Chapter 14: Refactoring
(Continuously) Improving the Design of Existing Code

Object-Oriented
Software Construction

Armin B. Cremers, Tobias Rho, Daniel Speicher, Holger Mügge
(based on Bruegge & Dutoit)

Outline

- Preface, Some Small Talk
- Safe changes to the design
- Clean code and good design
- Improving the design of existing code
  - Unmaintainable Classes: Stable but only specific
  - Low Cohesion, Identity Disharmonies, Tangling
  - High Coupling, Collaboration Disharmonies, Scattering
  - Classification Disharmonies
  - Useless Classes: Abstract but unstable
- Review

What‘s the responsibility of this class?

```
Class A {

  B<C> d;

  void e(
     F g,
     F h,
     int i) {
    ...
  }

  C j(
     int k) {
    ...
  }

}
```

What‘s the responsibility of this class?

```
Class A {

  B<C> d;

  void e(
     F g,
     F h,
     int i) {
    ...
  }

  C j(
     int k) {
    ...
  }

}
```

```
Class Library {

  List<Book> books;

  void addBook(
     String name,
     String author,
     int bookNo) {
    ...
  }

  Book getBook(
     int bookNo){
    ...
  }

}
```

```
Class PartsDepot {

  List<Part> parts;

  void addPart(
     String name,
     String manufacturer,
     int partNo) {
    ...
  }

  Part getPart(
     int partNo) {
    ...
  }

}
```

What does this method?

```
public static int s(int[] y, int x)
{
  int a = 0;
  int b = y.length-1;

  while(true) {
    int i = (a + b)/2;

    if (y[i] == x)
      return i;

    if (y[index] < x)
      a = i+1;
    else
      b = i-1;

    if (a > b) break;
  }

  return -1;
}
```

What does this method?

```
public static int binarySearch(int[] numbers, long searchedNumber)
{
  int start = 0;
  int end = numbers.length-1;

  while(true) {
    int middle = (start + end)/2;

    if (numbers[index] == searchedNumber)
      return index;

    if (numbers[index] < searchedNumber)
      start = middle+1;
    else
       end = middle-1;

    if (start > end) break;
  }

  return -1;
}
```

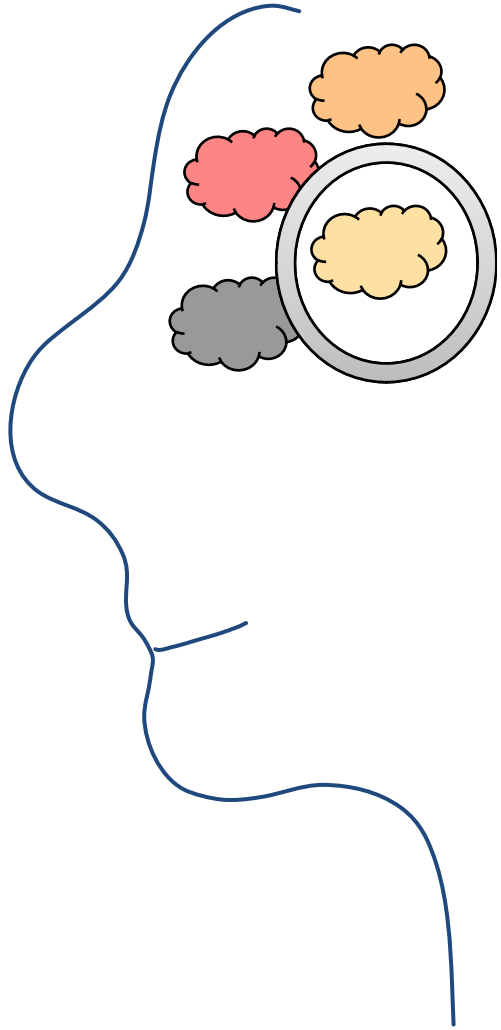
Clean Code! Naming is essential!

**Clean code doesn‘t make your code correct,
but it makes it much more difficult for a bug to hide.**

Bad Signs of Rotting Design, [M96]

- Rigidity
  - Code difficult to change
  - Management reluctance to change anything becomes policy
- Fragility
  - Even small changes can cause cascading effects
  - Code breaks in unexpected places
- Immobility
  - Code is so tangled that it's impossible to reuse anything
  - a module could be reused in another system, but the effort and risk of separating the module from original environment is too high
- Viscosity
  - Much easier to hack than to preserve original design.

Separation of Concerns



Let me try to explain to you, what to my taste is characteristic for all intelligent thinking. It is, that one is willing ***to study in depth an aspect of one's subject matter in isolation for the sake of its own consistency****, all the time knowing that one is occupying oneself only with one of the aspects.*

*We know that a program must be correct and we can study it from that viewpoint only; we also know that it should be efficient and we can study its efficiency on another day […] But nothing is gained – on the contrary – by tackling these various aspects simultaneously. It is what I sometimes have called "****the separation of concerns****" […]*

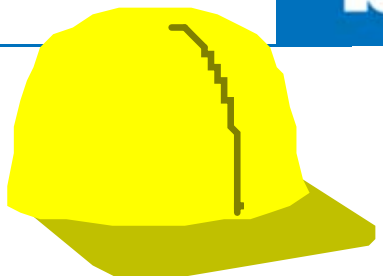
[Edsger W. Dijkstra, "On the role of scientific thought", 1974]

Outline

- Preface, Some Small Talk
- Safe changes to the design
- Clean code and good design
- Improving the design of existing code
  - Unmaintainable Classes: Stable but only specific
  - Low Cohesion, Identity Disharmonies, Tangling
  - High Coupling, Collaboration Disharmonies, Scattering
  - Classification Disharmonies
  - Useless Classes: Abstract but unstable
- Review

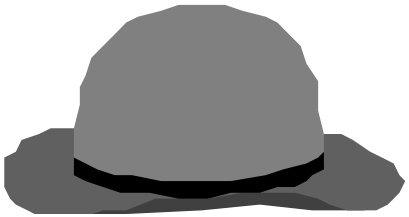
Course : Kent Beck's “Hats-metaphor"

- What is it I want to restructure?
- Is there a test? → Write test!
- Apply refactoring
- Test



Refactoring Hat

Programmer



Function Adding Hat

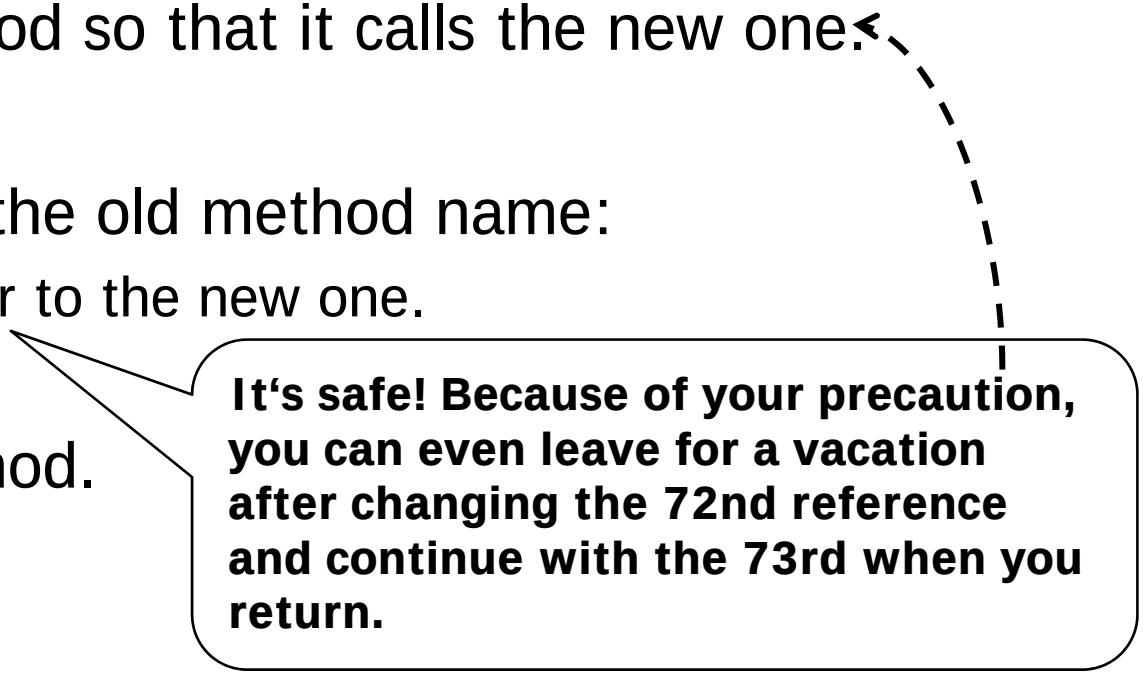
- What is it I want to add?
- Write test and see it fail
- Add functionality
- Test and see the test succeed

Safety: Small steps, Automation

- Fowler‘s book: Excellent guidance to execute Refactorings in small and safe steps.
- Integrated Development Environments like Eclipse offer many automated Refactorings:
  - In depth analysis of the preconditions.
  - Derivation of the required changes.
  - Preview lets the developer check the exected result.
  - Execution of the changes.
  - (Still small bugs every now and then.)
- Good tests are indispensible in both cases.

Example: Rename Method

- [Check for overriden and overrding methods.]
- Declare a new method with the new name.
- Copy the old body of code over to the new method.
- *Compile!*
- Change the old method so that it calls the new one.
- *Compile and test!*
- For all references to the old method name:
  - Change them to refer to the new one.
  - *Compile and test!*
- Remove the old method.
- *Compile and test!*



Automated Refactorings

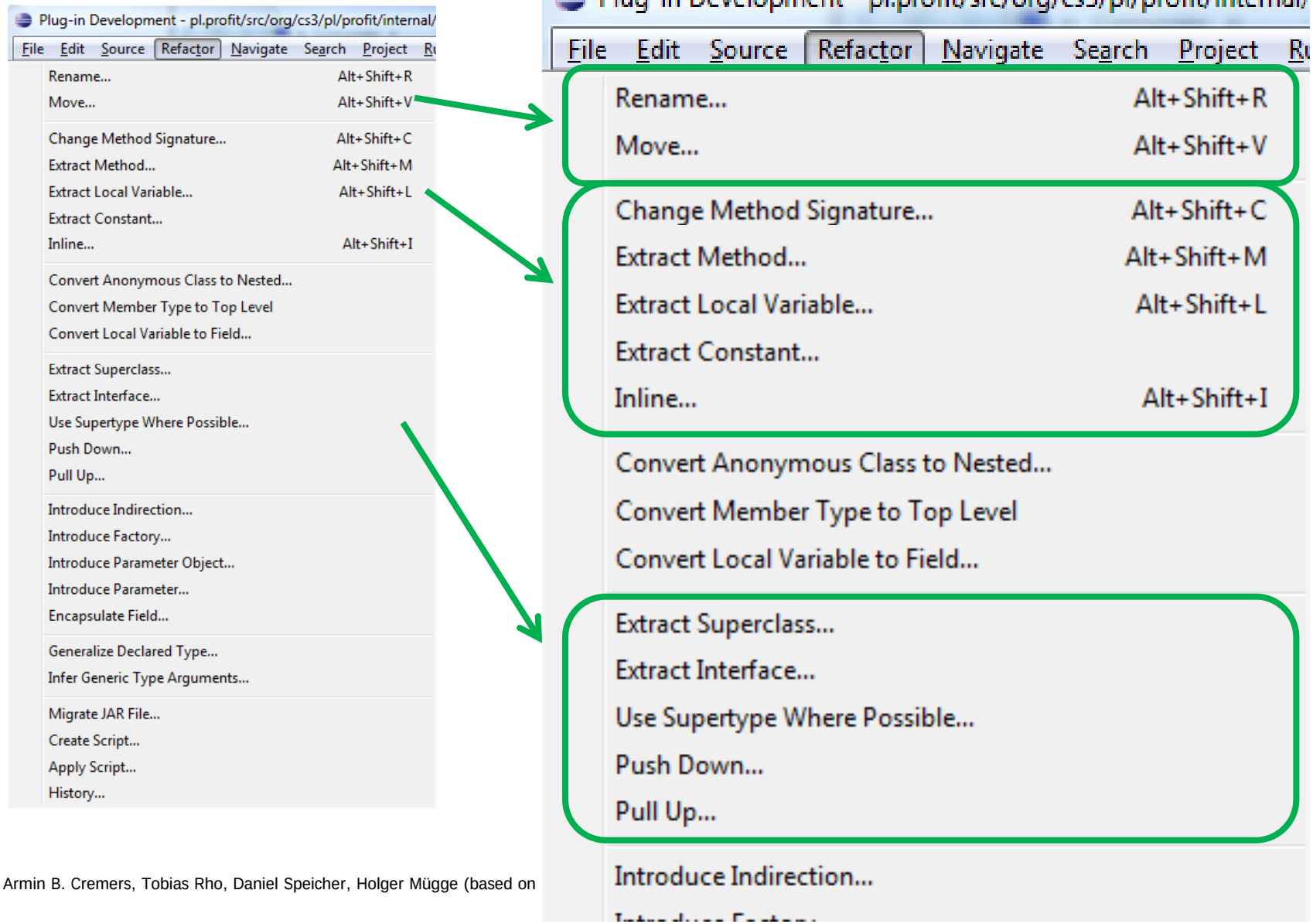
Demo

1: Rename method

2: Extract local variable

- Very fast! Almost reliable.
- You still need the expertise what to do where!
- Even there tool support exists, but can not replace your judgement.

Refactorings in Eclipse (Java)

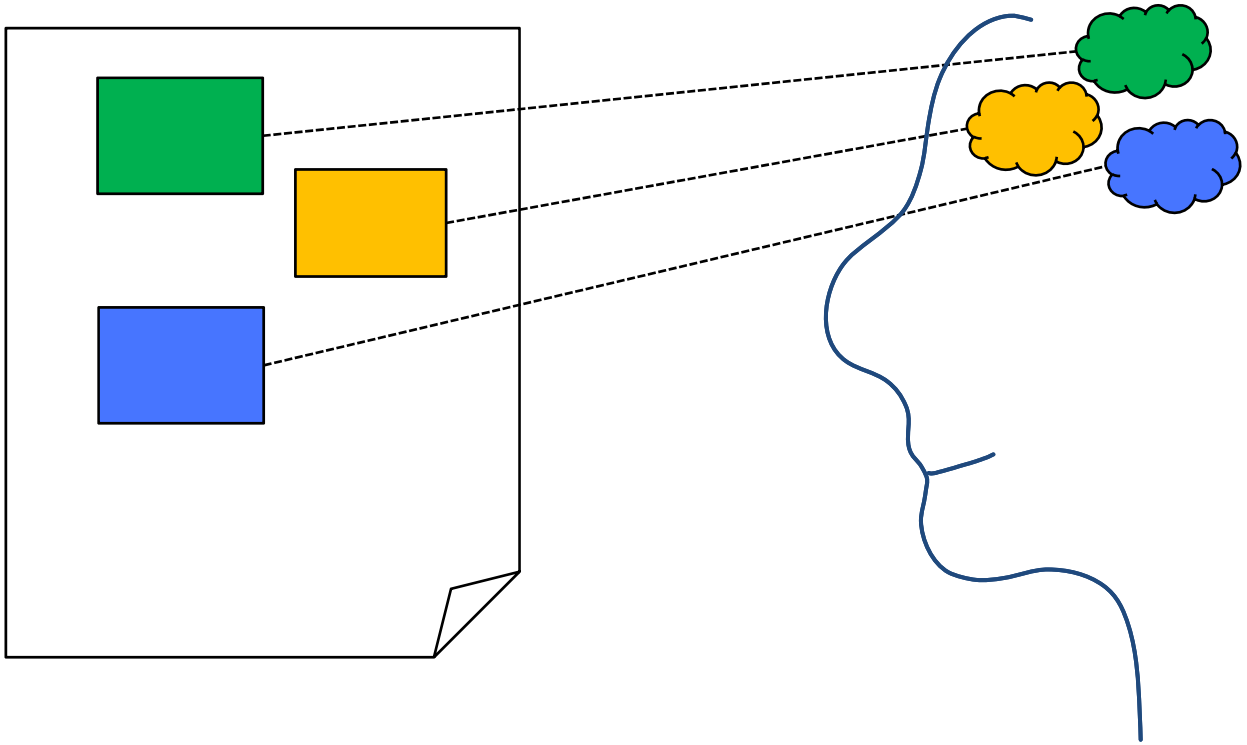


Armin B. Cremers, Tobias Rho, Daniel Speicher, Holger Mügge (based on

Outline

- Preface, Some Small Talk
- Safe changes to the design
- Clean code and good design
- Improving the design of existing code
  - Unmaintainable Classes: Stable but only specific
  - Low Cohesion, Identity Disharmonies, Tangling
  - High Coupling, Collaboration Disharmonies, Scattering
  - Classification Disharmonies
  - Useless Classes: Abstract but unstable
- Review

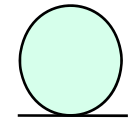
Concepts - Modules in 1-1 Relation



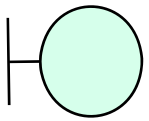
- → Comprehensibility
- ← Natural Implementation
- ← Easy identification of the parts to change

OO based Modularity is natural

- Object = Identity + State + Behavior
- Objects ⇔ Things
- Compounds consist of parts
- Abstraction, classification
  - Substitutability ⇔ Can be used/seen as ...
  - Inheritance ⇔ Is a ... and therefore has/can ...



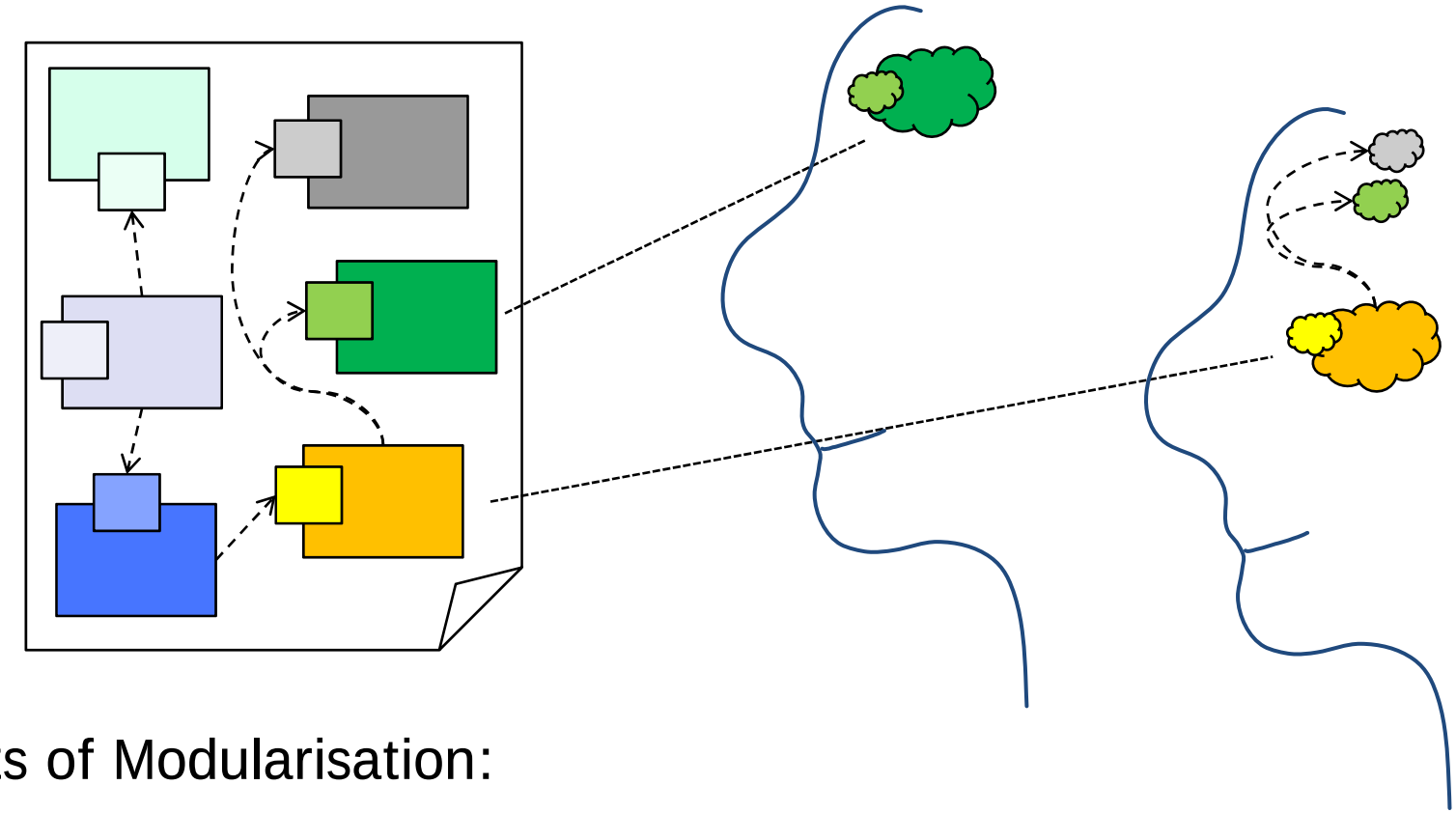
1-1 Modelling of real entities



Programming by combining virtual entities. The core success story of OO: Graphical user interfaces build from “buttons”, “windows”.

- → Classes are a giant step towards the 1-1 ideal

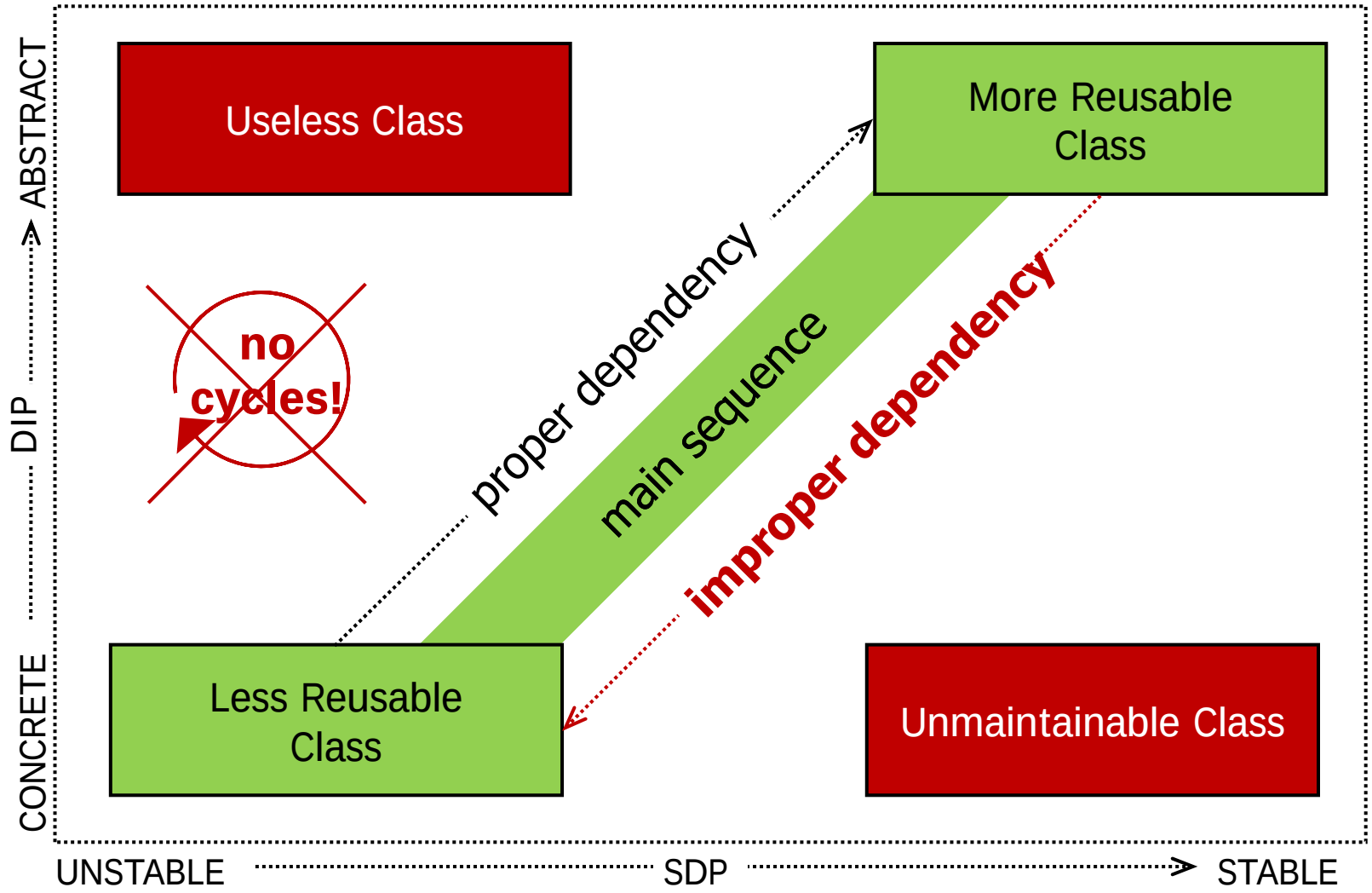
Visible Interfaces & Hidden Details



Benefits of Modularisation:

- Parallel Development + Changability + Comprehensibility
  [David L. Parnas: *On the Criteria to Be Used in Decomposing Systems into Modules*, 1972]
- Reusability of common / abstract Functionality
  [Dependency Inversion Principle: „Depend on abstractions not on concretions.”]

DIP/SDP, [M96]


Useless Class
More Reusable Class
no cycles!
proper dependency
main sequence
improper dependency
Less Reusable Class
Unmaintainable Class
CONCRETE
DIP
ABSTRACT
UNSTABLE
SDP
STABLE

Design Principles, [M96]

- Dependency Inversion Principle (DIP)
  - Depend upon Abstraction, Do NOT Depend upon Concretions.
- Acyclic Dependencies Principle (ADP)
  - The dependency structure for released component must be a Directed Acyclic Graph (DAG). There can be no cycles.
- Stable Dependencies Principle (SDP)
  - The dependencies between components in a design should be in the direction of stability. A component should only depend upon components that are more stable than it is.
- Stable Abstractions Principle (SAP)
  - The abstraction of a package should be proportional to its stability! Packages that are maximally stable should be maximally abstract. Instable packages should be concrete.

Outline

- Preface, Some Small Talk
- Safe changes to the design
- Clean code and good design
- Improving the design of existing code
  - Unmaintainable Classes: Stable but only specific
  - Low Cohesion, Identity Disharmonies, Tangling
  - High Coupling, Collaboration Disharmonies, Scattering
  - Classification Disharmonies
  - Useless Classes: Abstract but unstable
- Review

Outline

- Preface, Some Small Talk
- Safe changes to the design
- Clean code and good design
- Improving the design of existing code
  - Unmaintainable Classes: Stable but only specific
  - Low Cohesion, Identity Disharmonies, Tangling
  - High Coupling, Collaboration Disharmonies, Scattering
  - Classification Disharmonies
  - Useless Classes: Abstract but unstable
- Review

Unmaintainable Classes: Stable but only specific

Smell	Refactoring
Duplicated Code	Extract Method, Extract Class, Hide Delegate, Remove Middle Man
Comments	Extract Method, Introduce Assertion
Switch Statements	Replace Type Code with State/Strategy, Replace Type Code with Subclasses, Introduce Null Object, Replace Conditional with Polymorphism, Replace Parameter with Explicit Methods
Data Class	Move Method, Encapsulate Field, Encapsulate Collection
Data Clumps	Extract Class, Introduce Parameter Object, Preserve Whole Object
Primitve Obsession	Extract Class, Replace Data Value with Object, Replace Type Code with Class (Enumeration), Introduce Parameter Object, Replace Array with Object, Replace Type Code with State/Strategy, Replace Type Code with Subclasses
Long Parameter List	Introduce Parameter Object, Preserve Whole Object, Replace Parameter with Method
Long Method	Extract Subclass, Decompose Conditional, Replace Inheritance with Delegation, Replace Temp with Query
Large Class	Extract Class, Extract Subclass, Extract Interface, Duplicate Observed Data

Smell: Duplication

- The cost of code cloning:
- If you have to change the code of one clone, you probably have to change the code of every clone.
- Once you miss one clone and somebody else works on that code, it is not clear, which variant of the code is the correct one.

Smell: Comments

- Comments are good, aren't they?
- They are good: Document your public interface with JavaDoc (or similar tool for the language of your choice).

- In of method bodies comments smell.
- Writing comments might be an excuse for not writing clean code.
- Comments tend to become outdated.

- Comments that explain why you did something in a different than the obvious way are of course necessary. ("What" in body = smell, "Why" if necessary o.k.)

Extract Method

- Indication: Cohesive set of statements.
- How to handle? Extract in to a well-named method, replace statements with call to the method.

```
void printOwing(double amount) {
   printBanner();

   // print details
   System.out.println(“name“+_name);
   System.out.println(“amount“+ amount);
}
```

```
void printOwing (double amount) {
   printBanner();
   printDetails(amount);
}

void printDetails (double amount) {
   System.out.println (“name“+_name);
   System.out.println (“amount“+ amount);
}
```

Steps (1/2)

- Defining new and well-named methods
  - always „private“
- Copy the Code
- Searching for the local variables in extracted code
  - Variables that will be used just in new methods
    - local variables of new methods
  - Variables which are changed in new methods and will be used in old methods
    - If only one: give it back as the result of new method
    - more than one: Parts that can not be extracted!
      („Replace Temp with Query“ or try to „Split Temp Variable“)
  - Variables that will be read in new methods
    - Parameters of new methods

Steps (2/2)

- Compile
- In original method
  - Replace the extracted code with a call of the new method
  - Delete the declaration of local variables which have no use anymore
- Compile
- Test

Example: no local variables

```
void printOwing(double amount) {

   Enumeration e = orders.elements();
   double outstanding = 0.0;

   // print banner
   System.out.println("**********************");
   System.out.println("*** Customer owes ****");
   System.out.println("**********************");

   // calculate outstanding
   while (e.hasMoreElements()) {
      Order each = (Order) e.nextElement();
      outstanding += each.getAmount();

   // print details
   System.out.println(“name“+ _name);
   System.out.println(“amount“+ outstanding);
}
```

➔ Extraction of code for print banner

Example: no local variables

```
void printOwing(double amount) {

   Enumeration e = orders.elements();
   double outstanding = 0.0;

   printBanner();



   // calculate outstanding
   while (e.hasMoreElements()) {
      Order each = (Order) e.nextElement();
      outstanding += each.getAmount();

   // print details
   System.out.println(“name“+ _name);
   System.out.println(“amount“+ outstanding);
}
```

```
private void printBanner() {
   System.out.println("**********************");
   System.out.println("*** Customer owes ****");
   System.out.println("**********************");
}
```

- Extraction of code for print banner
  - Unaltered local variable: `outstanding`

Local variable which is not altered

```
void printOwing(double amount) {

   Enumeration e = orders.elements();
   double outstanding = 0.0;

   printBanner();




   // calculate outstanding
   while (e.hasMoreElements()) {
      Order each = (Order) e.nextElement();
      outstanding += each.getAmount();

   printDetails(outstanding);


}
```

```
private void printDetails(double outstanding) {
   System.out.println("name"+ _name);
   System.out.println("amount"+ outstanding);
}
```

- Extraction of codes for calculation
  - Local variable , which is altered and finally used: `outstanding`
  - Local variable , which is altered and will not be used anymore: `e`

Local variable that will be altered

```
void printOwing(double amount) {

   Enumeration e = orders.elements();
   double outstanding = 0.0;

   printBanner();




   outstanding = getOutstanding();



   printDetails(outstanding);


}
```

```
private double getOutstanding() {
   Enumeration e = orders.elements();
   double outstanding = 0.0;
   while (e.hasMoreElements()) {
      Order each = (Order) e.nextElement();
      outstanding += each.getAmount();
   }
   return outstanding;
}
```

- Extracting code for calculation
  - If local variable is assigned before in original method

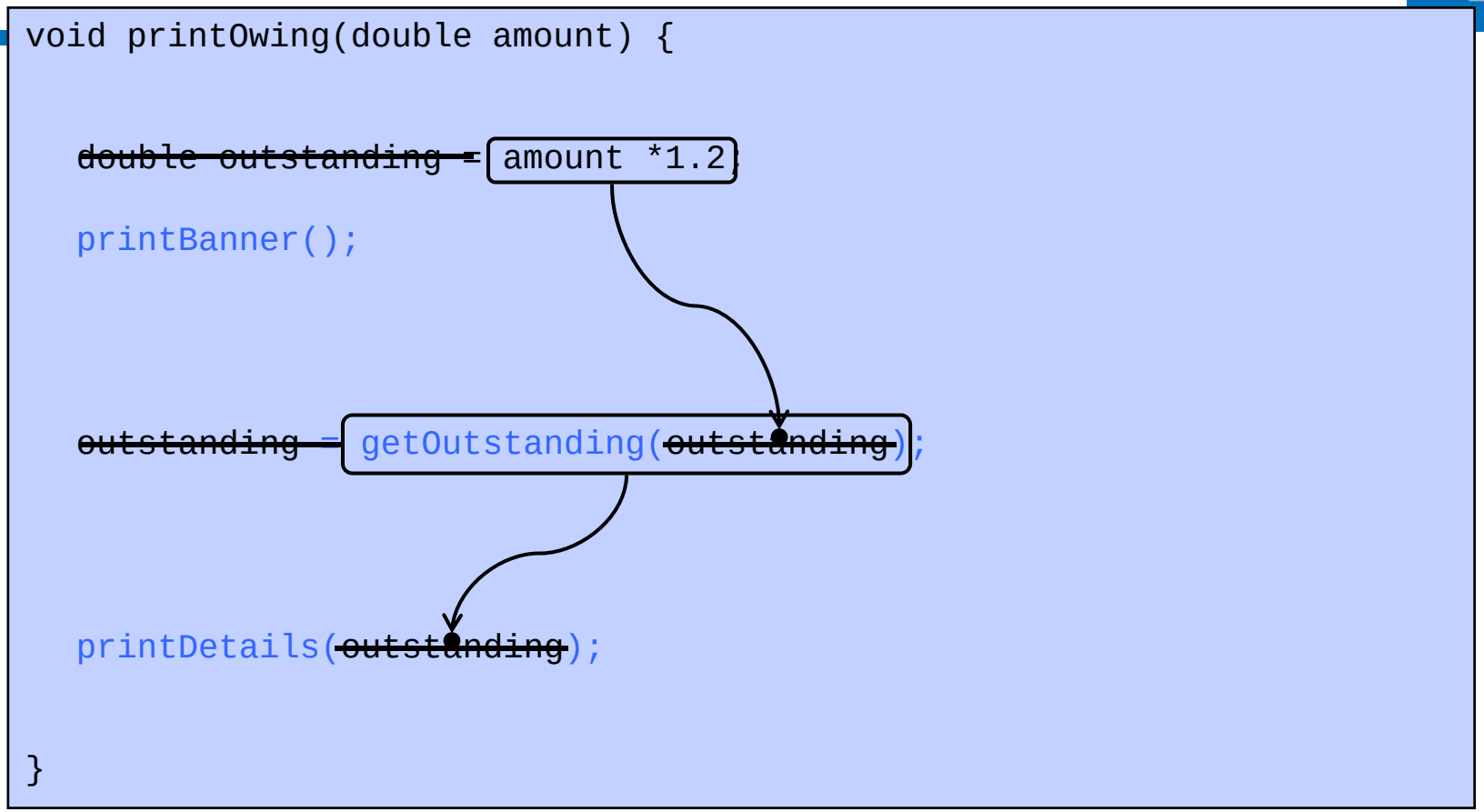
Example : local variable altered before

```
void printOwing(double amount) {


   double outstanding = amount *1.2;

   printBanner();




   outstanding = getOutstanding(outstanding);



   printDetails(outstanding);


}
```

```
private double getOutstanding(double startValue) {
   Enumeration e = orders.elements();
   double result = startValue;
   while (e.hasMoreElements()) {
      Order each = (Order) e.nextElement();
      result += each.getAmount();
   }
   return result;
}
```

- ➔ Now are more Rafactorings possible
  - ◆ twice „inline temp“ for assigning to local variable `outstanding`
  - ➔ In this way we can eliminate `outstanding` from `printOwing`-Method

Example: Elimination of „outstanding“



Example: Result

```
private double getOutstanding(double startValue) {
   Enumeration e = orders.elements();
   double result = startValue;
   while (e.hasMoreElements()) {
      Order each = (Order) e.nextElement();
      result += each.getAmount();
   }
   return result;
}

private void printBanner() {
   System.out.println("**********************");
   System.out.println("*** Customer owes ****");
   System.out.println("**********************");
}

private void printDetails(double outstanding) {
   System.out.println(“name“+ _name);
   System.out.println(“amount“+ outstanding);
}

void printOwing(double amount) {
   printBanner();
   printDetails(getOutstanding(amount *1.2));
}
```

Outline

- Preface, Some Small Talk
- Safe changes to the design
- Clean code and good design
- Improving the design of existing code
  - Unmaintainable Classes: Stable but only specific
  - Low Cohesion, Identity Disharmonies, Tangling
  - High Coupling, Collaboration Disharmonies, Scattering
  - Classification Disharmonies
  - Useless Classes: Abstract but unstable
- Review

Low Cohesion, Identity Disharmonies, Tangling

Smell	Refactoring
Feature Envy	Extract Method, Move Field, Move Method
Divergent Change	Extract Class
Temporary Field	Extract Class, Introduce Null Object

- Feature Envy: A method is more interested in features (fields and methods) of another class.
- Divergent Change: The reason a class has to change are too many. A class should have a clear responsibility.
- Temporary Field: A field that is set only in certain circumstances.

Outline

- Preface, Some Small Talk
- Safe changes to the design
- Clean code and good design
- Improving the design of existing code
  - Unmaintainable Classes: Stable but only specific
  - Low Cohesion, Identity Disharmonies, Tangling
  - High Coupling, Collaboration Disharmonies, Scattering
  - Classification Disharmonies
  - Useless Classes: Abstract but unstable
- Review

High Coupling, Collaboration Disharmonies, Scattering

Smell	Refactoring
Inappropriate Intimacy	Move Field, Move Method, Change Bidirectional Association to Undirectional, Hide Delegate, Replace Inheritance with Delegation
Shotgun Surgery	Move Field, Move Method, Inline Class
Message Chains	Hide Delegate

- Inappropriate Intimacy: Too strong mutual dependency on fields and methods of two or more classes.
- Shotgun Surgery: A certain kind of change requires to change the code in many different places. (=Scattering)
- Message Chain: You retrieve a reference to another object, to retrieve another reference to a third object, to ... to finally call a method that really does something.

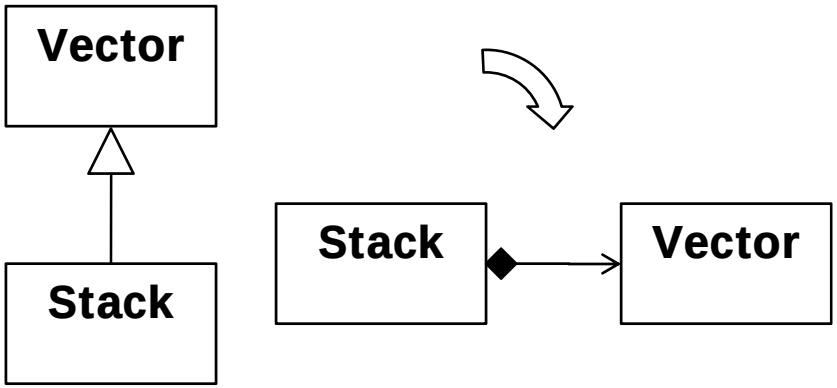
Outline

- Preface, Some Small Talk
- Safe changes to the design
- Clean code and good design
- Improving the design of existing code
  - Unmaintainable Classes: Stable but only specific
  - Low Cohesion, Identity Disharmonies, Tangling
  - High Coupling, Collaboration Disharmonies, Scattering
  - Classification Disharmonies
  - Useless Classes: Abstract but unstable
- Review

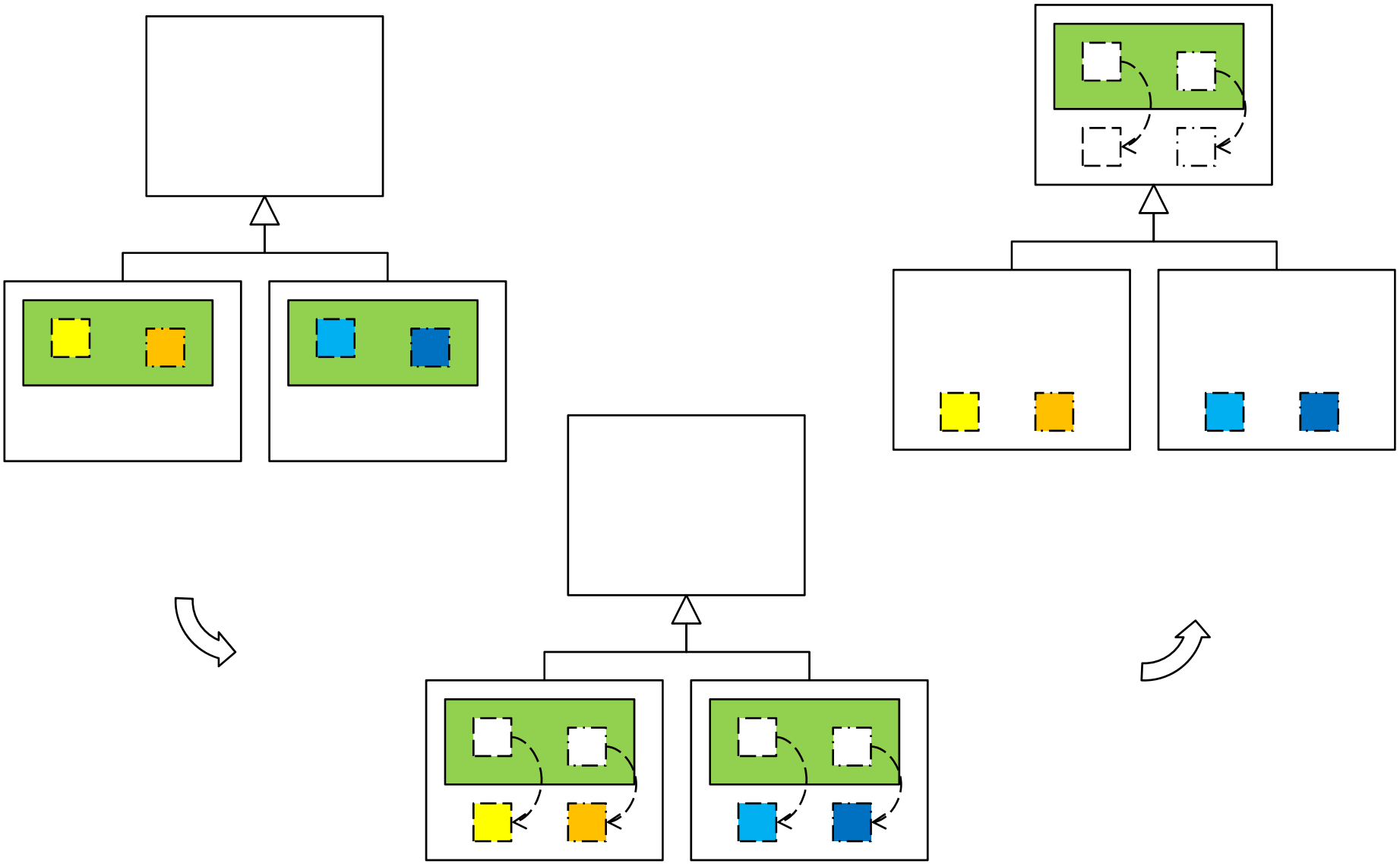
Classification Disharmonies

Smell	Refactoring
Refused Bequest	Replace Inheritance with Delegation
Duplicated Code (in siblings)	Pull Up Method, Form Template Method
Parallel Inheritance Hierarchies	Move Field, Move Method (from classes in one hierarchy to classes in the other)
Alternative Classes with Different Interface	Move Method, Rename Method

Replace Inheritance with Delegation



Form Template Method



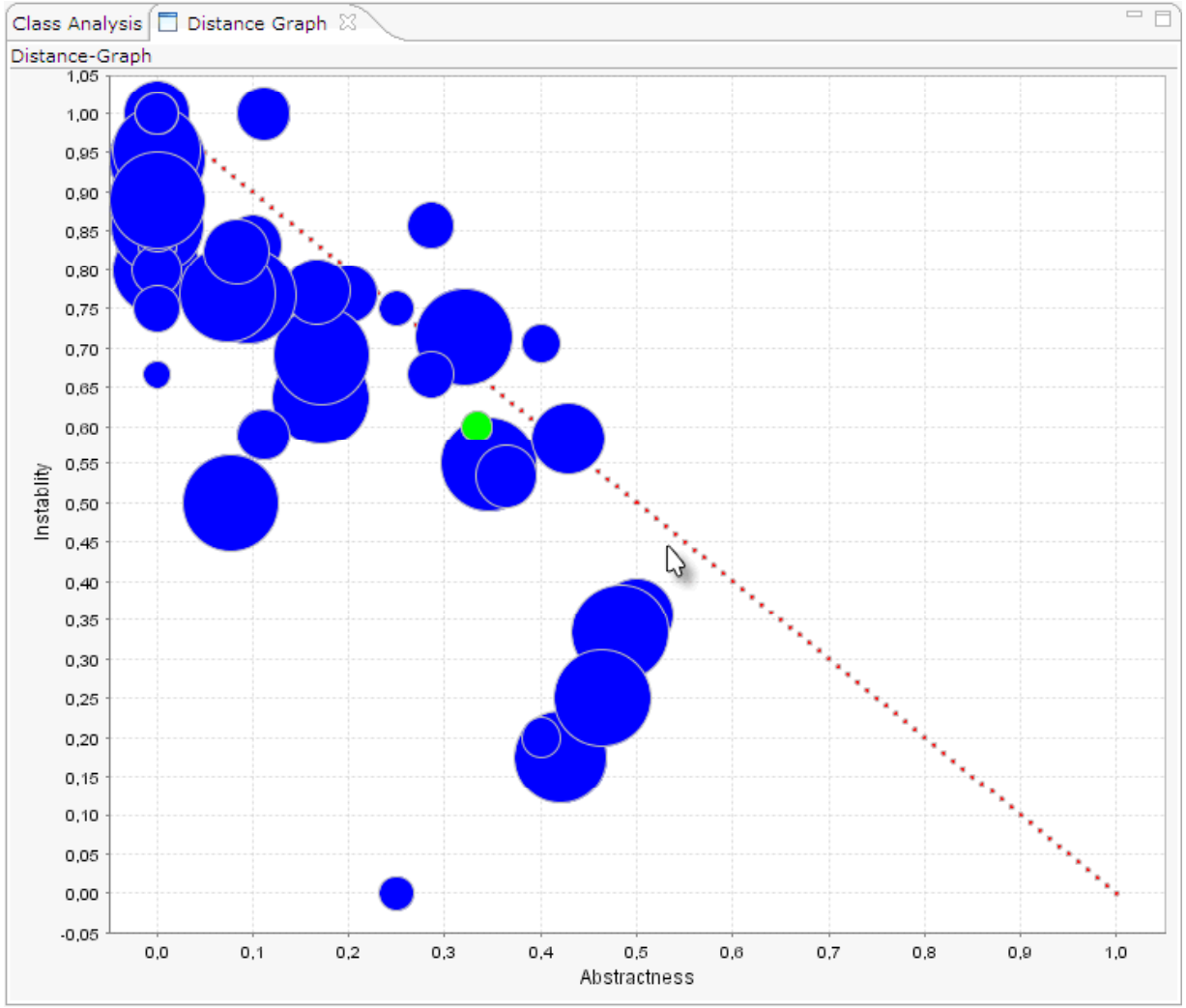
Outline

- Preface, Some Small Talk
- Safe changes to the design
- Clean code and good design
- Improving the design of existing code
  - Unmaintainable Classes: Stable but only specific
  - Low Cohesion, Identity Disharmonies, Tangling
  - High Coupling, Collaboration Disharmonies, Scattering
  - Classification Disharmonies
  - Useless Classes: Abstract but unstable
- Review

Useless Classes: Abstract but unstable

Smell	Refactoring
Speculative Generality	Collapse Hierarchy, Inline Class, Remove Parameter, Rename Class, Rename Field, Rename Interface, Rename Local Variable, Rename Method, Rename Package, Rename Parameter
Middle Man	Inline Method, Remove Middle Man, Replace Delegation with Inheritance
Lazy Class	Collapse Hierarchy, Inline Class

Screenshot: Code Analysis Plugin



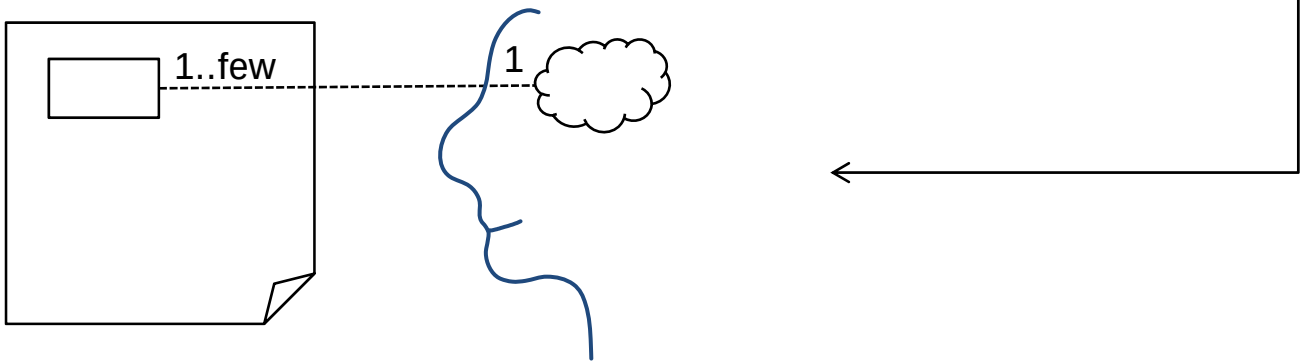
Outline

- Preface, Some Small Talk
- Safe changes to the design
- Clean code and good design
- Improving the design of existing code
  - Unmaintainable Classes: Stable but only specific
  - Low Cohesion, Identity Disharmonies, Tangling
  - High Coupling, Collaboration Disharmonies, Scattering
  - Classification Disharmonies
  - Useless Classes: Abstract but unstable
- Review

Master Craftsman‘s Advice

- Some time ago, Kent Beck offered the following **"rules" for simple design**.
  In priority order, the code must:

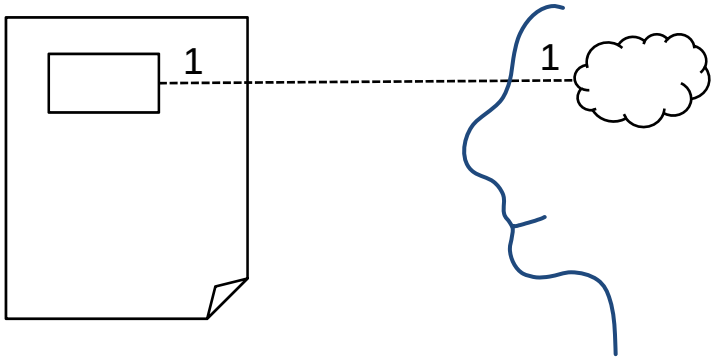
1. **Run all the tests**
2. **Contain no duplicate code**
3. **Express all the ideas the author wants to express**
4. **Minimize classes and methods**



XP: Refactor mercilessly

- **Refactor mercilessly** to keep the design simple as you go and to avoid needless clutter and complexity. Keep your code clean and concise so it is easier to understand, modify, and extend. **Make sure everything is expressed once and only once.** In the end it takes less time to produce a system that is well groomed.

  (Don Wells, http://www.extremeprogramming.org/rules/refactor.html)



Expressing ideas

b-it

- Expressing ideas **gives meaningful names to the classes and methods we have** and to those that are created by the duplication rule. As well, we are often moved to **create new classes and methods** just for improved expression. For example, it is common to **replace a switch statement with a few classes** with a polymorphic method. When we see the **essential idea** that underlies the switch, the classes, and the name of the method, tend to pop right out.

(Ron Jeffries: http://www.xprogramming.com/xpmag/expEmergentDesign.htm)

Literature

- [Fo99] Martin Fowler: Refactoring: Improving the Design of Existing Code (http://www.refactoring.com/catalog)
- M. Lanza, R. Marinescu: Object-Oriented Metrics in Practice, 2006. [LM06]
- [M96] Robert C. Martin:
  Design Principles and Design Patterns
  http://www.objectmentor.com/resources/articles/Principles_and_Patterns.PDF

Some tools for quality assessment

- Metrics – Plugin for Eclipse
  Calculation of common metrics (e.g. LCOM, WMC, etc.)
  http://sourceforge.net/projects/metrics
- Creole – Plugin for Eclipse
  Visualization of the strucure of the software
  http://www.thechiselgroup.org/creole
- CodeCrawler – Freeware (Mainly Smalltalk)
  Visualization of the strucure of the software, Metrics are represented by the shape an color or the rectangles and lines. (Polymetric view)
  (http://sourceforge.net/projects/codecrawler
- CAP - Code Analysis Plugin (http://cap.xore.de/)
- Checkstyle (http://checkstyle.sourceforge.net/)
- Simian (http://www.redhillconsulting.com.au/products/simian/)