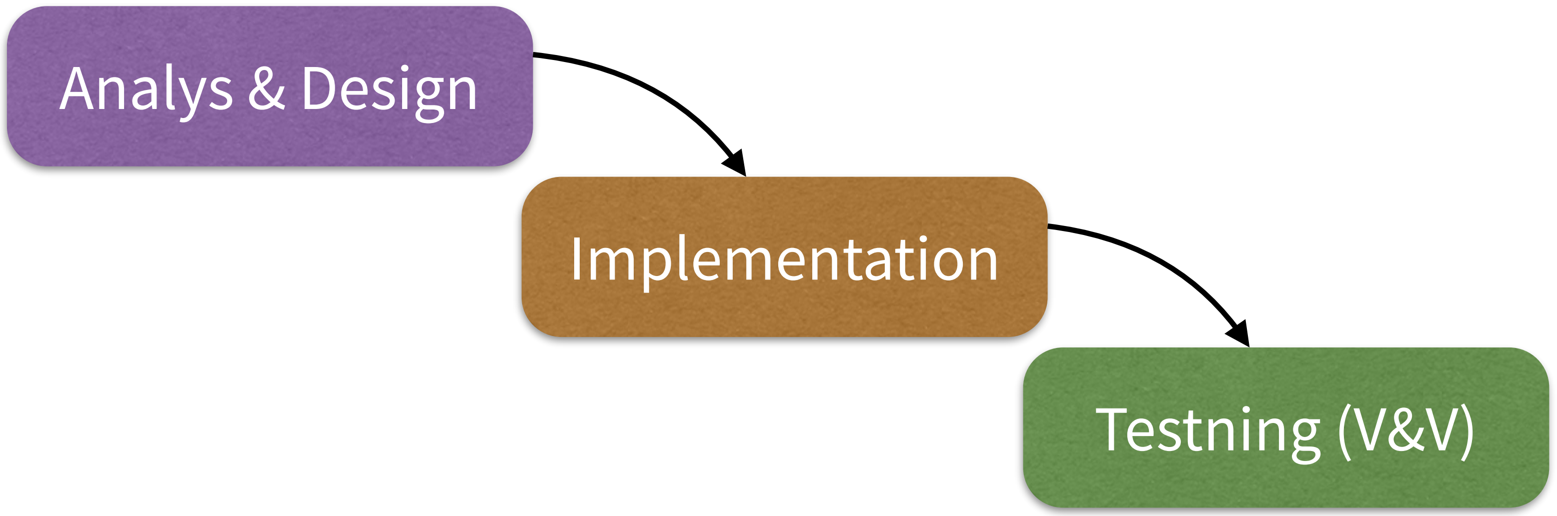
Lecture 23

Andreina Francisco
(based on slides by Tobias Wrigstad)

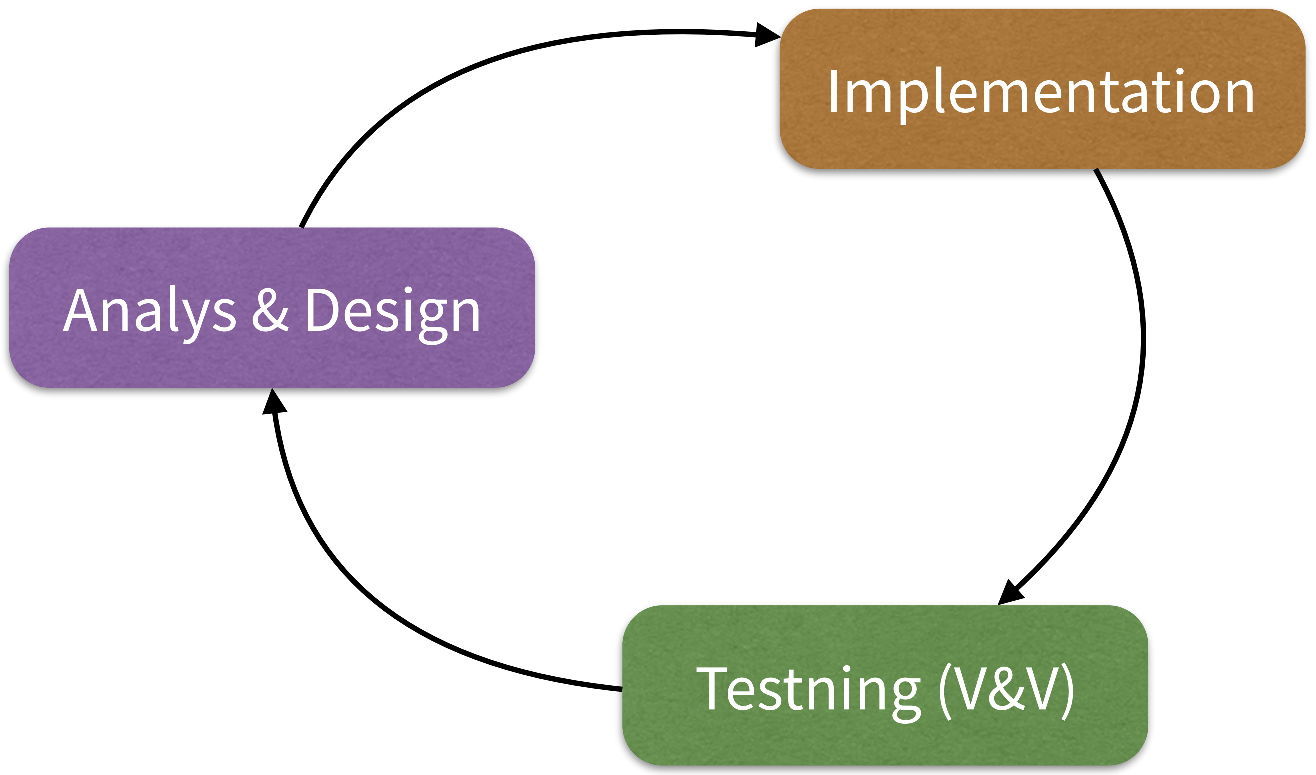
Refactoring

Traditional View of System Development



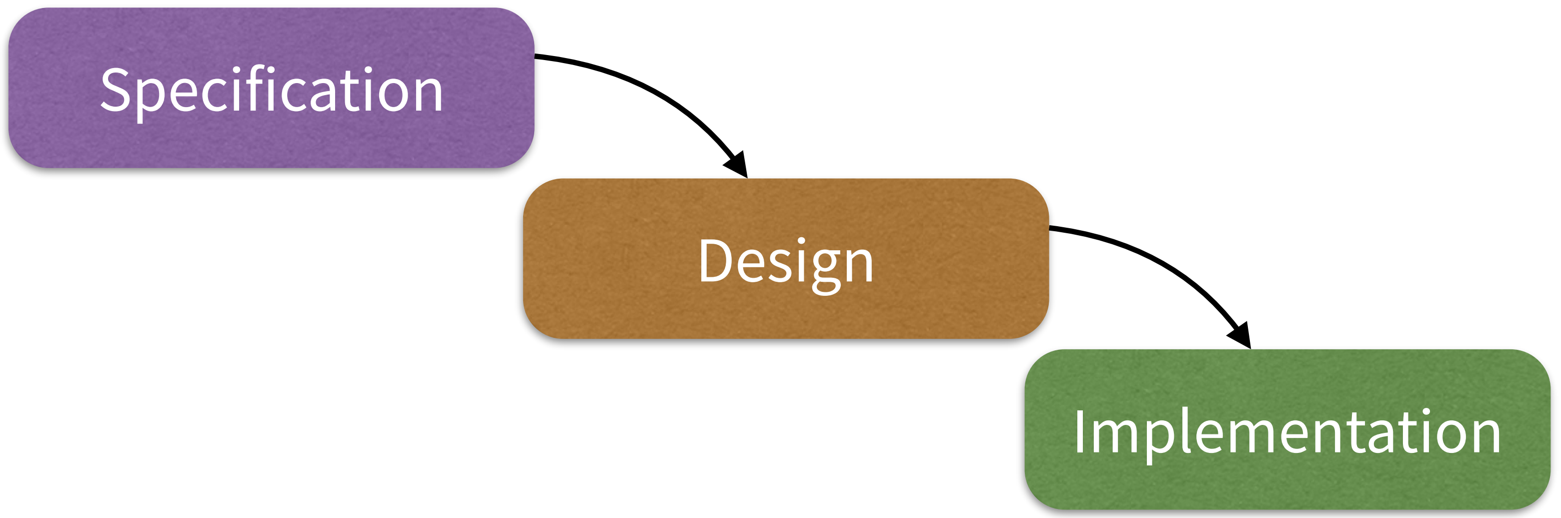
- The so-called "Waterfall model"
- Discrete steps that form a pipeline - each step provides input to the next step as output

Modern View of System Development



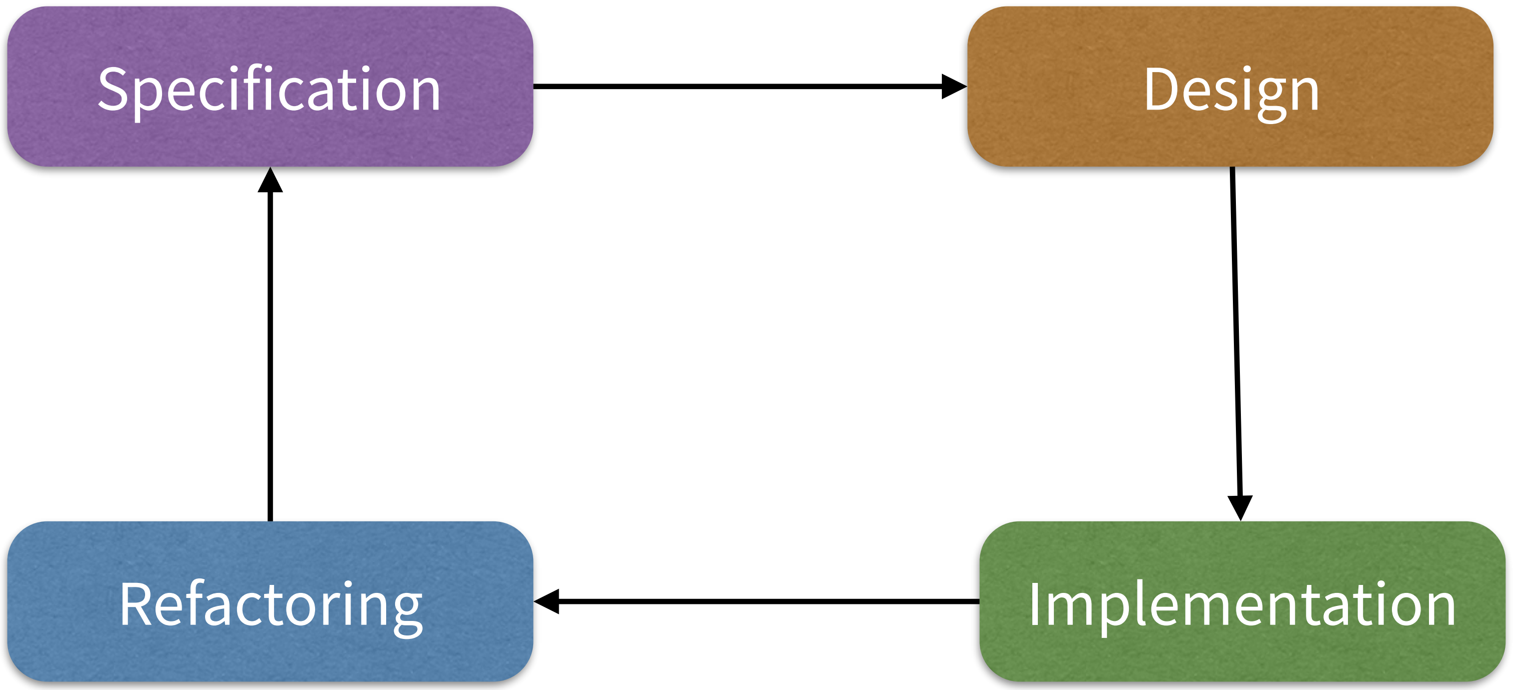
- Development is an iterative process - usually unprofitable / suboptimal to try to understand the problem before starting to implement
- Instead, use the implementation to drive understanding
- Continuous validation & verification - facilitated e.g. by always having a running system

Naive View of Implementation



- Same problem as the waterfall model — impossible to consider everything evenly
- Correctness is just one of many quality attributes (eg maintainable, readable, …)
- Time pressure and the like sometimes cause programmers to write substandard code
- Code is produced in a collective process — sometimes leads to e.g. duplications
- Process steps needed to gradually refine and correct — *refactoring*

Implementation Cycle



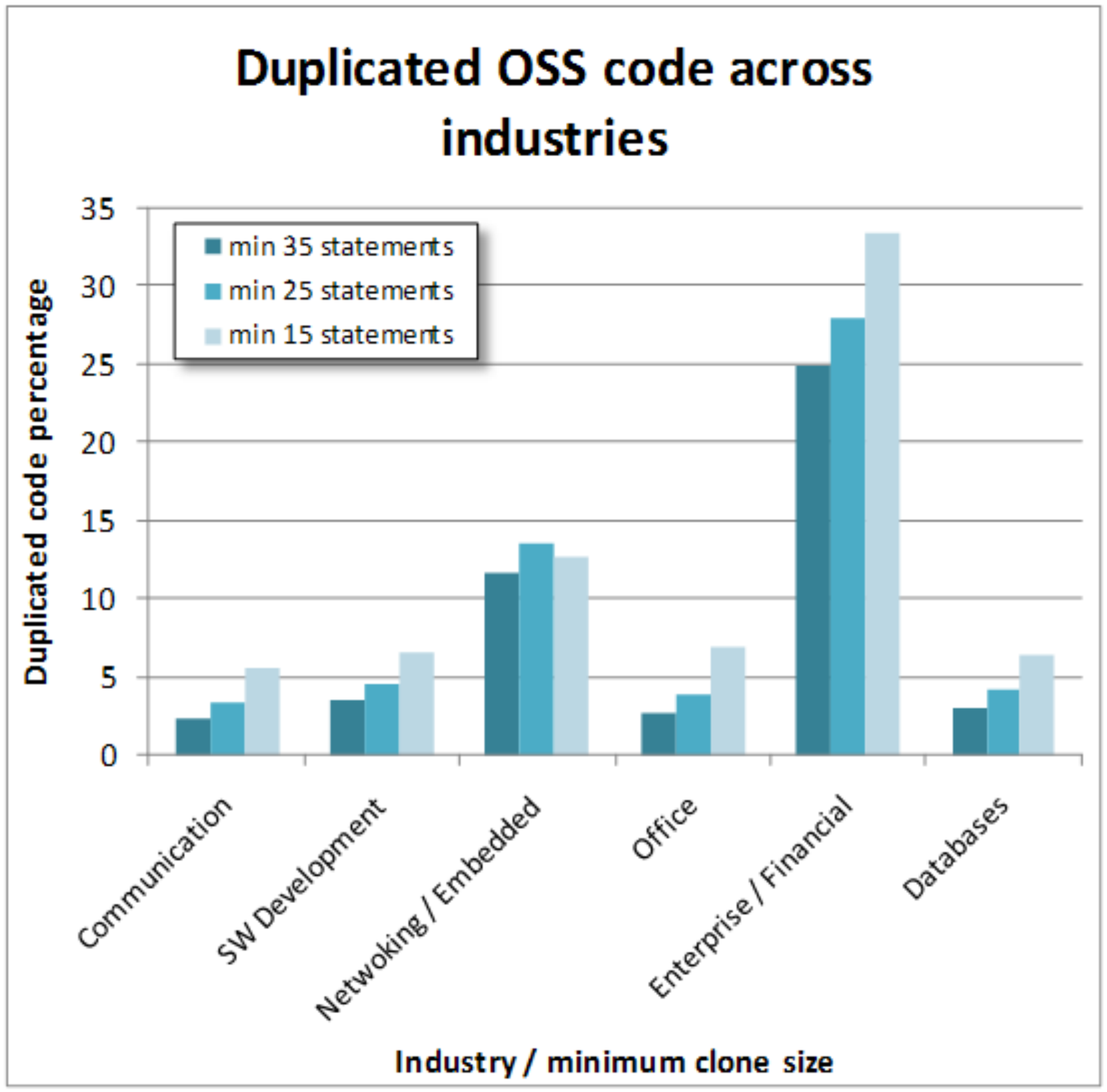
- New step — refactoring
- Changes to the code in order to make the code better without affecting what it does, or the performance of the system

    Change the structure of the code!!

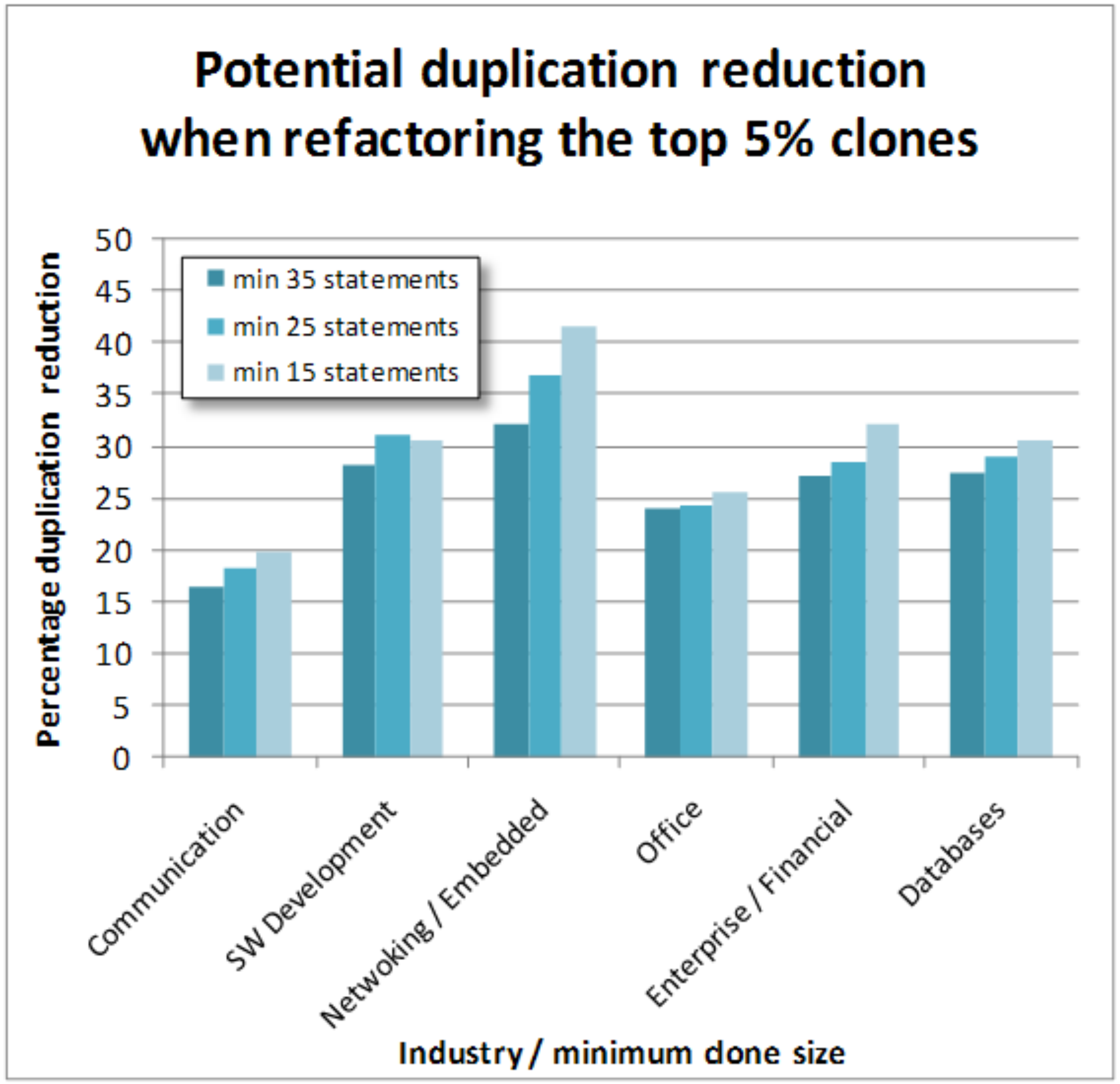
Refactoring

- The purpose of refactoring is to make code and design:
    - More maintainable
    - Easier to understand
    - Easier to change
    - Easier to expand with new functionality
- The term was popularised by Martin Fowler, see book Refactoring (AW, 1997)
- Fowler manages to capture many refactoring processes in named patterns
- Traditionally not always allowed in all workplaces (“if it ain’t broke, don’t fix it”)

Example: Code Duplication



a)



b)

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

Example: Maintenance

It is usually easier to maintain code that you have written yourself

- Easier to understand, follow your normal thinking paths
- Less respect for the code

The bulk of all system is maintenance of existing systems

- Ie maintenance of code that you have **not** written yourself

ERGO:

- Everyone should strive to write code that is as simple as possible to understand

“Bad smells” and rotting code

Code tends to rot over time

Many modifications under time pressure, with different mental models, etc.

We say that rotting code smells bad — “bad smells”

As a developer, our goal is to identify code that smells bad and clean it up

What gives off bad odour?

A **recognisable indicator** that something in the code may be wrong

All code can rot - even test code (not just production code)

Typical Bad Smells

- Magic constants
- Duplication
- Long methods
- Complicated conditionals
- Switch statements
- Large classes
- Divergent change
- Shotgun surgery
- Code comments

Typical Bad Smells

- Magic constants
- Duplication
- Long methods
- Complicated conditionals
- Switch statements
- Large classes

Intra-class-odour

- Divergent change
- Shotgun surgery
- Code comments

Inter-class-odour

Duplication

```
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);
                }
            }
        }
```

Duplication

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

Magic Constants

```java
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);
                }
```

Magic Constants

```
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);
                }
```

Long Methods

```
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);
                }
            }
        }
```

When is a Method TOO Long?

- Nested control structures with depths greater than 2
- Takes many parameters that radically change how the method behaves
- When its logic is duplicated in other methods
- Unnecessary noise, e.g. comments that are obvious, convenience methods not used, etc.
- It does not fit on a screen!!
- When the reader does not get an immediate and intuitive understanding of what it is doing
- …

Refactored Program [partially]

```
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);
    }
}
```

- ✅ *Duplication*
- ✅ *Long methods*
- ✅ *Magic constants*

Refactoring Patterns

- Code refactoring is a code transformation performed manually or with tool support

$$\mathit{code} \Rightarrow \mathit{code}$$

- Must be applied continuously, not just once a month
- A functioning set of tests is of great importance to reduce the risks associated with complex refactoring

The Refactoring Process

1. Make sure all tests pass
2. Identify what smells bad
3. Make a plan for how the code should be refactored
4. Implement the plan
5. Run all tests to make sure no changes / bugs / etc. sneaked in
6. Go to 1.

Refactoring Patterns [refactoring.com/catalog]

- 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());
```

So We Benefit from the Better Structure

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

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

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

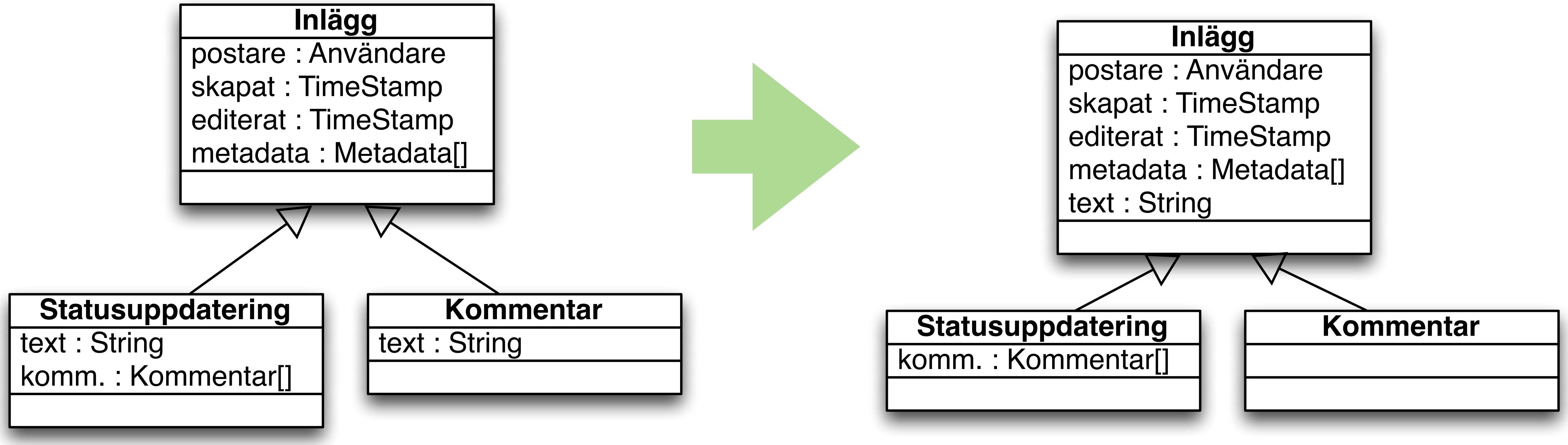
```java
Method s = findSetupMethod(c.getMethods());

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

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

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

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

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

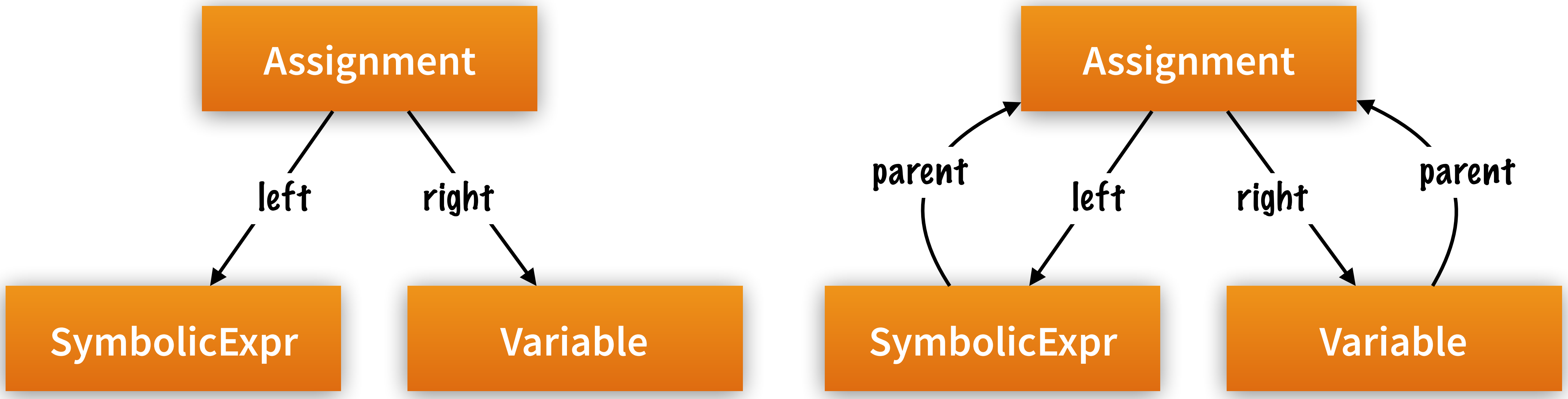
```java
Method s = findMethod(c.getMethods(), SETUP);

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

Pull Field Up



- This type of code can occur e.g. because different programmers have worked in parallel, or because there was a difference between Status Update and Comments previously but which no longer applies, etc.

Change Unidirectional Association to Bidirectional

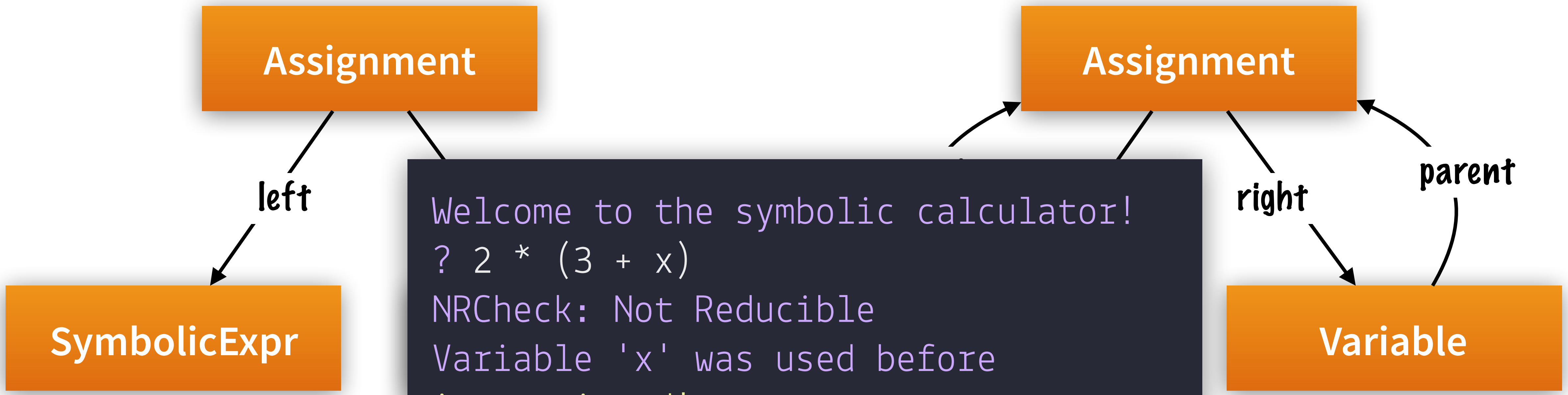


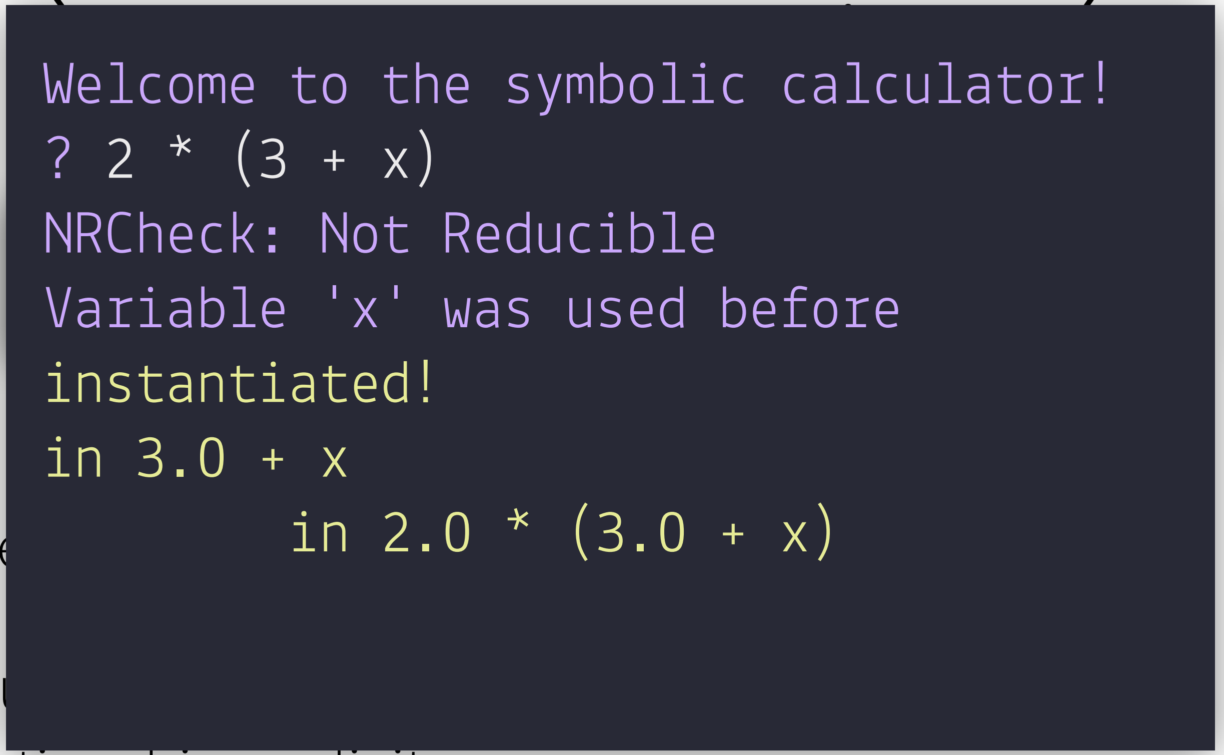
- Originally, only one direction was needed, now both are needed

  Can be calculated, but if the program often needs both directions, it can be a good idea to make the relationship explicit

  Solution in assignment 3: add `SymbolicExpression parent;` to the root class

Change Unidirectional Association to Bidirectional





```
Welcome to the symbolic calculator!
? 2 * (3 + x)
NRCheck: Not Reducible
Variable 'x' was used before
instantiated!
in 3.0 + x
        in 2.0 * (3.0 + x)
```

- Originally, only one dire

  Can be calculated, bu can be a good idea to make the relationship explicit

  Solution in assignment 3: add `SymbolicExpression parent;` to the root class

Single Responsibility Principle [Robert C. Martin]

- Each module (class) is responsible for part of the program's function
- Responsibility is encapsulated in the module (class)
- “There should never be more than one reason for a class to change”

- In assignment 3 — the traversing pattern (visitor interface)

  Eliminate the evaluation function from the AST tree

  Replace with a general principle to traverse a tree

  Allows us to build several different operations for a tree (eval, check, etc.)

“Rule of Three”

- The first time we do something - just do it
- The second time we need to do almost the same thing - copy it
- The third time we need to do almost the same thing - time to refactor!!

Typical Bad Smells

- Magic constants
- Duplication
- Long methods
- Complicated conditionals
- Switch statements
- Large classes

Intra-class-odour

- Divergent change
- Shotgun surgery
- Code comments

Inter-class-odour

Typical Bad Smells

- Magic constants
- Duplication
- Long methods
- Complicated conditionals
- Switch statements
- Large classes
- Divergent change
- Shotgun surgery
- Code comments
- Feature Envy
- Inappropriate intimacy
- Lazy class / freeloader
- Too many parameters
- Excessively long line of code (or God Line)
- …

Finishing, Refactoring

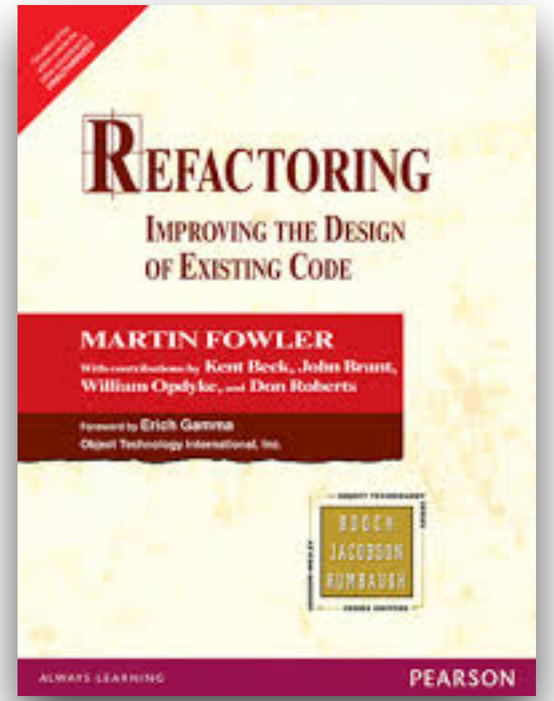
- Refactoring is necessary to prevent a program from messing up and having to be rewritten from scratch
- Several of you have experienced this during the course “the hard way”

  Refactoring can help, but maybe it’s too much to do for you right now

  The SIMPLE method encourages continuous (trivial) refactoring - now I want to encourage you to do more complicated refactoring for higher profits
- Assignment 4 starts with testing and then refactoring…

  … and then extensions!
- Look at refactoring.com/catalog to read about different designs

  You can learn a lot about programming by doing so



Assignment 4a — Sneak Preview

- Extension → Refactoring → Testing
- Expand the calculator to be more like a little interpreted programming language
  - Integer and floating point
  - Do not assign variables more than once
  - features
  - Conditional statements
- Test-driven development