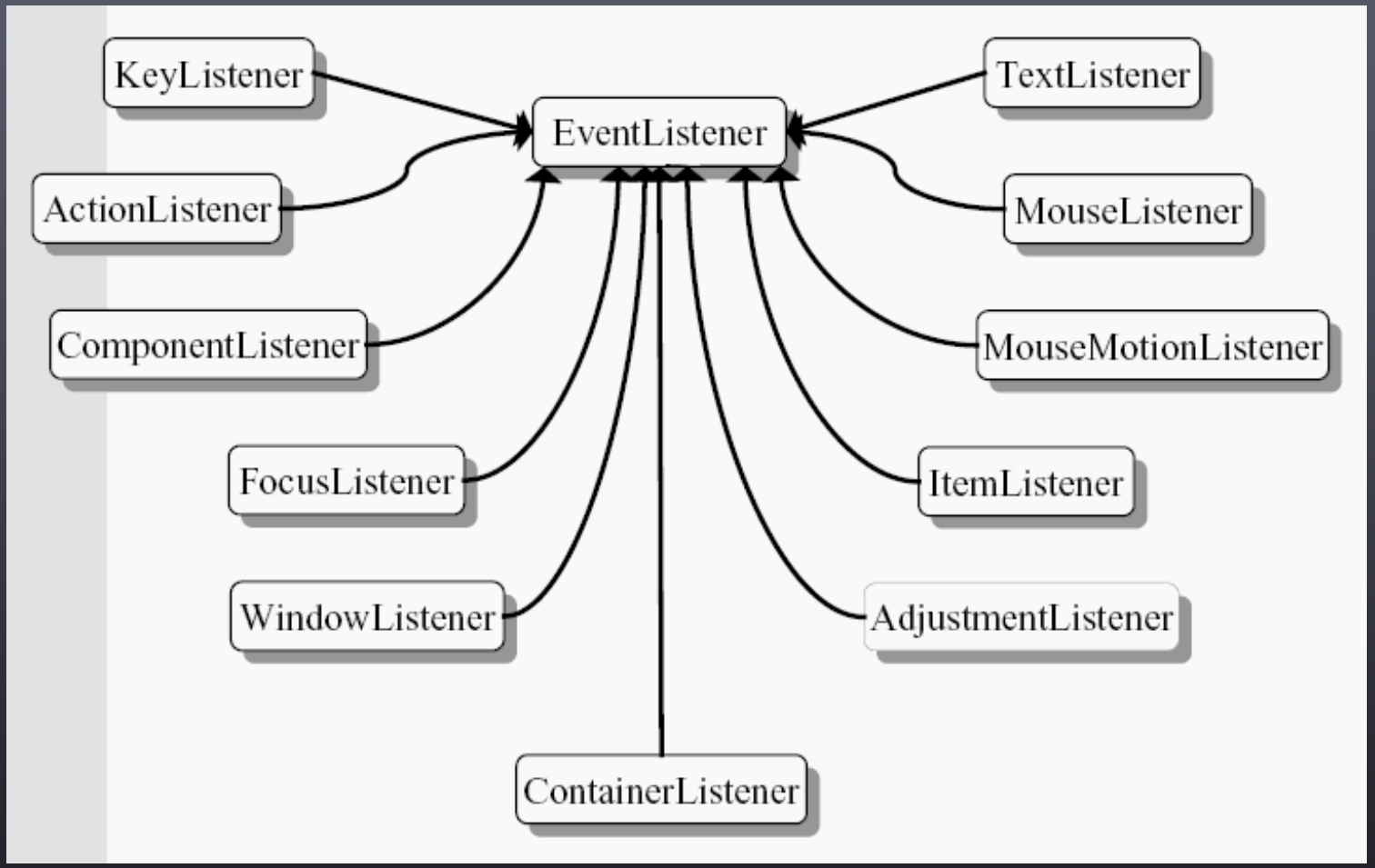
Problem med MVC

- Exemplet visar att det kan vara svårt att separera kontroller och vy i moderna grafiksystem.
- Men det lönar sig i princip alltid att separera datamodellen från applikationsmodellen, d.v.s. att hantera data på ett sådant sätt att det inte är beroende av hur det presenteras eller uppdateras.

Events i Java

- Påminner mycket om observer-mönstret, men de som "lyssnar" måste implementera mer än en metod.
- Varje metod motsvarar något som hänt hos enheten.

Events i Java



Events i Java

```
package MyTests;
import java.awt.*;
import java.awt.event.*;
public class MyFrame1 extends Frame implements WindowListener{
  public void windowOpened(WindowEvent e) {}
  public void windowClosing(WindowEvent e) {
    System.exit(0);
  }
  public void windowClosed(WindowEvent e) {}
  public void windowIconified(WindowEvent e) {}
  public void windowDeiconified(WindowEvent e) {}
  public void windowActivated(WindowEvent e) {}
  public void windowDeactivated(WindowEvent e) {}
  public static void main(String [] args) {
    MyFrame1 frame = new MyFrame1();
    frame.addWindowListener(frame);
    frame.setVisible(true);
  }
}
```

...som inre klass

```
package MyTests;
import java.awt.*;
import java.awt.event.*;
class MyWindowListener implements WindowListener {
  public void windowOpened(WindowEvent e) {}
  public void windowClosing(WindowEvent e) {
    System.exit(0);
  }
  public void windowClosed(WindowEvent e) {}
  public void windowIconified(WindowEvent e) {}
  public void windowDeiconified(WindowEvent e) {}
  public void windowActivated(WindowEvent e) {}
  public void windowDeactivated(WindowEvent e) {}
}

public class MyFrame2 extends Frame {
  public static void main(String [] args) {
    Frame frame = new MyFrame2();
    frame.addWindowListener(new MyWindowListener());
    frame.setVisible(true);
  }
}
```

Adaptors

- Förenklar hanteringen av events.
- Varje adaptor implementerar ett Listener-interface, men alla metoderna är tomma.
- Om man subklassar en adaptor behöver man alltså bara skriva precis den kod som behövs.

Adaptors

```
class MyWindowAdapter extends WindowAdapter {
  public void windowClosing(WindowEvent e) {
    System.exit(0);
  }
}

public class MyFrame3 extends Frame {
  public static void main(String [] args) {
    Frame frame = new MyFrame3();
    frame.addWindowListener(new MyWindowAdapter());
    frame.setVisible(true);
  }
}
```

...eller med anonym subklass

```
public class MyFrame4 extends Frame {
  public static void main(String [] args) {
    Frame frame = new MyFrame4();

    frame.addWindowListener(new WindowAdapter (){
      public void windowClosing(WindowEvent e) {
        System.exit(0);
      }
    });

    frame.setVisible(true);
  }
}
```