

Föreläsning 23

Tobias Wrigstad

*Profilering. Refactoring. JIT-
kompilering. Bytecode*



Vad innehåller en .class-fil?

```
ClassFile {  
    u4      magic;  
    u2      minor_version;  
    u2      major_version;  
    u2      constant_pool_count;  
    cp_info constant_pool[constant_pool_count-1];  
    u2      access_flags;  
    u2      this_class;  
    u2      super_class;  
    u2      interfaces_count;  
    u2      interfaces[interfaces_count];  
    u2      fields_count;  
    field_info fields[fields_count];  
    u2      methods_count;  
    method_info methods[methods_count];  
    u2      attributes_count;  
    attribute_info attributes[attributes_count];  
}
```

Vad innehåller en .class-fil?

```
ClassFile {  
    u4      magic;  
    u2      minor_version;  
    u2      major_version;  
    u2      constant_pool_count;  
    cp_info  constant_pool[constant_pool_count-1];  
    u2      access_flags;  
    u2      this_class;  
    u2      super_class;  
    u2      interfaces_count;  
    u2      interfaces[interfaces_count];  
    u2      fields_count;  
    field_info  fields[fields_count];  
    u2      methods_count;  
    method_info  methods[methods_count];  
    u2      attributes_count;  
    attribute_info  attributes[attributes_count];  
}
```

JVM:en är en stackmaskin

```
public class Greeter {  
    public Greeter(String s) { ... }  
    ...  
}  
  
new Greeter("Hello")
```

- En stack med varvat data och operationer

Varje operation pushar eller poppar något från stacken

Exempel: konstruera ett nytt objekt

```
new Greeter // klasser, konstanter, etc. är index in i konstantpoolen  
dup  
ldc "Hello"  
invokespecial
```

- En virtuell maskin har en väldigt enkel struktur

Datastruktur som representerar stacken

En loop som läser maskininstruktionen från toppen av stacken, utför den, poppar vad den behöver och pushar nya instruktioner som resultat

Inspektera en .class-file [Hello.class]

```
public class Hello {  
    public void greet(String s) {  
        System.out.println("Hello, " + s + "!");  
    }  
}
```

Inspektera en .class-file [javap -c Hello]

Compiled from "Hello.java"

```
public class Hello {
```

```
    Hello();
```

```
    Code:
```

```
        0: aload_0
```

```
        1: invokespecial #1
```

```
        4: return
```

```
// Method java/lang/Object."<init>":()V
```

```
public void greet(java.lang.String);
```

```
    Code:
```

```
        0: getstatic      #2
```

```
// Field java/lang/System.out:Ljava/io/PrintStream;
```

```
        3: new            #3
```

```
// class java/lang/StringBuilder
```

```
        6: dup
```

```
        7: invokespecial #4
```

```
// Method StringBuilder."<init>":()V
```

```
       10: ldc            #5
```

```
// String Hello,
```

```
       12: invokevirtual #6
```

```
// Method StringBuilder.append:(Ljava/lang/String;)Ljava/lang/StringBuilder;
```

```
       15: aload_1
```

```
       16: invokevirtual #6
```

```
// Method StringBuilder.append:(Ljava/lang/String;)Ljava/lang/StringBuilder;
```

```
       19: ldc            #7
```

```
// String !
```

```
       21: invokevirtual #6
```

```
// Method StringBuilder.append:(Ljava/lang/String;)Ljava/lang/StringBuilder;
```

```
       24: invokevirtual #8
```

```
// Method StringBuilder.toString:()Ljava/lang/String;
```

```
       27: invokevirtual #9
```

```
// Method PrintStream.println:(Ljava/lang/String;)V
```

```
       30: return
```

```
}
```

Inspektera en .class-file [Hello.class]

```
public class Hello {  
    public void greet(String s) {  
        StringBuilder sb = new StringBuilder();  
        sb.append("Hello, ");  
        sb.append(s);  
        sb.append("!");  
        System.out.println(sb.toString());  
    }  
}
```

```
public class Hello {  
    public void greet(String s) {  
        System.out.println(  
            new StringBuilder().append("Hello, ").append(s).append("!").toString()  
        );  
    }  
}
```

Compiled from "Hello.java"

```
public class Hello {
```

```
    public Hello();
```

```
        Code:
```

```
            0: aload_0
```

```
            1: invokespecial #1
```

```
                // Method java/lang/Object."<init>":()V
```

```
            4: return
```

```
    public void greet(java.lang.String);
```

```
        Code:
```

```
            0: new           #2
```

```
                // class java/lang/StringBuilder
```

```
            3: dup
```

```
            4: invokespecial #3
```

```
                // Method java/lang/StringBuilder."<init>":()V
```

```
            7: astore_2
```

```
            8: aload_2
```

```
            9: ldc           #4
```

```
                // String Hello,
```

```
           11: invokevirtual #5
```

```
                // Method java/lang/StringBuilder.append:(Ljava/lang/String;)Ljava/lang/StringBuilder;
```

```
           14: pop
```

```
           15: aload_2
```

```
           16: aload_1
```

```
           17: invokevirtual #5
```

```
                // Method java/lang/StringBuilder.append:(Ljava/lang/String;)Ljava/lang/StringBuilder;
```

```
           20: pop
```

```
           21: aload_2
```

```
           22: ldc           #6
```

```
                // String !
```

```
           24: invokevirtual #5
```

```
                // Method java/lang/StringBuilder.append:(Ljava/lang/String;)Ljava/lang/StringBuilder;
```

```
           27: pop
```

```
           28: getstatic     #7
```

```
                // Field java/lang/System.out:Ljava/io/PrintStream;
```

```
           31: aload_2
```

```
           32: invokevirtual #8
```

```
                // Method java/lang/StringBuilder.toString:()Ljava/lang/String;
```

```
           35: invokevirtual #9
```

```
                // Method java/io/PrintStream.println:(Ljava/lang/String;)V
```

```
           38: return
```

```
}
```



Compiled from "Hello.java"

```
public class Hello {
```

```
    public Hello();
```

```
        Code:
```

```
            0: aload_0
```

```
            1: invokespecial #1
```

```
                // Method java/lang/Object."<init>":()V
```

```
            4: return
```

```
    public void greet(java.lang.String);
```

```
        Code:
```

```
            0: getstatic      #2
```

```
                // Field java/lang/System.out:Ljava/io/PrintStream;
```

```
            3: new            #3
```

```
                // class java/lang/StringBuilder
```

```
            6: dup
```

```
            7: invokespecial #4
```

```
                // Method java/lang/StringBuilder."<init>":()V
```

```
           10: ldc            #5
```

```
                // String Hello,
```

```
           12: invokevirtual #6
```

```
                // Method java/lang/StringBuilder.append:(Ljava/lang/String;)Ljava/lang/StringBuilder;
```

```
           15: aload_1
```

```
           16: invokevirtual #6
```

```
                // Method java/lang/StringBuilder.append:(Ljava/lang/String;)Ljava/lang/StringBuilder;
```

```
           19: ldc            #7
```

```
                // String !
```

```
           21: invokevirtual #6
```

```
                // Method java/lang/StringBuilder.append:(Ljava/lang/String;)Ljava/lang/StringBuilder;
```

```
           24: invokevirtual #8
```

```
                // Method java/lang/StringBuilder.toString:()Ljava/lang/String;
```

```
           27: invokevirtual #9
```

```
                // Method java/io/PrintStream.println:(Ljava/lang/String;)V
```

```
           30: return
```

```
}
```



Reflection

- Ett program som har möjlighet att inspektera sig självt
- Kraftfullt framförallt i dynamiska programspråk — Java stöd är begränsat
- Kan skapa klasser, anropa metoder, etc.
- Kan inte på ett enkelt sätt

Ändra en metod/ta bort en metod/skapa en ny klass, etc.

Bättre stöd för introspection, dvs. att manipulera det som redan finns

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

public class ReflectionDemo {
    public static void main(String[] args) throws Exception {
        if (args.length != 3) {
            System.out.println("Usage: java ReflectionDemo ClassName MethodName string");
            return;
        }

        Class c = Class.forName(args[0]);
        Object o = c.newInstance();
        Method m = c.getMethod(args[1], new Class[] {String.class});
        m.invoke(o, args[2]);
    }
}
```

```
public class Hello {
    public void greet(String s) {
        System.out.println("Hello, " + s + "!");
    }
}
```

```
$ java ReflectionDemo Hello greet Tobias
Hello, Tobias
```



```
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);  
                }  
            }  
        }  
    }  
}
```

”TUnit”



JIT-kompilering till maskinkod

- Vi har beskrivit interpretatorloopen i stackmaskinen
- Hyfsat effektiv, men inte alls lika effektiv som maskinoptimerad kod
- Maskinoptimerad kod = plattformsspecifik
- Java är plattformsberoende — eftersom program körs i en mjukvarumaskin
- JIT-kompilering är ett försök att tillfredsställa båda behov

Kompilera bytekod till maskinkod under körning

Kompilering tar tid — för att de skall bli en prestandavinst måste vi kompilera selektivt, bara kod som är ”het”

Ytterligare problem: kod laddas in efterhand som programmet körs

- -Xint stänger av JIT-kompilering, pröva på intensivt program för att se skillnad

JIT-kompilering till maskinkod

- Vi har beskrivit interpretatorloopen i stackmaskinen
- Hyfsat effektiv, men inte alls lika effektiv som maskinoptimerad kod
- Maskinoptimerad kod = plattformsspecifik
- Java är plattformsoberoende — eftersom program körs i en mjukvarumaskin
- JIT-kompilering är ett försök att tillfredsställa båda behov

```
time java FibRec 40 — 1.449 s  
time java FibRec -Xint 40 — 79.775 s
```

Kompilera bytekod till maskinkod under körning

Kompilering tar tid — för att de skall bli en prestandavinst måste vi kompilera selektivt, bara kod som är ”het”

Ytterligare problem: kod laddas in efterhand som programmet körs

- -Xint stänger av JIT-kompilering, pröva på intensivt program för att se skillnad

```
$ time java ReflectionDemo Hello greet "Tobias" > /dev/null
1,60s user 0,67s system 117% cpu 1,945 total
```

```
$ time java ReflectionDemo Hello2 greet "Tobias" > /dev/null
1,50s user 0,65s system 113% cpu 1,898 total
```

```
$ time java -Xint ReflectionDemo Hello greet "Tobias" > /dev/null
16,93s user 0,81s system 99% cpu 17,924 total
```

```
$ time java -Xint ReflectionDemo Hello2 greet "Tobias" > /dev/null
17,69s user 0,86s system 98% cpu 18,764 total
```



Hur får man fram profileringsinformation från ett Java-program?

- JVM:en har utmärkt stöd för telemetri, kommandoradsargument vid körning

```
java -Xp $ java -Xrunprof Fib 40
Flat profile of 2,02 secs (144 total ticks): main
java -Xr
      Compiled + native   Method
java -Xr 97,9%   140 +     1   Fib.fibonacci
          97,9%   140 +     1   Total compiled

      Thread-local ticks:
      2,1%      3           Class loader

Global summary of 2,02 seconds:
100,0%   144           Received ticks
   0,7%    1           Compilation
   2,1%    3           Class loader
```


Hur kan vi förbättra detta programs prestanda?

Flat profile of 19.00 secs (223 total ticks): main

Interpreted	+	native	Method	
1.3%	1	+	0	java.lang.AbstractStringBuilder.append
1.3%	1	+	0	java.lang.String.<init>
2.6%	2	+	0	Total interpreted

Compiled	+	native	Method	
51.3%	0	+	40	java.lang.AbstractStringBuilder.expandCapacity
29.5%	23	+	0	java.lang.AbstractStringBuilder.append
10.3%	8	+	0	java.lang.StringBuilder.toString
6.4%	0	+	5	java.lang.String.<init>
97.4%	31	+	45	Total compiled

Thread-local ticks:
65.0% 145 Blocked (of total)

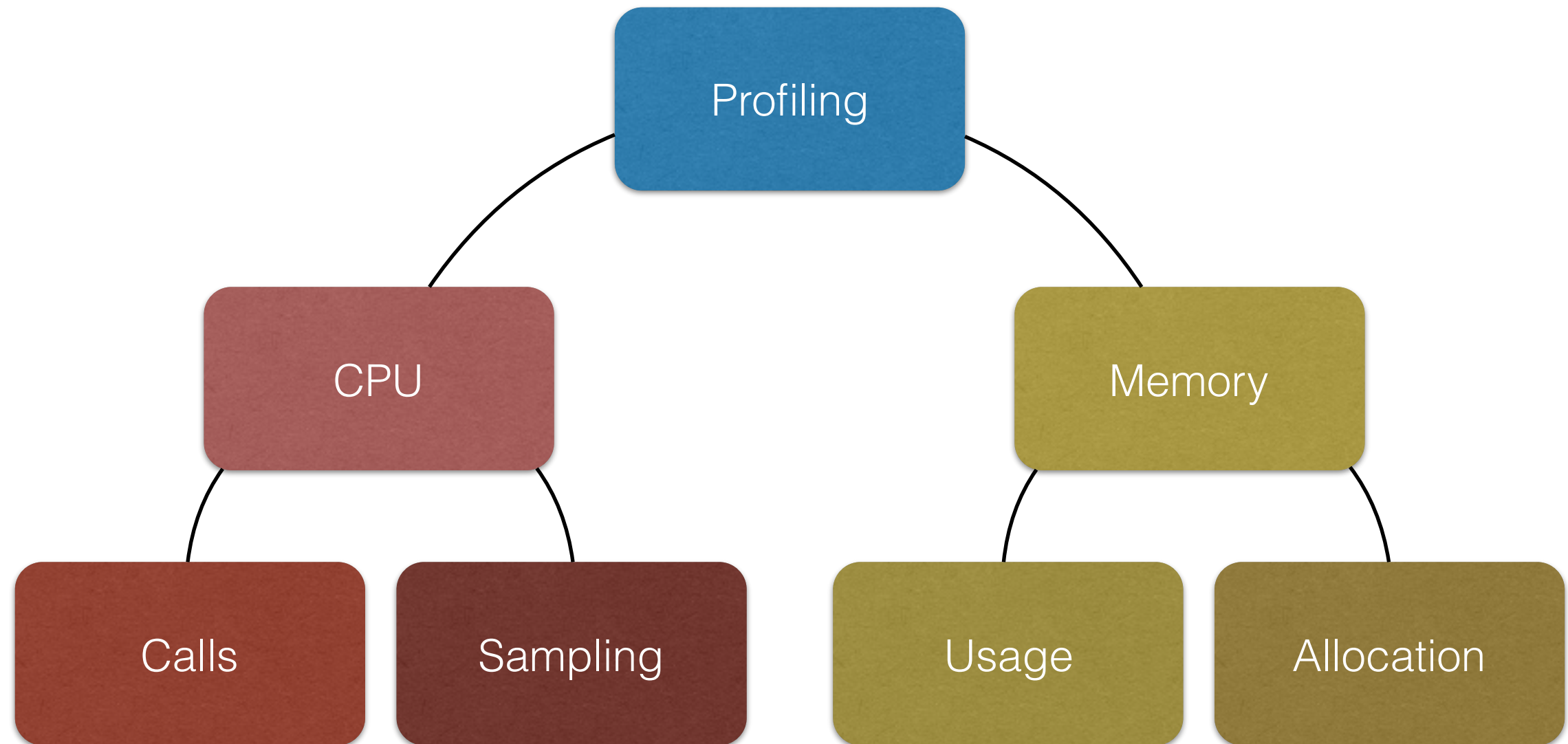
Flat profile of 0.01 secs (1 total ticks): DestroyJavaVM

Thread-local ticks:
100.0% 1 Blocked (of total)

Global summary of 19.01 seconds:
100.0% 929 Received ticks
74.6% 693 Received GC ticks
0.8% 7 Other VM operations



Typer av profilering



Sampling vs. Profiling

- Sampla: mät X gånger per tidsenhet

Inte komplett, men mycket tidseffektivt

- Profiling: stoppa in telemetry i koden som rapporterar varje användande

Komplett, mycket overhead, påverkar körtid negativt

- Använd båda!

Sampling är ofta en bättre teknik om CPU-prestanda är ditt mål

Slutord bytekod, JIT, profilering

- I vilken utsträckning måste man förstå JIT-kompilering och bytekod?

Det beror på vad man gör — men framförallt måste man förstå vad som sker under huven om man någon gång råkar ut för ett program som inte presterar bra

- Försök inte hjälpa JVM:en att generera bra bytekod

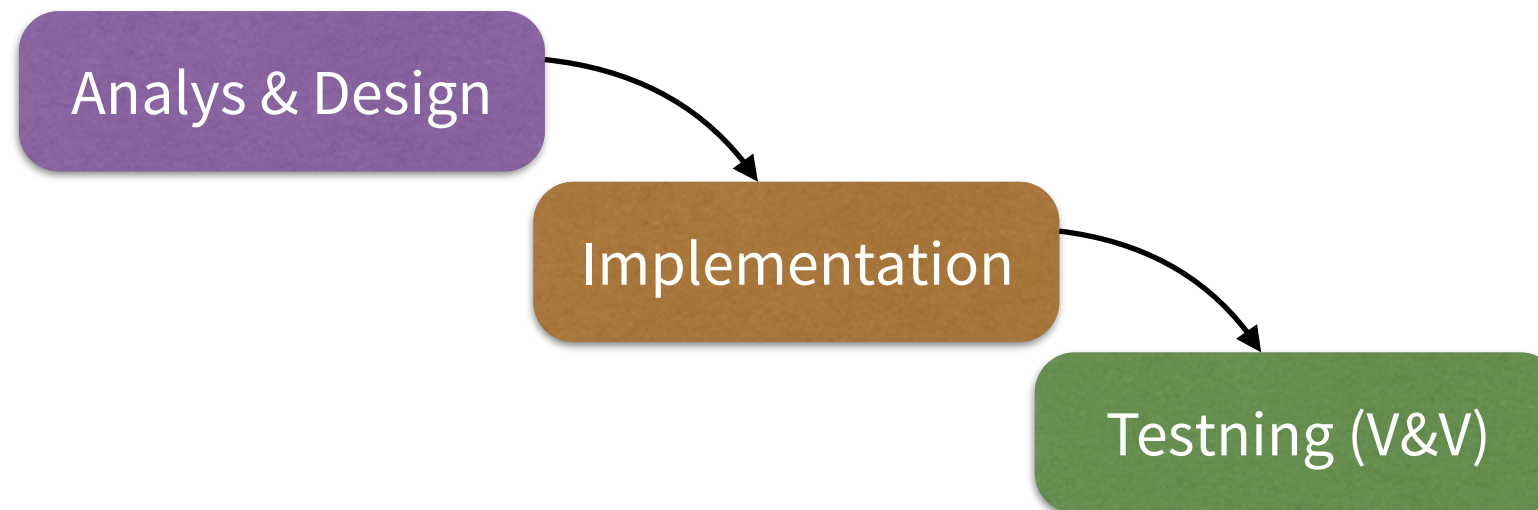
Den är optimerad för ”vanliga dödligas” kod

- Optimera aldrig utan att profilera fram vad som skall optimeras!

Vad är representativ indata/omständigheter för programmet?

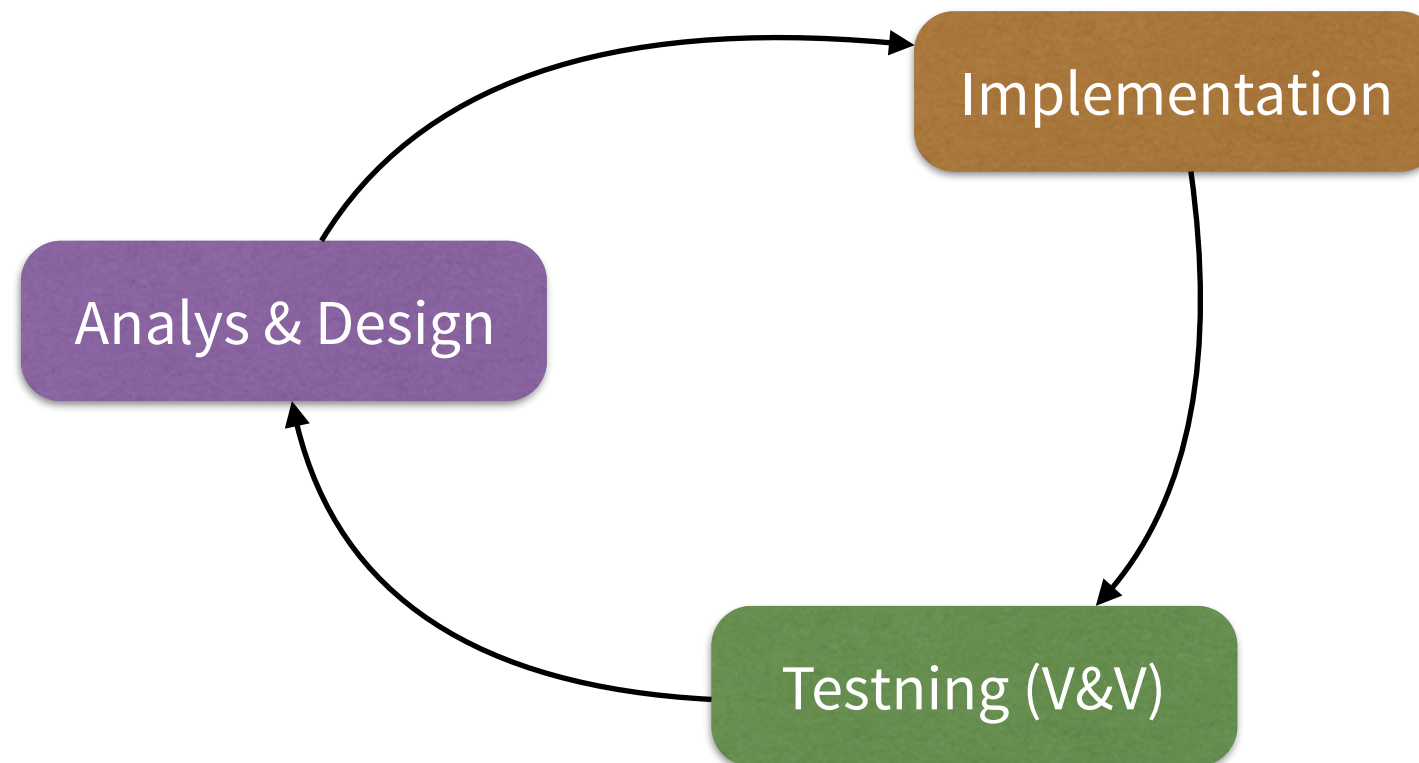
- Verifiera att en optimering inte är en falsk optimering genom att profilera igen!
- Undvik att optimera tills det inte är hållbart längre att inte göra det!

Traditionell syn på systemutveckling



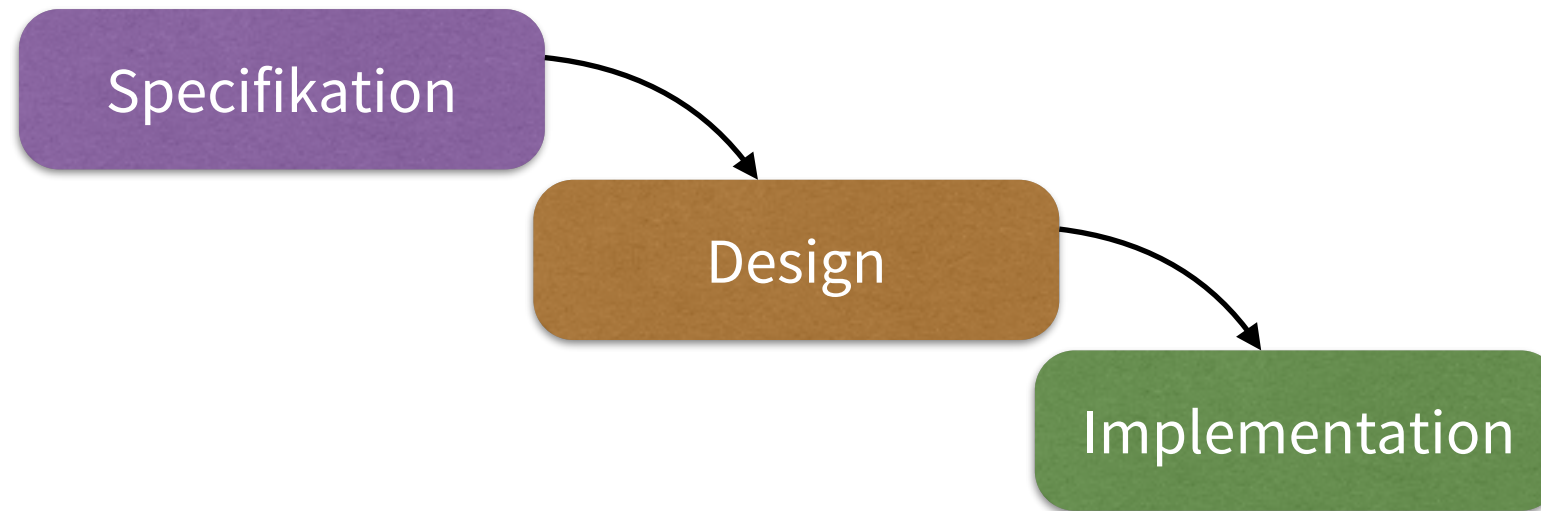
- Den s.k. "vattenfallsmodellen"
- Diskreta steg som bildar en pipeline — varje steg ger indata till nästa steg som utdata

Modern syn på systemutveckling (jmf. "agile")



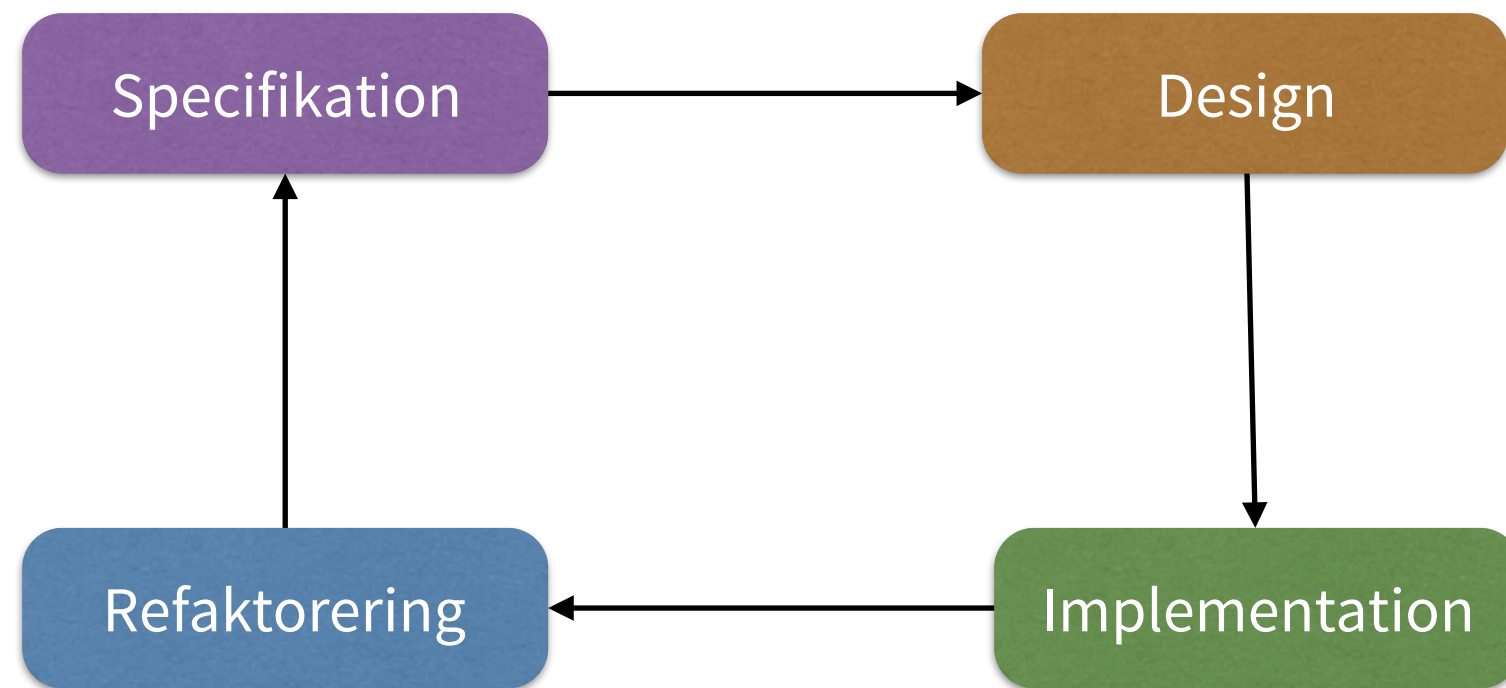
- Utveckling är en *iterativ process* — i regel lönlöst/suboptimalt att försöka förstå problemet innan man börjar implementera
- Använd istället implementationen för att driva fram förståelse
- Kontinuerlig *validering & verifiering* — underlättas t.ex. genom att alltid ha ett körande system

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

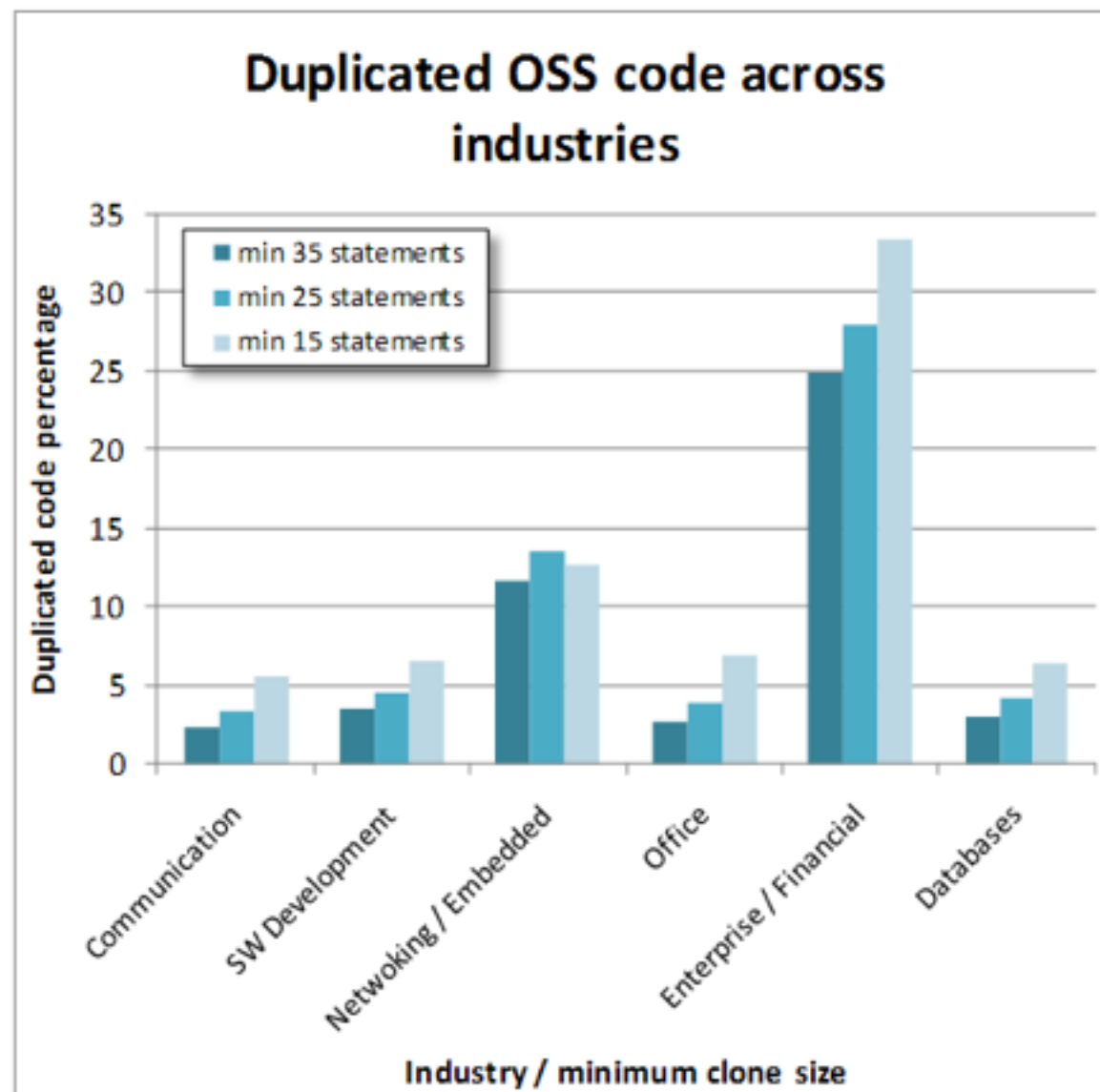
Lättare att förstå

Lättare att ändra

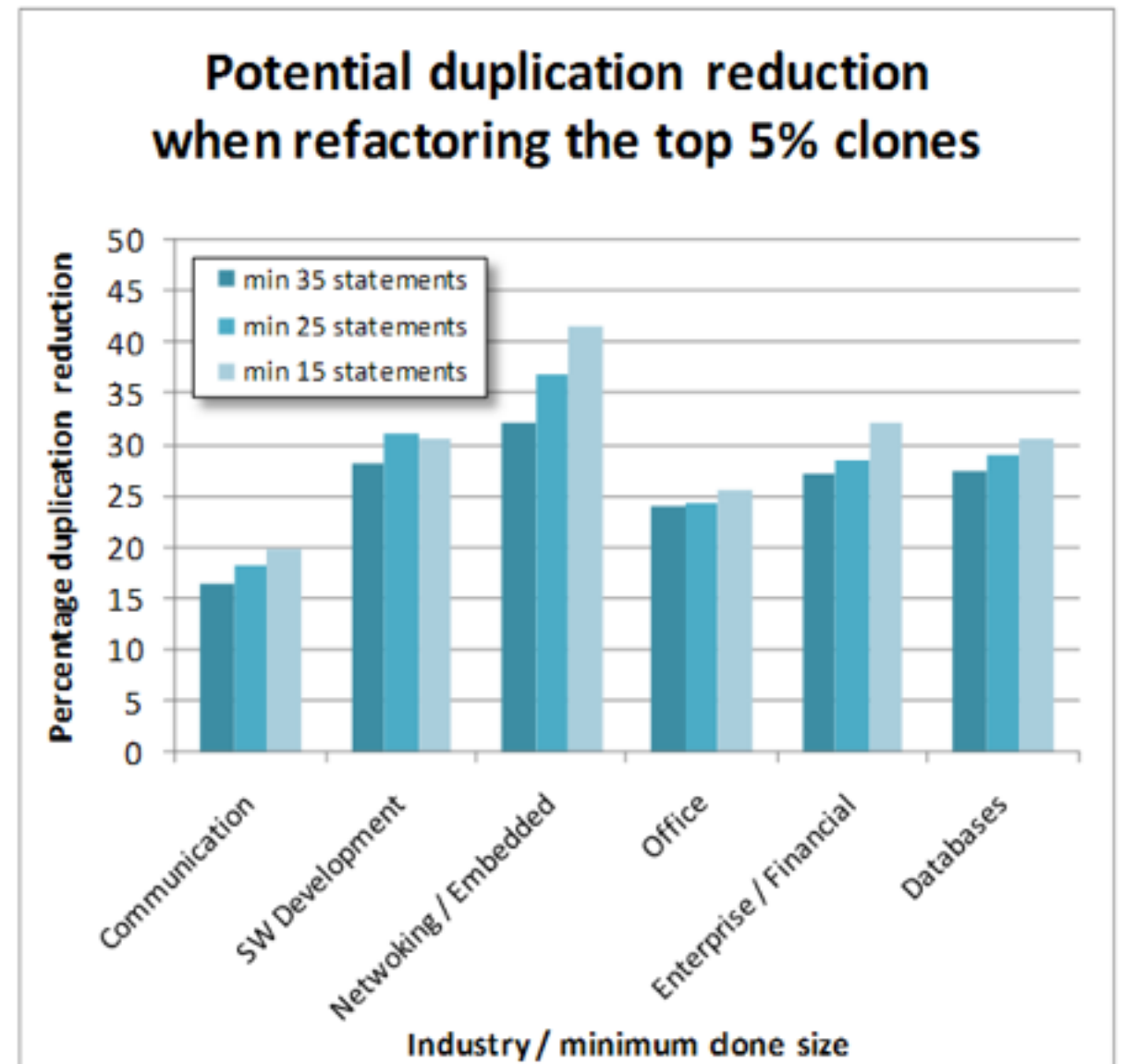
Enklare att utöka med ny funktionalitet

- 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)

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
- Uppprepningar
- Långa metoder
- Komplicerade villkorssatser
- Switchsatser
- Stora klasser
- Divergerande förändringar
- Shotgun-surgery
- Kodkommentarer

Typiska dåliga lukter

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

Intra-klass-lukt

- Divergerande förändringar
- Shotgun-surgery
- Kodkommentarer

Inter-klass-lukt

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);
                }
            }
        }
    }
}
```


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);
                }
            }
        }
    }
}
```

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);
    }
}
```

- ✓ *Uppprepningar*
- ✓ *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

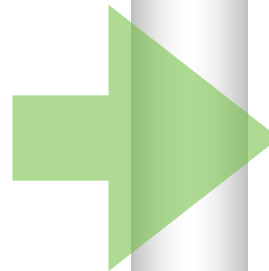
1. Se till att alla tester passerar (alla?)
2. Identifiera vad som luktar illa
3. Gör en plan för hur koden skall refaktoreras
4. Genomför planen
5. Kör alla tester för att se till att inga förändringar/buggar/etc. smög in
6. Gå till 1.

Refaktoreringsmönster [\[refactoring.com/catalogue\]](http://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

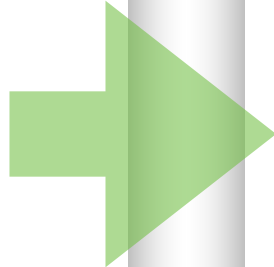
```
Method setup = null;  
for (Method m : c.getMethods()) {  
    if (m.getName().equals("setup")) { ... }  
}  
  
Method tearDown = null;  
for (Method m : c.getMethods()) {  
    if (m.getName().equals("tearDown")) { ... }  
}
```



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

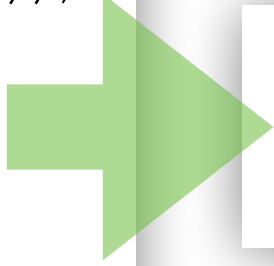
Extract Method

```
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")) { ... }  
}
```

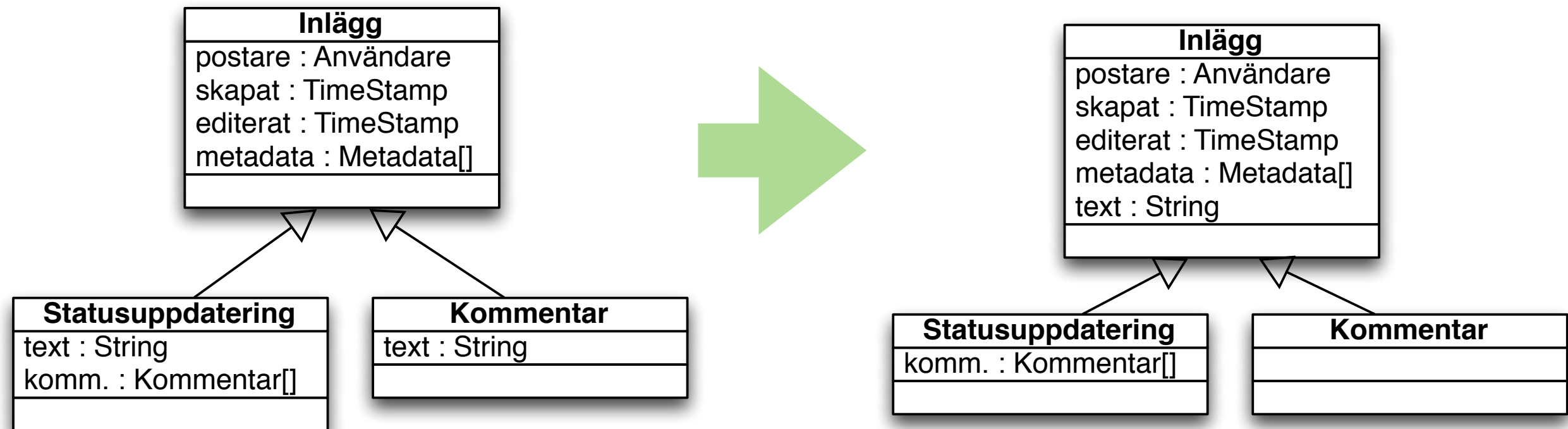
```
Method s = findSetupMethod(c.getMethods());  
  
Method tearDown = null;  
for (Method m : c.getMethods()) {  
    if (m.getName().equals("td")) { ... }  
}
```

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

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

”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 refaktorerera 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 uppmuntrar er till att göra mer komplicerad refaktorering för högre vinster

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

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

