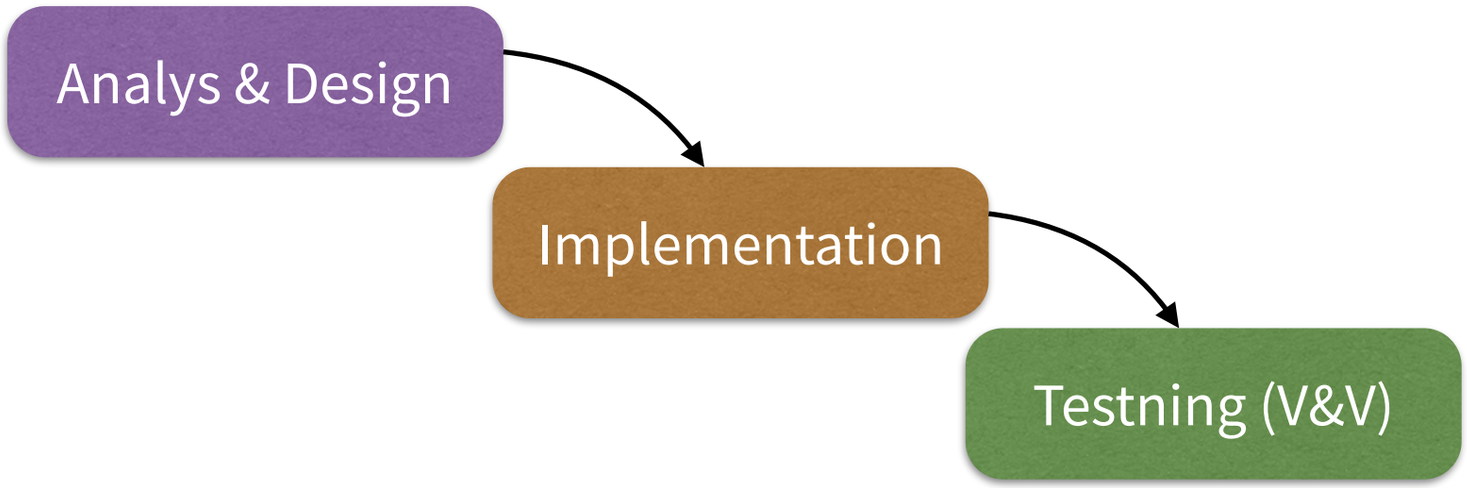
Föreläsning 23

Tobias Wrigstad

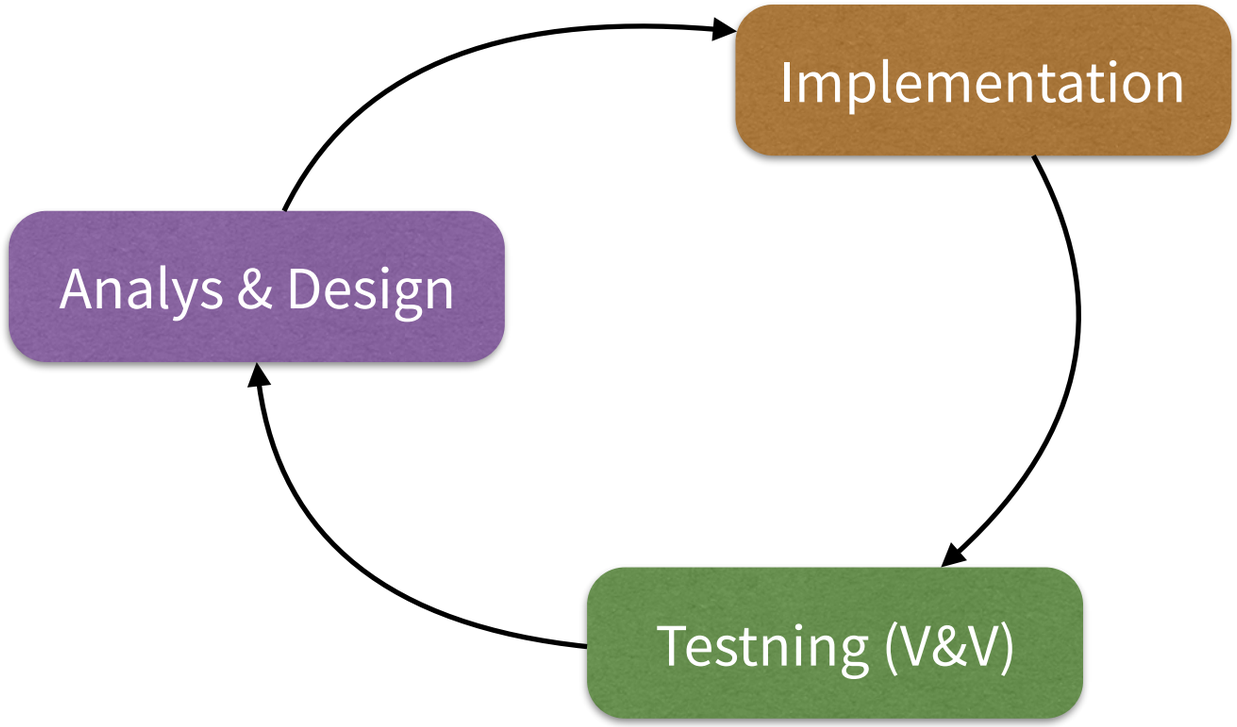
Refaktorering

Traditionell syn på systemutveckling



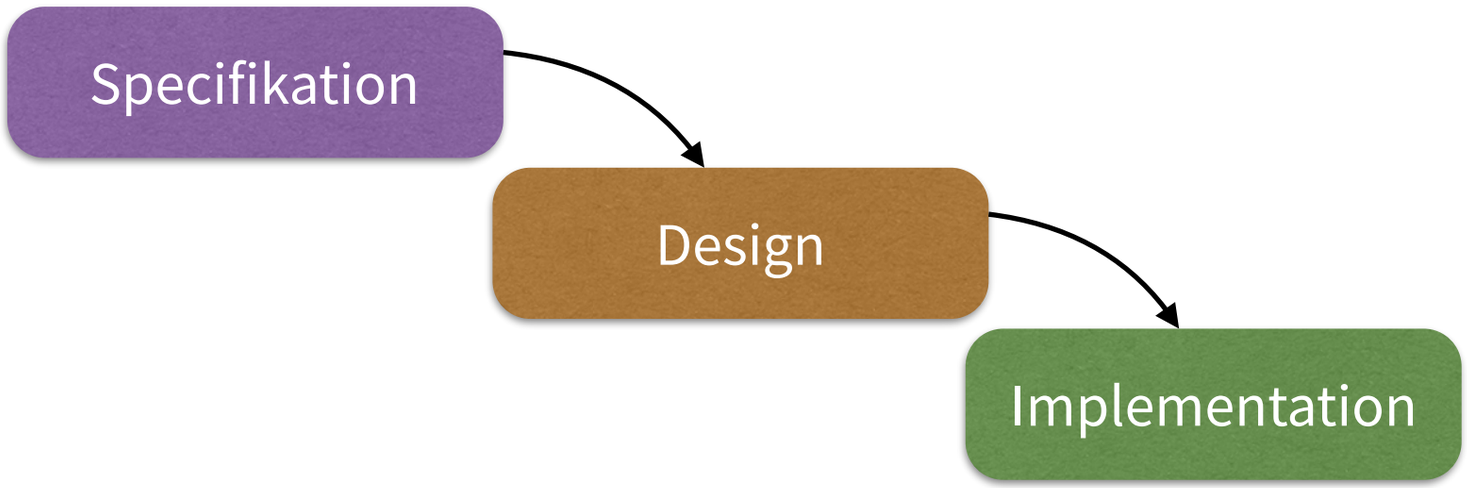
- Den s.k. ”vattenfallsmodellen”
- Diskreta steg som bildar en pipeline — varje steg ger indata till nästa steg som utdata

Modern syn på systemutveckling



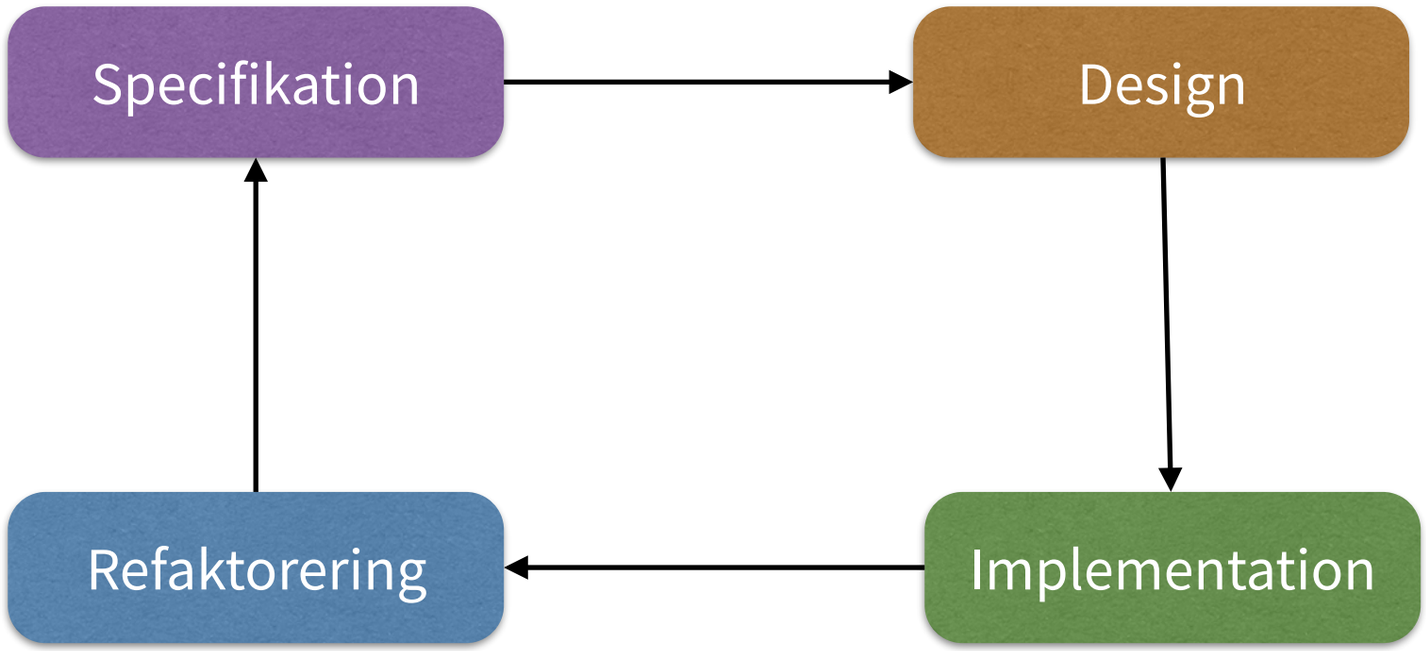
- Utveckling är en *iterativ process* — i regel lönlöst/suboptimalt att försöka förstå problemet innan man börjar implementera
- Använd istället implementationen för att driva fram förståelse
- Kontinuerlig *validering & verifiering* — underlättas t.ex. genom att alltid ha ett körande system

Naiv syn på implementation



- Samma problem som vattenfallsmodellen — omöjligt att beakta allt jämt
- Korrekthet är endast en av många kvalitetsattribut (t.ex. underhållsbar, läsbar, …)
- Tidspress och dylikt medför ibland att programmerare skriver undermålig kod
- Kod produceras i en kollektiv process — leder ibland till t.ex. dupliceringar (jmf. NIH)
- Behövs processteg för att gradvis förfina och ställa tillrätta — *refactoring*

Implementationscykeln



- Nytt steg — refaktorering
- Förändringar i koden i syfte att göra koden *bättre* utan att påverka vad den gör, eller systemets prestanda

  Ändra kodens struktur

Refactoring

- Syftet med refaktorering är att göra kod och design

  Mer underhållsbart

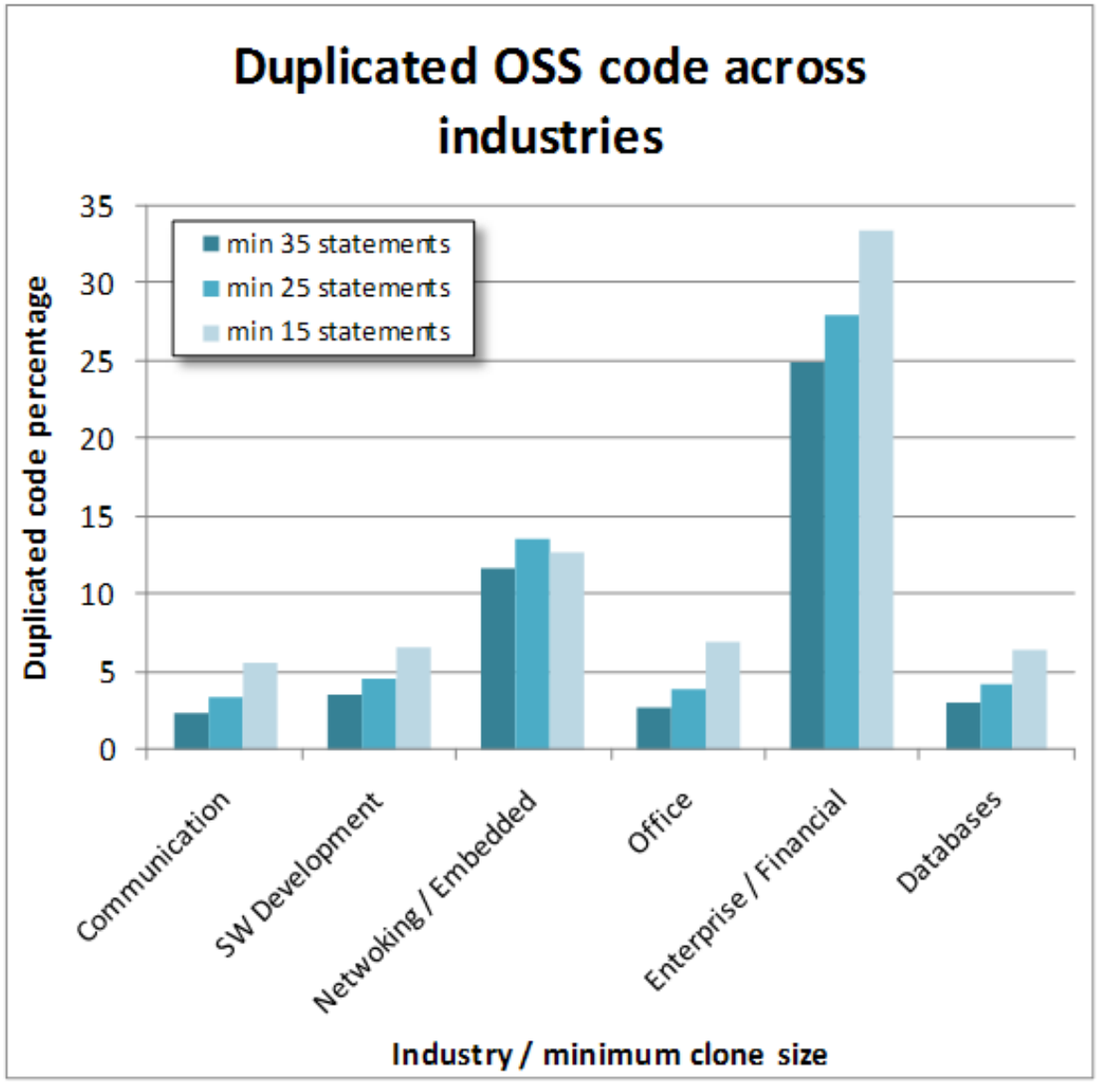
  Lättare att förstå

  Lättare att ändra

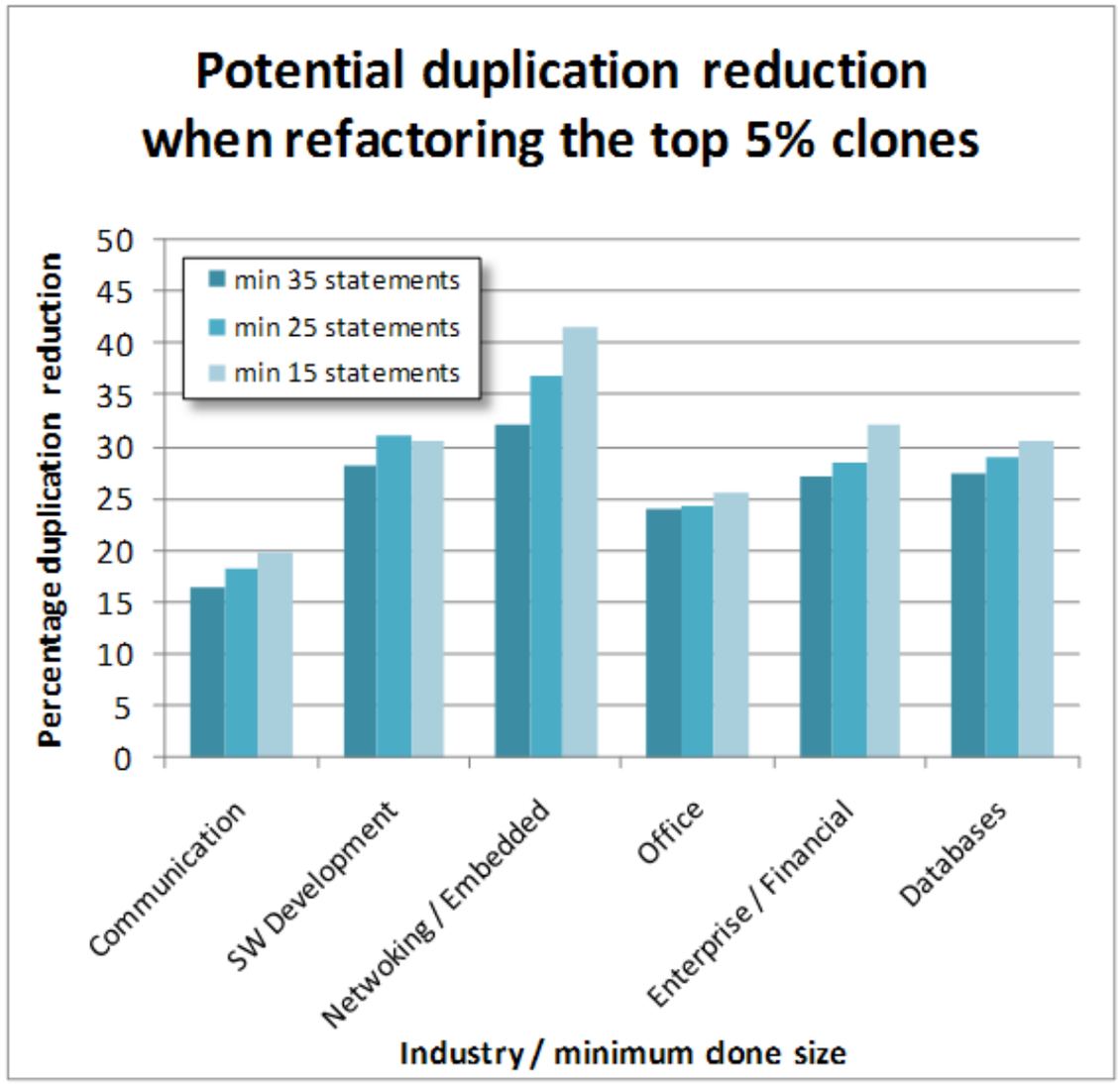
  Enklare att utöka med ny funktionalitet

- Termen populariserades av Martin Fowler, se boken Refactoring (AW, 1997)
- Fowler lyckas fånga många refaktoreringsprocesser i namngivna mönster
- Traditionellt inte alltid tillåtet på alla arbetsplatser (”if it ain’t broke, don’t fix it”)

Exempel: kodduplicering



a)



b)

Via https://solidsourceit.wordpress.com/2012/08/03/does-source-code-duplication-matter/

Exempel: underhåll

Det är i regel enklare att underhålla kod som du själv har skrivit

Enklare att förstå, följer dina normala tankebanor

Mindre respekt för koden

Lejonparten av all systemutveckling är underhåll av existerande system

Dvs. underhåll av kod som du **inte** skrivit själv

ERGO:

Alla bör bemöda sig om att skriva kod som är så enkel som möjligt att förstå

”Bad smells” och ruttnande kod

Kod tenderar att ruttna över tid

Många modifieringar under tidspress, med olika mentala modeller, etc.

Vi säger att ruttnande kod luktar illa — bad smells

Som utvecklare är det vårt mål att identifiera kod som luktar illa och städa up den

Vad avger dålig lukt?

En **igenkänningsbar indikator** på att något i koden kan vara fel

All kod kan ruttna — även testkod (alltså inte bara produktionskod)

Typiska dåliga lukter

- Magiska konstanter
- Upprepningar
- Långa metoder
- Komplicerade villkorssatser
- Switchsatser
- Stora klasser
- Divergerande förändringar
- Shotgun-surgery
- Kodkommentarer

Typiska dåliga lukter

- Magiska konstanter
- Upprepningar
- Långa metoder
- Komplicerade villkorssatser
- Switchsatser
- Stora klasser

- Divergerande förändringar
- Shotgun-surgery
- Kodkommentarer

Intra-klass-lukt

Inter-klass-lukt

Upprepningar

```
public class TUnit {
    public static void main(String[] args) throws Exception {
       for (String className : args) {
            Class c = Class.forName(className);
            Object o = c.newInstance();

            Method setup = null;
            Method tearDown = null;
            for (Method m : c.getMethods()) {
                if (m.getName().equals("setup")) { setup = m; break; }
            }

            for (Method m : c.getMethods()) {
                if (m.getName().equals("tearDown")) { tearDown = m; break; }
            }

            for (Method m : c.getMethods()) {
                if (m.getName().startsWith("test") && m.getParameterCount() == 0) {
                    if (setup != null) setup.invoke(o);
                    m.invoke(o);
                    if (tearDown != null) tearDown.invoke(o);
                }
            }
        }
```

Upprepningar

```
public class TUnit {
    public static void main(String[] args) throws Exception {
       for (String className : args) {
            Class c = Class.forName(className);
            Object o = c.newInstance();

            Method setup = null;
            Method tearDown = null;
            for (Method m : c.getMethods()) {
                if (m.getName().equals("setup")) { setup = m; break; }
            }

            for (Method m : c.getMethods()) {
                if (m.getName().equals("tearDown")) { tearDown = m; break; }
            }

            for (Method m : c.getMethods()) {
                if (m.getName().startsWith("test") && m.getParameterCount() == 0) {
                    if (setup != null) setup.invoke(o);
                    m.invoke(o);
                    if (tearDown != null) tearDown.invoke(o);
                }
            }
        }
```

Magiska konstanter

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

public class TUnit {
    public static void main(String[] args) throws Exception {
        if (args.length < 1) {
            System.out.println("Usage: java TUnit TestClass1 ... ");
            return;
        }

        for (String className : args) {
            Class c = Class.forName(className);
            Object o = c.newInstance();

            Method setup = null;
            Method tearDown = null;
            for (Method m : c.getMethods()) {
                if (m.getName().equals("setup")) setup = m;
                if (m.getName().equals("tearDown")) tearDown = m;
                if (setup != null && tearDown != null) break;
            }

            for (Method m : c.getMethods()) {
                if (m.getName().startsWith("test") && m.getParameterCount() == 0) {
                    if (setup != null) setup.invoke(o);
                    m.invoke(o);
                    if (tearDown != null) tearDown.invoke(o);
                }
```

Magiska konstanter

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

public class TUnit {
    public static void main(String[] args) throws Exception {
        if (args.length < 1) {
            System.out.println("Usage: java TUnit TestClass1 ... ");
            return;
        }

        for (String className : args) {
            Class c = Class.forName(className);
            Object o = c.newInstance();

            Method setup = null;
            Method tearDown = null;
            for (Method m : c.getMethods()) {
                if (m.getName().equals("setup")) setup = m;
                if (m.getName().equals("tearDown")) tearDown = m;
                if (setup != null && tearDown != null) break;
            }

            for (Method m : c.getMethods()) {
                if (m.getName().startsWith("test") && m.getParameterCount() == 0) {
                    if (setup != null) setup.invoke(o);
                    m.invoke(o);
                    if (tearDown != null) tearDown.invoke(o);
                }
```

Långa metoder

```
public class TUnit {
    public static void main(String[] args) throws Exception {
        if (args.length < 1) {
            System.out.println("Usage: java TUnit TestClass1 ... ");
            return;
        }

        for (String className : args) {
            Class c = Class.forName(className);
            Object o = c.newInstance();

            Method setup = null;
            Method tearDown = null;
            for (Method m : c.getMethods()) {
                if (m.getName().equals("setup")) setup = m;
                if (m.getName().equals("tearDown")) tearDown = m;
                if (setup != null && tearDown != null) break;
            }

            for (Method m : c.getMethods()) {
                if (m.getName().startsWith("test") && m.getParameterCount() == 0) {
                    if (setup != null) setup.invoke(o);
                    m.invoke(o);
                    if (tearDown != null) tearDown.invoke(o);
                }
            }
        }
```

När är en metod för lång egentligen?

- Nästlade kontrollstrukturer med djup större än 2
- Tar många parametrar som radikalt ändrar hur metoden skall bete sig
- När dess logik är duplicerad i andra metoder
- Onödigt brus, t.ex. kommentarer som är uppenbara, bekvämlighetsmetoder som inte används, etc.
- Den ryms inte på en skärmsida
- När den som läser inte får en omedelbar och intuitiv förståelse för vad den gör
- …

Refaktorerat program [partiell]

```
public class TUnit {
    public static final String SETUP                = "setup";
    public static final String TEAR_DOWN            = "tearDown";
    public static final String TEST_METHOD_PREFIX = "test";

    public static void main(String[] args) throws Exception {
       for (String className : args) {
          Class c = loadClass(className);
          runTestSuite(c);
       }
    }

    public Method findMethod(Class c, String name) { ... }
    public Method findTestMethods(Class c) { ... }
    public void runTest(Method s, Method td, Method test) { ... }

    public static void runTestSuite(Class c) {
       Method setup = findMethod(c, SETUP);
       Method tearDown = findMethod(c, TEAR_DOWN);

       Method[] testMethods = findTestMethods(c);
       for (Method m : testMethods) runTest(setup, teardown, m);
    }
}
```

✅ *Upprepningar*

✅ *Långa metoder*

✅ *Magiska konstanter*

Refaktoreringsmönster

- En refaktorering är en kodtransformation som utförs manuellt eller med verktygsstöd

$$kod \Rightarrow kod$$

- Skall tillämpas *kontinuerligt*, inte med en månads mellanrum
- En fungerande uppsättning tester är av stor vikt för att minska riskerna vid komplex refaktorering

Refaktoreringsprocessen

1. Se till att alla tester passerar (alla?)
2. Identifiera vad som luktar illa
3. Gör en plan för hur koden skall refaktoreras
4. Genomför planen
5. Kör alla tester för at se till att inga förändringar/buggar/etc. smög in
6. Gå till 1.

Refaktoreringsmönster [refactoring.com/catalogue]

- Add parameter
- Change bidirectional association to unidirectional
- Change reference to value
- Change unidirectional association to bidirectional
- Change Value to Reference
- Collapse Hierarchy
- Consolidate Conditional Expression
- Consolidate Duplicate Conditional Fragments
- Convert Procedural Design to Objects
- Decompose Conditional
- Duplicate Observed Data
- Encapsulate Collection
- Encapsulate Downcast
- Encapsulate Field
- Extract Class
- Extract Hierarchy
- Extract Interface
- Extract Method
- Extract Subclass
- Extract Superclass
- Hide Delegate
- Hide Method
- Inline Class
- Inline Method
- Rename Constant

Extract Method

```
Method setup = null;
for (Method m : c.getMethods()) {
  if (m.getName().equals("setup")) { ... }
}

Method tearDown = null;
for (Method m : c.getMethods()) {
  if (m.getName().equals("tearDown")) { ... }
}
```

```
Method setup = findSetupMethod(c.getMethods());

Method tearDown = null;
for (Method m : c.getMethods()) {
  if (m.getName().equals("tearDown")) { ... }
}
```

Extract Method

```
Method setup = null;
for (Method m : c.getMethods()) {
  if (m.getName().equals("setup")) { ... }
}

Method tearDown = null;
for (Method m : c.getMethods()) {
  if (m.getName().equals("tearDown")) { ... }
}
```

```
Method setup = findSetupMethod(c.getMethods());

Method tearDown = null;
for (Method m : c.getMethods()) {
  if (m.getName().equals("tearDown")) { ... }
}
```

```
Method s = findSetupMethod(c.getMethods());

Method tearDown = null;
for (Method m : c.getMethods()) {
  if (m.getName().equals("td")) { ... }
}
```

```
Method s = findSetupMethod(c.getMethods());

Method td = findTDMethod(c.getMethods());
```

Så drar vi nytta av den bättre strukturen

```
Method setup = null;
for (Method m : c.getMethods()) {
  if (m.getName().equals("setup")) { ... }
}

Method tearDown = null;
for (Method m : c.getMethods()) {
  if (m.getName().equals("tearDown")) { ... }
}
```

```
Method setup = findSetupMethod(c.getMethods());

Method tearDown = null;
for (Method m : c.getMethods()) {
  if (m.getName().equals("tearDown")) { ... }
}
```

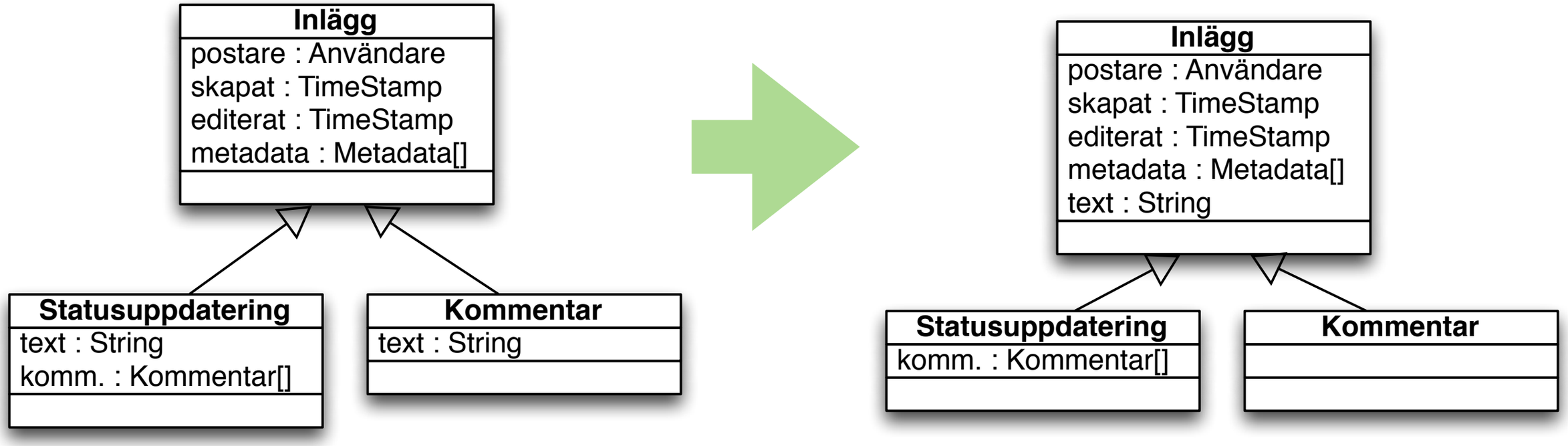
```
Method s = findSetupMethod(c.getMethods());

Method tearDown = null;
for (Method m : c.getMethods()) {
  if (m.getName().equals("td")) { ... }
}
```

```
Method s = findSetupMethod(c.getMethods());

Method td = findTDMethod(c.getMethods());
```

```
Method s = findSetupMethod(c.getMethods());

Method td = findTDMethod(c.getMethods());
```

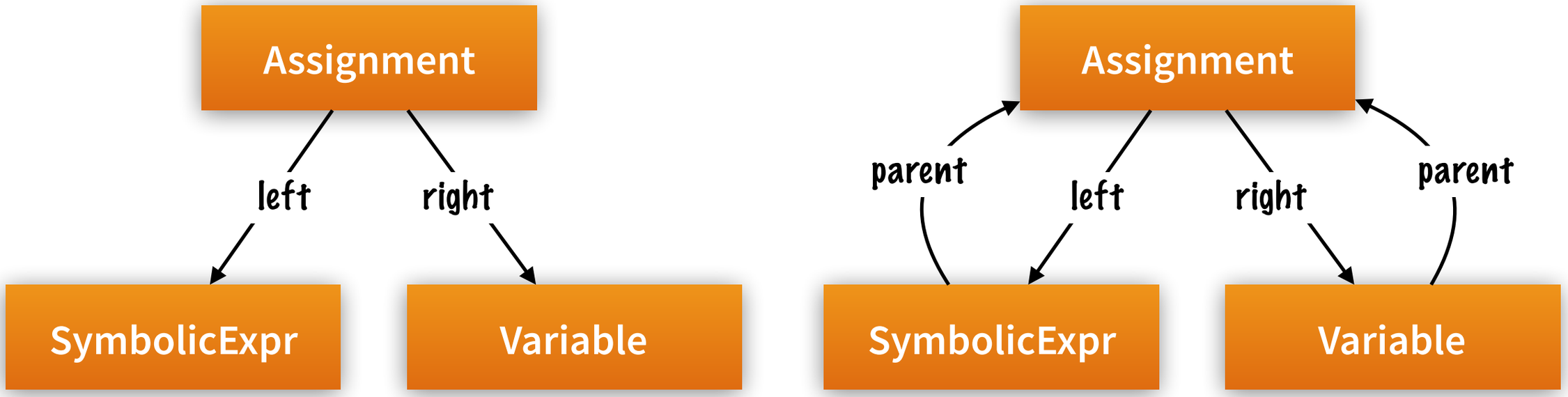
```
Method s = findMethod(c.getMethods(), SETUP);

Method td = findMethod(c.getMethods(), TD);
```

Pull Up Field



- Denna typ av kod kan uppstå t.ex. för att olika programmerare har arbetat parallellt, eller för att någon skillnad mellan `Statusuppdatering` och `Kommentarer` tidigare har funnits men som inte längre gäller, etc.

Change unidirectional association to bidirectional

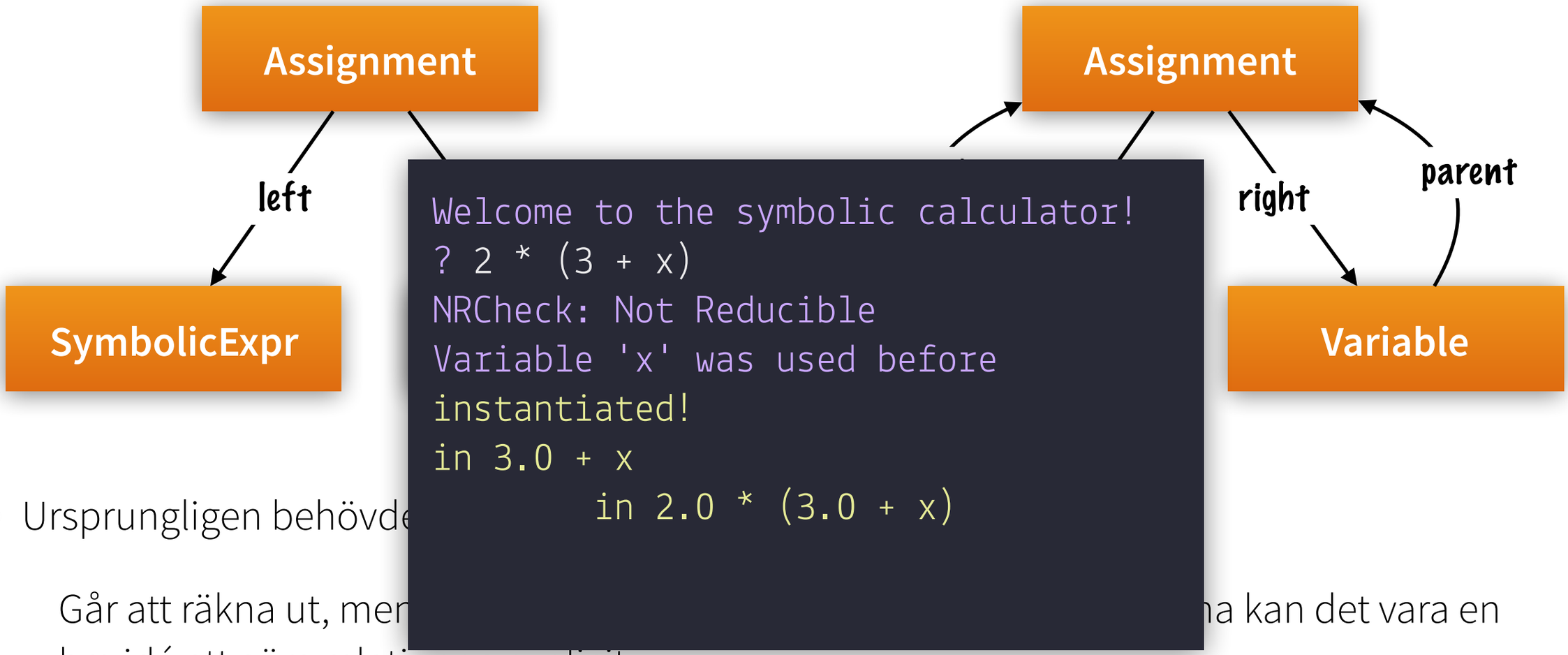


- Ursprungligen behövdes enbart ena riktningen, nu behövs båda

  Går att räkna ut, men om programmet ofta behöver båda riktningarna kan det vara en bra idé att göra relationen explicit

  Lösning i inlupp 3: lägg till `SymbolicExpression parent;` i rotklassen

Change unidirectional association to bidirectional



- Ursprungligen behövde

  Går att räkna ut, men na kan det vara en bra idé att göra relationen explicit

  Lösning i inlupp 3: lägg till `SymbolicExpression parent;` i rotklassen

Single Responsibility Principle [Robert C. Martin]

- Varje modul (klass) har ansvar för en del av programmets funktion
- Ansvaret är inkapslat i modulen (klassen)
- ”Det skall bara finnas en anledning till att en klass ändras”

- I inlämningsuppgift 3 — visitor pattern

  Bryta ut evalueringsfunktionen från AST-trädet

  Ersätt med generell princip för att besöka ett träd

  Tillåter oss att bygga flera olika visitors för ett träd (eval, check, etc.)

”Rule of Three”

- Första gången vi skall göra något — bara gör det
- Andra gången vi skall göra nästan samma sak — kopiera det
- Tredje gången vi skall göra nästan samma sak — dags att refaktorera

Avslutning, refaktorering

- Att refaktorera är nödvändigt för att ett program inte skall ruttna ihop och behöva skrivas om från grunden
- Flera av er har upplevt detta under kursen ”den hårda vägen”

  Refaktorering kan hjälpa, men kanske för mycket att göra för er just nu

  SIMPLE-metoden uppmuntrar till kontinuerlig (trivial) refaktorering — nu vill jag uppmuntra er till att göra mer komplicerad refaktorering för högre vinster

- Inlupp 4 startar med testning för att sedan göra refaktorering…

  …och sedan utökningar!

- Titta på refactoring.com/catalogue för att läsa om olika mönster

  Man kan lära sig en del om programmering genom att göra det