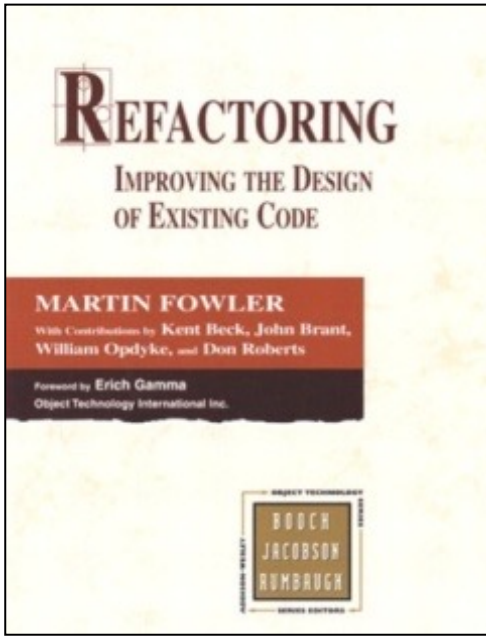
Refactoring

Mariano Ceccato

mariano.ceccato@univr.it

Definition

Fowler's definition
"A change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior"
Martin Flowler, *Refactoring,* page 53



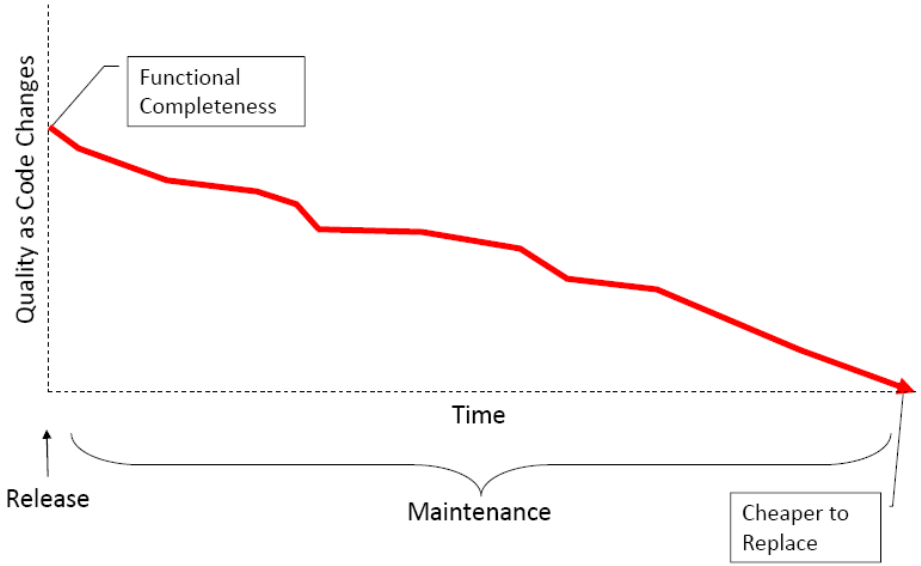
Refactoring – Improving the Design of Existing Code.
Martin Fowler
Addison Wesley, 2000

Refactoring

- **Refactoring** is the process of making improvements to a program to slow down degradation through change.
- You can think of refactoring as *preventative maintenance* that reduces the problems of future change.
- Refactoring involves modifying a program to improve its structure, reduce its complexity or make it easier to understand.
- When you refactor a program, you should not add functionality but rather concentrate on program improvement.

Why to refactor code

- To limit *design decay/erosion*
- To improve code readability
- To simplify code
- To clean-up existing code
- To simplify the testing phase
- To simplify future maintenance

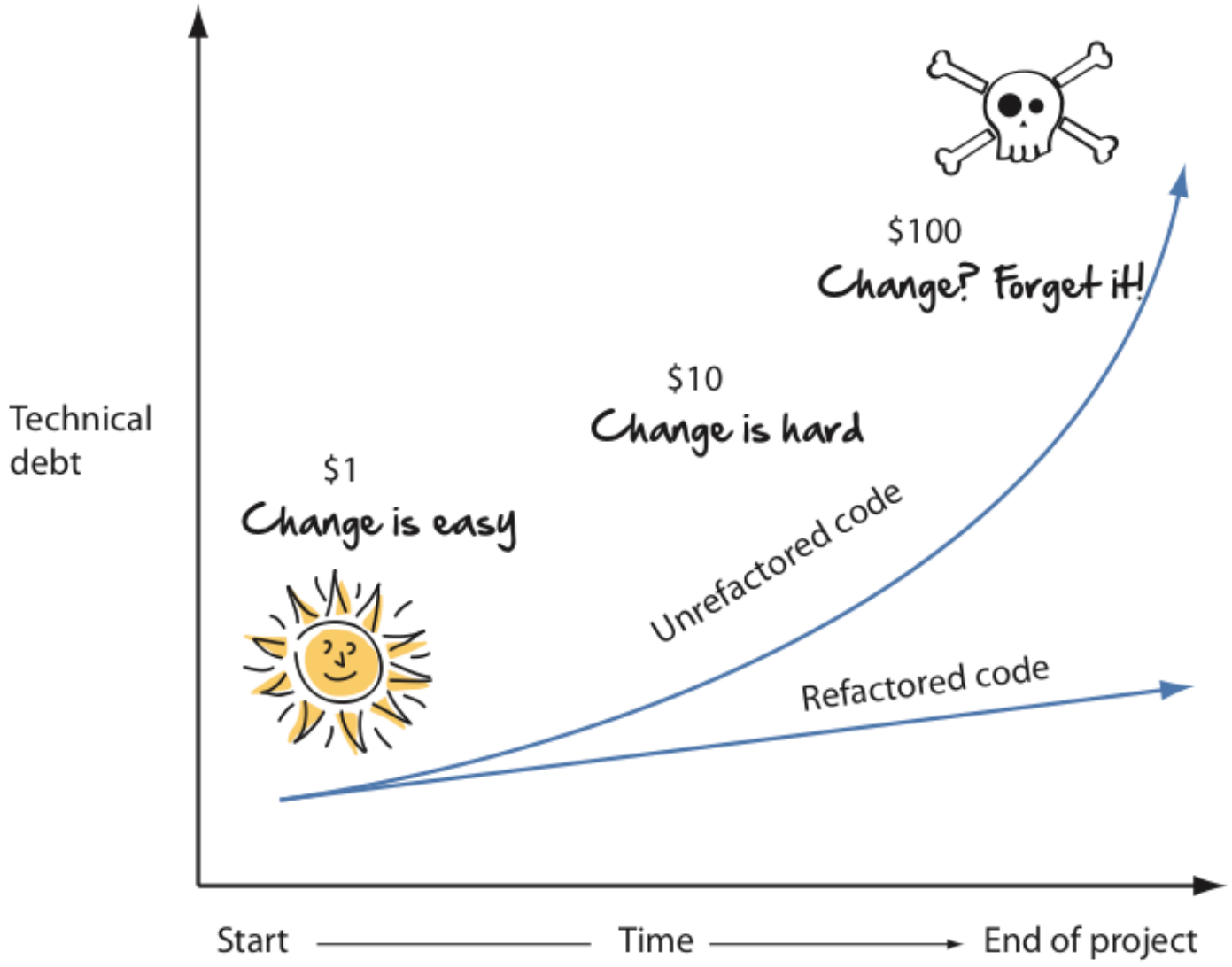


Refactoring Vs reengineering

- **Re-engineering** takes place after a system has been maintained for some time and maintenance costs are increasing
  - You use automated tools to process and re-engineer a legacy system to create a new system that is more maintainable

- **Refactoring** is a continuous process of improvement throughout the development and evolution process
  - It is intended to avoid the structure and code degradation that increases the costs and difficulties of maintaining a system.

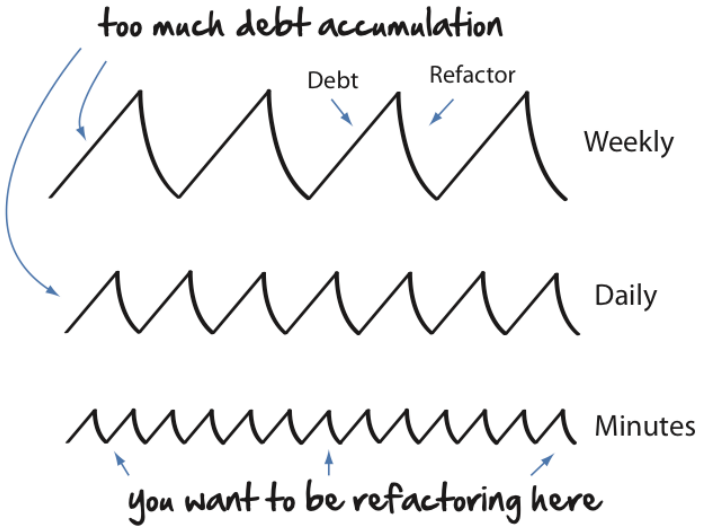
Refactoring opportunities



When to apply refactoring

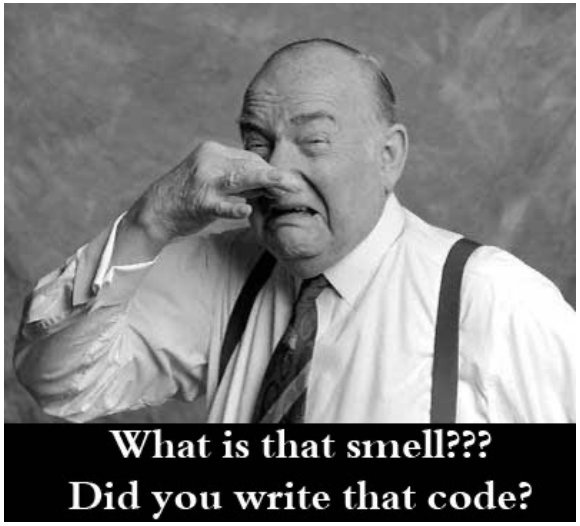
- Not planned
- Apply refactoring when:
  - Before a new feature need to be added
    - Faster and easier on refactored code
  - When a bug is fixed
  - When a code smell is identified

Apply refactoring as often as possible during development



Code smell

- Indicator that *something is wrong with the code*
  - Just hint, not certainty
  - Maybe just programming style, maybe something that reduce understandability, maybe a more serious problem is hidden
  - Often software metric tools identify that the code is affected by code smells



Code smells

- **Duplicate code**: The same or very similar code may be included at different places in a program.
  - This can be removed and implemented as a single method or function that is called as required
- **Long methods**: If a method is too long
  - It should be redesigned as shorter methods
- **Switch-case statements**: These often involve duplication, where the switch depends on the type of a value. The switch statements may be scattered around a program.
  - In object-oriented languages, you can often use polymorphism to achieve the same thing
- **Data clumping**: Data clumps occur when the same group of data items (fields in classes, parameters in methods) re-occur in several places in a program.
  - These can often be replaced with an object that encapsulates all of the data
- **Speculative generality**: This occurs when developers include generality in a program in case it is required in the future.
  - This can often simply be removed.

Code smells

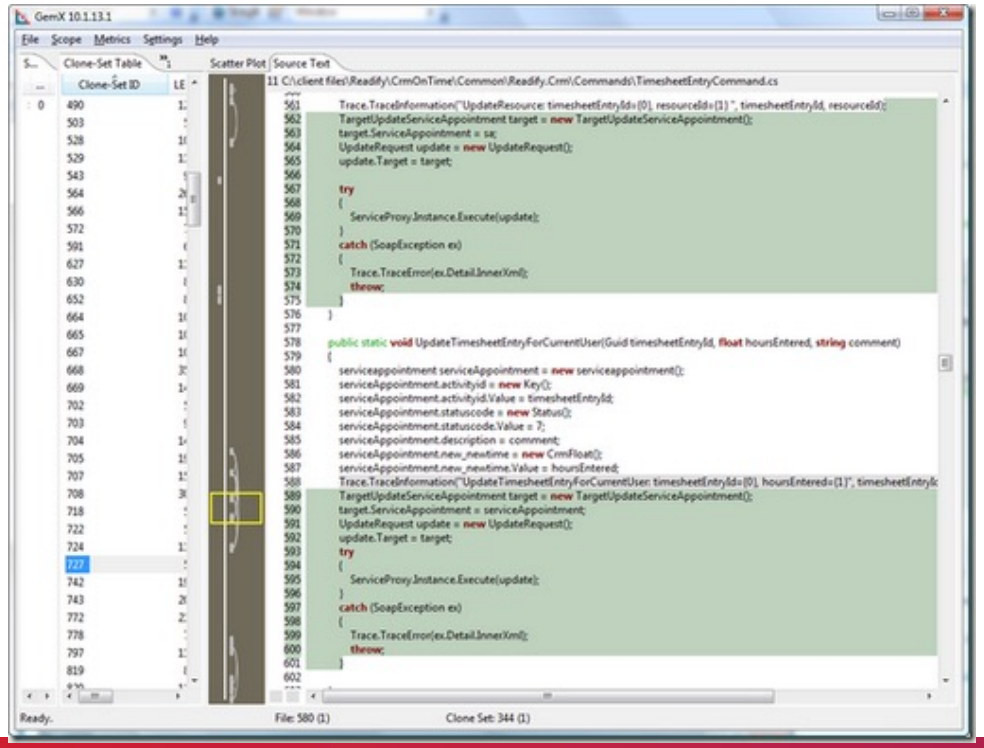
- Too much code:
  - Long method
  - Large class
  - Duplicate code (clone)
  - Dead code (code that is not executed)
  - Long parameter list
- Not enough code
  - Classes with little code
  - Data class (only getter/setter)
  - Empty catch clause
- Outside code
  - Excessing commenting

Pericoloso è opportuno gestire sempre gli errori

```
try {
    File.ReadAllText("test.txt");
}
catch{ }
```

Software clones

- Duplicated code
  - With or without changes
- *Bug propagation* (copy-pasted defects)
- Maintenance problems: all the clones should be detected and changed
- Prominence: 5% - 20% of a software system contains duplicated code

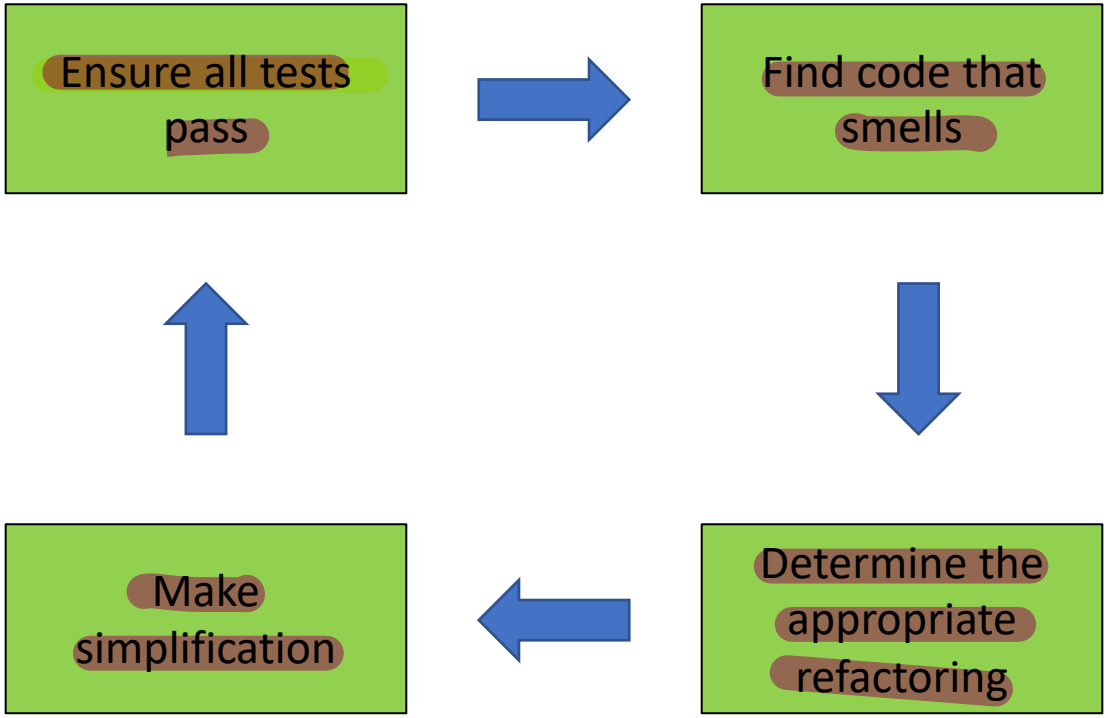


Clone examples

```java
01 package test;
02
03 public class TestFileOne {
04
05   public int factorial(int n){
06     if(n == 0){
07       return 1;
08     }else{
09       return n * factorial(n-1);
10     }
11   }
12
13   public int gcdOne(int a, int b) {
14     while (b != 0) {
15       if (a > b) {
16         a = a - b;
17       } else {
18         b = b - a;
19       }
20     }
21     return a;
22   }
23
24   public int mul(int a, int b){
25     int n = 0;
26     for(int i = 0; i < b; i++){
27       n += a;
28     }
29     return n;
30   }
31 }
```

```java
01 package test;
02
03 public class TestFileTwo {
04
05   public int factorial(int n){
06     if(n == 0){
07       return 1;
08     }else{
09       return n * factorial(n-1);
10     }
11   }
12
13   public int gcdTwo(int c, int d) {
14     while (d != 0) {
15       if (c > d) {
16         c = c - d;
17       } else {
18         d = d - c;
19       }
20     }
21     return c;
22   }
23
24   public double mul(double a, long b){
25     double n = 0.0;
26     for(long i = 0l; i < b; i++)
27       n += a;
28
29     return n;
30   }
31 }
```

Refactoring process



Refactoring pace

- Find a code smell
- Change a SMALL part of the code
  - Follow a clearly defined procedure (refactoring catalogue)
- Build
- Run test cases

Don't get ahead of yourself!

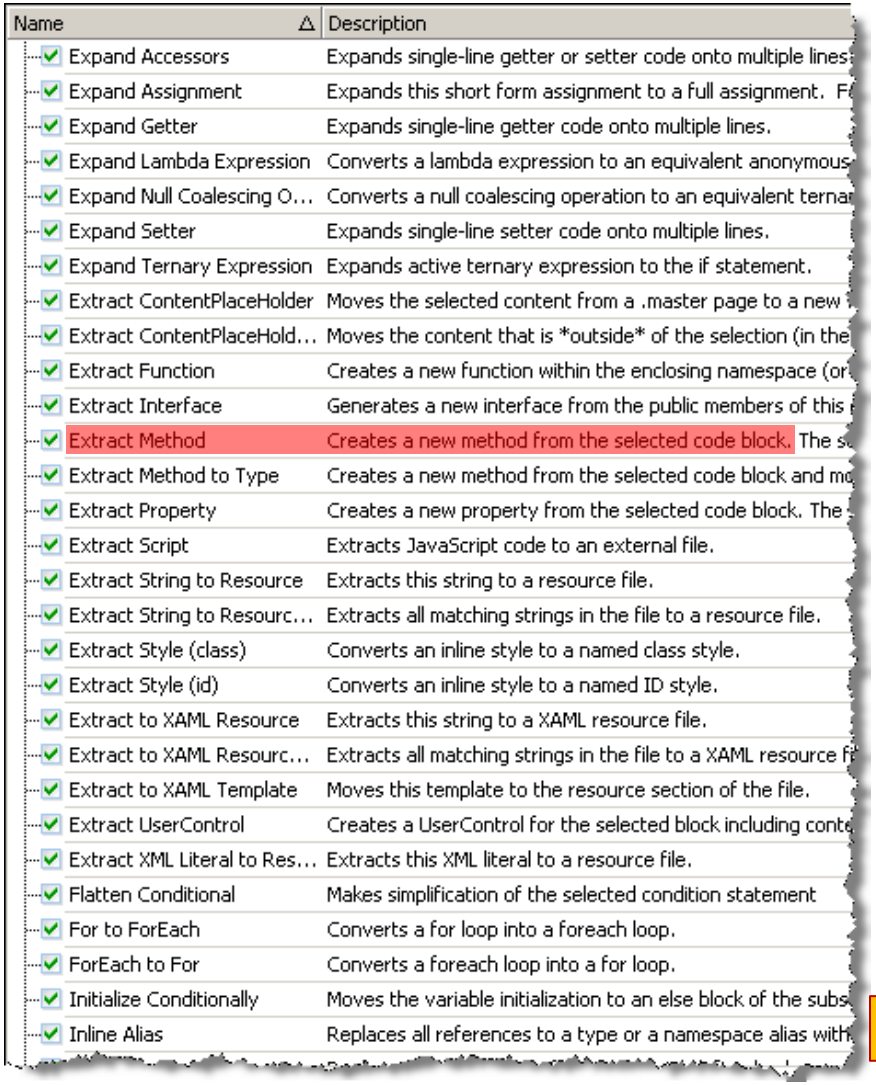
From a smell to refactoring

- Cheatsheet: http://people.scs.carleton.ca/~jeanpier/4004F19/T4-%20About%20Refactoring/0a-%20Smells%20to%20Refactorings.pdf

Smell	Description	Refactoring
Incomplete Library Class	Occurs when responsibilities emerge in our code that clearly should be moved to a library class, but we are unable or unwilling to modify the library class to accept these new responsibilities. [F 86]	Introduce Foreign Method [F 162]
		Introduce Local Extension [F 164]
Indecent Exposure	This smell indicates the lack of what David Parnas so famously termed information hiding [Parnas]. The smell occurs when methods or classes that ought not to be visible to clients are publicly visible to them. Exposing such code means that clients know about code that is unimportant or only indirectly important. This contributes to the complexity of a design. [K 42]	Encapsulate Classes with Factory [K 80]
Large Class	Fowler and Beck note that the presence of too many instance variables usually indicates that a class is trying to do too much. In general, large classes typically contain too many responsibilities. [F 78, K 44]	Extract Class [F 149]
		Extract Subclass [F 330]
		Extract Interface [F 341]
		Replace Data Value with Object [F 175]
		Replace Conditional Dispatcher with Command [K 191]
		Replace Implicit Language with Interpreter [K 269]
		Replace State-Altering Conditionals with State [K 166]
Long Method	In their description of this smell, Fowler and Beck explain several good reasons why short methods are superior to long methods. A principal reason involves the sharing of logic. Two long methods may very well contain duplicated code. Yet if you break those methods into smaller methods, you can often find ways for the two to share logic. Fowler and Beck also describe how small methods help explain code. If you don't understand what a chunk of code does and you extract that code to a small, well-named method, it will be easier to understand the original code. Systems that have a majority of small methods tend to be easier to extend and maintain because they're easier to understand and contain less duplication. [F 76, K 40]	Extract Method [F 110]
		Compose Method [K 123]
		Introduce Parameter Object [F 295]
		Move Accumulation to Collecting Parameter [K 313]
		Move Accumulation to Visitor [K 320]
		Decompose Conditional [F 238]
		Preserve Whole Object [F 288]
		Replace Conditional Dispatcher with Command [K 191]
		Replace Conditional Logic with Strategy [K 129]
		Replace Method with Method Object [F 135]
		Replace Temp with Query [F 120]

F - Fowler, Martin. Refactoring: Improving the Design of Existing Code
K - Kerievsky, Joshua. Refactoring to Patterns

Refactoring catalogue



Name	Description
Expand Accessors	Expands single-line getter or setter code onto multiple lines
Expand Assignment	Expands this short form assignment to a full assignment. F
Expand Getter	Expands single-line getter code onto multiple lines.
Expand Lambda Expression	Converts a lambda expression to an equivalent anonymous
Expand Null Coalescing O...	Converts a null coalescing operation to an equivalent terna
Expand Setter	Expands single-line setter code onto multiple lines.
Expand Ternary Expression	Expands active ternary expression to the if statement.
Extract ContentPlaceHolder	Moves the selected content from a .master page to a new
Extract ContentPlaceHold...	Moves the content that is *outside* of the selection (in the
Extract Function	Creates a new function within the enclosing namespace (or
Extract Interface	Generates a new interface from the public members of this
Extract Method	Creates a new method from the selected code block. The s
Extract Method to Type	Creates a new method from the selected code block and mo
Extract Property	Creates a new property from the selected code block. The
Extract Script	Extracts JavaScript code to an external file.
Extract String to Resource	Extracts this string to a resource file.
Extract String to Resourc...	Extracts all matching strings in the file to a resource file.
Extract Style (class)	Converts an inline style to a named class style.
Extract Style (id)	Converts an inline style to a named ID style.
Extract to XAML Resource	Extracts this string to a XAML resource file.
Extract to XAML Resourc...	Extracts all matching strings in the file to a XAML resource f
Extract to XAML Template	Moves this template to the resource section of the file.
Extract UserControl	Creates a UserControl for the selected block including conte
Extract XML Literal to Res...	Extracts this XML literal to a resource file.
Flatten Conditional	Makes simplification of the selected condition statement
For to ForEach	Converts a for loop into a foreach loop.
ForEach to For	Converts a foreach loop into a for loop.
Initialize Conditionally	Moves the variable initialization to an else block of the subs
Inline Alias	Replaces all references to a type or a namespace alias with

- Many refactoring are small and simple
  - Low level refactoring
    - *Rename method*
    - *Extract method*
- Low level refactoring is a building block (or enable) more complex refactoring
  - *Replace conditional with polymorphism*
- https://refactoring.com/catalog/

The catalogues is quite long, and always expanding

How to apply refactoring

- Two possible ways:
  - Apply refactoring manually
  - Use an automated tool
- Clearly the automated tool is more convenient
  - E.g., if a class name is changed, all the references should be changed consistently
    - The *constructor*
    - The file name
    - All the classes referencing the changed class
- However, automated tools support only the simple refactoring
- Anyway, test cases should be run in both the cases

International Symposium on Leveraging Applications of Formal Methods

ISoLA 2016: Leveraging Applications of Formal Methods, Verification and Validation: Foundational Techniques pp 517-531 | Cite as

Safer Refactorings

Authors Authors and affiliations

Anna Maria Eilertsen, Anya Helene Bagge, Volker Stolz

Conference paper

First Online: 05 October 2016

Part of the Lecture Notes in Computer Science book series (LNCS, volume 9952)

Abstract

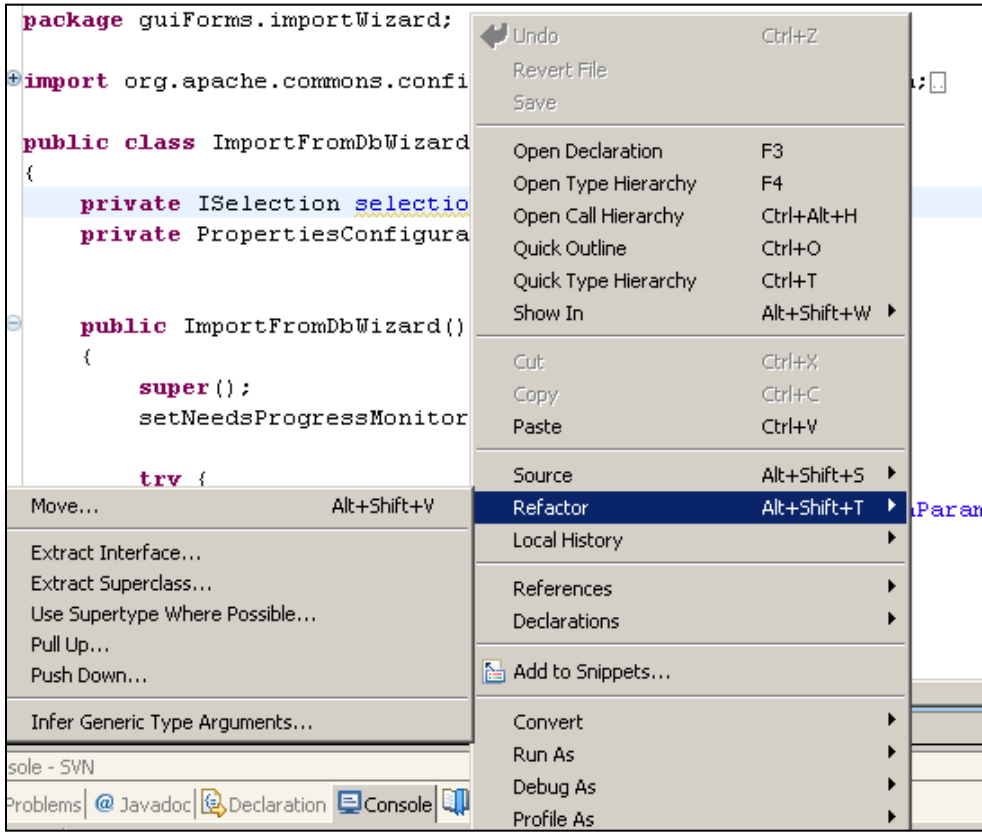
Refactorings often require semantic correctness conditions that amount to software model checking. However, IDEs such as Eclipse's Java Development Tools implement far simpler checks on the structure of the code. This leads to the phenomenon that a seemingly innocuous refactoring can change the behaviour of the program. In this paper we demonstrate our technique of introducing runtime checks for two particular refactorings for the Java programming language: Extract And Move Method, and Extract Local Variable. These checks can, in combination with unit tests, detect changed behaviour and allow identification of which specific refactoring step introduced the deviant behaviour.

Tools

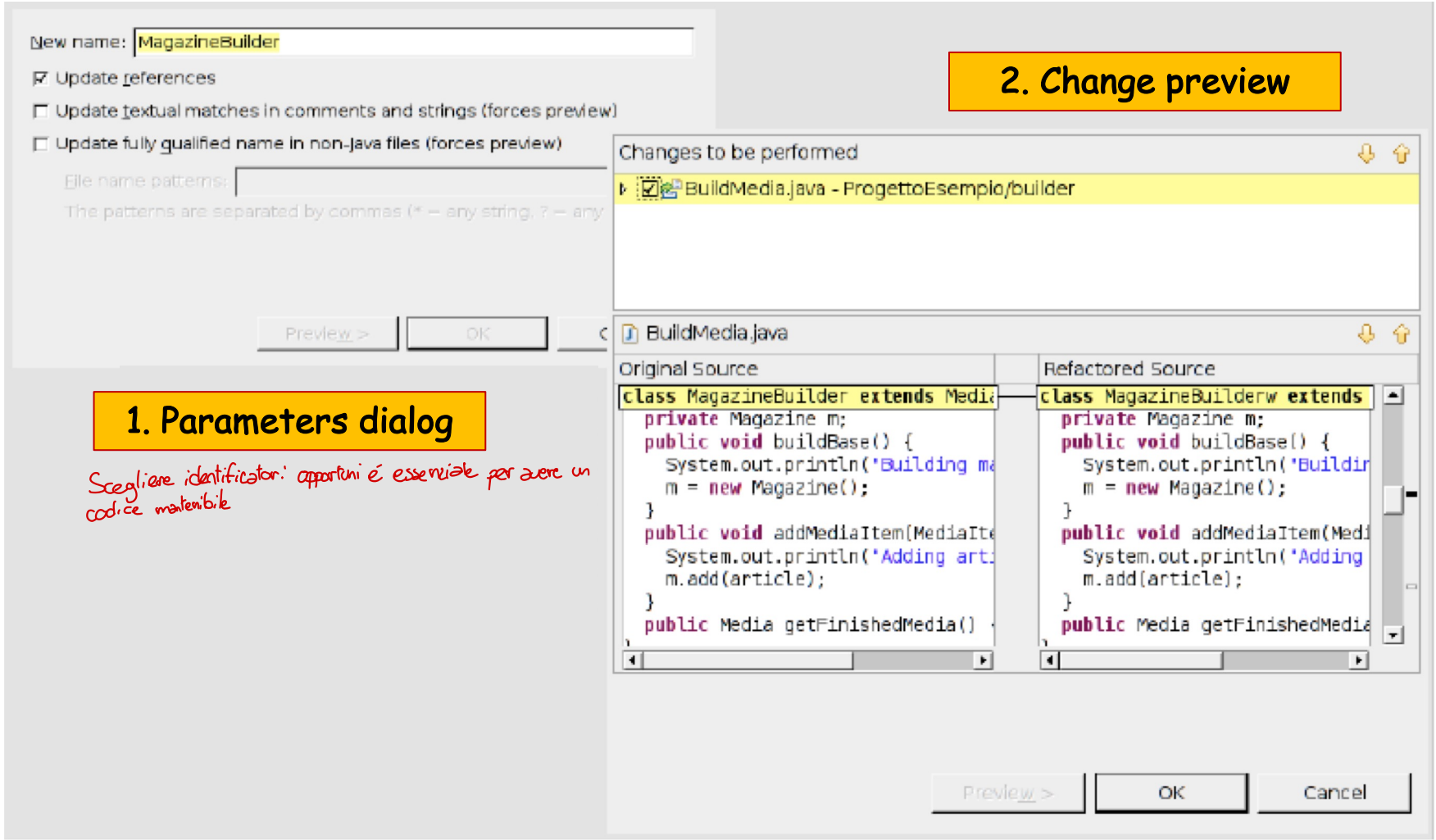
- Some IDE supports some simple refactoring
  - Eclipse
  - IntelliJ
  - NetBeans
- There exists specific plug-ins that suggests more complex refactoring
  - **Jdeodorant** (for Eclipse) detect some code smells and applies corresponding refactoring (on user request)
    - Good class -> Extract class
  - **ReShaper**: (for Microsoft Visual Studio)

Refactoring in Eclipse

1. Highlight the fragment of code of interest
2. Right click to open contextual menu
3. Select *"Refactor"*
4. Several refactoring options are proposed, depending on the selected code portion

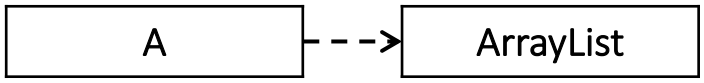


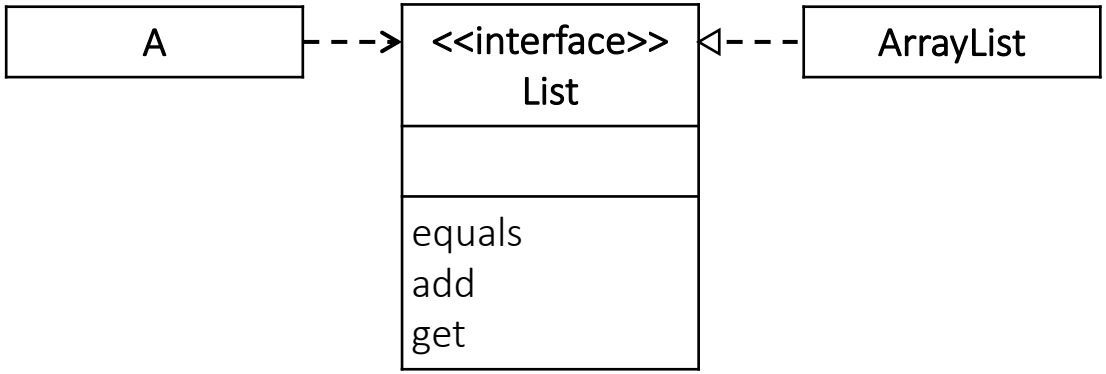
Rename a class

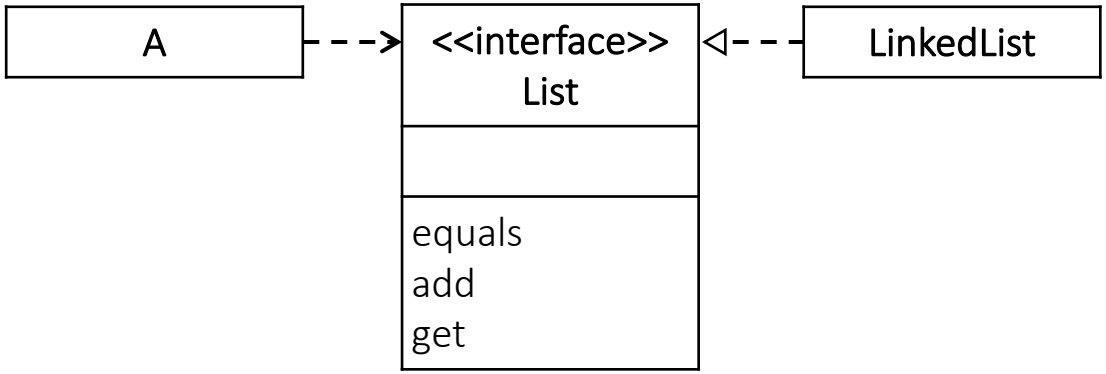


Scegliere identificatori appropriati è essenziale per avere un codice manutenibile

Extract Interface

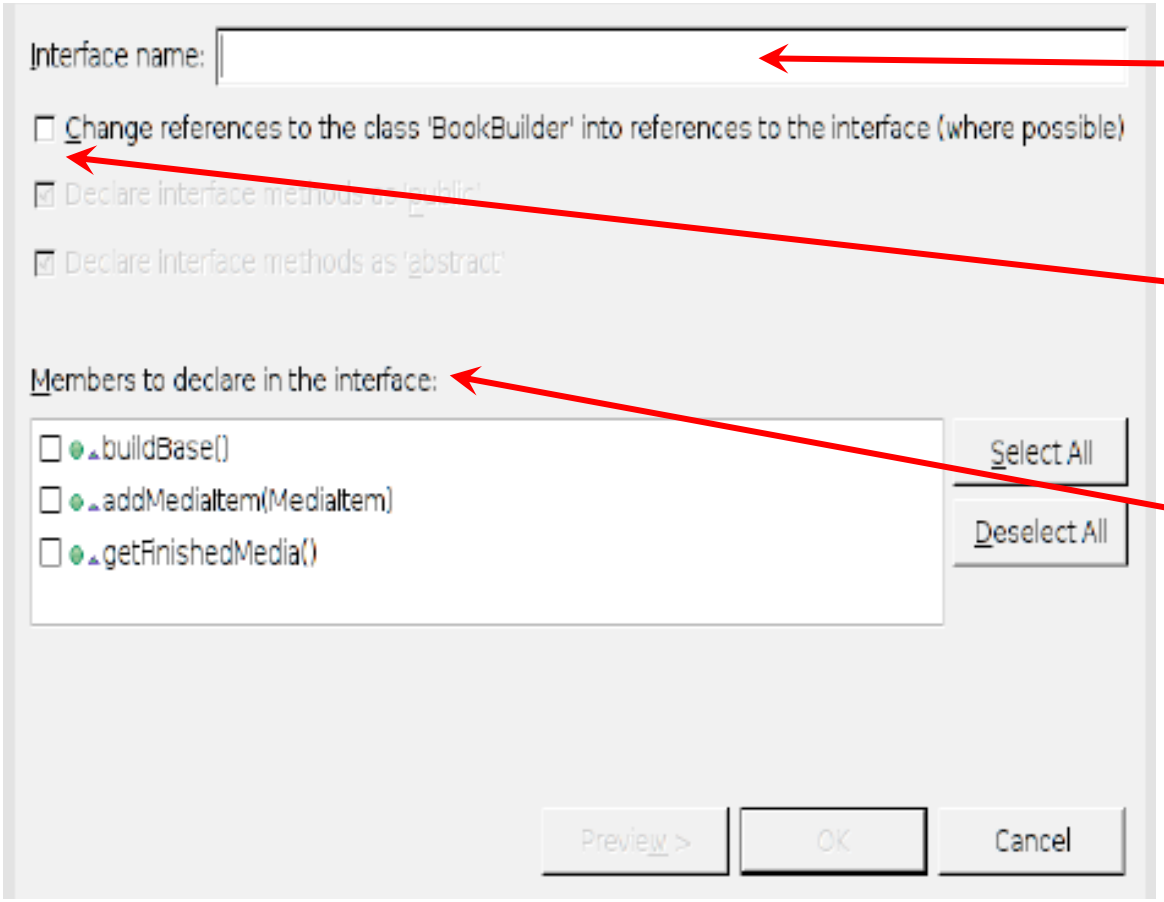






- Separation a class implementation from its interface
- Reduced coupling helps in evolution

Extract interface



1. Enter new interface name

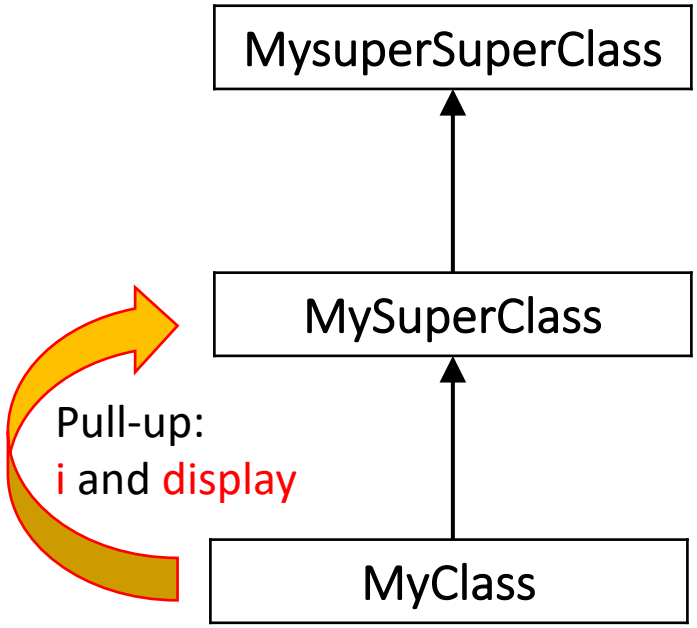
2. Change all the class refences to references to the new interface (where possible)

3. Specify what class methods should be part of the interface

Pull-up refactoring

```
class MySuperSuperclass {
}

class MySuperclass extends MySuperSuperclass {
}

class MyClass extends MySuperclass {
    int i, j;

    void display() {
    }
}

class MyOtherClass extends MySuperclass {
}
```

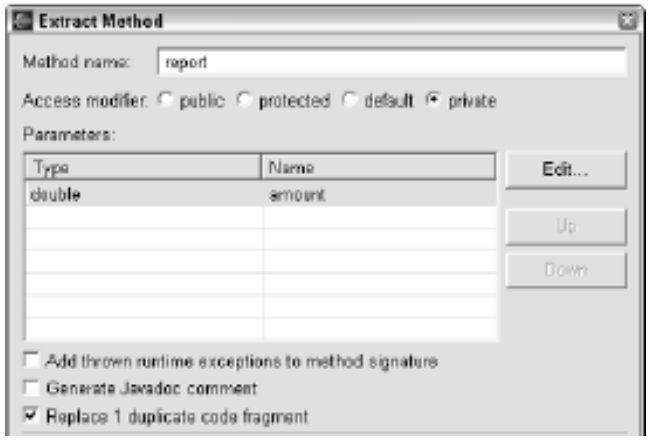
```
class MySuperSuperclass {
}

class MySuperclass extends MySuperSuperclass {

    int i;

    void display() {
    }
}

class MyClass extends MySuperclass {
    int j;
}

class MyOtherClass extends MySuperclass {
}
```



Extract method

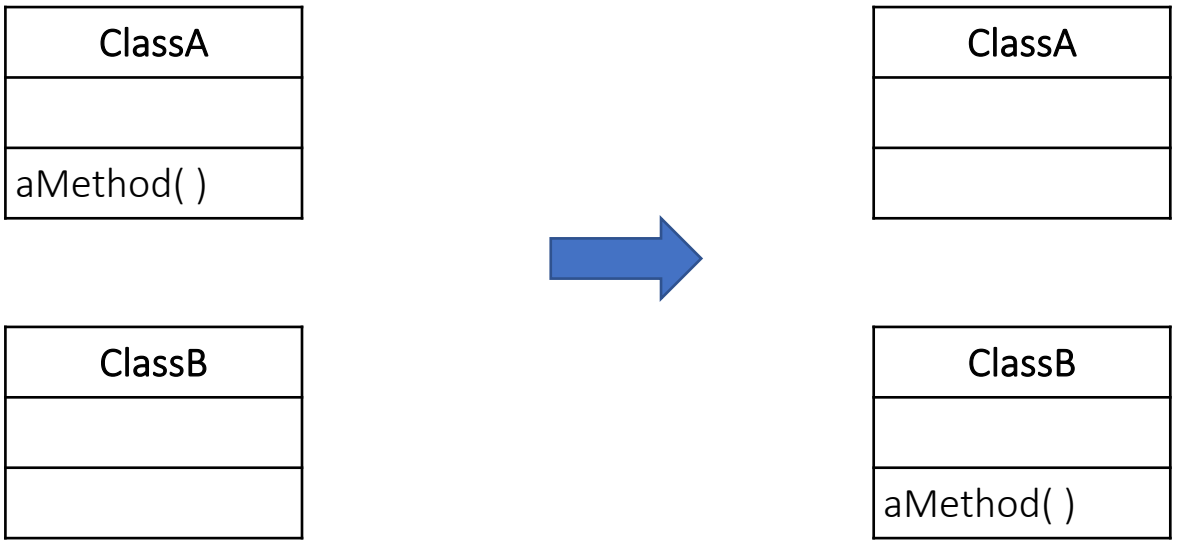
```
public class Account {
    String name;

    double balance;

    void doDeposit(double amount) {
        balance += amount;
        System.out.println("Name:               " + name);
        System.out.println("Transaction amount: " + amount);
        System.out.println("Ending balance:     " + balance);
    }

    void doWithdrawl(double amount) {
        balance -= amount;
        System.out.println("Name:               " + name);
        System.out.println("Transaction amount: " + amount);
        System.out.println("Ending balance:     " + balance);
    }
}
```

```
public class Account {
    String name;

    double balance;

    void doDeposit(double amount) {
        balance += amount;
        report(amount);
    }

    void doWithdrawl(double amount) {
        balance -= amount;
        report(amount);
    }

    private void report(double amount) {
        System.out.println("Name:               " + name);
        System.out.println("Transaction amount: " + amount);
        System.out.println("Ending balance:     " + balance);
    }
}
```



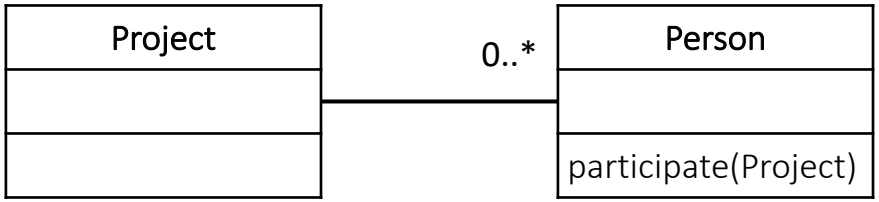
Move method

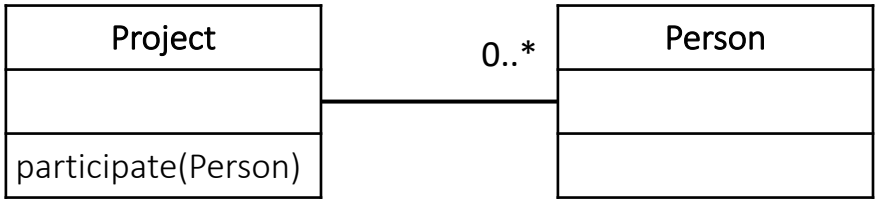
- When a class has too much behavior or when two classes collaborate too much or are too coupled
- A method is moved from one class to another one to reduce coupling



Move method

```
class Project {
  Person[] participants;
}

class Person {
  int id;
  boolean participate(Project p) {
    for(int i=0; i<p.participants.length; i++) {
          if (p.participants[i].id == id) return(true);
    }
    return(false);
  }
}

... if (x.participate(p)) ...
```






```
class Project {
  Person[] participants;
  boolean participate(Person x) {
    for(int i=0; i<participants.length; i++) {
          if (participants[i].id == x.id) return(true);
    }
    return(false);
  }
}

class Person {
  int id;
}

... if (p.participate(x)) ...
```

Replace temp with query

temp field field

```
double basePrice = _quantity * _itemPrice;
if (basePrice > 1000)
     return basePrice * 0.95;
else
    return basePrice * 0.98;
```

- Local variables are only visible in local scope, so they encourage long methods
- A query method is available to all the class methods

```
double basePrice() {
     return _quantity * _itemPrice;
}
```

```
...
if (basePrice() > 1000)
     return basePrice() * 0.95;
else
     return basePrice() * 0.98;
.
...
```

Replace parameter with method

- Methods with many parameters are difficult to understand.
- The formal parameter list should be as short as possible
- In case a method can obtain a value, it should not be a formal parameter

```
int basePrice = _quantity * _itemPrice;
discountLevel = getDiscountLevel();
double finalPrice = discountedPrice (basePrice, discountLevel);
```

```
int basePrice = _quantity * _itemPrice;
double finalPrice = discountedPrice (basePrice);
```

A value is computed and then used as actual parameter

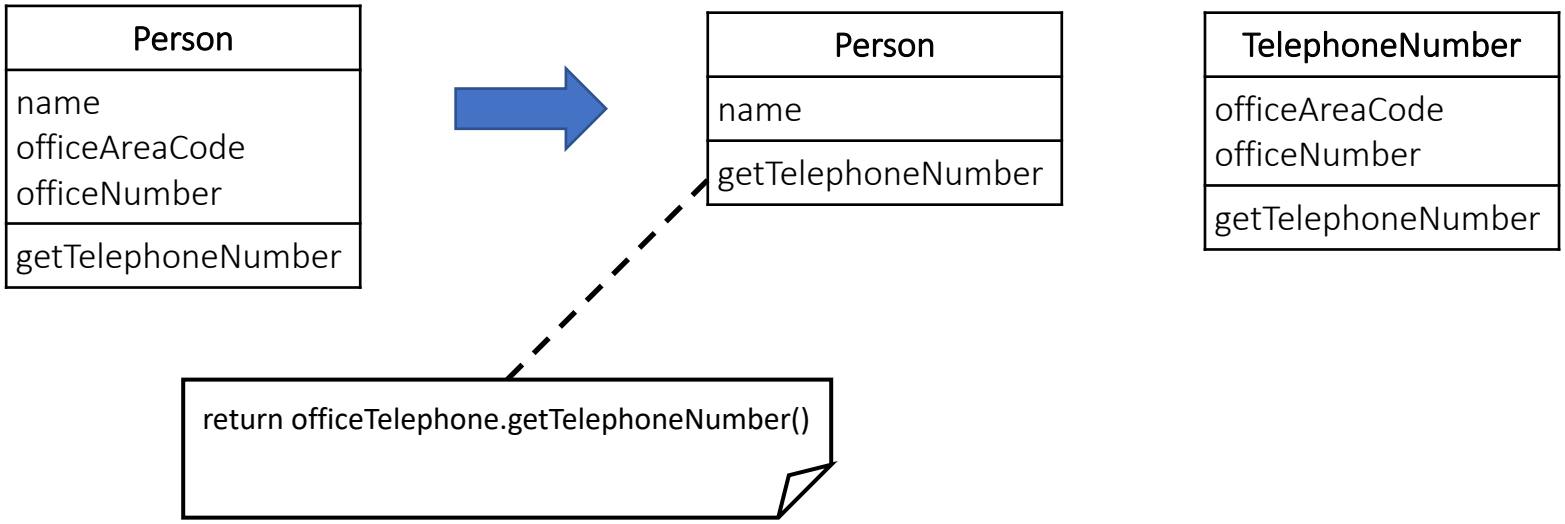
Now *discountedPrice()* is fetching *discountLevevel*

Extract class

- A class with too many responsibilities
  - *God class* or *Blob class* (low cohesion)

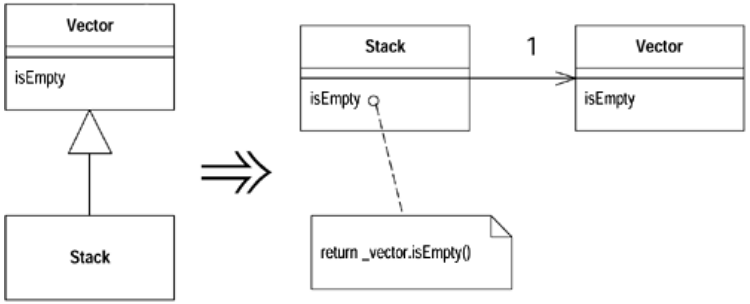
possibile identificare dei sottoinsiemi poco collegati ma con molti collegamenti interni

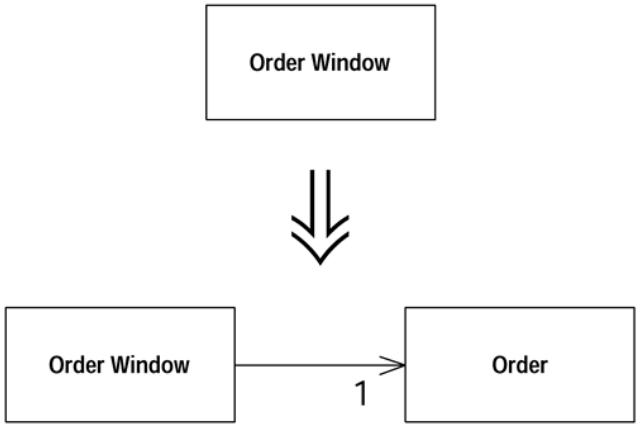
- Let's create a new class where we move some responsibilities (methods and fields)
- In case we do not mean to change the interface, delegation should be used

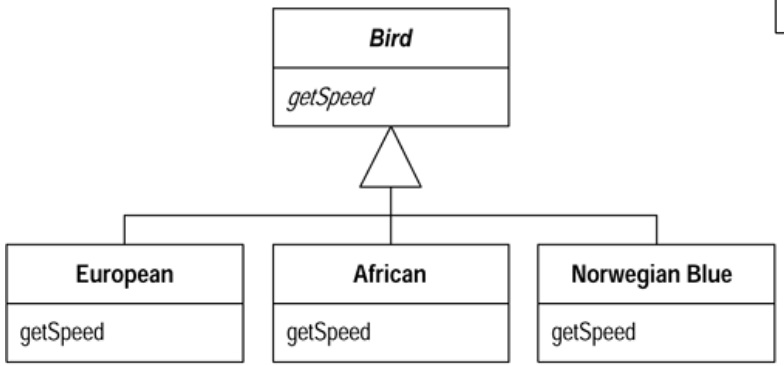


More complex refactoring

- Replace Inheritance with Delegation
- Replace Conditional with Polymorphism
- Separate Domain from Presentation

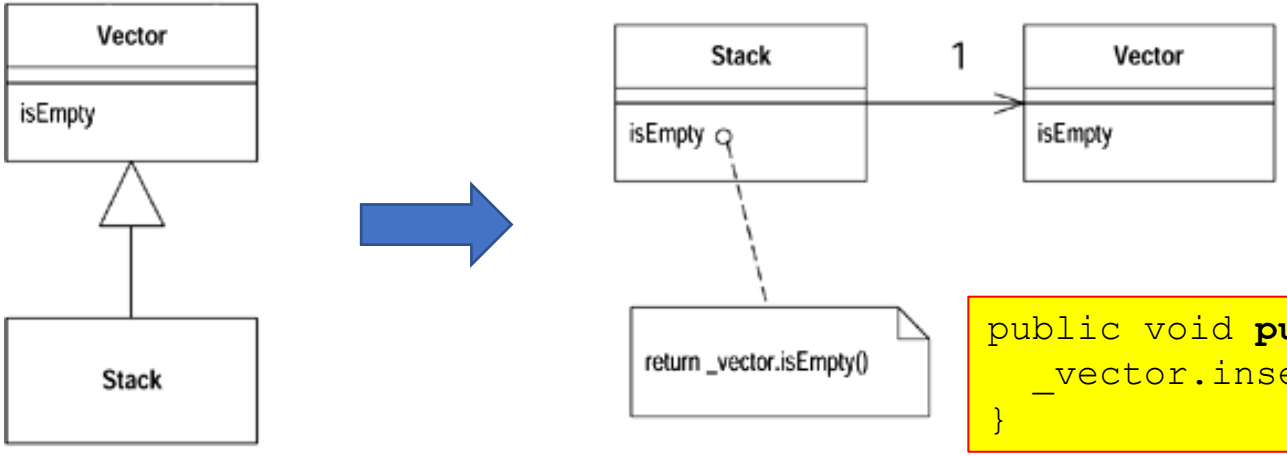






Replace inheritance with delegation

- A subclass only uses a portion of the superclass, but is not interested in the rest of it
- E.g., **Stack** is implemented starting from a **Vector**, but method *insertElementAt* makes no sense in a Stack



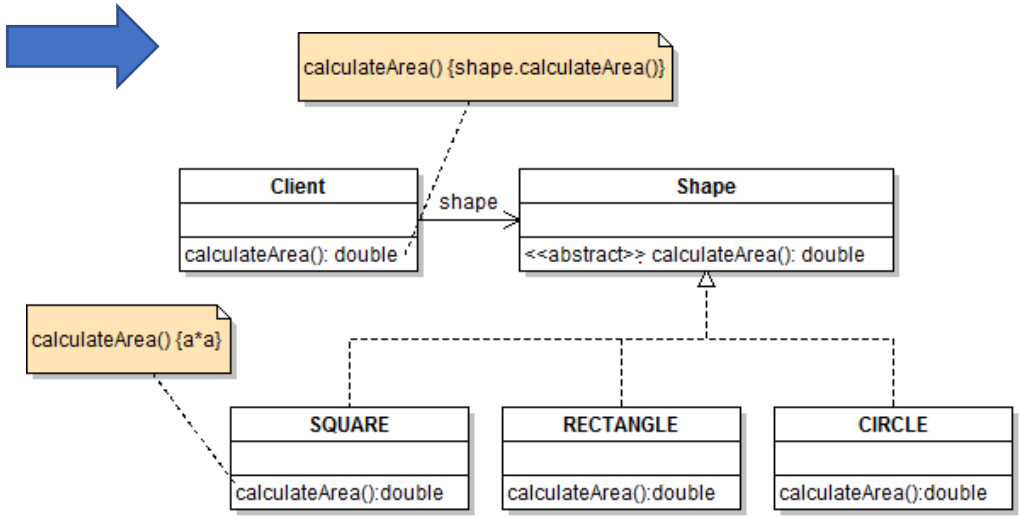

```
public void push(Object element) {
   _vector.insertElementAt(element,0);
}
```

```
public Object pop() {
   Object result = _vector.removeElementAt(0);
   return result;
}
```

Replace Conditional with Polymorphism

- A condition that chose among many behaviors, depending on a variable

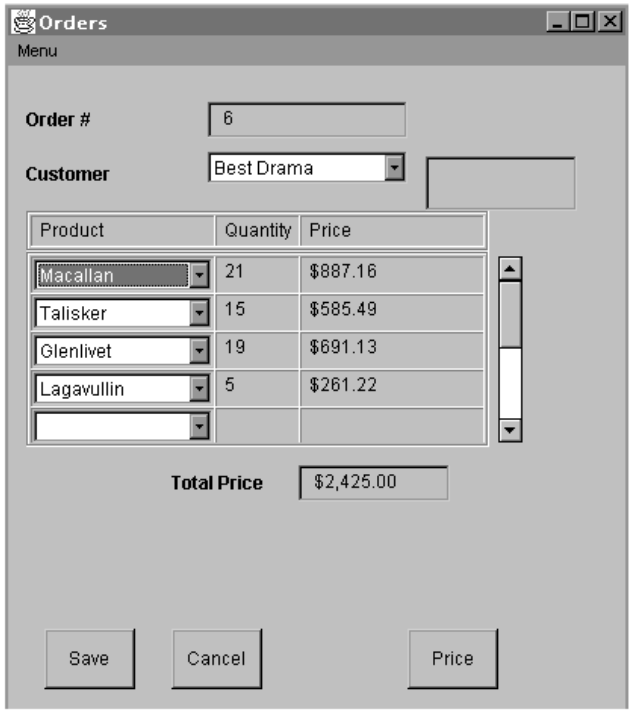
```
public class Client {
 private double a;
 private double b;
 private double r;
 ...
 public double calculateArea(int shape) {
     double area = 0;
     switch(shape) {
         case SQUARE:
             area = a * a;
             break;
         case RECTANGLE:
             area = a * b;
             break;
         case CIRCLE:
             area = Math.PI * r * r;
             break;
     }
     return area;
 }
```

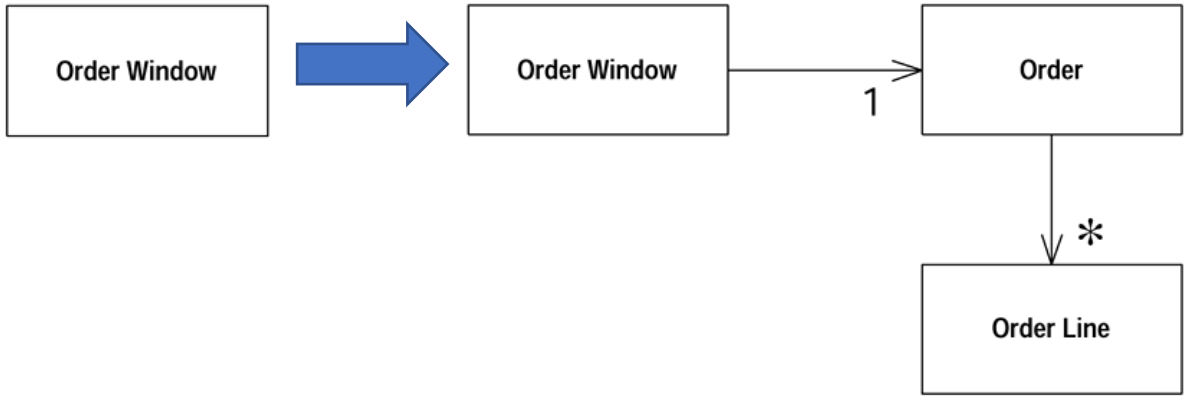


Extract class (Shape)
Move *calculateArea* to *Shape*
Create a new subclass for each *case* entry
Create a new method *calculateArea* in each subclass
Turn *calculateArea* in *Shape* abstract

Separate Domain from Presentation

- A GUI class contains also business logic





Many simple refactoring steps are needed:

- Extract class
- Move field
- Move method
- Extract method
- …

References

- Refactoring for everyone: How and why to use Eclipse's automated refactoring features. By David Gallardo https://www.ibm.com/developerworks/library/os-ecref/

- Fowler, Martin Refactoring – Improving the Design of Existing Code, Addison Wesley, 2000

- Fowler, Martin. Refactoring Home Page.
  - www.refactoring.com
  - https://www.refactoring.com/catalog/

- https://sourcemaking.com/refactoring