Inför projektet:

Vanliga problem att tänka på

Kodlukt

jonkv@ida

- **Hur märker man att det finns problem i koden?**
  - Många olika sätt…

- En sorts indikation: **Kodlukt / code smell**
  - Något man kan se "**på ytan**" på koden,
    som indikerar att det ***kan finnas*** ett problem
  - Lukten är inte problemet, utan en varningssignal
    (som inte *alltid* stämmer)
  - Handlar ofta om kodens **struktur**:
    Den kanske *gör vad den ska*, men på fel sätt
  - Fixas ofta genom **refactoring**:
    Kod skrivs om, men har fortfarande samma resultat

Denna föreläsning:
Fokus på problem – ibland kodlukt

Undvik upprepning, redundans
Skriv generell kod
som kan återanvändas

Duplicated code

Redundans?

jonkv@ida

- I vissa sammanhang kan redundans ha sina poänger…
  - https://www.dailymotion.com/video/x2hwqnp
  - *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
  - **Mer kod att underhålla**
  - **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…

Onödigt många metoder?
Använd metodparametrar

Onödigt många metoder

jonkv@ida

- Kod ska skrivas **effektivt** – 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) {
    …;
}
```

Onödigt många metoder (2)

jonkv@ida

- Metodparameter är ofta **effektivare**: Vi gör om **kod till data**

```
if (shouldBeUp()) {
   foo.activateUp();
} else {
   foo.deactivateUp();
}
```

```
foo.setUp(shouldBeUp());
```

Mycket kortare!

```
foo.activateUp();
```

```
foo.setUp(true);
```

Ingen försämring!

Upprepa inte samma mönster:
Kodrader, uttryck, ...

```
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);
} }
```

Användbart även om det bara är några rader…

```
public void createGUI() {
   Font font =
      new Font("Times", Font.PLAIN, 10);

   JTextField nameField = new JTextField();
   nameField.setColumns(30);
   nameField.setFont(font);

   JTextField addressField = new JTextField();
   addressField.setColumns(40);
   addressField.setFont(font);

   JTextField cityField = new JTextField();
   cityField.setColumns(30);
   cityField.setFont(font);
}
```

```
public void createGUI() {
   Font font =
      new Font("Times", Font.PLAIN, 10);

   JTextField nameField = createTextField(font, 30);
   JTextField addressField = createTextField(font, 40);
   JTextField cityField = createTextField(font, 30);
}

private JTextField createTextField
      (final Font font, final int columns) {
   JTextField field = new JTextField();
   field.setColumns(columns);
   field.setFont(font);
   return field;
}
```

…eftersom vi undviker repetition
och skapar nya meningsfulla begrepp
som ger en bra översikt över metoden

…och enkelt!

```
 8      public void createGUI() {
 9          Font font = new Font( name: "Times", Font.PLAIN, size: 10);
10
11          JTextField nameField = new JTextField();
12          nameField.setColumns(30);
13          nameField.setFont(for
14
15          JTextField addressFie
16          addressField.setColu
17          addressField.setFont(
18
19          JTextField cityField
20          cityField.setColumns(
21          cityField.setFont(for
22      }
23  }
24
```

Upprepning av uttryck

- Extrahera **upprepade uttryck** till namngivna variabler

```
public void method() {
    int a = 1;
...
    int b = a + anotherClass.intValue();
    int c = b + anotherClass.intValue();
}
```

```
public void method() {
    int a = 1;
...
    int number = anotherClass.intValue();
    int b = a + number;
    int c = b + number;
}
```

(Snabbare kod)

Namnet kan ge en *förklaring* till deluttrycket

Lättare att se att delarna är *medvetet* identiska

IDEA kan hjälpa till: Refactor | Extract | Variable

Upprepning i konstruktorer

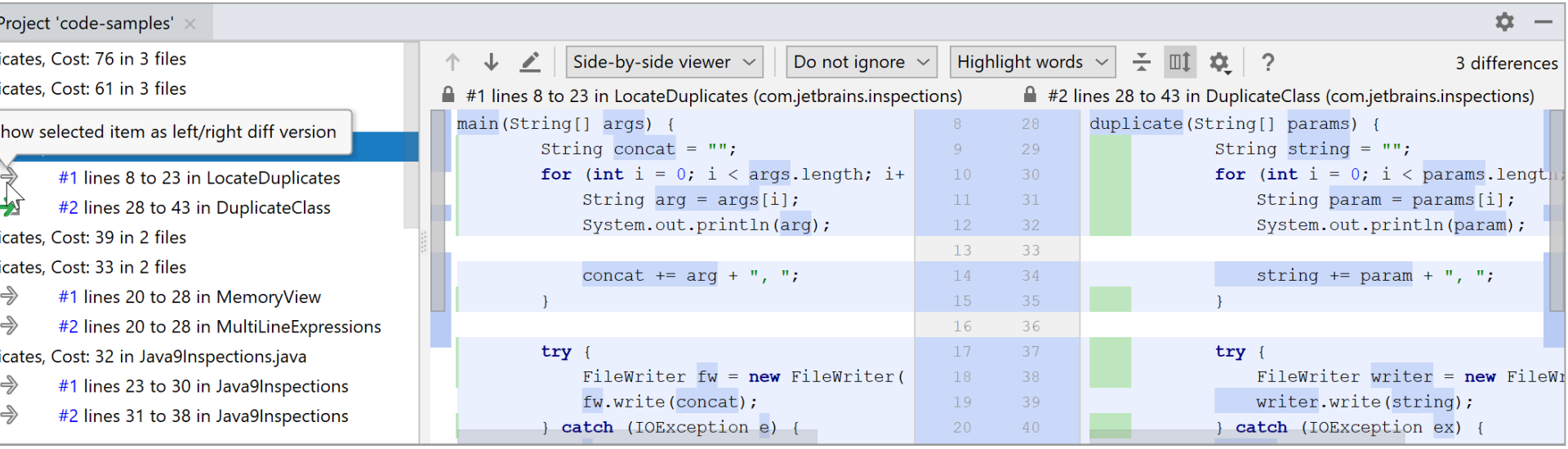
- Många konstruktorer ➔ upprepad kod?
  - En konstruktor kan anropa en annan – en "kedja"

```
class Circle {
  double x, y, r;
  Circle(double x, double y, double r) {
    this.x = x; this.y = y; this.r = r;
    // Testa att r >= 0
    // Beräkna arean
    // Mer initialiseringskod...
  }
  Circle(Circle other) {
    this(other.x, other.y, other.r);
  }
}
```

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

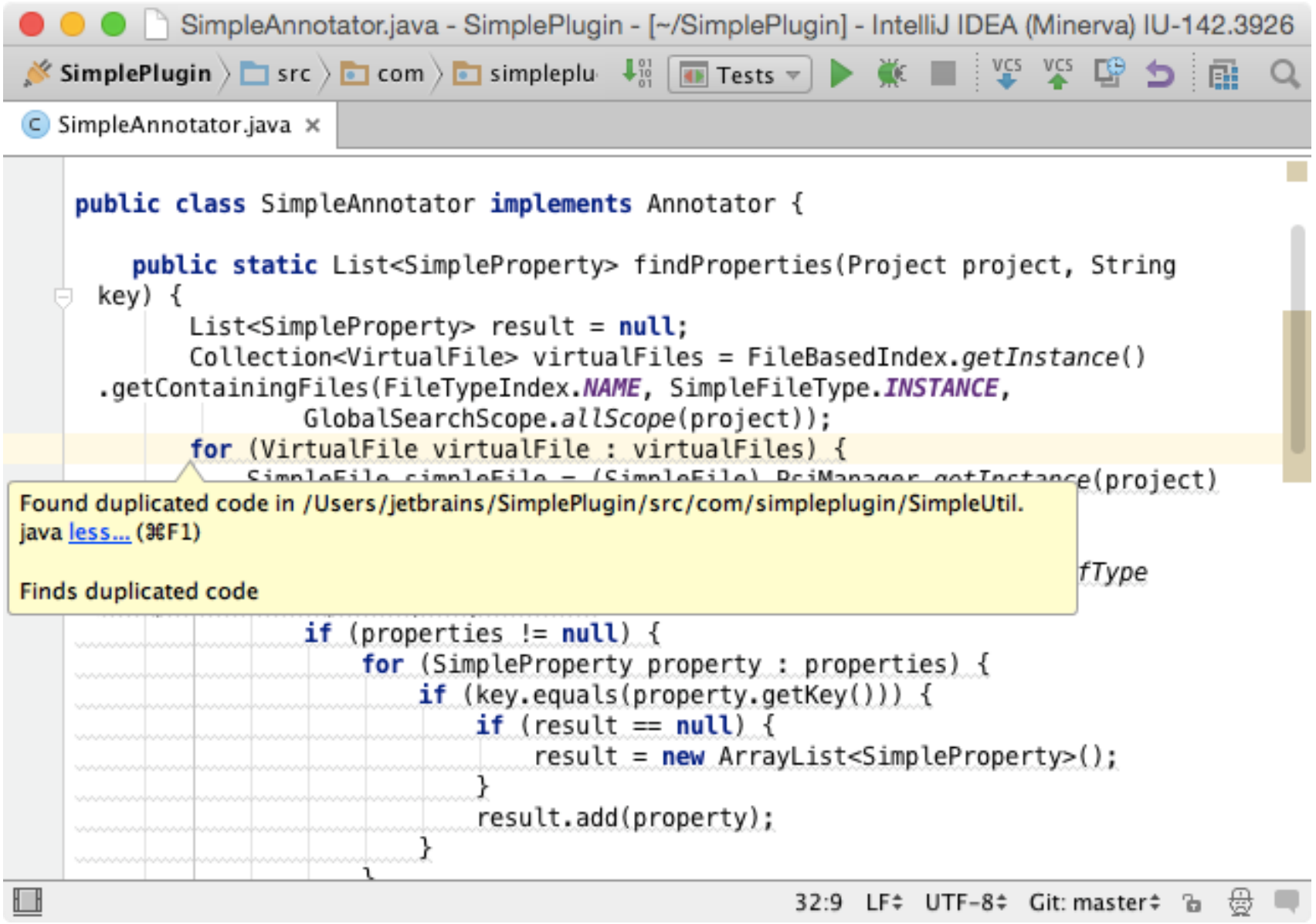
Att hitta upprepad kod

- IDEA kan hitta vissa former av **uppenbar repetition**
  - Inspektion: *Duplicated Code Fragment*
  - https://www.jetbrains.com/help/idea/analyzing-duplicates.html



Kodinspektionen hjälper också – ni gör resten
Alltför redundant kod ger komplettering!

Kan även köras ”on the fly”



Listor eller enskilda variabler?
Loopar eller enskilda satser?

Listor 1

jonkv@ida

- För att modellera en **fast uppsättning objekt**:
  - Kan vara frestande att använda **flera variabler**

```
public class Game {
   // Vi har två spelare, så vi använder två separata fält...
   private Player player1;
   private Player player2;

   // ...
   private Level level1;
   private Level level2;
   private Level level3;
}
```

***Fungerar* säkert...**

Men det är omöjligt att prata om
”alla spelare”,
”alla nivåer”,
”nivåerna mellan 2 och 7”,
...

Listor 2

jonkv@ida

- Eller kod som denna:

```
public class Game {
   private Level level1;
   private Level level2;
   private Level level3;

   private void setLevel(int level) {
      if (level == 1)
         this.level = level1;
      else if (level == 2)
         this.level = level2;
      else if (level == 3)
         this.level = level3;
   }
}
```

Onödigt mycket kod

Måste uppdatera koden när man skapar nya nivåer, inte bara nivålistan

Lätt att göra fel någonstans

Listor 3

jonkv@ida

- Kan lätt leda till kod som denna:

```
public class Game {
   private Player player1;
   private Player player2;

   private void playerTakesPowerup(int playerID) {
      if (playerID == 1)
         player1.addPowerup();
      else if (playerID == 2)
         player2.addPowerup();
   }
}
```

Ser koden ut så här
går det alltid att göra bättre…

Listor 4

jonkv@ida

- För att **begreppet** ”spelarna”
  ska kunna **materialiseras** i ett **objekt** ”spelarna”:

```
public class Game {
   private List<Player> players;

    private void tick() {
      for (Player player : players) { … }
   }

   public Player getPlayer(int index) {
      return players.get(index);
   }
}
```

Kan prata om ”spelarna”,
kan loopa över ”spelarna”

Om du tänker att något ska göras ”för alla”,
ska du väldigt sällan göra detta separat ”för A” och sedan ”för B”,
även om det bara finns A och B (2 objekt).
Låt tanken ”för alla” synas i koden!

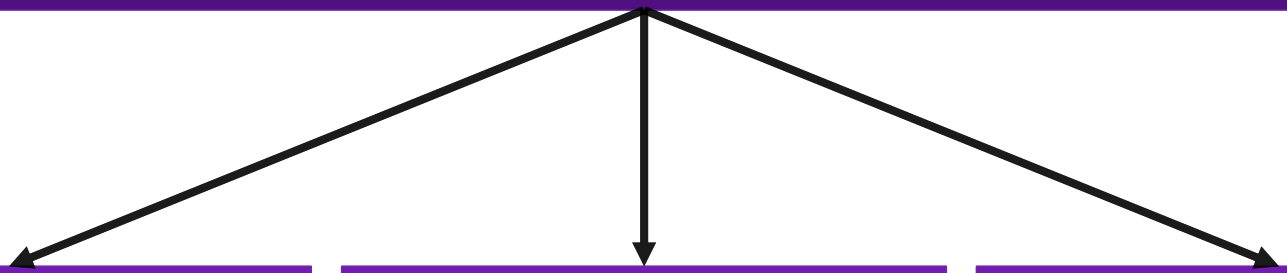
Låt koden direkt reflektera / visa upp underliggande idéer / begrepp

Mysterious name
(…and *lack of names*)

Namngivning: Varför 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"

Begrepp 1

- Commodore 64 BASIC V2:

”Anropa en funktion”:
GOSUB 1000
Tänker vi så?

```
10 PRINT CHR$(147)
20 SP = 20: ZE = 3: A$ = "Good Morning!": GOSUB 1000: GOSUB 2000
30 SP = 10: ZE = 3: A$ = "I'm the Commodore 64": GOSUB 1000: GOSUB 2000
40 SP = 12: ZE = 6: A$ = "And what is your name ?": GOSUB 1000
100 END
1000 REM cursor positioning and printing
1010 POKE 211,SP :POKE 214, ZE: SYS 58640 : PRINT A$
1020 RETURN
2000 REM delay-loop
2010 FOR X=0 TO 3000: NEXT X
2020 RETURN
```

POKE 211, SP → set_column(SP)…

Knappast.
Vore bättre att kunna skriva print_at(20, 3, ”Good Morning!”)

Vi vill ha begreppet print_at()!

Begrepp 2

- Java:
  - Det går utmärkt att skriva…

```
public class Game {
   public void moveEnemies() {
      …
      …
      …
      double newX = x + Math.sqrt((x2-x1)*(x2-x1) + (y2-y1)*(y2-y1));
      …
      …
}
```

Men det uttrycker inte vad vi tänkte!

Vad tänkte vi?

Om vi inte vet vad programmeraren tänkte,
hur vet vi då om det är korrekt implementerat?

Begrepp 3

- Java:
  - Det är lite bättre att skriva:

```
public class Game {
   public void moveEnemies() {
      …
      …
      …
      double distance = Math.sqrt((x2-x1)*(x2-x1) + (y2-y1)*(y2-y1));
      double newX = x + distance;
      …
      …
}
```

Ge namn till deluttryck!

Nu ser vi lite mer av vad programmeraren tänkte!

Begrepp 4

jonkv@ida

- Java:
  - Det är ännu bättre att skriva:

Som att utöka språket med meningsfulla begrepp!

```
public class Game {
   public double distance(double x1, double y1, double x2, double y2) {
      return Math.sqrt((x2-x1)*(x2-x1) + (y2-y1)*(y2-y1));
   }
   public void moveEnemies() {
      …
      …
      double newX = x + distance(x1, y1, x2, y1);
      …
      …
}
```

Själva avståndsberäkningen är ett meningsfullt begrepp

Som en metod kan detta återanvändas och anropas igen

Uttrycket reflekterar direkt en tanke:
Nya värdet är x plus avståndet mellan…

Begrepp 5

jonkv@ida

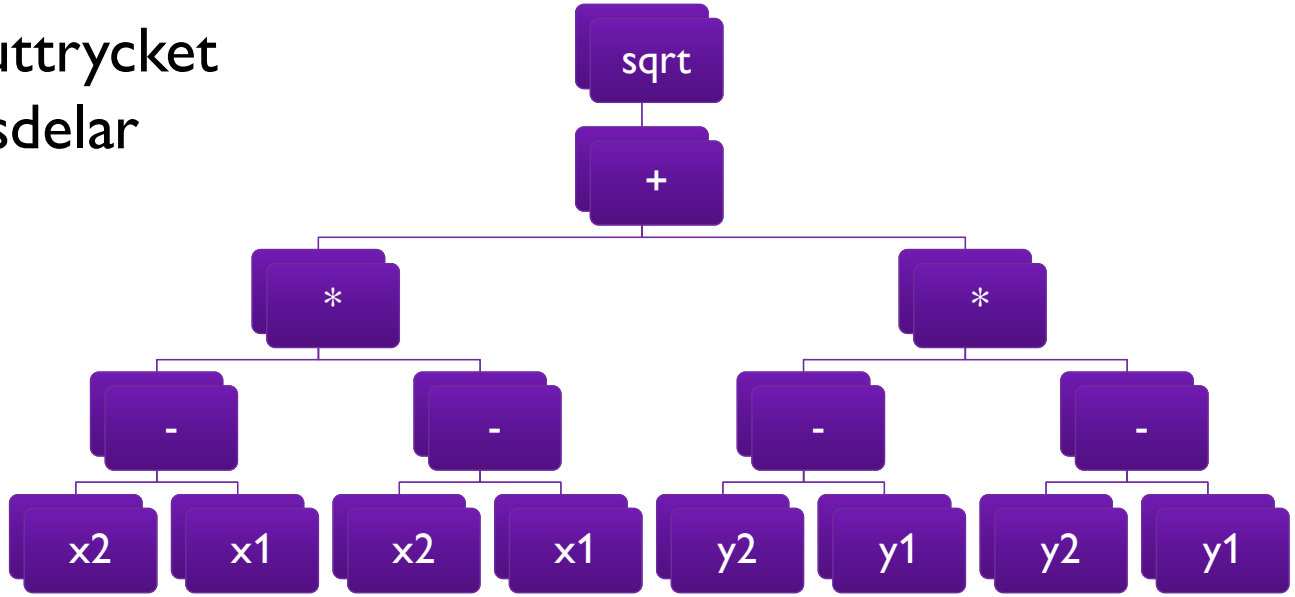
- Bättre att införa *punkter* som egna begrepp (finns i java.awt.Point)!
  - Vi vill ha avstånd mellan två **punkter/saker**, inte mellan **4 flyttal**…

```
public class Game {
   public double distance(Point p1, Point p2) {
      return Math.sqrt((p2.x – p1.x)*(p2.x – p1.x) +
                       (p2.y – p1.y)*(p2.y – p1.y));
   }
   public void moveEnemies() {
      …
      …
      double newX = x + distance(enemy.getPos(), player.getPos());
      …
      …
}
```

Begrepp 6

jonkv@ida

- Det matematiska uttrycket har också beståndsdelar
  - Kan införa namn på olika delar




```
public class Game {
   public double distance(Point p1, Point p2) {
      double deltaX = p2.x – p1.x;
      double deltaY = p2.y – p1.y;
      return Math.sqrt(deltaX * deltaX + deltaY * deltaY);
   }
}
```

VIKTIGT!
Vi får MÅNGA inlämningar med alltför lite namngivna värden

Dela upp koden i lagom många meningsfulla metoder

Long method

Begrepp

- Tidigare nämnde vi **egna datatyper** som **nya begrepp**
  - Det **finns redan** heltal, flyttal, sanningsvärden, …
  - Vi **inför själva** begreppet ”kund”, ”bankkonto”, …

Datatyper som begrepp i programspråk

- Varje ny **datatyp** blir ett **nytt meningsfullt begrepp**!
  - Vi **samlar ihop** information – och ger den också ett **namn**
    - ➔ Nya ord införs i vårt programmeringsspråk (Customer, BankAccount); inte fast i språkets egna begrepp (lista, dict, …)
    - ➔ Vi kan lättare ***skriva*** kod, ***förstå*** existerande kod: **Begreppen** kan anpassas mer till ***hur vi tänker*** (”en kund”, inte ”en lista med dessa 17 kundrelaterade värden”)
  - **Söndra och härska** (divide and conquer)!
    - Dela upp programmeringen i **delar av lämplig storlek**
    - Se till att varje enskild del kan **förstås i detalj**

Nu har vi samlat ihop information – men hur bearbetar vi den?

Nya ”saker” med egna namn, utökar vår vokabulär

Recept

- Pannkaksreceptet **lagaPannkaka()**, **utan bakningsbegrepp**:
  - … 1000 rader…
  - **För** handen till vispen och **greppa** den.
  - **Lyft** vispen med handen medan du **undviker** att kollidera med andra föremål.
  - …
  - **För** handen i cirkulära rörelser…
  - …

Omöjligt att läsa 3000 rader recept och förstå helheten!

Inför nya meningsfulla begrepp: ”vispa”, …

- **lagaPannkaka()**, enligt Arla:
  - **Vispa** ut mjölet i hälften av mjölken till en slät smet. **Vispa** i resterande mjölk, ägg och salt.
  - **Låt** smeten svälla ca 10 min.
  - **Smält** smör i en stekpanna och **häll** ner i smeten. **Grädda** tunna pannkakor.

Nu kan vi förstå receptet i sin helhet, på en högre nivå: Lagom långt, lagom abstraktionsnivå!

Metoden vispa(): En ny handling med eget namn

Grundläggande och fördefinierade begrepp

jonkv@ida

- När vi ska få datorn att **göra något**:
  - Det **finns redan** handlingar i språket
    - Kontrollstrukturer – loopar, villkor, …
    - Tilldelning av värden
    - Jämförelser
  - Det finns **bibliotek av färdig funktionalitet – färdiga begrepp**
    - **System.out.println()** – *att skriva ut något på skärmen*

Begrepp och metoder

jonkv@ida

- **Kan** bygga upp allt med ett **minimalt antal nya begrepp**

```
class MyGameComponent {
   void tick() { // Necessary: Called 50 times per second
      … 300 lines of code, updating object positions,
      checking for collisions,
      updating scores …
   }
}
```

Svårt att få översikt!

- Dela i **flera** metoder med tydligt **eget ansvar** → skapa nya **begrepp**

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

Lätt att se
vad som händer under tick()

Definiera nytt meningsfullt
begrepp

Grundläggande och fördefinierade begrepp

- Även **korta ”mönster”** kan bli till egna begrepp
  - **System.out.println()** – *att skriva ut något, med radbrytning efter*
    - Uppbyggt av primitivare begrepp!
    - ```
      public void println(String x) {
          synchronized (this) {
              print(x);
              newLine();
          }
      }
      ```

Kunde ha krävt att användaren anropade:
System.out.print(x);
System.out.newLine();

Men println är ett användbart eget begrepp!

Även om det bara gäller 2 rader…
      ```

Balansera metodstorlek

- Hur långt ska vi gå?
  - 1000 metoder med 2 rader i varje?
    - Lätt att förstå varje enskild metod
    - Svårt att förstå sammanhanget
  - 2 metoder med 1000 rader i varje?
    - Tvärtom.
  - Hitta en balans!

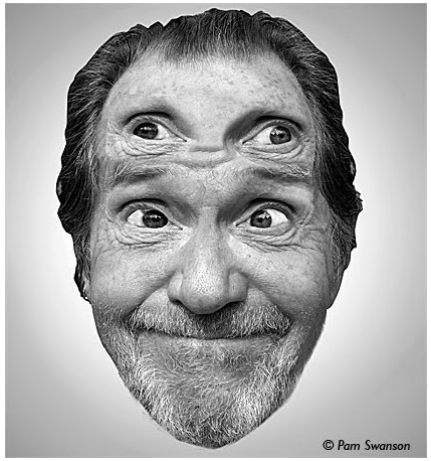
Metodlängd (1)

- Hur lång får en metod vara?
  - *"The first rule of functions is that they should be small.*
    The second rule of functions is that they should be smaller than that.
    Functions should not be 100 lines long.
    Functions should hardly ever be 20 lines long."
    - -- Robert Martin, *Clean Code: A Handbook of Agile Software Craftsmanship*
  - *"The routine should be allowed to grow organically up to 100-200 lines.*
    Decades of evidence say that routines of such length
    [are] no more error prone than shorter routines."
    - -- Steve McConnell, *Code Complete*

Metodlängd (2): Motsatser

- Varför dessa motsatser?
  - Ju fler kockar desto sämre soppa

  - Fyra ögon ser bättre än två



För varje regel finns en lika stark och motsatt motregel…

Metodlängd (3): Läsbarhet

jonkv@ida

- **Mål**: Kod ska vara **läsbar**, **lätt att förstå**, **lätt att underhålla**
  - Långa metoder ➔ **varningssignal**!
    - Är metoden osammanhängande? Repetitiv? Gör den för mycket?
    - Antagligen finns meningsfulla delar som kan bli egna metoder!

  - Många korta metoder ➔ **varningssignal**!
    - Är det för många metoder för att hålla reda på?
    - Skulle det bli enklare om man kombinerade dem?

Hitta rätt balans
Prioritera målet (läsbarhet) istället för medlet (uppdelningen)

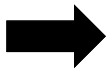
Men:
"För osammanhängande" metoder, och repetitiv kod,
är vanligare i inlämningar och ger ofta komplettering

Metodlängd (4): Uppdelning

jonkv@ida

Dela upp ➔ nya begrepp/namn

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



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

Förbättrar ofta läsbarhet

IDEA kan ibland hjälpa till: Refactor | Extract | Method

Undvik magiska konstanter:
Låt namn visa upp vad värdena betyder

Excessive use of literals

Magiska konstanter

- **Vanligt problem**: Många magiska konstanter
  - Tveka inte att ge **namn** till dem – då får allt en **betydelse**
  - player.setX(32 + 4 * getWidth() / 5)
    - Det fanns nog en idé bakom detta
    - Men vad?
  - if (player.getSpeed() >= 5) { … }
    - Vad är det för speciellt med 5 egentligen? En maxfart? En tröskel?
    - Förekommer 5 på flera platser? Om jag vill ändra 5, måste jag ändra något annat?
  - Gör kodens betydelse uppenbar – introducera namn!

Konstanter: Var deklareras de?

jonkv@ida

- Mönster: **Samla alla konstanter**
  - ```
    public class Constants {
        public static final double PI = 3.14159;
        public static final int BUTTON_POS = 200;
        public static final String PLAYER_NAME_1
                = "Main Player";
    }
    ```

Ofta dålig indelning, *antimönster*: ”Här är allt som är konstant i hela projektet”

- ***Konstanterna hör ofta till någon annan*** – placera dem där!
  - GameGUI hanterar gränssnittet [och vet/äger allt om det, inklusive positioner]
  - ```
    public class GameGUI {
        public final static int BUTTON_POS = 200;
        …
    }
    ```

Bättre: ”Här är allt som har med grafiska gränssnittet att göra”

Använd hellre data än kod

Undvik **hårdkodning**

Data: EnumMap

jonkv@ida

Tillkommer ny SquareType måste vi ändra koden, kompilera om
Vill vi ändra en färg …

```
SquareType st = board.get(x,y);
switch (st) {
   case EMPTY:
      g2d.setColor(WHITE);
      break;
   case L:
      g2d.setColor(BLACK);
      break;
   case I:
      g2d.setColor(GREEN);
      break;
   case Z:
      g2d.setColor(BLUE);
      break;
   …
```

Kan skicka med olika färgkartor, läsa en Map från fil, …

Data är flexiblare!

```
EnumMap<SquareType, Color) colors =
   new EnumMap<>();

colors.put(SquareType.EMPTY, WHITE);
colors.put(SquareType.L, BLACK);
…

SquareType st = board.get(x,y);
g2d.setColor(colors.get(st));
```

Gäller i princip alla if/switch-kedjor där man bara vill hitta ett *värde*:
Sådant ska man slå upp i en mappning

```
if (name.equals(”Cow”)) icon = ”cow.jpg”;
```

Data: EnumMap

jonkv@ida

Tillkommer nya nivåer måste vi ändra denna kod…

Jobbig kod att läsa!

```
switch (level) {
   case 1:
      this.speed = 7
      this.bonus = 3
      break;
   case 2:
      this.speed = 10;
      this.bonus = 5;
      break;
  …
}
```

Kan lätt skapa nya nivåer, skicka runt dem som data, spara på fil!

Inte upprepad kod, många rader per nivå

```
public class Level {
   private final int speed;
   private final int bonus;
   … constructor, getters, …
}

List<Level> levels = List.of(
   new Level(7, 3), new Level(10, 5)
);

this.speed = level.getSpeed();
this.bonus = level.getBonus();
```

Information ska inte representeras som procedurell kod!

More Maps

jonkv@ida

Mer kod som representerar data

```
public String getNextMap() {
   switch (currentMap) {
      case "Map1":
         return "Map2";
      case "Map2":
         return "Map3";
      case "Map3":
         return "Map3-extended";
      case "Map3-extended":
         currentMap = "Map12";
      case "Map12":
         return null;
   }
}
```

Kan lätt skapa nya sekvenser, skapa dem *on the fly*, modifiera dem, använda slumpen, …

```
// … skapa en Map<String,String> …

public String getNextMap() {
   return nextMap.get(currentMap);
}
```

Lite om lagring av information

Typer att använda

Globala variabler?

Välj "rätt" lagring

jonkv@ida

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

```
public enum HorizontalDirection {
   LEFT, STILL, RIGHT;
}
public class GenericPlayer {
   private HorizontalDirection movingHorizontally;
}
```

Alla möjliga värden är rimliga

➔ vi har gjort en feltyp omöjlig

Globala variabler:
Fall inte för frestelsen!

Exempel: Poänglista i Tetris (1)

jonkv@ida

- Exempel: **Poänglista i Tetris**
  - **Just nu** ska det bara finnas **en** lista:
    Den är i någon mening *global*
  - Några delar av koden (GUI, spara-på-fil, …)
    behöver tillgång till den

- Hmmm. När en variabel är **static**
  behöver man inte hitta ”rätt objekt”…
  - Och är den **public**
    kan vem som helst komma åt den…
  - *”Trevligt! Mindre data att skicka runt!”*
    …eller?

Exempel: Poänglista i Tetris (2)

- **Varning 1A:** Lagra all information *direkt* som statiska variabler?

```
public class HighscoreList
{
   public static List<Score> scores;
   public static int maxScore;
}
```

Alla fält är statiska och publika ➔ vem som helst kommer åt dem

Inte objektorienterat:
Det finns inget *objekt* av HighscoreList-typ
➔ kan inte skicka listan som parameter,
kan inte skapa flera listor,
kan inte använda ärvning [senare], …

KOMPLETTERING!

Exempel: Poänglista i Tetris (3)

- **Varning 1B:** Privata statiska variabler?

```
public class HighscoreList
{
   private static List<Score> scores;
   private static int maxScore;
   public static void addScore(Score newScore) {
      scores.add(newScore);
      scores.sort(...);
   }
   public static Score getHighestScore() {
      return scores.get(0);
   }
}
```

Alla fält är statiska men privata

Användning: Anropa ”global funktion”
HighscoreList.addScore(theScore);

Samma problem som tidigare
Informationen är fortfarande global,
helt i onödan

KOMPLETTERING!

Exempel: Poänglista i Tetris (4)

jonkv@ida

- **Varning 2**, **Singleton**: Börja som en vanlig klass…

```
public class HighscoreList {
   public List<Score> scores;
   public void addScore(Score newScore) { … }
   public Score getHighestScore() { … }
```

  - …men se till att bara **en** instans skapas, och ge **global** tillgång till den

```
   // Privat konstruktor, så bara klassen själv kan skapa instanser.
   private HighscoreList() { … }

   // Klassen skapar ETT objekt av denna typ
   public static final HighscoreList
      INSTANCE = new HighscoreList();
}
```

(1) Kan fortfarande komma åt listan utan att ha "fått" den
➔
Svårare att förstå dataflödet

(2): Fortfarande ett fundamentalt designbeslut att det bara kan finnas 1 global highscorelista –
får liknande konsekvenser!

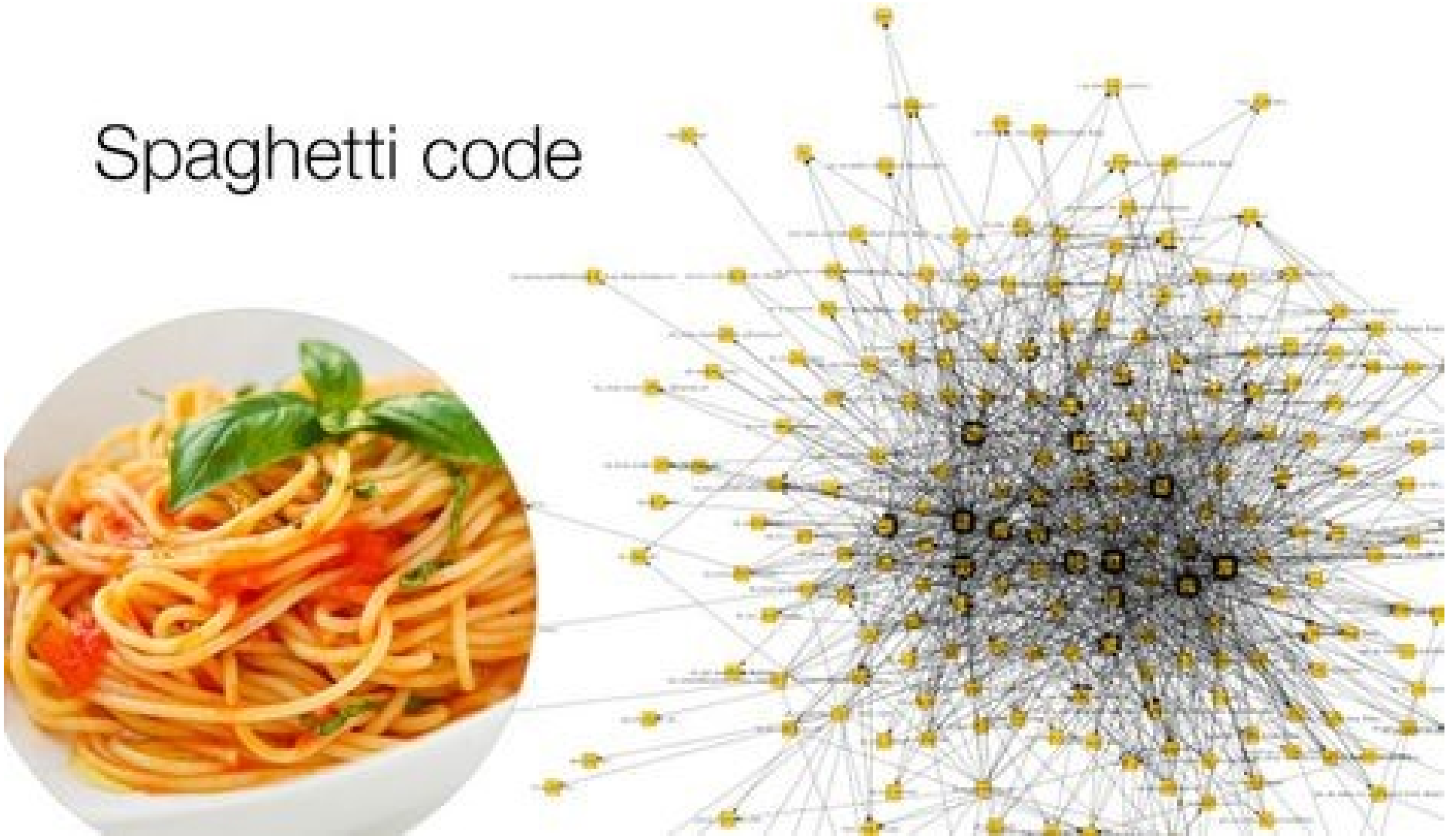
  - …och använd sedan denna instans

```
HighscoreList hs = HighscoreList.INSTANCE;
hs.addScore(theScore);
```

Static: Röriga beroenden

jonkv@ida

- Global tillgång ger **röriga beroenden**, en form av **spaghettikod**



Exempel: Poänglista i Tetris (5)

jonkv@ida

- Bäst: Låt HighscoreList vara en **helt vanlig** klass…

```
public class HighscoreList {
   public HighscoreList() { … }

   public List<Score> scores;
   public void addScore(Score newScore) { … }
   public Score getHighestScore() {  }
}
```

- …Men hur ser vi till att det bara blir **en** highscorelista?

Exempel: Poänglista i Tetris (6)

jonkv@ida

- **Skapa** helt enkelt bara **en**
  - Du har ju kontrollen över din kod!

```
public class Tetris {
   public static void main(String[] args) {
      HighscoreList highscores = new HighscoreList();
      Board board = new Board(highscores, …);
      HighscoreViewer viewer = new HighscoreViewer(highscores, …);
      Controller ctrl = new Controller(board, …);
   }
}
```

Skicka info till de som behöver!

Finns bara 1 lista: Här skapas den

Finns bara 1 spelbräde: Här skapas det

Vill du någon gång ha 2 listor är allt förberett!

Dokumentera... lagom!

Python: Docstrings

Java: **Javadoc**

Javadoc-kommentarer

- Kan dokumentera **klasser**, **fält**, **metoder**, **konstruktorer**, …
  - Kommentar omedelbart **ovanför** en deklaration

```
/**
 * Removes all mappings from this map (optional operation).
 *
 * @throws UnsupportedOperationException clear
 *   is not supported by this map.
 */
public void clear() {
   …
}
```

Startar med /**

Typ av info kan markeras…

Slutar med */

https://www.oracle.com/technetwork/java/javase/documentation/index-137868.html

Javadoc (2)

- Korrekt format är viktigt: Javadoc används av **verktyg**!
  - IntelliJ IDEA: Tryck Ctrl-Q för att se Quick Documentation

```
@SpringBootApplication
public class DemoApplication {

    public static void main(String[] args) {
        SpringApplication.run(
    }
}
```

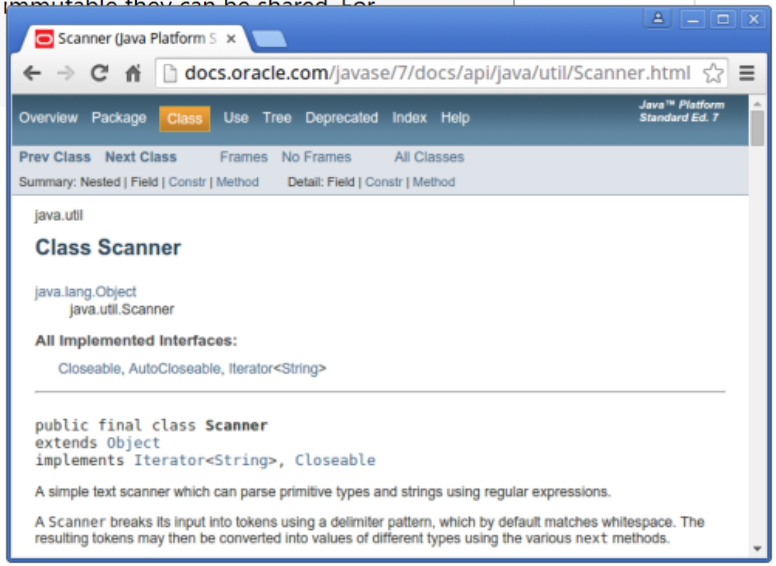
```
java.lang
public final class String
extends Object
implements java.io.Serializable, Comparable<String>, CharSequence
```

The `String` class represents character strings. All string literals in Java programs, such as `"abc"`, are implemented as instances of this class.

Strings are constant; their values cannot be changed after they are created. String buffers support mutable strings. Because String objects are immutable they can be shared. For example:

```
String str = "abc";
```

  - Javadoc (kommandorad): Skapa websidor för API-dokumentation



Kan inte säga ”as above”, går inte att ersätta med annan kommentarstyp eller flytta!

Javadoc (3)

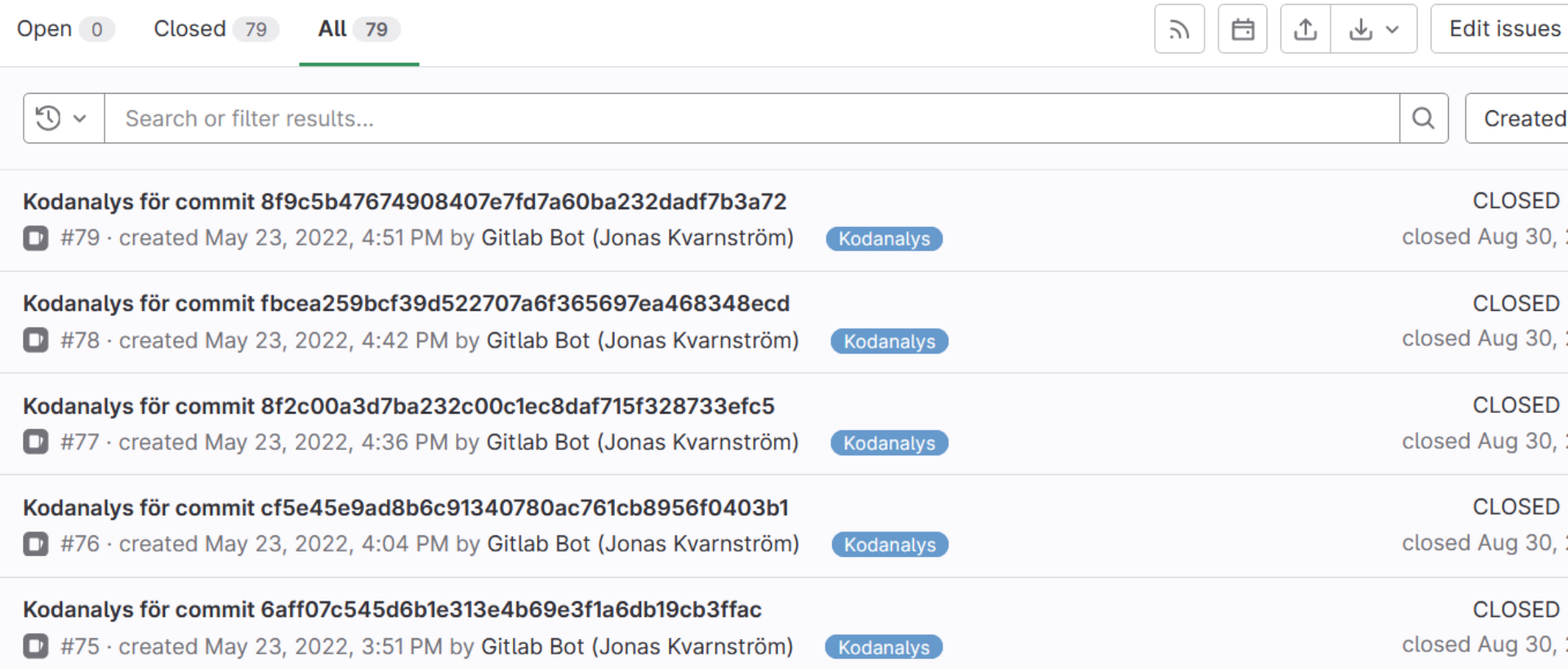
jonkv@ida

- Krav på javadoc i projektet:
  - Alla klasser
  - Alla publika fält
  - *Där det behövs* – mindre uppenbara delar av projektet

Mer om kodinspektion

Kodinspektion

- Varje gång ni pushar till Gitlab:
  - *Automatisk kodinspektion körs*
  - *Ger issues i Gitlab, tilldelade till er*



Kodinspektion (2)

jonkv@ida

Closed Issue created May 23, 2022, 4:51 PM by **Gitlab Bot (Jonas Kvarnström)** Reopen issue

Reporter

Kodanalys för commit `8f9c5b47`

Kodanalysen för commit `8f9c5b47` kan nu laddas ner som en bilaga.

Analysen inkluderar IDEAs inspektioner, men även många utökade inspektioner som är inspirerade av problem som är mer specifika för just labbarna och kursprojekten. Tanken med detta är inte att öka kraven på er, utan att göra det enklare att förstå sådant som redan diskuteras i kursmaterialet och kursböckerna!

0 0

Kodinspektion (3)

Översikt över kod

Varningar med olika vikt (”importance”)

Olika vikt för olika betyg: Tips om vad man ska fokusera på för att höja betyget

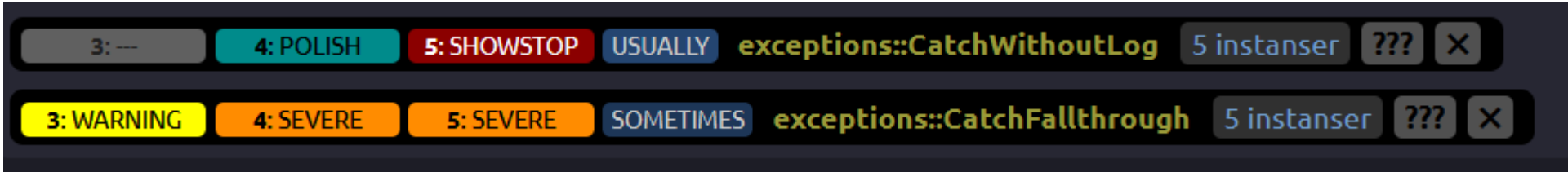
Kodinspektion (4)

- Showstoppers
  - Omöjligt att få ett visst betyg med en *giltig* showstopper för det betyget
  - Giltig = korrekt varning (verktyget ger *varningar* men *kan ha fel*)
    - Hur ofta är den korrekt? SOMETIMES, USUALLY, ALMOST ALWAYS, ALWAYS

Kodinspektion (5)

jonkv@ida

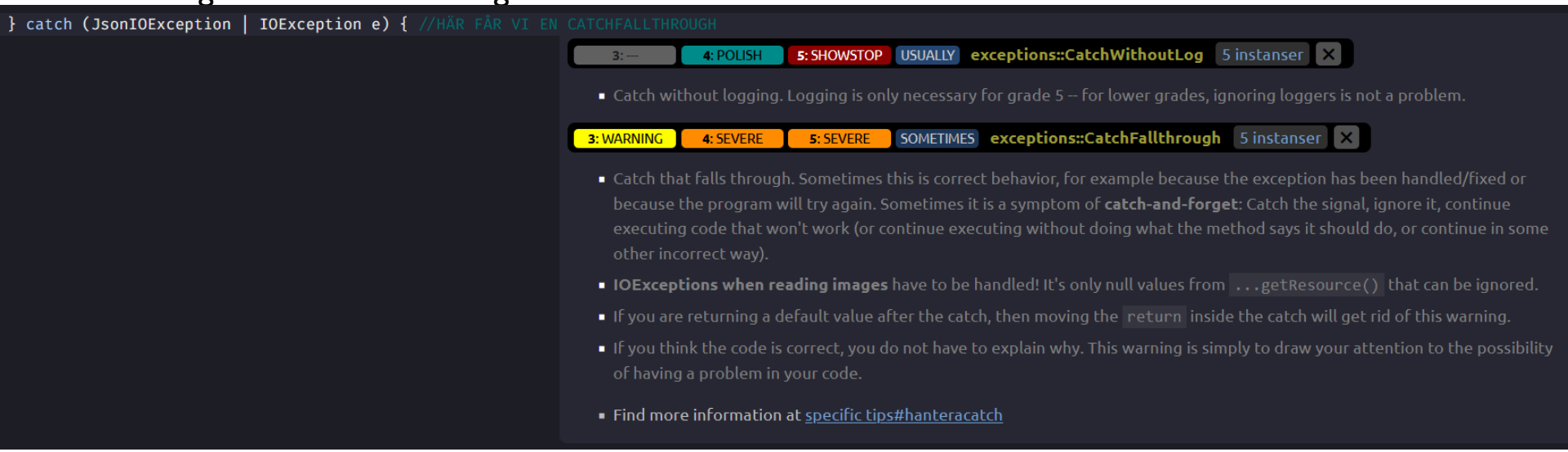
- Nivå SEVERE:
  - Starkt negativt!
  - Till skillnad från SHOWSTOP kan man få godkänt ändå
  - Men bara om resten av projektet kompenserar genom mycket hög kvalitet
- Nivå WARNING:
  - Vanlig varning, bör inte finnas så många
- Nivå WEAKWARN:
  - Svag varning, ska inte finnas så *väldigt* många
- Nivå POLISH:
  - Inget större problem, ändå något att tänka på om man vill polera ordentligt



Kodinspektion (6)

jonkv@ida

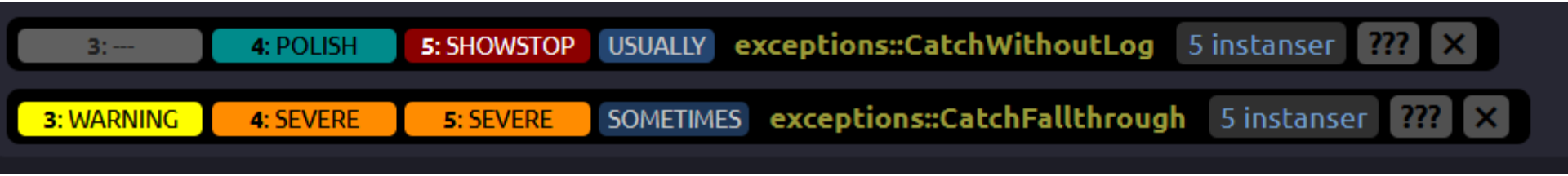
- Varningar visas inne i koden
  - Med **förklaringar**
    - Catch that falls through. Sometimes this is correct behavior, for example because the exception has been handled/fixed or because the program will try again. Sometimes it is a symptom of catch-and-forget: Catch the signal, ignore it, continue executing code that won't work (or continue executing without doing what the method says it should do, or continue in some other incorrect way).
    - IOExceptions when reading images have to be handled! It's only null values from ...getResource() that can be ignored.
    - If you are returning a default value after the catch, then moving the return inside the catch will get rid of this warning.



Kodinspektion (7)

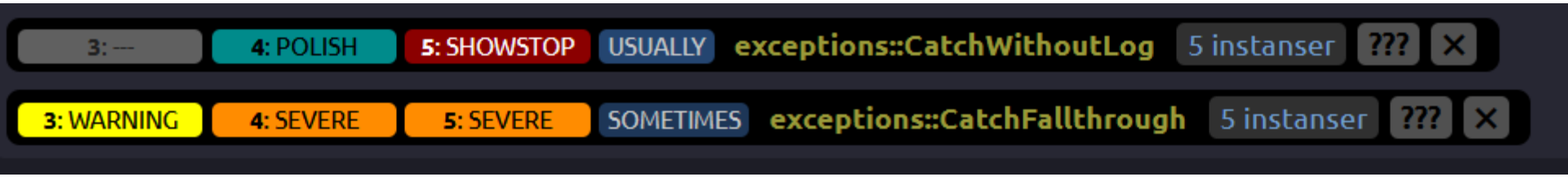
jonkv@ida

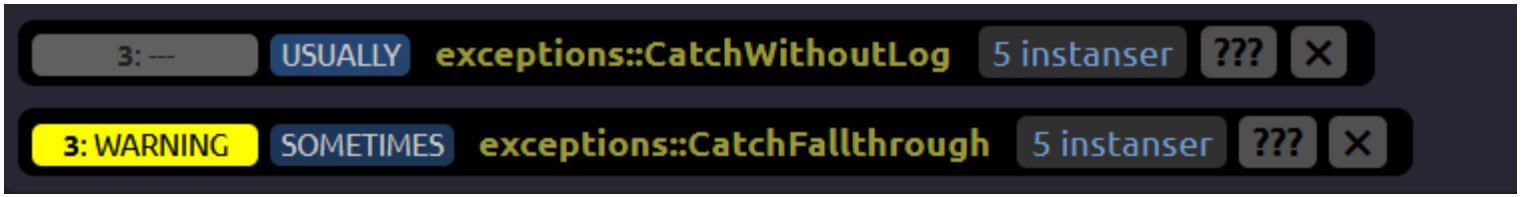
- Upprepade varningar ➔ kollapsade förklaringar
  - Klicka ”???” för att expandera förklaringen



Kodinspektion (8)

- Fokusera på specifikt betyg? Klicka betygsvalen…





Kodinspektion (9)

- Allvarliga fel i själva projektet ➔ stor röd felruta i början
  - Saknade filer, osv
  - Ska **inte lämnas in**:
    Projektet **kan inte inspekteras och granskas**,
    ger ingen återkoppling
  - (Showstopper ➔ får inte godkänt men får ändå återkoppling)

Inget IDEA-projekt

I detta Git-repo hittar vi inget incheckat IDEA-projekt: Ingen `.idea` -katalog och ingen `*.ipr` -fil. Därför kan koden inte analyseras.

Ett korrekt konfigurerat IDEA-projekt krävs för den slutliga inlämningen, så skapa det gärna redan nu.
Inlämningar utan sådant projekt ger komplettering utan kodgranskning.

Det går sedan bra att fortsätta med själva utvecklingsarbetet i en annan miljö.

Kodinspektion (10)

jonkv@ida

- Varför kodinspektion?
  - **Inte** för att **ge er mer att göra**
  - För att **ge er en bättre chans att höja kvaliteten**
    där handledaren **ändå skulle ha kommenterat och hittat problem**
  - För att ge er en **översikt** över **de potentiella problem** som
    kan hittas automatiskt, och deras vikt (importance),
    så ni får en **helhetsbild**
  - För att **lära ut** sådant som man lär sig bäst genom **träning**,
    genom att **skriva kod och få kommentarer**