Mer om OO-programmering och modellering

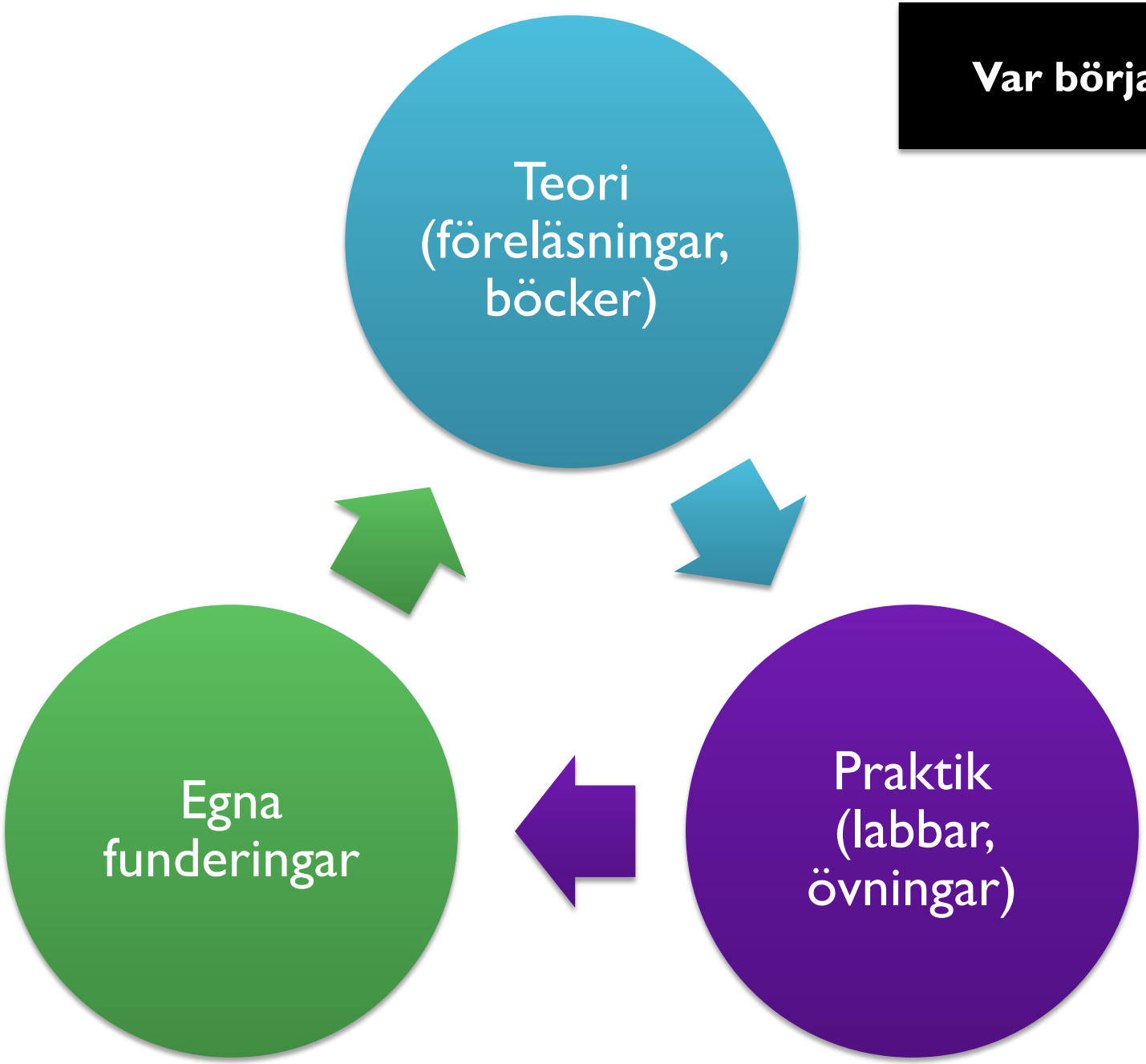
Principer och konkreta projektråd

jonas.kvarnstrom@liu.se – 2016

Vad ett program gör är viktigt

Hur det görs är lika viktigt!

Var börjar vi?



Uppdelning i klasser och metoder

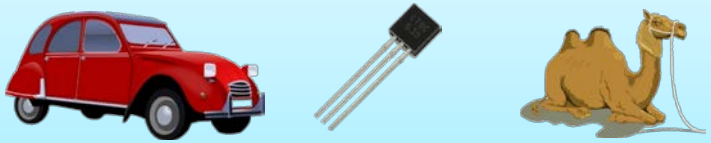
Hur vet vi
vilka klasser vill vi ha?

Funktionalitet

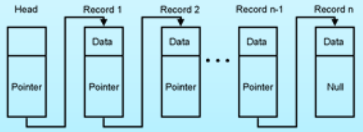
jonkv@ida

- Analys: Vilken **funktionalitet** behövs i programmet?

Objekt i verkligheten/domänen



Datastrukturer



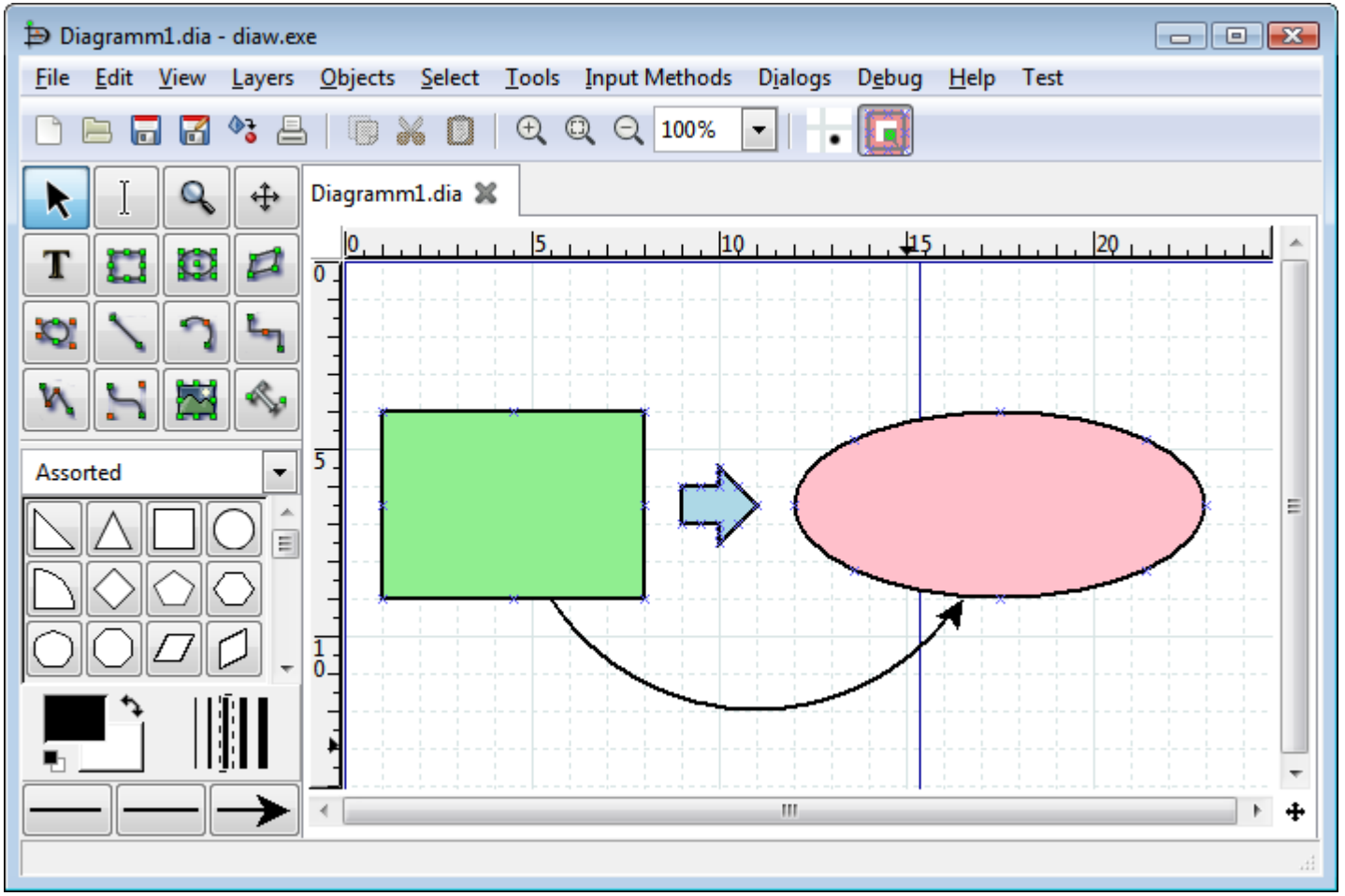
Vad är viktigt?

Klasser:
Car, Transistor, Camel
(Vehicle, Component, Animal)

Klasser:
List, Map, File, InputStream
RentalSchedule, …

Exempel (1)

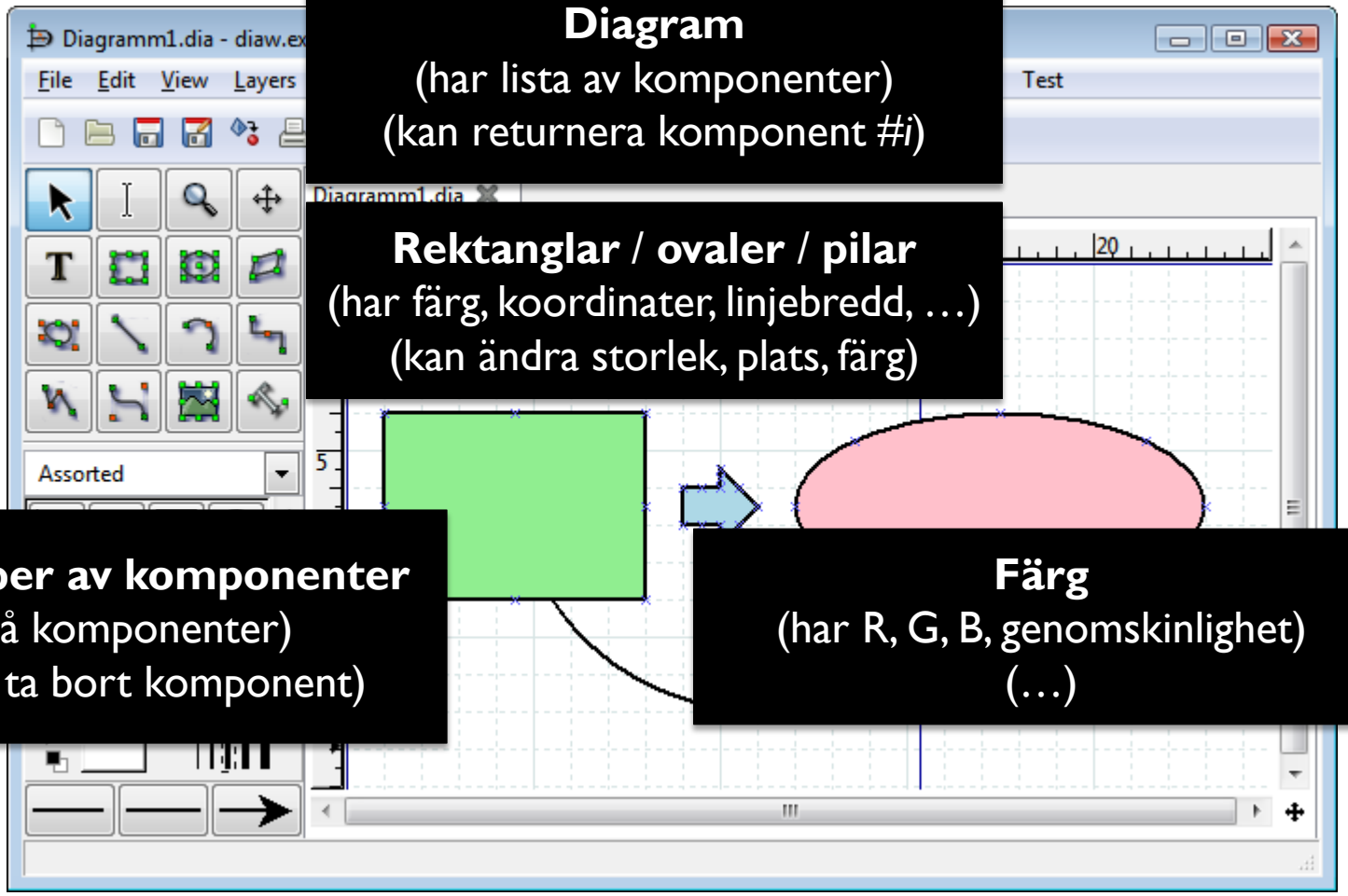
- Ett vektorbaserat **ritprogram**:



Exempel (2)

jonkv@ida

- I applikationsdomänen *vektordiagram*:

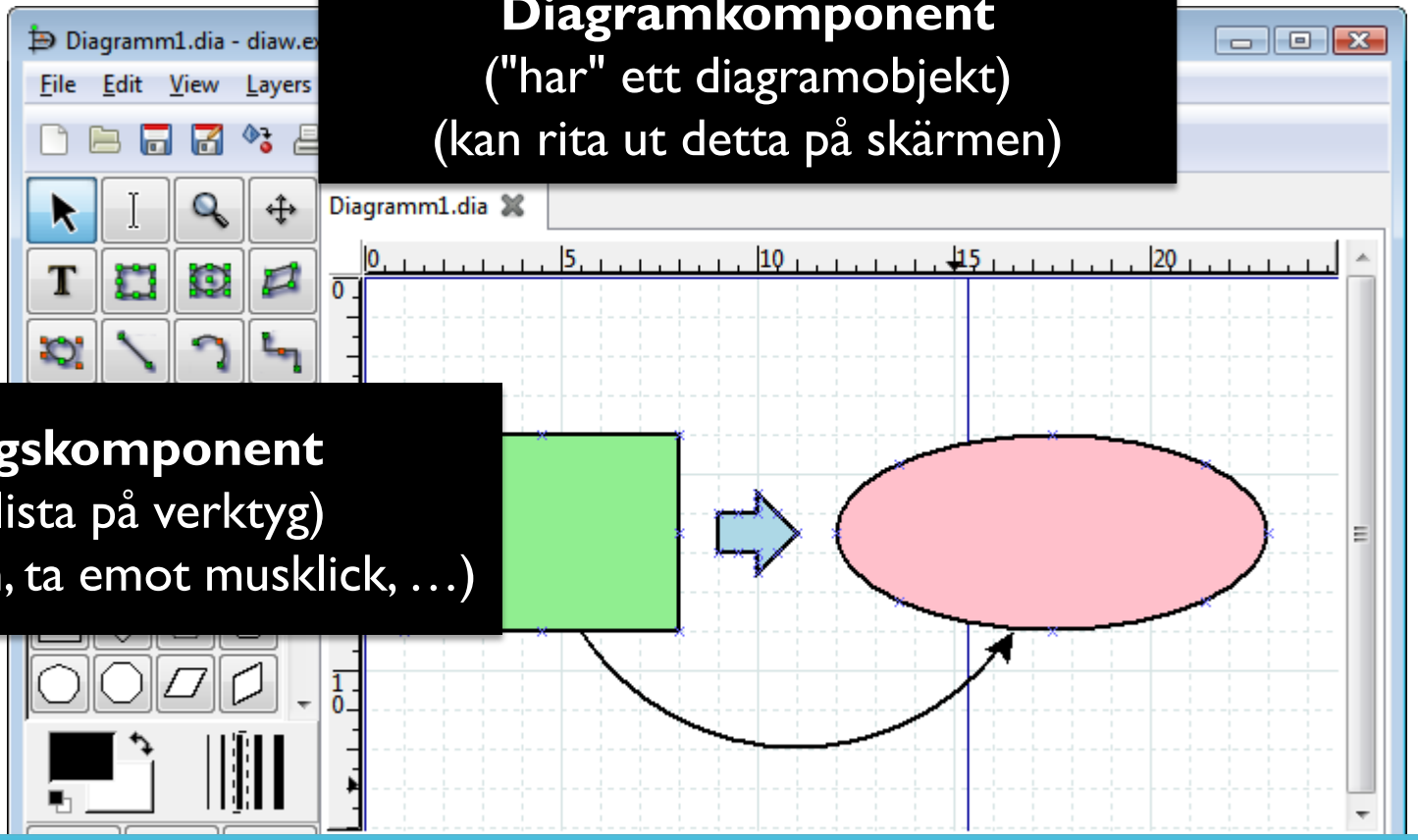


Exempel (3)

jonkv@ida

- Vi behöver ett **grafiskt gränssnitt**

Diagramkomponent
("har" ett diagramobjekt)
(kan rita ut detta på skärmen)



Verktygskomponent
(har en lista på verktyg)
(kan rita ut den, ta emot musklick, …)

…och så vidare

När gör vi uppdelningen?

När görs designen?

jonkv@ida

- **När** kan vi "välja" klasser?
  - **Vattenfallsmodellen**?
    1. Kravspecifikation
    2. Design ➔ mjukvaruarkitektur
    3. Implementation
    4. Integration
    5. Testning, felsökning
    6. Installation
    7. Underhåll

1. Tänk ut exakt vad som behövs, i detalj
2. Programmera exakt dessa klasser

  - **"Kontinuerlig design"?**
    - Liknande begrepp finns i:
      - Extreme programming
      - Agile software development
      - …

Tänk lagom mycket!

1. Fundera på övergripande design
2. Fundera mer detaljerat på en delmängd
3. Börja programmera
4. Vid behov: Tänk om, *refactoring*!

Gör så gott du kan från början – men var inte rädd att ändra dig!

Inledande designarbete

jonkv@ida

- Ett av många sätt att tänka:

Hur vet vi
om vi **valt rätt uppdelning**?

Några principer kan ge ledning...

Bra: Hög cohesion = sammanhållning

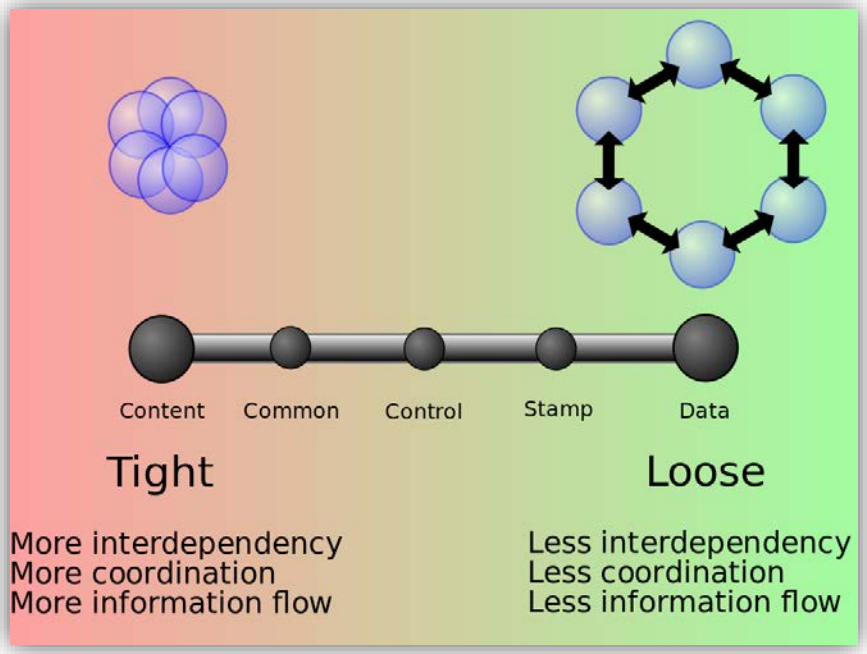
jonkv@ida

- Klassens funktionalitet ska **vara meningsfull** och **hänga ihop**
  - Bör gå att ge en **kort sammanfattning** av klassen
    - "En **List** är en sekvens av godtyckliga element"
    - "En **LayoutEngine** kan göra en specifik typ av layout för ett **Diagram**"
  - "En **MiscWorker** genererar slumptal, sparar filer och visar hjälptexter"?
    - ➔ Något är troligen fel…

Men: Undvik splittring, minska koppling

jonkv@ida

- Men **gå inte för långt**!
  - Varje extra klass ger **ytterligare ett begrepp** att hålla reda på
    - ListStructure, ListPrinter, ListSorter, ListRandomizer, … ➔ för mycket!
    - Listfunktionerna är troligen **tillräckligt** sammanhängande för att vara **en** klass
  - Varje extra **koppling** (**coupling**) till en annan klass kan ge problem
    - Täta beroenden mellan klasser ➔ varje ändring påverkar många klasser – dåligt



Låg coupling = få kopplingar mellan klasser

jonkv@ida

- Låg **coupling** = färre kopplingar **mellan** klasser
  - Funktionalitet som **hör ihop** bör vara **samlad**
    - (Bra om det räcker att läsa **en** klass för att förstå hur listor fungerar)

- Som alltid måste man hitta en **balans**: Ansvarsområden ska vara
  - **Sammanhängande**
  - Lätta att **beskriva**
  - **Lagom** stora

Single Responsibility Principle

jonkv@ida

Hör ihop med:

Single Responsibility Principle

1. En klass bör ha **ett enda ansvarsområde**
   - "Göra **en** sak" – inte generera slumptal + spara filer
2. Ansvarsområdet bör hanteras **helt inom klassen**
   - Inte vara utspritt

Ökar cohesion

Minskar coupling

```
class GameComponent extends JComponent {
   public void paintComponent(...) {
      player.drawImage(...);
      if (player.getY()+50 < ball.getY()+32) { ... }
      ...
   }
}
```

50 råkar vara höjden på spelarfiguren.
Varför ska GUI-komponenten veta detta,
medan spelarobjektet innehåller bilden + koordinater?

SINGLE RESPONSIBILITY PRINCIPLE

Just Because You Can, Doesn't Mean You Should

SRP och sammanhållning för metoder

jonkv@ida

- **SRP** + önskan om **hög sammanhållning** gäller även metoder!
  - Varje metod ska ha ett **tydligt ansvar**, "göra **en** sak"

```
class MyGameComponent {
   void tick() { // Called 50 times per second
      ... A lot of code updating object positions ...
      ... A lot of code checking for collisions ...
      ... A lot of code updating scores ...
   }
}
```

3 separata uppgifter,
svårt att få översikt

```
class MyGameComponent {
   void tick() { // Called 50 times per second
      updatePositions();
      checkCollisions();
      updateScores();
   }
   void updatePositions() {  ... }
}
```

Lätt att se
vad som händer under tick()

En metod som gör **en** sak

Organisation via namnrymder
(Paket i Java)

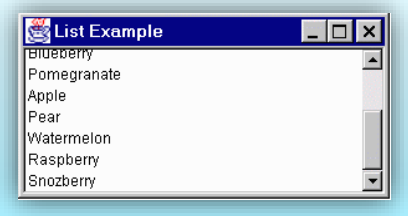
Namnrymder

- **Enkla klassnamn** kan ge **namnkollisioner**
  - Datastrukturen List och GUI-elementet List?
    - Kan döpa om to ListStructure, ListComponent...
    - ➔ Måste alltid använda långa namn, även om vi inte jobbar med GUI
  - Din liststruktur och någon annans?
    - Döp om till JonasListStructure, …?

Många språk har distinkta namnrymder

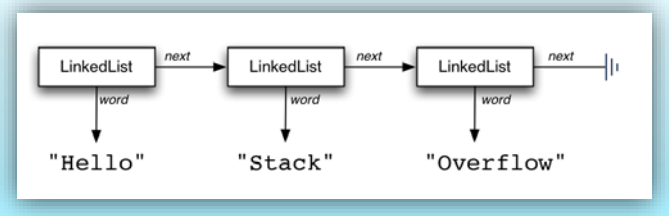
"GUI-namnrymd"

List ➔



"Datatyps-namnrymd"

List ➔



Namnrymder ger oss enklare namngivning, men också organisation

Repetition: Namnrymder i Python

jonkv@ida

- Från Python:

```
>>> import newton
This program tries to find the square root of a number.
Enter a number: 2
1.5
1.4166666666666665
1.4142156862745097
1.4142135623746899
1.414213562373095
>>> newton.find_root()
This program tries to find the square root of a number.
Enter a number: 10
3.5
3.178571428571429
3.162319422150883
3.1622776604441363
3.162277660168379
```

Gör "newton" till en namnrymd (med innehållet från filen newton.py)

Fullständigt namn: namnrymd.funktion

Ger möjlighet till t.ex. "gui.List" och "datatypes.List"

Paket 1: Namnrymder

- I Java ingår **varje klass** i ett **paket** (som är en namnrymd)
  - ```
    package se.liu.ida.jonkv.graphics.shapes;
    class Circle {
        ...
    }
    ```
  - ```
    se.liu.ida.jonkv.graphics.shapes.Circle c1;
    ```

Standard för **unika namn**:
Börja med domännamn, baklänges

Python	Java
Om vi gör "import" skapar vi en namnrymd, där filens innehåll placeras	**Klassen deklarerar vilken namnrymd den tillhör – detta gäller alltid!**
Kod i fil A hittar bara kod i fil B om den "importeras"	**Skriv paketnamn.klass så hittas klassen utan "import"**

Hur hittar Java klassen?

Paket 2: Katalogstruktur

jonkv@ida

- Javas källkod lagras i en **katalogstruktur** motsvarande paketen
  - rotkatalog ~/mysource
    - katalog se
      - katalog liu
        - katalog ida
          - katalog jonkv
            - katalog graphics
              - katalog shapes
                - fil Circle.java
                - fil GraphicCircle.java

Kan bli många nivåer

Utvecklingsmiljöer hjälper till att hålla reda på detta…

- Ange rotkatalogen ~/mysource – så hittas allt annat från den
  - På kommandoraden: Miljövariabeln **CLASSPATH**
  - I övrigt: Beror på utvecklingsmiljön

Paket 3: Existerande

- Många paket finns i Javas klassbibliotek

java.lang:
Object, Class, String, Thread, …

java.util:
List, ArrayList, Date, Calendar, Timer, Formatter, …

java.io:
Reader, InputStream, …

Paket 4: Import förkortar namnen

- Men nu blir det väl lite jobbigt…
  - javax.swing.filechooser.FileFilter filter =
    new javax.swing.filechooser.FileFilter();

- Undvik med **Javas** variant av **import**
  - **package** se.liu.ida.jonkv.io;
    import javax.swing.filechooser.FileFilter;

```
class FileSaver {
    …
    FileFilter filter = new FileFilter();
    …
}
```

Talar bara om var FileFilter finns!

Om jag *säger* FileFilter
menar jag
javax.swing.filechooser.FileFilter

Ungefärlig motsvarighet i Python:

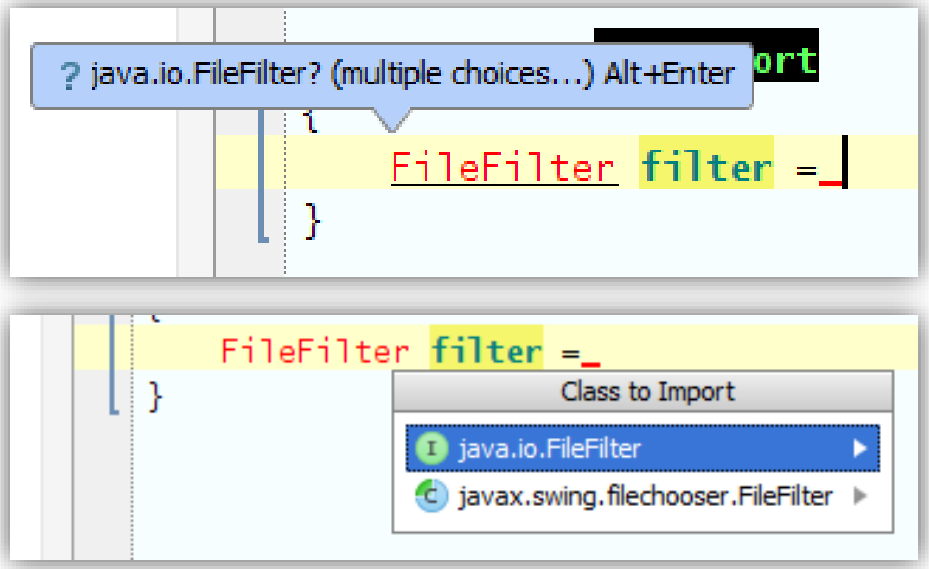
from javax.swing.filechooser
import FileFilter

Paket 5: Automatisk import

jonkv@ida

- Vissa klasser **importeras automatiskt**
  - Alla klasser i samma paket
  - Alla klasser i java.lang (”systemklasser” som Object, String)

- Moderna utvecklingsmiljöer hjälper till med annan import



Paket 6: Import med "*"

jonkv@ida

- Kan importera **alla klasser i ett paket** med "*"
  - `import javax.swing.filechooser.*;`
    `FileFilter filter = new FileFilter();`

Motsvarighet i Python:

```
from javax.swing.filechooser
import *
```

Undvik: Kan ge problem om nya klasser adderas till ett importerat paket…

Skriv i Java 1.0…

```
import java.awt.*;// Har en List-klass
import java.util.*; // Ingen List-klass…
class MyClass {
   List list; // Unikt!
}
```

Kompilera med Java 7…

```
import java.awt.*;// Har en List-klass
import java.util.*; // Har också en…
class MyClass {
   List list; // ???  Kompileringsfel!
}
```

Paket 7: Använd dem!

jonkv@ida

- Om inget paket anges:
  - Klassen hamnar i "unnamed package"
  - **Använd inte detta** – ange alltid paketnamn!

- För att:
  - Undvika namnkollisioner
  - Förbättra läsbarhet, underhåll, organisation:
    Vilka klasser är **relaterade**?
  - Undvika problem med verktyg som antar att paketnamn finns

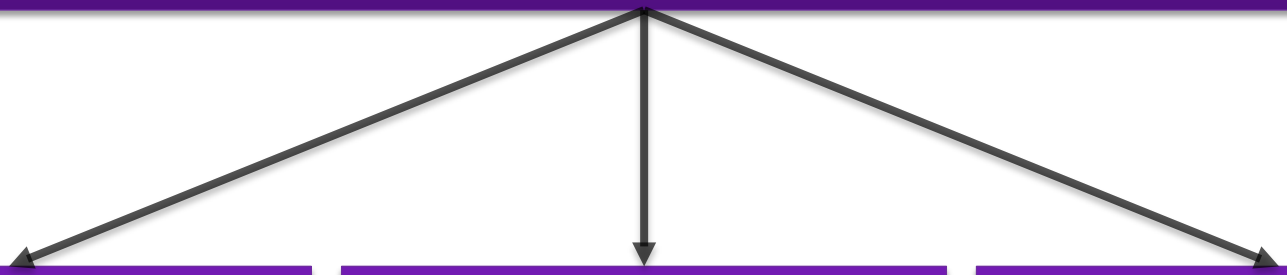
De flesta av projekten i kursen bör ha flera "delpaket"

Namngivningsprinciper

Namngivning 1: Varför är det viktigt?

jonkv@ida

Att använda namn, och ge beskrivande namn,
***ersätter* till viss del**
kommentarer och dokumentation!



Lättare och snabbare att skriva

Lättare och snabbare att läsa

Lättare att hålla uppdaterat – man ser om namnet är "ur synk"

Namngivning 2: Namngivna konstanter

jonkv@ida

Använd namn!

```
if (state == 14)
```

```
if (state == State.RUNNING)
```

```
paint(xpos + 42, ...)
```

```
paint(xpos + block.getSize(), ...)
```

```
paint(xpos + 41, ...)
```

```
paint(xpos + block.getSize() - 1, ...)
```

Förbättrar läsbarhet, förenklar ändringar

IDEA kan ibland hjälpa till: Refactor | Extract | Constant

Namngivning 3: Namngivna konstanter

jonkv@ida

Använd namn!

```
if (state == 14) {
    jump();
}
```

```
if (state == State.RUNNING) {
    jump();
}
```

Var är det enklast att se felet?

Namngivning 4: (Namngivna) metoder

jonkv@ida

Använd namn!

```
class MyGameComponent {
    void tick() {
        // Updating object pos
        …
        // Checking for collisions
        …
        // Updating scores
        …
    }
}
```

```
class MyGameComponent {
    void tick() {
        updatePositions();
        checkCollisions();
        updateScores();
    }
    void updatePositions() {
        …
    }
}
```

Förbättrar läsbarhet om segmenten är ”komplexa/långa” (balans!)

IDEA kan ibland hjälpa till: Refactor | Extract | Method

Namngivning 5: Komplettering

jonkv@ida

Använd namn!

Magiska konstanter luktar illa:
Påverkar läsbarhet, försvårar underhåll → *komplettering*

```
if (jumpCounter < 50) ...
if (wobbleValue + xCoord < 500) ...
```

```
if (changePicture == 0) {
      if (i == 4) {
            i = 0;
      }
      changePicture = 3;
      this.setImage(imageList.get(i));
      i++;
}
```

Namngivning 6: Information

jonkv@ida

- Använd **informativa** namn!
  - En metod som flyttar alla fiender ett steg framåt:
    - update(); // Moves all enemies
      ➔ moveEnemies();
  - En lista som innehåller alla uppkopplingar i en FTP-klient:
    - c // Contains all current connections
      ➔ connections, currentConnections
  - En boolean-variabel som är sann om spelet är igång:
    - if (startGame)
      ➔ if (gameRunning)

Namngivning 7: Namngivningsstandard

jonkv@ida

- (Och följ Javas namngivningsstandard…)
  - **Paket**: se.liu.ida.jonkv
  - **Klasser**: Raster, ImageSprite
  - **Gränssnitt**: RasterDelegate, Comparator
  - **Metoder**: run(), runFast(), getBackground()
  - **Variabler**: i, myWidth
  - **Konstanter**: static final int MIN_WIDTH, MAX_WIDTH

Modellering och ärvning

Liskov Substitution Principle

Är en kvadrat en sorts rektangel ?

LSP – återkomsten

jonkv@ida

- Liskov Substitution Principle:
  - *Let q(x) be a property provable about objects x of type T. Then q(y) should be provable for objects y of type S, where S is a subtype of T.*

- Förenklat (och till viss del försvagat):
  - En subtyp måste uppfylla supertypens kontrakt

LSP mellan oföränderliga objekt

jonkv@ida

- Anta dessa kontrakt:

Square
Har 4 sidor
Alla 4 sidor är garanterat lika långa
Alla vinklar är 90 grader
Immutable (oföränderlig)

Rectangle
Har 4 sidor
Motstående sidor är lika långa
Alla vinklar är 90 grader
Immutable (oföränderlig)

En kvadrat är en sorts rektangel

Alla krav från rektangelklassen är uppfyllda
Allt vi kan bevisa om rektanglar kan vi också bevisa om kvadrater

OK att mer specifik *immutable* klass ärver från mindre specifik klass

LSP mellan föränderliga objekt

jonkv@ida

- Anta istället:

Rectangle
Har 4 sidor
Motstående sidor är lika långa
Alla vinklar är 90 grader
Har setWidth(), setHeight()

Square extends Rectangle:

Måste också ha setWidth(), setHeight()

Ärv set*() från Rectangle?

```
square.setHeight(200);
square.setWidth(100);
square.getHeight() != square.getWidth()
```

Bryter mot Square:s kontrakt!

Låt set*() ändra höjd *och* bredd?

```
square.setHeight(200);
square.setWidth(100);
square.getHeight() ➔ 100
```

Bryter mot Rectangle:s kontrakt!

Inte OK att mindre flexibel klass ärver från mer flexibel klass

Gränssnitt –
eller abstrakt klass?

Abstrakt klass eller gränssnitt? (1)

jonkv@ida

Gränssnitt

- Kan implementera flera

class LinkedList
implements List, Queue, …

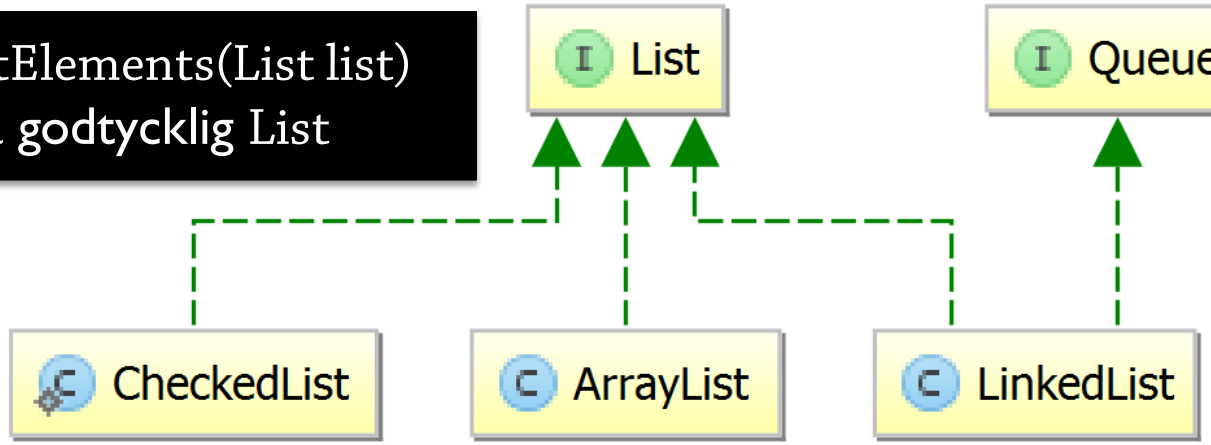
Kan vara både List och Queue

- Ingen kod, bara löften / kontrakt

interface List { … }

Alla som implementerar måste ha kod för alla metoder

void printElements(List list)
kan ta godtycklig List



Vem som helst kan implementera List:
Även om man också *implementerar* Queue, eller *utökar* en klass

Gränssnitt ger ingen gemensam kod att ärva

Abstrakt klass eller gränssnitt? (2)

jonkv@ida

Abstrakta klasser

- Ger löften *och* kod

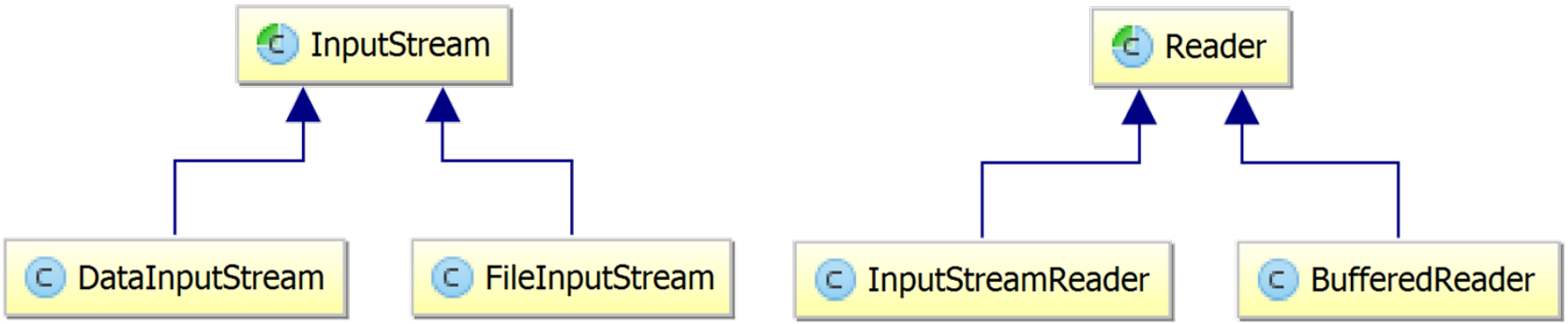
abstract class Reader { … }
abstract class InputStream { … }

Klasser tillhandahåller kod som alla subklasser använder

- Kan bara utöka *en*

class BufferedReader
extends Reader [och inget mer]

Ingen klass kan vara Reader *och* InputStream



Subklasser till InputStream och Reader
ärver användbar kod…

Abstrakt klass eller gränssnitt? (3)

jonkv@ida

- Javas I/O-klasser har suboptimal design…
  - "Basen" i dessa hierarkier är abstrakta klasser, inte gränssnitt
  - ➔ I/O-kod deklarerar parametrar, fält, variabler av *klasstyperna* InputStream och Reader!

```
List readElementsFrom(InputStream stream) {
    …
}
```

För att skapa en ny strömtyp som metoden kan använda, *måste* man utöka InputStream

Då kan klassen inte utöka något annat…

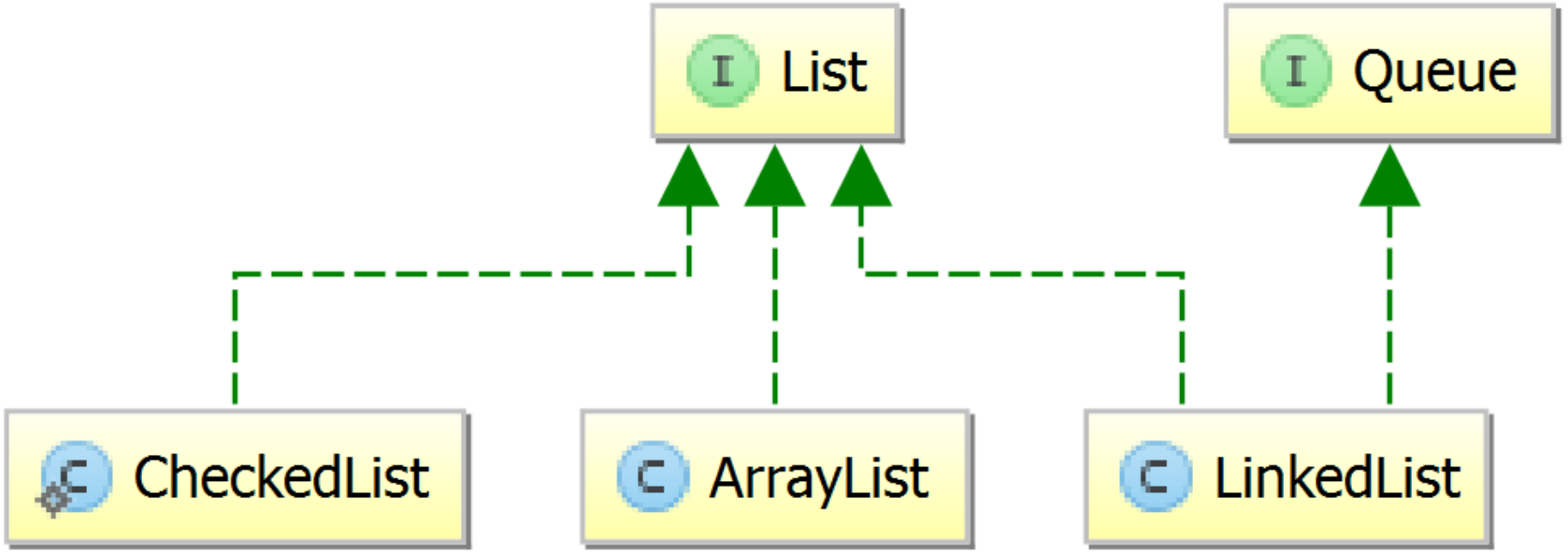
Hade List och Queue varit abstrakta klasser, kunde LinkedList inte ha varit både List and Queue!

Abstrakt klass eller gränssnitt? (4)

jonkv@ida

- Bra teknik: Använd båda!

Hierarkiernas "baser" *är* gränssnitt
(List, Queue)
➔ används för deklarationer etc.



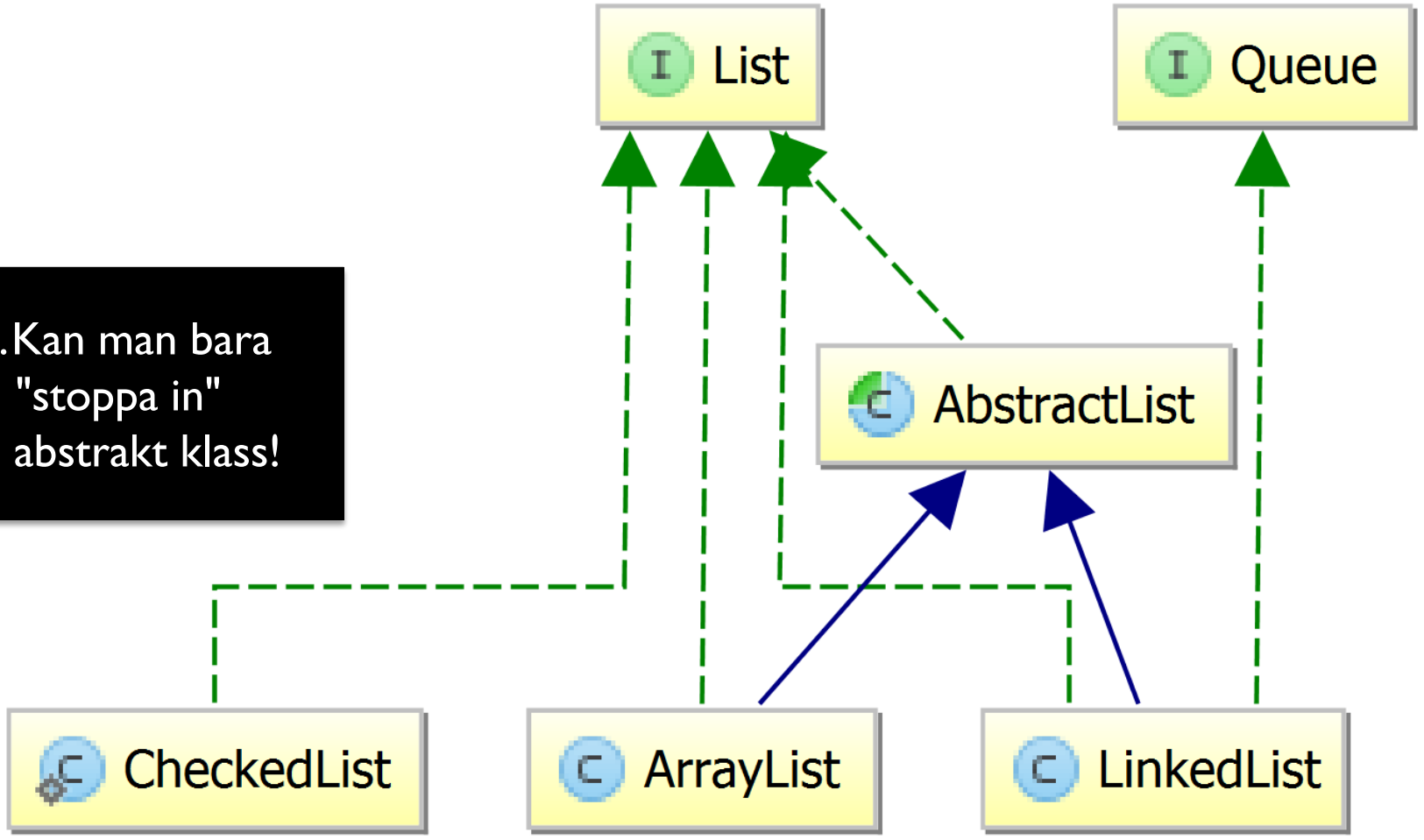
Man *kan* implementera dessa direkt
➔ full frihet

Om man har mycket gemensam kod,
och kan tjäna på att ha
en abstrakt klass…

Abstrakt klass eller gränssnitt? (5)

jonkv@ida

…Kan man bara
"stoppa in"
en abstrakt klass!



Metoder kräver/returnerar fortfarande **List**, inte **AbstractList**

➔ kan ta/returnera en CheckedList skriven *utan* den abstrakta klassen

Kodlukt

Kodlukt

jonkv@ida

- **Kodlukt / code smell**:
  - Inte nödvändigtvis en bugg i sig
  - Något man kan se "**på ytan**" på koden,
    som ofta indikerar att det finns ett **djupare problem**
  - Fixas ofta genom **refactoring**: Kod skrivs om, men gör exakt samma sak

Redundans och upprepning

Redundans?

- I vissa sammanhang kan redundans ha sina poänger…
  - *This parrot is no more. It has ceased to be.*
    It's expired and gone to meet its maker.
    This is a late parrot. It's a stiff. Bereft of life, it rests in peace.
    If you hadn't nailed it to the perch, it would be pushing up the daisies.
    It's rung down the curtain and joined the choir invisible.
    This is an ex-parrot.

Undvik redundans

- I programmering ska redundans och upprepning undvikas!
  - Gör det **svårare att förstå** koden
    - Mer att läsa
    - Måste se om upprepningen är exakt lika eller skiljer sig på något sätt
  - Leder till **mer kod att underhålla**
  - Leder till **risk att inte all kod uppdateras** på samma sätt

DRY-principen: Don't Repeat Yourself!
DIE: Duplication Is Evil

WET: Write Everything Twice
WET: We Enjoy Typing…

Redundans på klassnivån (1)

jonkv@ida

```
class ArrayList implements List {
   Object[] elements = new Object[42];
   int length = 0;
   int size() { return length; }
   Object get(int index) { return elements[index]; }
   void add(Object element) { … }
   void set(int index, Object element) { …
   int indexOf(Object o) {
      for (int i = 0; i < length; i++)
         if (get(i).equals(o)) return i;
      return –1;
   }
}
```

Tidigare exempel:
För mycket kod upprepas!

```
class LinkedList implements List {
   Node head = null, tail = null;
   int length = 0;
   int size() { return length; }
   Object get(int index) { … }
   void add(Object element) { … }
   void set(int index, Object element) { … }
   int indexOf(Object o) {
      for (int i = 0; i < length; i++)
         if (get(i).equals(o)) return i;
      return –1;
   }
}
```

Inträffar ofta i projekt…

Redundans på klassnivån (2)

jonkv@ida

```
abstract class GenericList implements List {
   int length = 0;
   int size() { return length; }
   int indexOf(Object o) {
      for (int i = 0; i < length; i++)
         if (get(i).equals(o)) return i;
      return –1;
   }
}
```

Tidigare diskuterad lösning:
Flytta kod till gemensam superklass
(abstrakt eller konkret!)

```
class ArrayList extends GenericList {
   Object[] elements = new Object[42];
   Object get(final int index) { return elements[index]; }
   void add(final Object element) { … }
   void set(int index, Object element) { … }
}
```

```
class LinkedList extends GenericList {
   Node head = null, tail = null;
   Object get(final int index) { … }
   void add(final Object element) { … }
   void set(int index, Object element) { … }
}
```

Redundans på klassnivån (3)

jonkv@ida

```
abstract class GenericList implements List {
   int indexOf(Object o) {
      for (int i = 0; i < size(); i++)
         if (get(i).equals(o)) return i;
      return –1;
   }
}
```

Relaterat problem i projekt:
Ärvningen finns där,
men **medlemmar** repeteras ändå!

```
class ArrayList extends GenericList {
   int length = 0;
   int size() { return length; }
   Object[] elements = new Object[42];
   …
}
```

Flytta till superklassen!

```
class LinkedList extends GenericList {
   int length = 0;
   int size() { return length; }
   Node head = null, tail = null;
   …
}
```

Redundans på klassnivån (4)

jonkv@ida

```
abstract class GenericList implements List {
  int length = 0;
  int size() { return length; }
  int indexOf(Object o) {
    for (int i = 0; i < size(); i++)
      if (get(i).equals(o)) return i;
    return –1;
  }
}
```

Relaterat problem i projekt:
Ärvningen finns där,
men **fält** repeteras ändå!

```
class ArrayList extends GenericList {
  int length = 0;
  Object[] elements = new Object[42];
  …
}
```

Ger **skuggning** av fält:
Varje ArrayList har TVÅ fält med namnet length

Vilket som används beror på pekartypen

Slösar minne
Svårt att förstå¨!

```
class LinkedList extends GenericList {
  int length = 0;
  Node head = null, tail = null;
  …
}
```

Onödiga klasser

Onödiga klasser (1)

jonkv@ida

```
class GenericPlayer {
   int strength;
   Image img;
   GenericPlayer(int strength, Image img) {
      …
   }
}
```

Subklasser ska ha eget **beteende**:
Nya eller ändrade metoder!

Här ett extremt exempel:
Bara konstruktorn skiljer sig

```
class BasicPlayer extends GenericPlayer {
   BasicPlayer() {
      super(20, getImage("basicplayer.png"));
   }
}
```

Detta anger inte beteende,
bara *data* (fältvärden)

```
class SuperPlayer extends GenericPlayer {
   SuperPlayer() {
      super(50, getImage("superplayer.png"));
   }
}
```

Dålig lösning:
Onödiga klasser

Kan bara anropa
new GenericPlayer(…)
med parameter istället
➔ slipper två klasser

Onödiga klasser (1): Fabriksmetoder

jonkv@ida

- Om man ändå vill ge namn för att förenkla:
  - Använd fabriksmetoder

```
class PlayerFactory {
   Player createBasicPlayer() {
      return new GenericPlayer(20, getImage("basicplayer.png"));
   }
   Player createSuperPlayer() {
      return new GenericPlayer(50, getImage("superplayer.png"));
   }
}
```

  - Behöver inte alltid en egen "fabriksklass"
    - Anropas create* bara från GameMechanics?
    - Då kanske de ska ligga i GameMechanics istället

Onödiga klasser (2)


```
class GenericPlayer {
    int strength;
    Image img;
    GenericPlayer(int strength, Image img) {
        ...
    }
}
```

```
class BasicPlayer extends GenericPlayer {
    void leapForward() {
        x += 10;
    }
}
```

```
class SuperPlayer extends GenericPlayer {
    void leapForward() {
        x += 20;
    }
}
```

Här finns metoder med olika kod
Men den enda skillnaden är i data!
Ska **inte** ha olika klasser

Onödiga klasser (2): Lösning

jonkv@ida

```
class GenericPlayer {
    int strength;
    int leapLength;
    Image img;
    GenericPlayer(int strength, Image img, int leapLength) {
        ...
    }
}
```

Bra, generell lösning:

Kan lätt ange *godtycklig* hopplängd
Kan hitta på hopplängder *under programmets körning* (ingen ny klass krävs)

Använd inte klasser när värden räcker

Onödigt många metoder

Onödigt många metoder

jonkv@ida

- Även på andra sätt ska kod skrivas **effektivt** – möjliga exempel:

```
public void activateUp() {
    up = true;
}
public void deactivateUp() {
    up = false;
}
```

```
public void setUp(boolean up) {
   this.up = up;
}
```

```
public void moveLeft() {
    …;
}
public void moveRight() {
    …;
}
```

```
public void move(Direction dir) {
    …;
}
```

Onödig upprepning av kodmönster

Upprepning av kodmönster (1)

jonkv@ida

```
public void checkKeyboard() {
   if (isPressed(Key.UP)) {
      player.directions.add(Direction.NORTH);
   } else {
      player.directions.remove(Direction.NORTH);
   }
  if (isPressed(Key.DOWN)) {
     player.directions.add(Direction.SOUTH);
  } else {
     player.directions.remove(Direction.SOUTH);
  }
  if (isPressed(Key.LEFT)) {
     player.directions.add(Direction.WEST);
  …
```

Extrahera en hjälpmetod!

```
public void checkKeyboard() {
   checkKey(Key.UP, Direction.NORTH);
   checkKey(Key.DOWN,Direction.SOUTH);
   checkKey(Key.LEFT, Direction.WEST);
   checkKey(Key.RIGHT, Direction.EAST);
}
```

```
private void checkKey(Key key, Direction dir) {
    if (isPressed(key)) {
      player.directions.add(dir);
   } else {
      player.directions.remove(dir);
} }
```

Upprepning av kodmönster (2)

jonkv@ida

- Extrahera **upprepade uttryck** till namngivna variabler

NameProcessor.java

```
static void printNames(final String fullName) {
        System.out.println(fullName.substring(fullName.indexOf(" ") + 1));
        System.out.println(fullName.substring(0, fullName.indexOf(" ")));
}
```

Multiple occurrences found

Replace this occurrence only

Replace all 2 occurrences

NameProcessor.java

```
static void printNames(final String fullName) {
    int firstNameEndIndex = fullName.indexOf(" ");
    System.out.println(fullName.substring(firstNameEndIndex + 1));
        System.out.println(fullName.substring(0, firstNameEndIndex));
}
```

Snabbare kod

Namnet ger en *förklaring* till deluttrycket

Lättare att se att delarna är *medvetet* identiska

IDEA kan hjälpa till: Refactor | Extract | Variable

Upprepning i konstruktorer

jonkv@ida

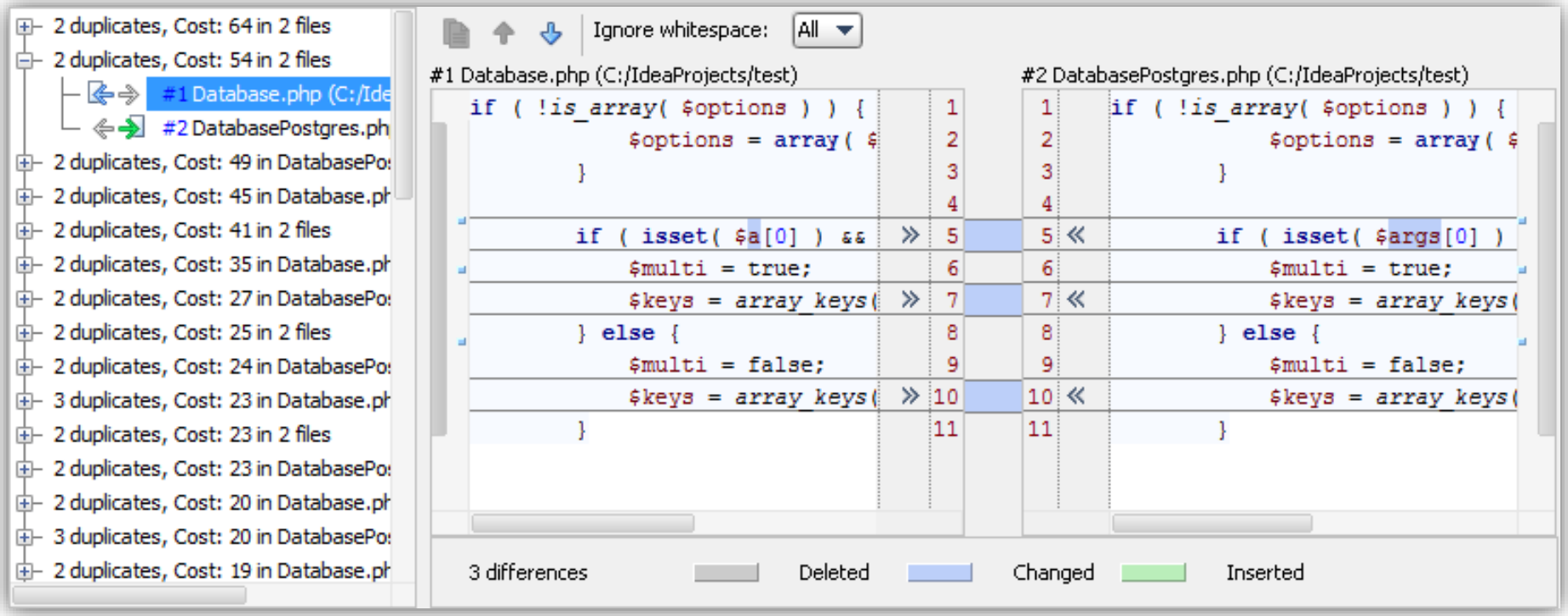
- Många konstruktorer ➔ upprepad kod?
  - En konstruktor kan anropa en annan – en "kedja"

```
class Circle {
  double x, y, r;
  Circle(double x, double y, double r) {
    this.x = x; this.y = y; this.r = r;
    // Lots of common initialization code
  }
  Circle(double r) {
    this(0.0,  0.0, r);
  }
}
```

Specialsyntax:
this(args)
som *första* sats i en konstruktor
anropar en annan konstruktor

Att hitta upprepad kod

- IDEA kan hitta vissa former av **uppenbar repetition**
  - Analyze | Locate Duplicates



Ni gör resten – alltför ineffektiv kod ger komplettering!

Redundant lagring

Redundans i lagring (fält)

jonkv@ida

```
class GenericPlayer {
   // Can't move left and right at the same time,
   // so some values are forbidden
   // ➔ we could get an inconsistency
   // if we update incorrectly
   boolean movingLeft;
   boolean movingRight;
}
```

```
enum HorizontalDir {
   LEFT, STILL, RIGHT;
}
class GenericPlayer {
   HorizontalDir horizMove;
}
```

Alla värden rimliga

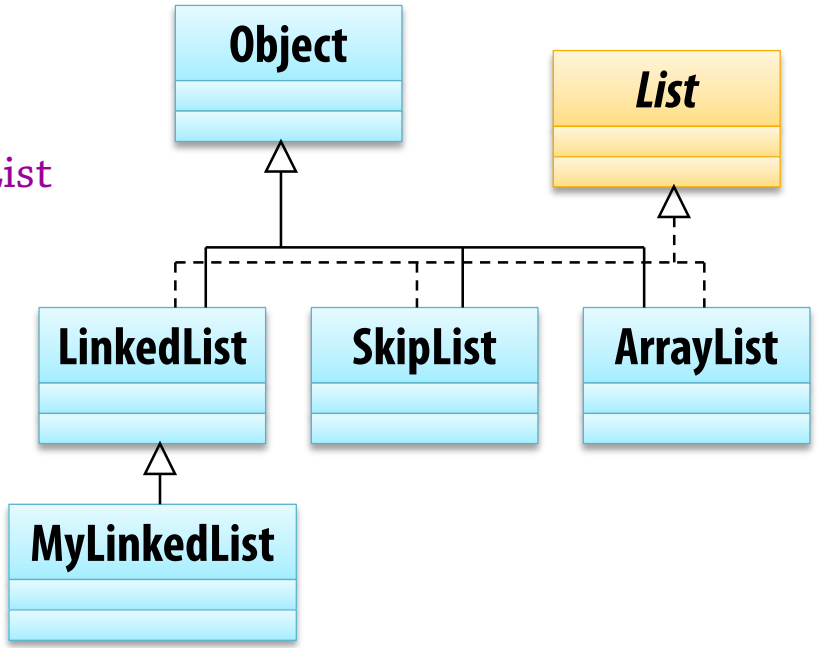
➔ vi har gjort en feltyp omöjlig

Egen typkontroll i Java
(RTTI, Run-Time Type Identification)

Typkontroll: Operatorn instanceof

jonkv@ida

- Vilken typ har ett visst objekt?
  - Exakt typ: object.getClass() == TypeName.**class**
    - **if** (obj.getClass() == LinkedList.**class**) {
      // **Exakt** typkontroll:
      // obj pekar på ett objekt som skapats med "**new** LinkedList()"
      }
  - Subtyp: object **instanceof** TypeName
    - **if** (obj **instanceof** LinkedList) {
      // **Subtypskontroll**:
      // obj pekar på ett objekt av typ LinkedList
      // eller en subklass som MyLinkedList
      }
    - *Allt* är **instanceof** Object!



Typkontroll: Operatorn instanceof (2)

jonkv@ida

- **Kan** användas för att göra olika saker beroende på typ…

```
void checkCollisions() {
    for (Thing t : thingsOnScreen) {
        if (player.intersects(t)) {
            if (t instanceof Wall) {
                // Få spelaren att studsa tillbaka
            } else if (t instanceof Bullet) {
                // Få spelaren att förlora hälsa
            } else if (t instanceof Powerup) {
                player.addPU((Powerup)t);
            } else if …
        }
```

Göra olika beroende på typ…?

Det är ju detta subtypspolymorfism är till för!

UNDVIK! Polymorfism är oftast bättre!

Från *Effective C++*, av Scott Meyers :

"Anytime you find yourself writing code of the form "if the object is of type T1, then do something, but if it's of type T2, then do something else," slap yourself.

Varför är polymorfism (oftast) bättre?

jonkv@ida

```
void checkCollisions() {
   for (Thing t : thingsOnScreen) {
      if (player.intersects(t)) {
         if (t instanceof Wall) {
            // Make player bounce back
         } else if (t instanceof Bullet) {
            // Make player lose health
         } else if (t instanceof Powerup) {
            player.addPU((Powerup)t);
         } else if ...
      }
   }
}
```

Centraliserat: Lägger man till en ny sorts Thing, måste denna metod *också* ändras!

Bräckligt, inte modulärt!

***Single Responsibility Principle*:**
Kollisionshanteraren borde inte bestämma *hur* alla saker på skärmen interagerar med spelaren!

Varför är polymorfism (oftast) bättre?


```
void checkCollisions() {
   for (Thing t : thingsOnScreen) {
      if (player.intersects(t)) {
         t.handleCollisionWith(player);
      }
   }
}
```

1. Be saken hantera detta

4. Polymorfism och dynamisk bindning ➔ rätt implementation väljs!

```
interface Thing {
   void handleCollisionWith(Player p);
}

class Bullet implements Thing {
   void handleCollisionWith(Player p) {
      // Make player lose health
   }
}

class Powerup implements Thing {
   void handleCollisionWith(Player p) {
      p.addPowerUp(this);
} }
```

2. Thing lovar att alla Thing-klasser kan hantera kollision

3. Alla Thing-klasser har sin egen implementation, bestämmer sitt beteende