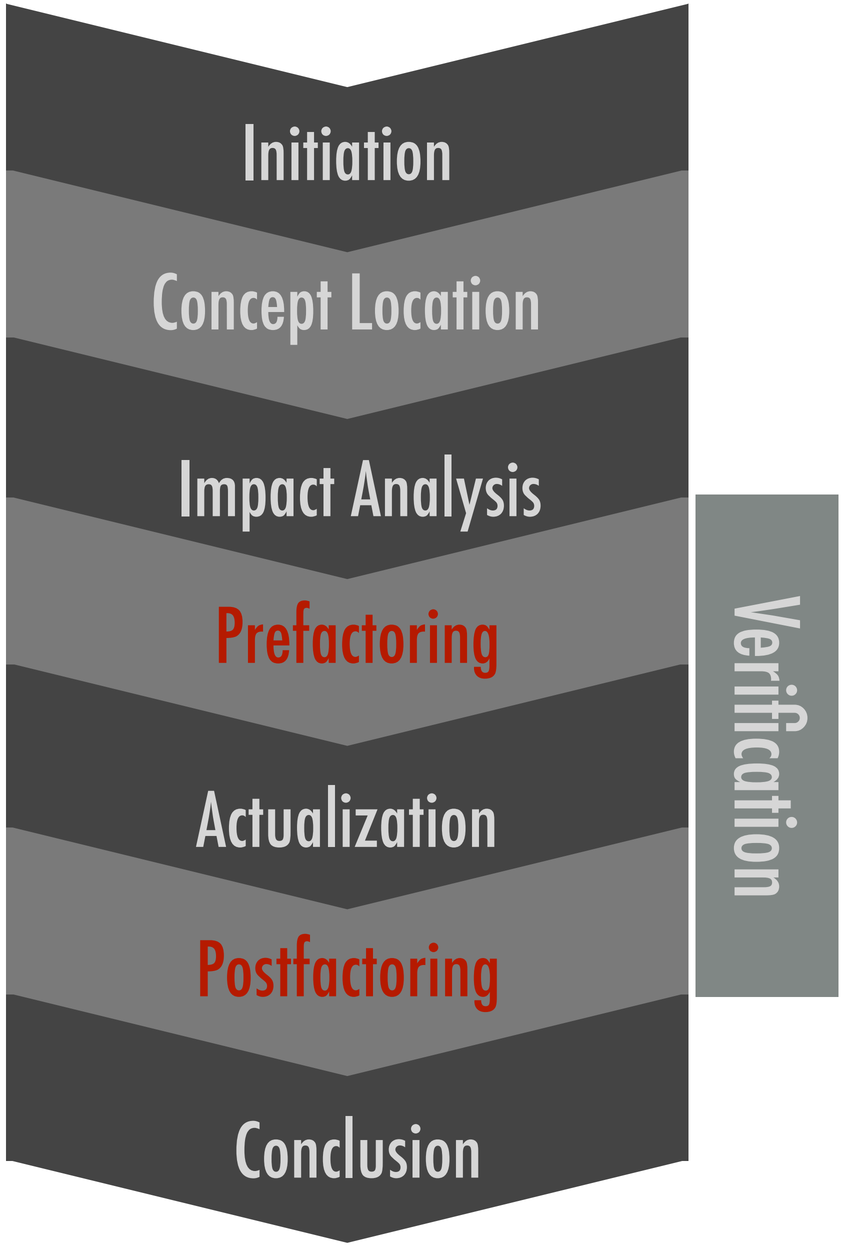
Refactoring

Prof. Romain Robbes

Refactoring improves the structure of code



The programmer makes the necessary changes to make actualization easier (prefactoring), and to prevent the decay caused by the change (postfactoring).

Software systems inevitably decay, say the Laws of Software Evolution

Increasing Complexity – As an E-type system evolves its complexity increases unless work is done to maintain or reduce it.

Declining Quality – The quality of E-type systems will appear to be declining unless they are rigorously maintained and adapted to operational environment changes.

Inevitably?

Increasing Complexity – As an E-type system evolves its complexity increases unless work is done to maintain or reduce it.

Declining Quality – The quality of E-type systems will appear to be declining unless they are rigorously maintained and adapted to operational environment changes.

Refactoring is a key set of techniques to prevent decay

Outline

Principles of refactoring

Refactoring are changes that do not change the behavior of the system

but, they improve its internal structure

```
if (isSpecialDeal()) {
   total = price * 0.95;
   ~~send();~~
} else {
   total = price * 0.98;
   ~~send();~~
}
```

```
if (isSpecialDeal()) {
    total = price * 0.95;
} else {
    total = price * 0.98;
}
send();
```

Refactoring is very pratical

Often, we don’t get the design right the first time
(but we’re close!).
Or, the requirement changes make a design change necessary.

Refactoring provides techniques for evolving the design in small **incremental** steps. Refactoring should by **systematic** and **safe**.

Benefits of refactoring

Code size often decreases (sometimes not).

Confusing structures are simplified, and hence easier to maintain and understand.

```
if (isSpecialDeal()) {
     total = price * 0.95;
     send();
} else {
     total = price * 0.98;
     send();
}
```

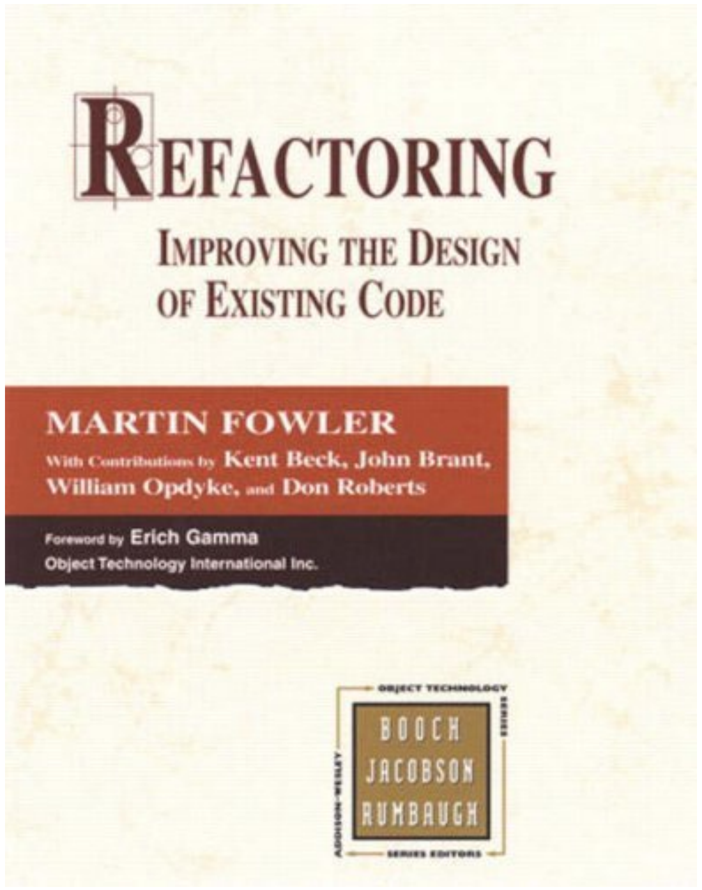
```
if (isSpecialDeal()) {
     total = price * 0.95;
} else {
     total = price * 0.98;
}
send();
```

Refactoring strives to make code understandable and maintenable

“Any fool can write code that a computer can understand. Good programmers write code that humans can understand.”

–Martin Fowler

A (very) recommended reading


REFACTORING

IMPROVING THE DESIGN OF EXISTING CODE

MARTIN FOWLER

With Contributions by Kent Beck, John Brant, William Opdyke, and Don Roberts

Foreword by Erich Gamma

Object Technology International Inc.

OBJECT TECHNOLOGY SERIES

BOOCH JACOBSON RUMBAUGH

ADDISON-WESLEY

SERIES EDITORS

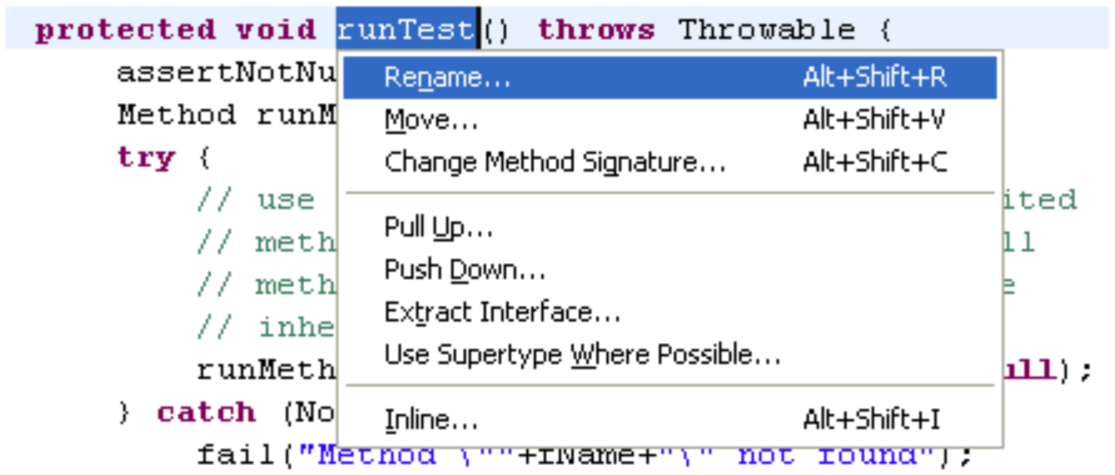
Refactoring, according to Fowler:

Refactoring (noun): a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior.

Refactoring (verb): to restructure software by applying a series of refactorings without changing its observable behavior.

How can we make refactoring safe?

1. Use refactoring patterns (cf. design patterns). Each refactoring is well defined.
2. Test constantly! (cf unit testing, next lecture). If we break the code, we try again.
3. Use tools. Eclipse makes a lot of refactorings automatic.



```
protected void runTest() throws Throwable {
    assertNotNu
    Method runM
    try {
        // use                                          ited
        // meth                                         ll
        // meth                                         e
        // inhe
        runMeth                                         ull);
    } catch (No
        fail("Method \""+fName+"\" not found");
```

Why should you refactor?

To prevent decay in the design: quick changes have that effect.
To make software easier to understand: less duplication, clear names, better design.

To ease finding bugs: code is easier to understand, less duplication.

To program faster: less bugs, less time spent reading code, better design.

When should you refactor?

Three strikes rule: “The first time you do something, you just do it. The second time you do something similar, you wince at the duplication, but you do the duplicate thing anyway. The third time you do something similar, you refactor.”

Before and after performing changes (pre and post factoring).
During code reviews.

When to be careful about refactoring:

When dealing with databases, or several languages in general.

When you don’t have all the code (APIs, etc). For APIs, deprecation may be needed.

Code Smells

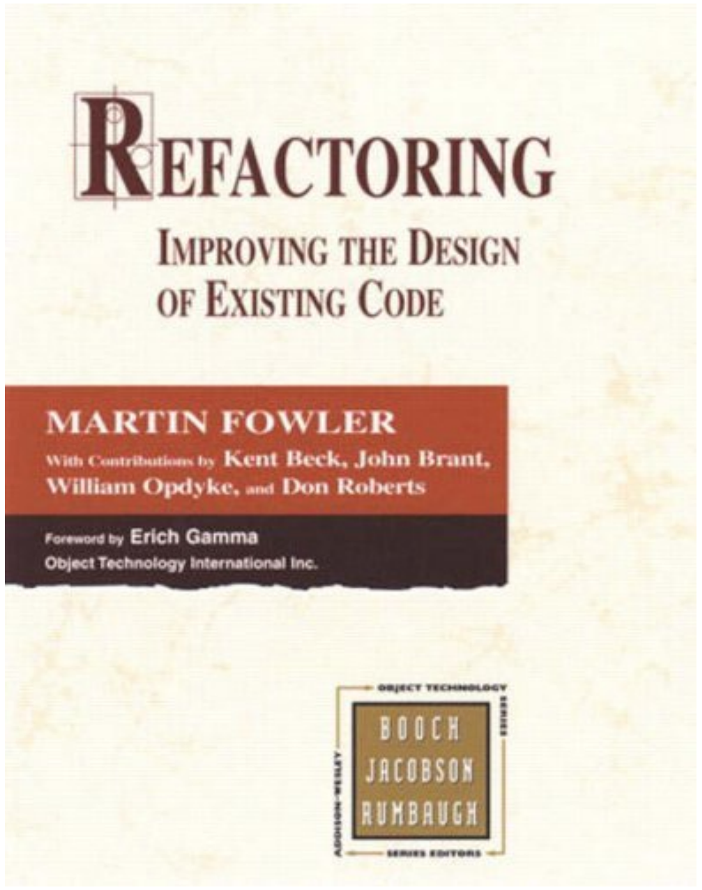
Code smells are early signs of code decay

"If it stinks, change it."
–Grandma Beck, discussing child-rearing philosophy

A code smell is a hint that something might be wrong, not a certainty. Further inspection is needed. If it's a problem, use refactoring to make the smell go away.

There are many code smells:

See chapter 3 of:


REFACTORING
IMPROVING THE DESIGN OF EXISTING CODE
MARTIN FOWLER
With Contributions by Kent Beck, John Brant, William Opdyke, and Don Roberts
Foreword by Erich Gamma
Object Technology International Inc.
OBJECT TECHNOLOGY SERIES
BOOCH JACOBSON RUMBAUGH
SERIES EDITORS
ADDISON-WESLEY

A nice "cheatsheet":
http://www.industriallogic.com/wp-content/uploads/2005/09/smellstorefactorings.pdf

Duplicated code

Increases program size and makes maintenance more difficult:

If you change one copy, and not the other, you may get a bug!

Long method

Long methods are more difficult to understand!

What’s “long”?
- If you need to scroll ...
- More than 7-8 (Smalltalk)
- More than 10-20 lines (Java)

Large class

The class has many instance variables and/or many methods.
The class is probably doing too much.
It probably has a low cohesion.

Long parameter list

Having too many parameters makes the method hard to understand, hard to use, and hard to learn. Especially if several parameters are of the same type.

Are all the parameters necessary?
Maybe some of the parameters can be grouped as a single object?

Divergent change

Symptom of low cohesion: some changes affect one part of the class, other changes affect another part.

Would it make sense to split it in two?

Shotgun surgery

The opposite of Divergent change: some changes affect several classes.
This is a symptom of high coupling between the classes.

Feature envy

A method in class A uses a lot of methods from class B.
Maybe it should be moved to class B?

Data clumps

Data from different classes and/or parameters
are often used together.

Maybe they should be moved in the same place.
For parameters, maybe they should be grouped in
a single, coherent entity.

Primitive obsession

Parameters are (several) primitive types, and not classes.
Consider using an appropriate class instead.

Switch statements

Lots of complex conditions. Tend to be associated with duplication. If a lot of similar switches exist in the program, maybe we should use polymorphism.

Parallel inheritance hierarchies

Two inheritance hierarchies have the same shape; e.g., a “model” tree and a “UI” tree.

This can be seen as a special case of Shotgun surgery as it may be necessary to change “sister classes” together.

Speculative generality

A part of the system is too generic without a good reason (for instance, a Factory design pattern with only one type of objects to create).

Sometimes, we anticipate change too much ...the code may become too complex because of this and could be simplified.

Temporary field

An instance variable in an object is used only part of the time.

Should it belong to the object for its entire lifetime? Can it be removed instead, or moved elsewhere?

Message chains

```
dog.leg().knee().bend();
```

This is a possible violation of Demeter’s law, which states that an object should only “talk” to its direct references, and not access objects via long method chains.

Middle man

A class delegates most of its work to another class.
Is the class really necessary?

Possible cause: too many levels of indirection?
(A possible special case of speculative generality)

Inappropriate intimacy

An object is accessing the instance variables of another object.
A class has no business accessing another classe’s instance variables!

This breaks the encapsulation and leaks implementation details.
Note that using getters and setters is a very marginal improvement.

Inheritance can also be a cause for this, as instance variables of the superclass may be visible to the subclasses, even if they shouldn’t be.

Alternative class with different interfaces

Some methods do the same thing, but with different names or parameters. This is bad for consistency and as a consequence makes the APIs of the system harder to learn.

Data class

The class has instance variables, getters, and setters. That’s it.

Objects should have behavior!

Opposite: Large or God class (usually manipulating data classes).

Refused bequest (legacy)

A subclass does not use the superclasses’ behavior.
E.g., overriding a non-abstract method, but not calling super.
Or, overriding a method to to do nothing or to throw an error.

This indicates that the class hierarchy may be wrong.

Comments

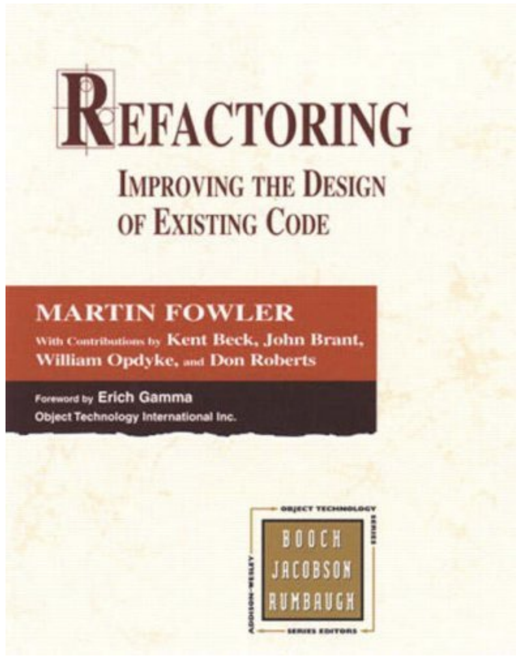
Comments are good!

But, not when they are explaining code that is too complicated. “Comments are often used as a deodorant”.

A common case is comments explaining a block of code, that may be replaced with call to a new method containing the block of code, and that has a descriptive name. (See “extract method” refactoring later on).

A refactoring catalogue

Read the book!


REFACTORING
IMPROVING THE DESIGN OF EXISTING CODE
MARTIN FOWLER
With Contributions by Kent Beck, John Brant, William Opdyke, and Don Roberts
Foreword by Erich Gamma
Object Technology International Inc.
BOOCH
JACOBSON
RUMBAUGH

The book has 72 refactorings (the website, 93!).
We can't see all of them; we talk about the most important ones.

http://www.refactoring.com

Rename method/class/instance variable

A concept in the code has an unclear name.
Rename it, and change all the usages of it.

Example: the concept of Money is too generic, rename it to Salary.

Add parameter

A method needs a new parameter. Add the parameter to the method's signature, and add a default parameter to all calls of the method.

Extract method

A group of statements in a long method performs a cohesive function.
Give it a name, and put it in a new 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);
}
```

Replace temp with query

An expression is stored in a temp.
Extract the method, and replace the temp with a method call.
Now you can use the method somewhere else!

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

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

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

Move method

A method in class A uses many features in class B (feature envy).

Move the code of A to B. You can make the method in A delegate to the one in B, or just remove it.

A composite refactoring: Extract class

One class is doing the work of two classes.
Create a new class, and move the relevant behavior there (with the move method refactoring).

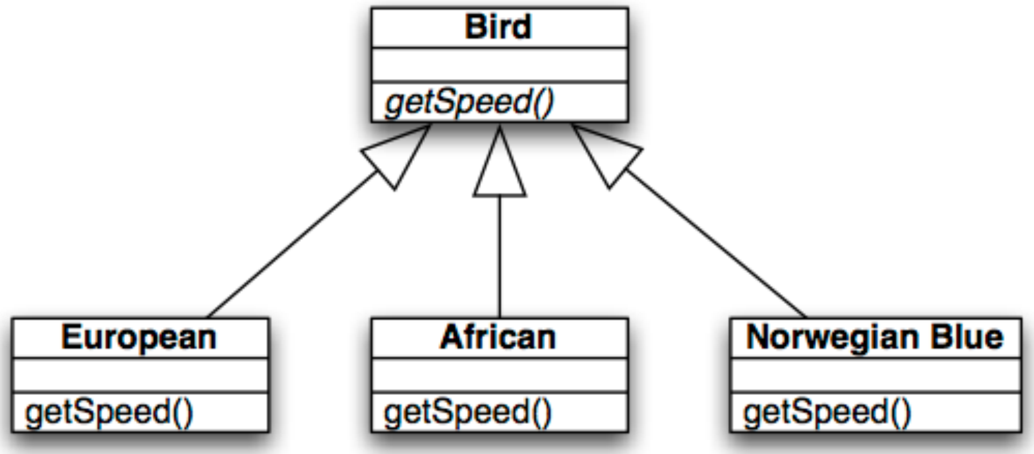
E.g. A customer class has a telephone number instance variable, and several methods related to this instance variable only, like area code, etc. (Divergent change).

Create a TelephoneNumber class, move the code there.
Additional benefit: now the customer can have two phone numbers!

Replace conditional with polymorphism

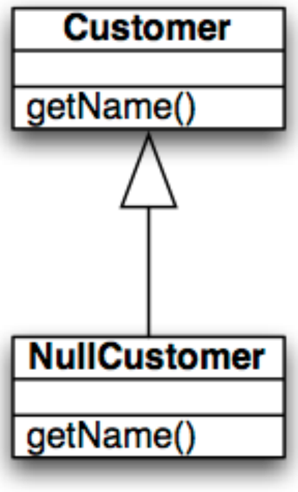
You have a switch statement testing the type of an object.
Make a polymorphic method, with each condition in a separate implementation.

```
double getSpeed() {
  switch (_type) {
    case EUROPEAN:
      return getBaseSpeed();
    case AFRICAN:
      return getBaseSpeed() - getLoadFactor() *
      _numberOfCoconuts;
    case NORWEGIAN_BLUE:
      return (_isNailed) ? 0 : getBaseSpeed(_voltage);
  }
  throw new RuntimeException("Unreachable")
}
```



Introduce null object

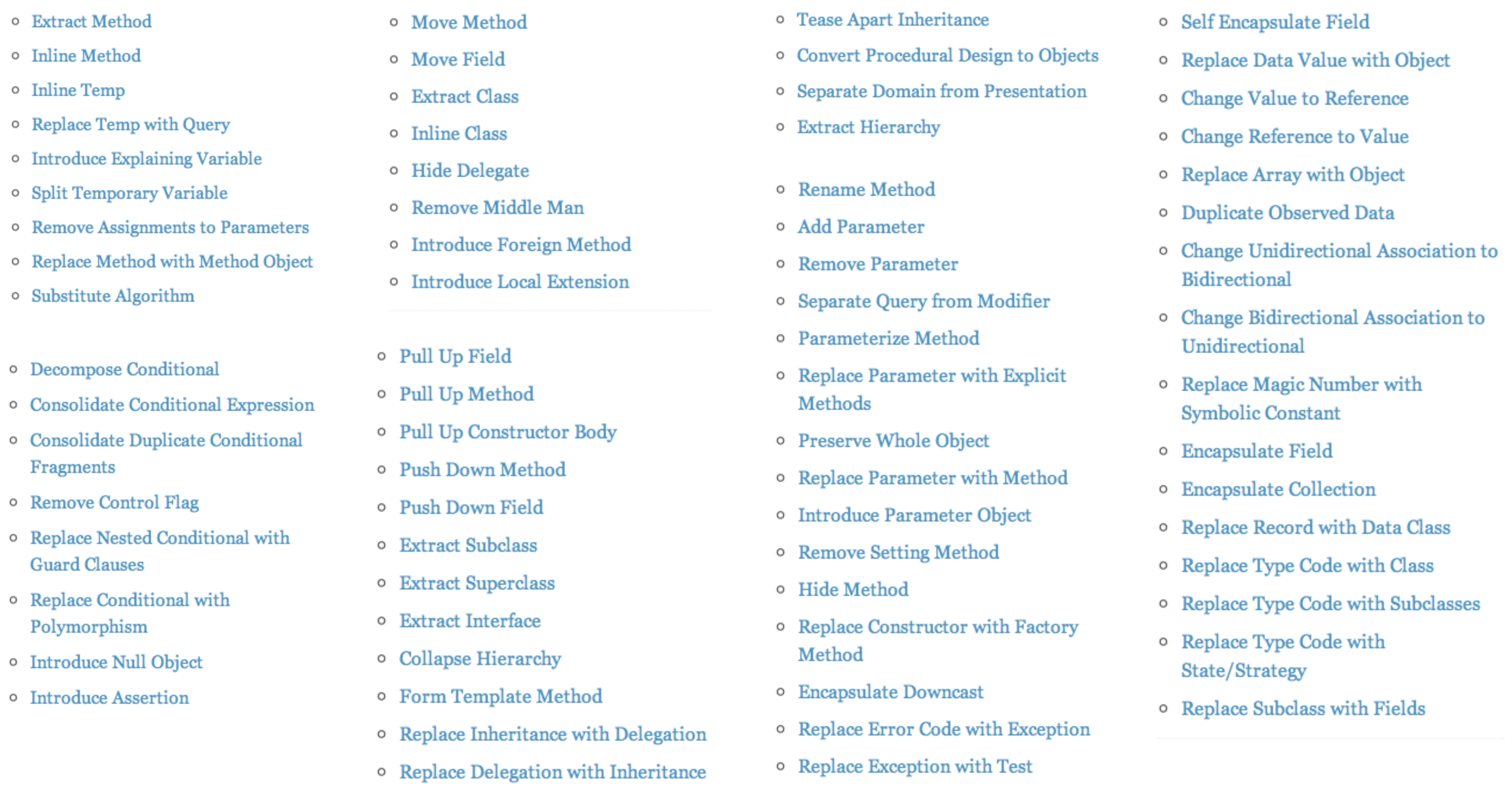
The code checks if an object is null. Create a new object with default values.



```
...
Customer c = findCustomer(...);
...
if (customer == null) {
   name = "occupant"
} else {
   name = customer.getName()
}
if (customer == null) {
...
```

```
public class nullCustomer {
   public String getName() { return "occupant";}
}
==============================
Customer c = findCustomer(...);
name = c.getName();
```

There are a few more refactorings ...



- Extract Method
- Inline Method
- Inline Temp
- Replace Temp with Query
- Introduce Explaining Variable
- Split Temporary Variable
- Remove Assignments to Parameters
- Replace Method with Method Object
- Substitute Algorithm

- Decompose Conditional
- Consolidate Conditional Expression
- Consolidate Duplicate Conditional Fragments
- Remove Control Flag
- Replace Nested Conditional with Guard Clauses
- Replace Conditional with Polymorphism
- Introduce Null Object
- Introduce Assertion

- Move Method
- Move Field
- Extract Class
- Inline Class
- Hide Delegate
- Remove Middle Man
- Introduce Foreign Method
- Introduce Local Extension

- Pull Up Field
- Pull Up Method
- Pull Up Constructor Body
- Push Down Method
- Push Down Field
- Extract Subclass
- Extract Superclass
- Extract Interface
- Collapse Hierarchy
- Form Template Method
- Replace Inheritance with Delegation
- Replace Delegation with Inheritance

- Tease Apart Inheritance
- Convert Procedural Design to Objects
- Separate Domain from Presentation
- Extract Hierarchy

- Rename Method
- Add Parameter
- Remove Parameter
- Separate Query from Modifier
- Parameterize Method
- Replace Parameter with Explicit Methods
- Preserve Whole Object
- Replace Parameter with Method
- Introduce Parameter Object
- Remove Setting Method
- Hide Method
- Replace Constructor with Factory Method
- Encapsulate Downcast
- Replace Error Code with Exception
- Replace Exception with Test

- Self Encapsulate Field
- Replace Data Value with Object
- Change Value to Reference
- Change Reference to Value
- Replace Array with Object
- Duplicate Observed Data
- Change Unidirectional Association to Bidirectional
- Change Bidirectional Association to Unidirectional
- Replace Magic Number with Symbolic Constant
- Encapsulate Field
- Encapsulate Collection
- Replace Record with Data Class
- Replace Type Code with Class
- Replace Type Code with Subclasses
- Replace Type Code with State/Strategy
- Replace Subclass with Fields

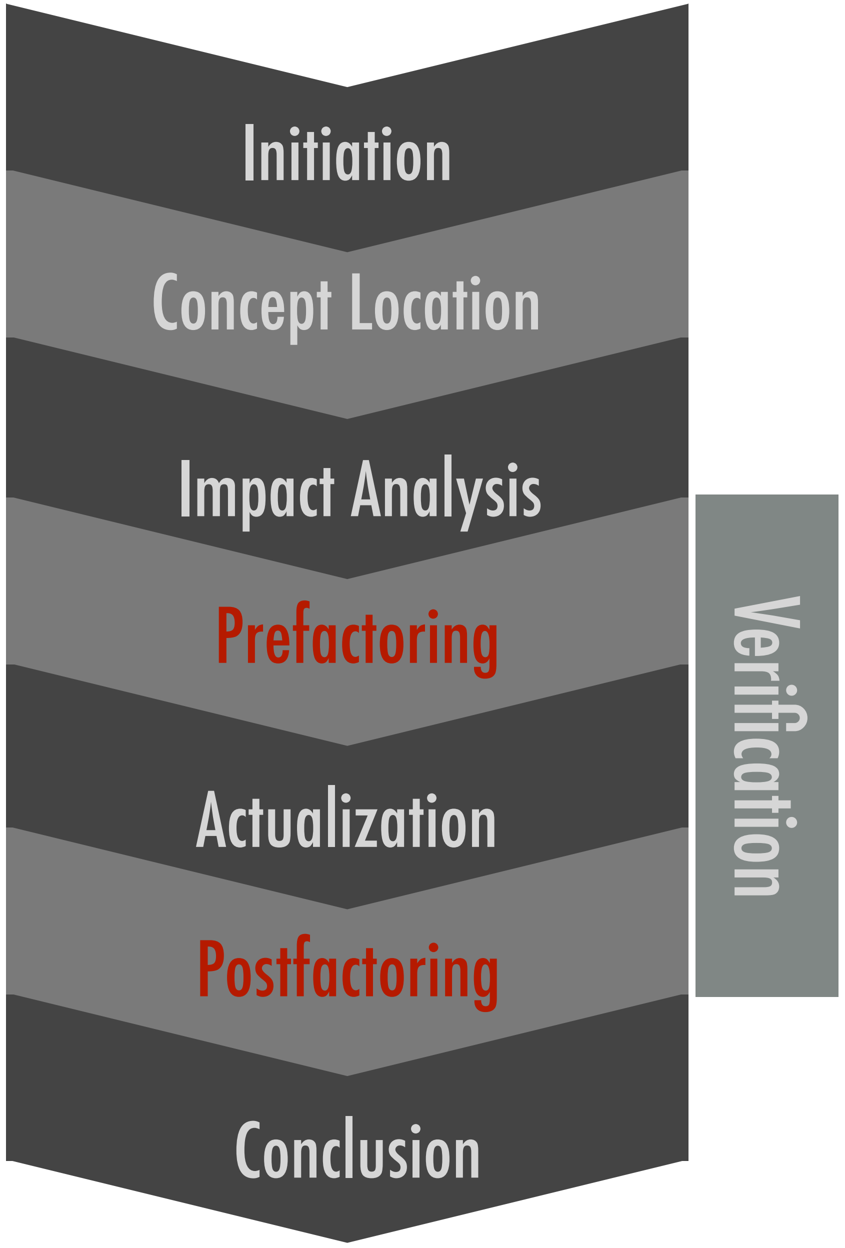
There are a few empirical studies on refactoring

A recent study shows that refactored binaries in Windows tend to have less dependencies, and less bugs.

On the other hand, other studies showed that incomplete refactorings are defect prone. Programmers are not aware of the presence of automated refactoring tools!

Pre and post-factoring

Refactoring improves the structure of code



The programmer makes the necessary changes to make actualization easier (prefactoring), and to prevent the decay caused by the change (postfactoring).

The goal of prefactoring is to make the change easier to do

Examples of useful refactorings:
Extract class, extract method, add parameter

We stop prefactoring when the new structure makes the change actualization easy.

The goal of postfactoring is to prevent decay after the change

Example of useful refactorings:
Rename method, instance variable, class ...

We stop when the quality of the code is good enough and all the new code smells are treated.

Prefactoring and postfactoring are both refactoring phases

Any refactoring can be applied during each phase if necessary.

Examples

Refine a concept by introducing a base class

Change request: “add sparse matrices to the program”.

We found this class:

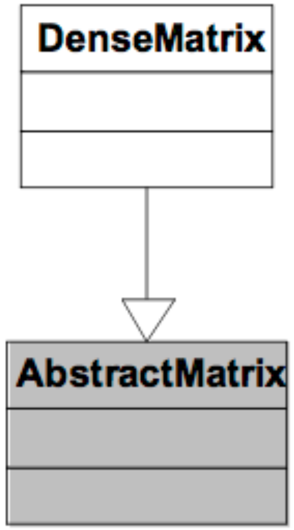
```
class Matrix {
 protected:
  int elements [10000], columns, rows;
 public:
  Matrix();
  void inverse();
  void multiply (Matrix& m);
  int get(int x, int y);
  void put(int x, int y, int value);
}
```

Steps:

- Rename `Matrix` to `DenseMatrix`
- Create superclass `AbstractMatrix`
- Replace references to elements with `get()` and `put()` in `DenseMatrix`
- Add abstract methods `get()` and `put()` to `AbstractMatrix`
- Move common methods (`inverse()`,`multiply()`) and variables (`columns`, `rows`) to `AbstractMatrix`

Result

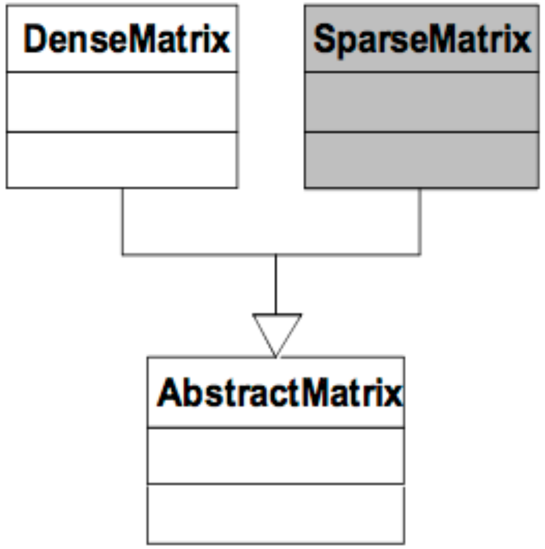
```
class AbstractMatrix {
 protected:
     int columns, rows;
 public:
     Matrix();
     void inverse();
     void multiply (AbstractMatrix& m);
     virtual int get(int x, int y);
     virtual void put(int x, int y, int value);
}
```



```
class DenseMatrix: public AbstractMatrix {
 protected:
     int elements [10000];
 public:
     DenseMatrix();
     int get(int x, int y);
     void put(int x, int y, int value);
}
```

After actualization

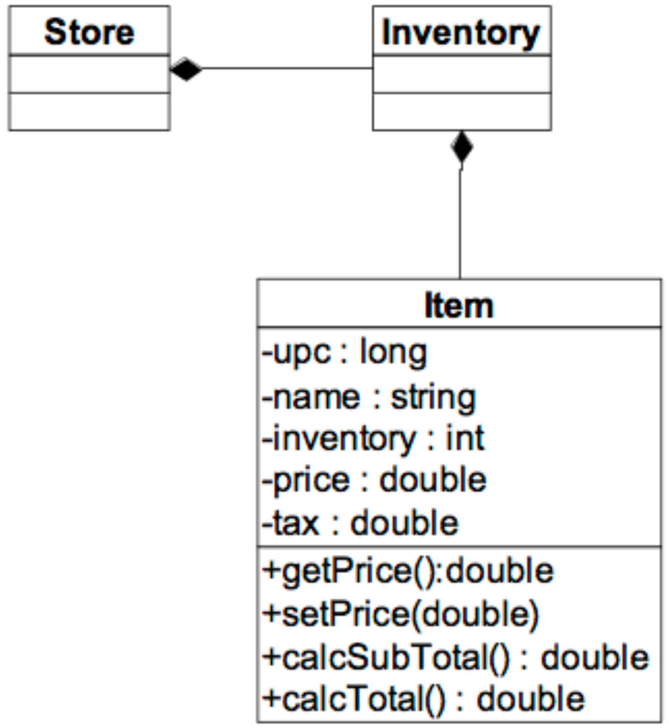
```
class AbstractMatrix {
 protected:
     int columns, rows;
 public:
     Matrix();
     void inverse();
     void multiply (AbstractMatrix& m);
     virtual int get(int x, int y);
     virtual void put(int x, int y, int value);
}
```



```
class SparseMatrix: public AbstractMatrix {
 protected:
     List elements;
 public:
     DenseMatrix();
     int get(int x, int y);
     void put(int x, int y, int value);
}
```

Extract class to make a concept more concrete

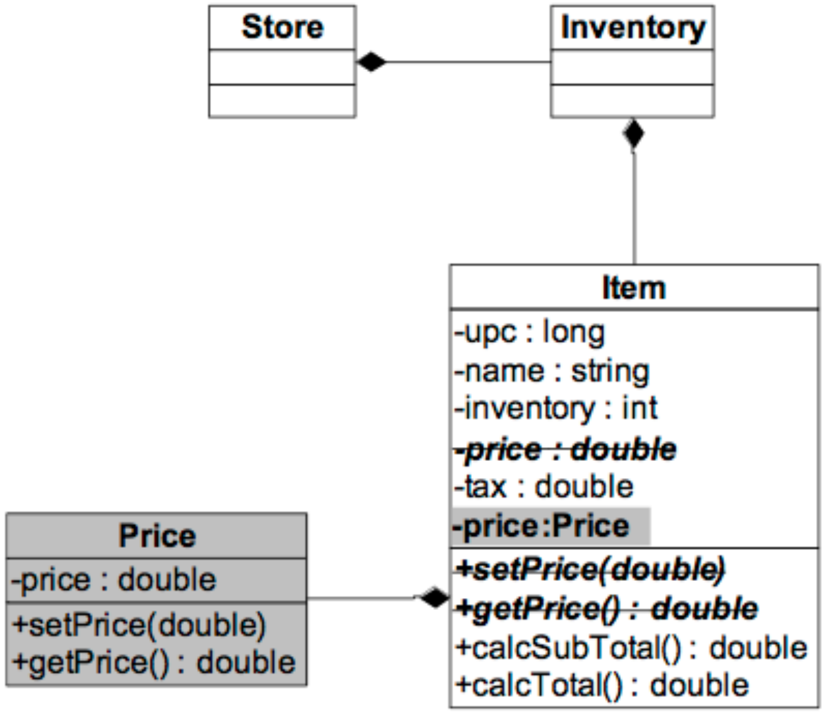
Before adding promotions, we need to extract the concept of price out of Item.



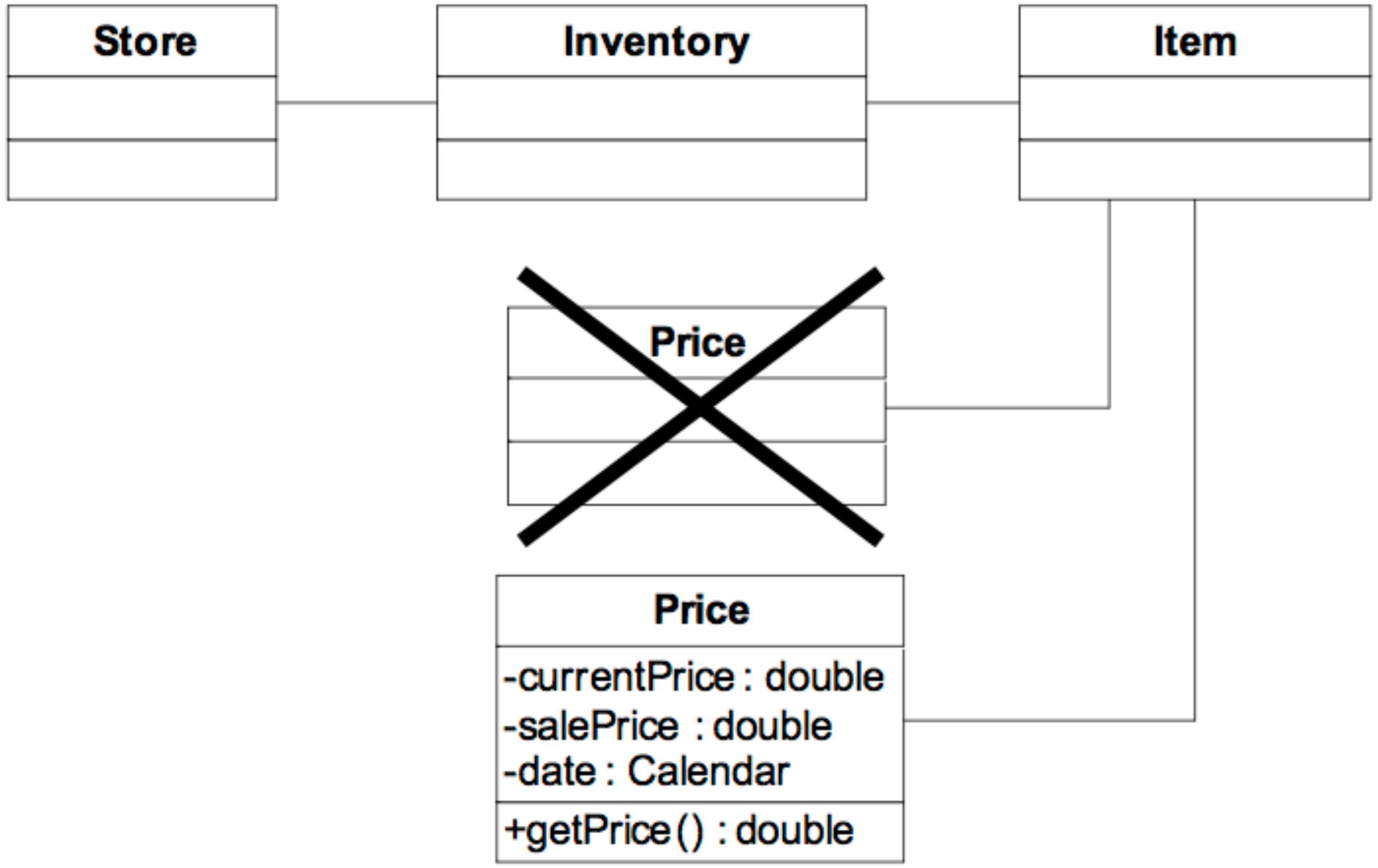
After refactoring

```
public double calcSubTotal(int numberToSell) {
  if (numberToSell < 1) {
    return 0.0;
  } else {
    ~~// return numberToSell * price;~~
    return numberToSell *  price.getPrice();
  }
}

public double calcTaxTotal(int numberToSell) {
     if (numberToSell < 1) {
         return 0.0;
    } else {
       ~~// return numberToSell * price;~~
       return numberToSell
           *  price.getPrice()
           *  (1 + tax);
    }
}
```



After actualization



A bit of postfactoring, maybe?

```
public double calcSubTotal(int numberToSell) {
     if (numberToSell < 1) {
        return 0.0;
   } else {
      return numberToSell *  price.getPrice();
   }
}

public double calcTaxTotal(int numberToSell) {
    if (numberToSell < 1) {
        return 0.0;
    } else {
      return numberToSell
          *  price.getPrice()
          *  (1 + tax);
    }
}
```

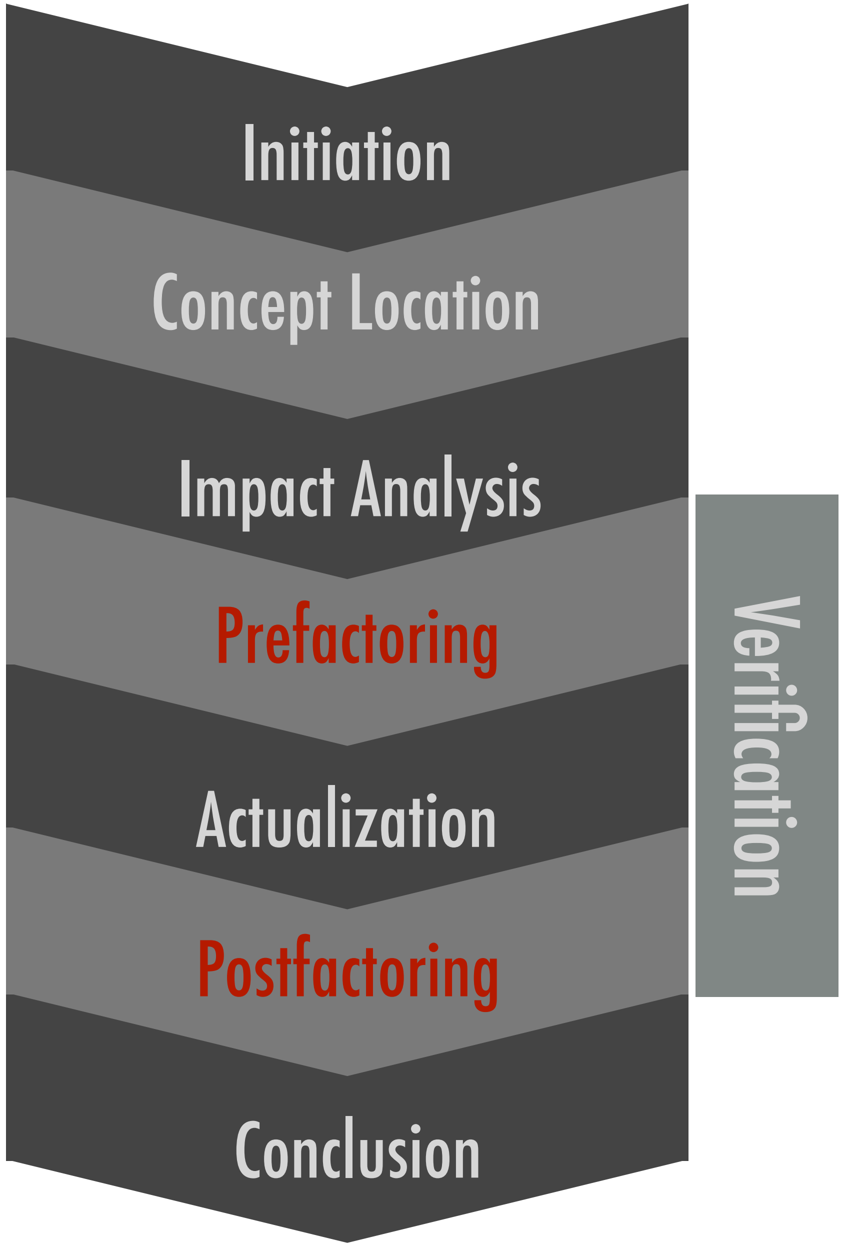
Much better

```
public double calcSubTotal(int numberToSell) {
     if (numberToSell < 1) {
        return 0.0;
   } else {
      return numberToSell *  price.getPrice();
   }
}

public double calcTaxTotal(int numberToSell) {
    return calcSubTotal() *  (1 + tax);
}
```

Conclusions

Refactoring improves the structure of code



The programmer makes the necessary changes to make actualization easier (prefactoring), and to prevent the decay caused by the change (postfactoring).

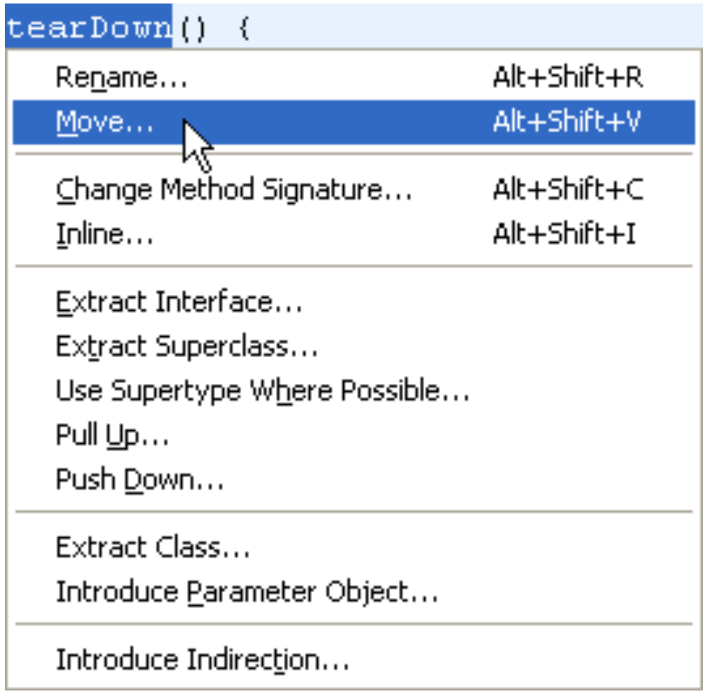
Refactoring is a key technique to prevent code decay

Use it constantly! Watch out for code smells; when needed, remove them with refactorings.

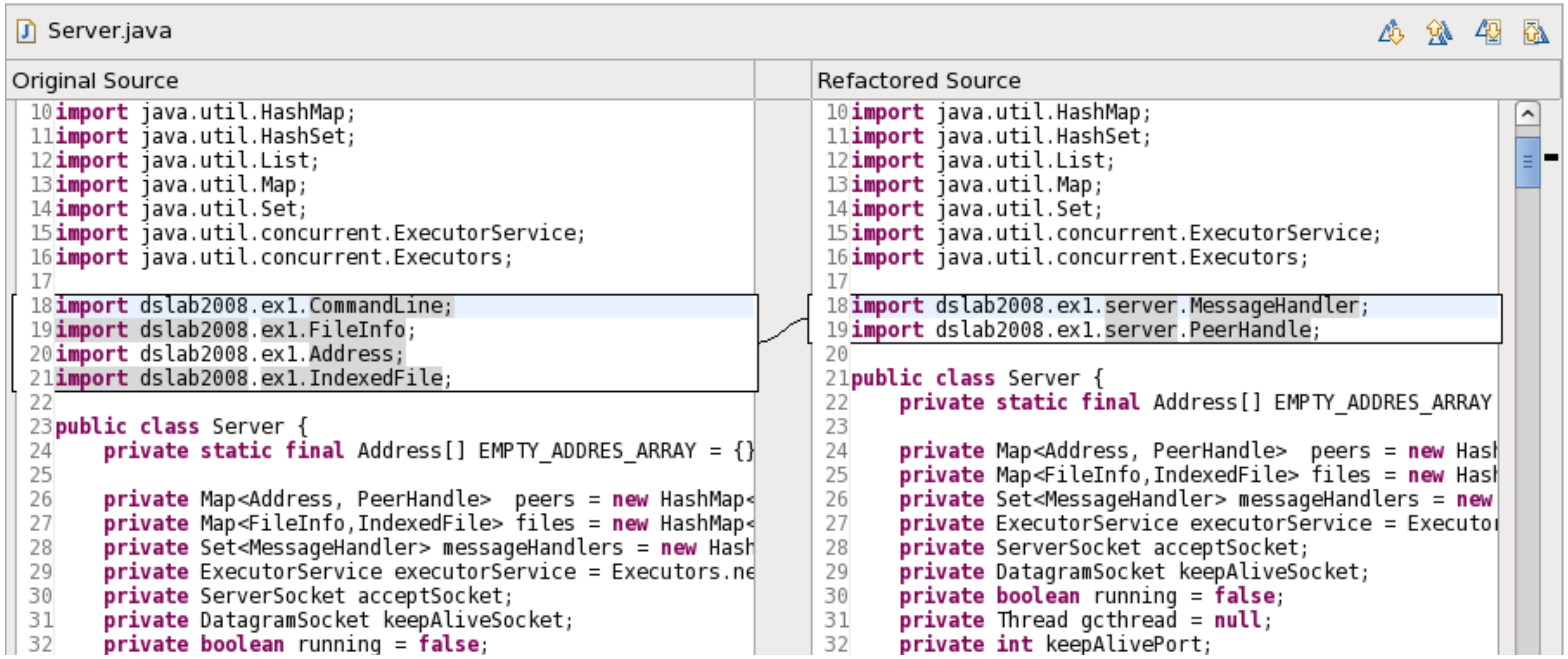
Use refactorings to make concepts more explicit before applying software changes (prefactoring).

Use refactorings to cleanup after the changes (postfactoring).

Using IDEs, refactorings can become second-nature



Fully automatic (safe!) refactorings, with previews


Server.java

Original Source

```
import java.util.HashMap;
import java.util.HashSet;
import java.util.List;
import java.util.Map;
import java.util.Set;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

import dslab2008.ex1.CommandLine;
import dslab2008.ex1.FileInfo;
import dslab2008.ex1.Address;
import dslab2008.ex1.IndexedFile;

public class Server {
    private static final Address[] EMPTY_ADDRES_ARRAY = {}

    private Map<Address, PeerHandle>  peers = new HashMap<
    private Map<FileInfo,IndexedFile> files = new HashMap<
    private Set<MessageHandler> messageHandlers = new Hash
    private ExecutorService executorService = Executors.ne
    private ServerSocket acceptSocket;
    private DatagramSocket keepAliveSocket;
    private boolean running = false;
```

Refactored Source

```
import java.util.HashMap;
import java.util.HashSet;
import java.util.List;
import java.util.Map;
import java.util.Set;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

import dslab2008.ex1.server.MessageHandler;
import dslab2008.ex1.server.PeerHandle;

public class Server {
    private static final Address[] EMPTY_ADDRES_ARRAY

    private Map<Address, PeerHandle>  peers = new Hash
    private Map<FileInfo,IndexedFile> files = new Hash
    private Set<MessageHandler> messageHandlers = new
    private ExecutorService executorService = Executo
    private ServerSocket acceptSocket;
    private DatagramSocket keepAliveSocket;
    private boolean running = false;
    private Thread gcthread = null;
    private int keepAlivePort;
```