

# **DESIGN PATTERNS**

**COURSE 9**

# PREVIOUS COURSE

- ☐ Behavior patterns
- ☐ Other patterns

# CURRENT COURSE

- ❑ **Refactoring**

- ❑ Way refactoring

- ❑ Some refactoring examples

# SOFTWARE EVOLUTION

- ❑ **Problem: You need to modify existing code**

- ❑ extend/adapt/correct/...

- ❑ **(Bad) Solution:**

- ❑ Just add new features

- ❑ Consequence:

- ❑ Design decays
    - ❑ Duplicated code
    - ❑ Long methods / classes , ...

- ❑ **(Good) Solution:**

- ❑ First make code simpler => Refactor
  - ❑ Add new features

- ❑ Consequence:

- ❑ Code stays simple

# REFACTORING CONSIDERED HARMFUL

- ❑ **From the standpoint of a manager, refactoring can appear to be dangerous!**
  - ❑ If my developers spend their time “cleaning up the code” then that's less time implementing required functionality
    - ❑ ...and my schedule is slipping as it is!
- ❑ **To address these concerns, refactoring needs to be**
  - ❑ systematic
  - ❑ incremental
  - ❑ safe

# WHAT IS REFACTORING?

- ❑ **“The process of changing a software system in such a way that it does not alter the external behaviour of the code, yet improves its internal structure.”**

Martin Fowler, “Refactoring: Improving the Design of Existing Code”, Addison-Wesley, 1999.

- ❑ **“A behaviour-preserving source-to-source program transformation.”**

Don Roberts, “Practical analysis for Refactoring”, PhD Thesis, University of Illinois, 1999.

- ❑ **“A change to the system that leaves its behaviour unchanged, but enhances some non-functional quality - simplicity, flexibility, understandability, ...”**

Kent Beck, “eXtreme Programming Explain: Embrace Change”, Addison-Wesley, 2000.

# WAY TO REFACTOR?

- ☐ Refactoring **improves** the **design** of your system
- ☐ Refactoring makes your software **easier** to **understand**
  - ☐ because structure is improved
  - ☐ duplicated code is removed
  - ☐ etc.
- ☐ Refactoring helps you **find bugs**
  - ☐ because it promotes a deep understanding of the code
- ☐ Refactoring helps you **program faster**
  - ☐ because a good design enables progress
- ☐ Prevent **“design decay”**
- ☐ **Clean** up **messes** in the code
- ☐ **Simplify** the code
- ☐ **Reduce debugging** time
- ☐ Redoing things is fundamental to every creative process

# HOW TO MAKE A SAFE REFACTORING

## ☐ First, make it **systematic**

- ☐ e.g. use refactoring patterns, like the ones discussed in Fowler's book
- ☐ Follow a refactoring process

## ☐ Second, **test constantly!**

- ☐ Each time you finish a refactoring, you run your test suite to confirm that your system's functionality has stayed the same
- ☐ This assumes, you have test already!



# PREREQUISITES FOR REFACTORING

- ☐ Tests
- ☐ Coding standards
- ☐ Continuous integration
- ☐ Collective code ownership
- ☐ **Pair programming**
- ☐ Simple design

# THE REFACTORING PROCESS

- ❑ **When you systematically apply refactoring, you wear two hats**
  - ❑ add functionality
  - ❑ refactoring
- ❑ **Don't try to clean the code when doing the former**
- ❑ **Don't try to add features when doing the latter**
- ❑ **Refactoring is not just arbitrary restructuring**
  - ❑ Code must still work
  - ❑ Small steps only so the semantics are preserved (i.e. not a major re-write)
  - ❑ Unit tests to prove the code still works
  - ❑ Code is
    - ❑ More loosely coupled
    - ❑ More cohesive modules
    - ❑ More comprehensible

# WHEN TO REFACTOR

## ☐ You should refactor

- ☐ **Any time** that you see a better way to do things
  - ☐ “Better” means making the code easier to understand and to modify in the future
- ☐ You can do so **without breaking** the **code**
  - ☐ Unit tests are essential for this

## ☐ You should not refactor

- ☐ Stable code that won't need to change
- ☐ Someone else's code
- ☐ Unless the other person agrees to it or it belongs to you
  - ☐ Not an issue in Agile Programming since code is common

# WHEN TO REFACTOR

## ☐ When should you refactor?

- ☐ Any time you find that you can **improve the design** of existing code
- ☐ You detect a **“bad smell”** (an indication that something is wrong) in the code

## ☐ When can you refactor?

- ☐ You should be in a supportive environment (agile programming team, or doing your own work)
- ☐ You are familiar with common refactoring
- ☐ Refactoring tools also help
- ☐ You should have an adequate set of unit tests

# WHAT TO REFACTOR?

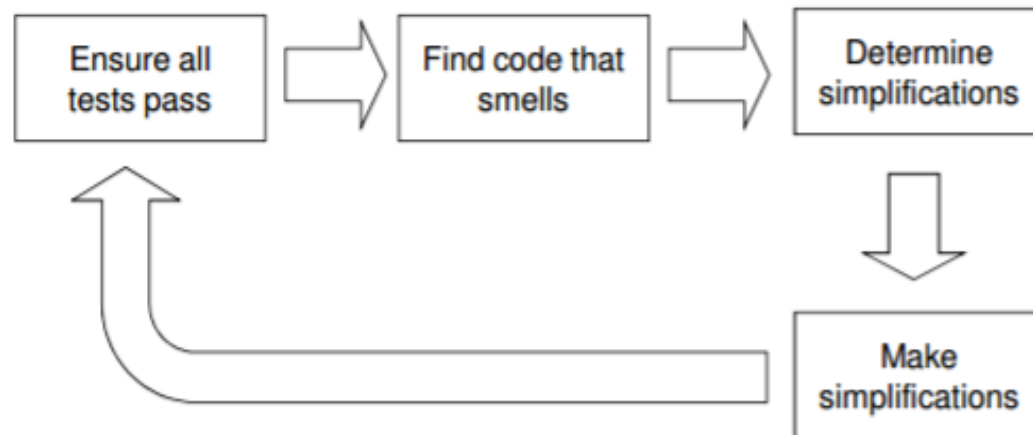
- ☐ **Make sure your tests pass**
- ☐ **Find some code that “smells”**
- ☐ **Determine how to simplify this code**
- ☐ **Make the simplifications**
- ☐ **Run tests to ensure things still work correctly**
  - ☐ You eventually have to adapt your tests
- ☐ **Repeat the simplify/test cycle until the smell is gone**

# REFACTORING STEPS

- ❑ **Save / backup / checking** the code before you mess with it.
  - ❑ If you use a well-managed version control repo, this is done.
- ❑ **Write unit tests** that verify the code's external correctness.
  - ❑ They should pass on the current poorly designed code.
  - ❑ Having unit tests helps make sure any refactor doesn't break existing behavior (regressions).
- ❑ **Analyze** the code to decide the **risks** and **benefits** of refactoring.
  - ❑ If it is too risky, not enough time remains, or the refactor will not produce enough benefit to the project, don't do it.

# REFACTORING PROCESS

- ❑ Make a **small change**
  - ❑ a single refactoring
- ❑ Run all the tests to ensure everything still works
- ❑ If everything works, move on to the next refactoring
- ❑ If not, fix the problem, or undo the change, so you still have a working system



# PROBLEMS WITH REFACTORING

- ❑ **Taken too far**, refactoring can lead to incessant tinkering with the code, trying to make it perfect
- ❑ Refactoring code when the **tests don't work** or tests when the application doesn't work leads to potentially dangerous situations
- ❑ **Databases** can be **difficult** to refactor
  - ❑ code is easy to change; databases are not
- ❑ **Refactoring published interfaces** can cause **problems** for the **code that uses those interfaces**



# WHY (SOME) DEVELOPERS DON'T LIKE IT

- ☐ Lack of understanding
- ☐ Short-term focus
- ☐ Not paid for overhead tasks like refactoring
- ☐ Fear of breaking current program

# CODE SMELLS EXAMPLES

## ☐ If it stinks, change it

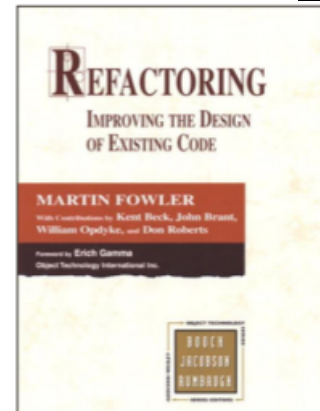
- ☐ Code that can make the design harder to change

## ☐ Examples:

- ☐ Duplicate code
- ☐ Long methods
- ☐ Big classes
- ☐ Big switch statements
- ☐ Long navigations (e.g., `a.b().c().d()`)
- ☐ Lots of checking for null objects
- ☐ Data clumps (e.g., a Contact class that has fields for address, phone, email etc.) – similar to non-normalized tables in relational design
- ☐ Data classes (classes that have mainly fields/properties and little or no methods)
- ☐ Un-encapsulated fields (public member variables)

# SOME TYPES OF REFACTORING

- ☐ refactoring to fit design patterns
- ☐ renaming (methods, variables)
- ☐ extracting code into a method or module
- ☐ splitting one method into several to improve cohesion and readability
- ☐ changing method signatures
- ☐ performance optimization
- ☐ moving statements that semantically belong together near each other
- ☐ naming (extracting) "magic" constants
- ☐ exchanging idioms that are risky with safer alternatives
- ☐ clarifying a statement that has evolved over time or is unclear
- ☐ See also <http://www.refactoring.org/catalog/>



# HOW TO REFACTOR

## ❑ Manualy

## ❑ Refactoring tool

- ❑ Eclipse (and some other IDEs) provide significant support for refactoring

Refactor	
Rename...	⌘R
Move...	⌘V
Change Method Signature...	⌘C
Extract Method...	⌘M
Extract Local Variable...	⌘L
Extract Constant...	
Inline...	⌘I
Convert Anonymous Class to Nested...	
Convert Member Type to Top Level	
Convert Local Variable to Field...	
Extract Superclass...	
Extract Interface...	
Use Supertype Where Possible...	
Push Down...	
Pull Up...	
Introduce Indirection...	
Introduce Factory...	
Introduce Parameter...	
Encapsulate Field...	
Generalize Declared Type...	
Infer Generic Type Arguments...	
Migrate JAR File...	
Create Script...	
Apply Script...	
History...	

# EXTRACT METHOD

- ❑ You have a code fragment that can be grouped together.
- ❑ Turn the fragment into a method whose name explains the purpose of the method.
- ❑ Inverse of **Inline Method**

```
void printOwing() {  
    printBanner();  
    //print details  
    System.out.println ("name: " + _name);  
    System.out.println("amount " + getOutstanding());  
}
```

```
void printOwing() {  
    printBanner();  
    printDetails(getOutstanding());  
}  
void printDetails (double outstanding) {  
    System.out.println ("name: " + _name);  
    System.out.println ("amount " + outstanding);  
}
```

# INLINE METHOD

- ❑ A method's body is just as clear as its name.
- ❑ Put the method's body into the body of its callers and remove the method.
- ❑ Inverse of **Exact Method**

```
int getRating() {  
    return (moreThanFiveLateDeliveries()) ? 2 : 1;  
}  
boolean moreThanFiveLateDeliveries() {  
    return _numberOfLateDeliveries > 5;  
}
```

```
int getRating() {  
    return (_numberOfLateDeliveries > 5) ? 2 : 1;  
}
```

# RENAME METHOD

- ❑ The name of a method does not reveal its purpose.
- ❑ Change the name of the method.

```
class Customer {  
    double getInvcdtlmt();  
}
```

```
class Customer {  
    double getInvoiceableCreditLimit();  
}
```

# REMOVE PARAMETER

- ☐ A parameter is no longer used by the method body.
- ☐ Remove it.
- ☐ inverse of Add Parameter
- ☐ Naming: In IDEs this refactoring is usually done as part of "Change Method Signature"

Customer getContact(Date)

Customer getContact()



# ADD PARAMETER

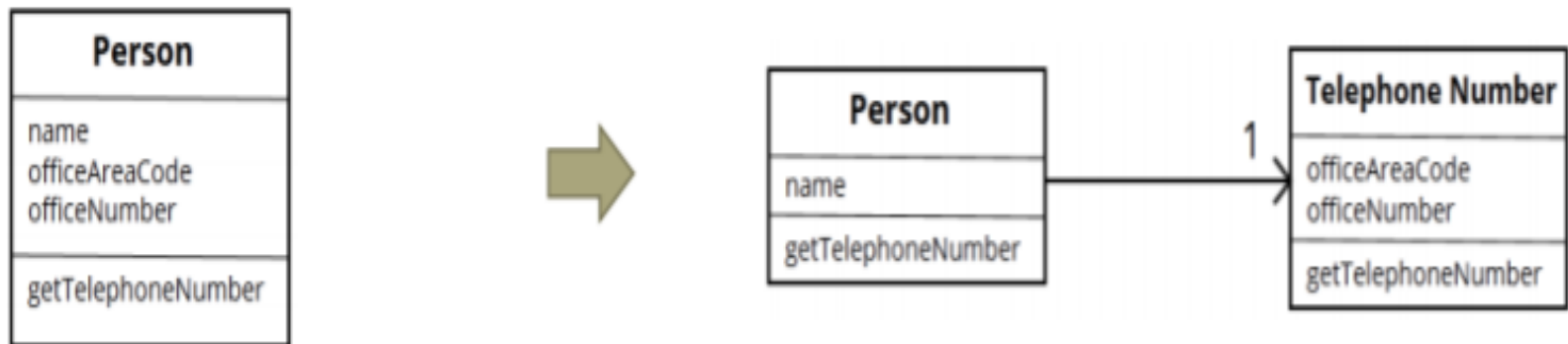
- ❑ A method needs more information from its caller.
- ❑ Add a parameter for an object that can pass on this information.
- ❑ Inverse of **Remove Parameter**
- ❑ Naming: In IDEs this refactoring is usually done as part of "Change Method Signature"

Customer getContact()

Customer getContact(Date data)

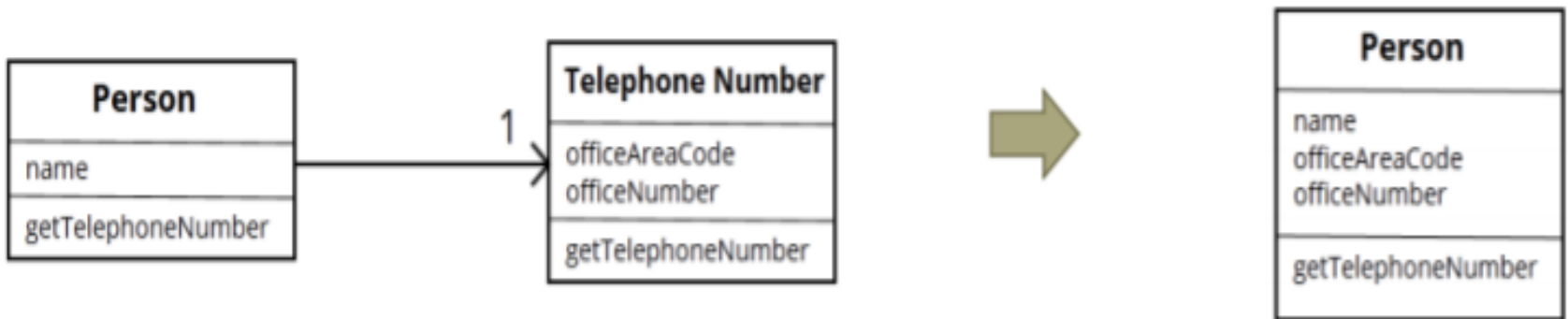
# EXTRACT CLASS

- ❑ You have one class doing work that should be done by two.
- ❑ Create a new class and move the relevant fields and methods from the old class into the new class.
- ❑ Inverse of **Inline Class**



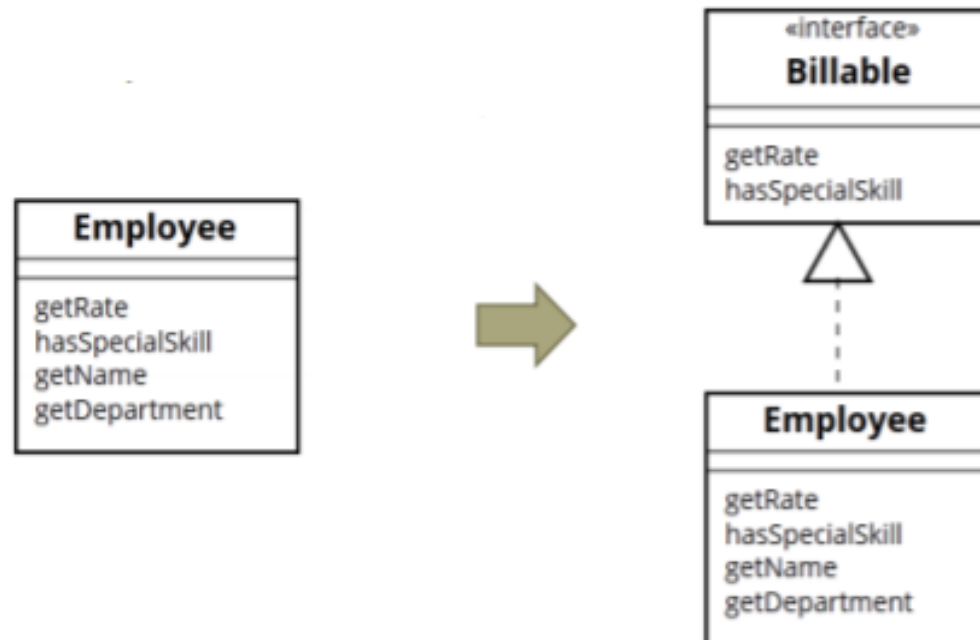
# INLINE CLASS

- ❑ A class isn't doing very much.
- ❑ Move all its features into another class and delete it.
- ❑ Inverse of **Extract Class**, **Extract Interface**



# EXTRACT INTERFACE

- ❑ Several clients use the same subset of a class's interface, or two classes have part of their interfaces in common.
- ❑ Extract the subset into an interface.
- ❑ Inverse of **Inline Class**



# REPLACE ERROR CODE WITH AN EXCEPTION

- ❑ A method returns a special code to indicate an error.
- ❑ Throw an exception instead.

```
int withdraw(int amount) {  
    if (amount > _balance)  
        return -1;  
    else  
        _balance -= amount; return 0;  
}
```

```
void withdraw(int amount) throws BalanceException {  
    if (amount > _balance)  
        throw new BalanceException();  
    _balance -= amount;  
}
```

# REPLACE EXCEPTION WITH TEST

- ❑ You are throwing an exception on a condition the caller could have checked first.
- ❑ Change the caller to make the test first.

```
double getValueForPeriod (int periodNumber) {  
    try {  
        return _values[periodNumber];  
    } catch (ArrayIndexOutOfBoundsException e) {  
        return 0;  
    }  
}
```

```
double getValueForPeriod (int periodNumber) {  
    if (periodNumber >= _values.length)  
        return 0;  
    return _values[periodNumber];  
}
```

# CONSOLIDATE CONDITIONAL EXPRESSION

- ❑ You have a sequence of conditional tests with the same result.
- ❑ Combine them into a single conditional expression and extract it.

```
double disabilityAmount() {  
    if (_seniority < 2) return 0;  
    if (_monthsDisabled > 12) return 0;  
    if (_isPartTime) return 0;  
    // compute the disability amount
```

```
double disabilityAmount() {  
    if (isNotEligibleForDisability()) return 0;  
    // compute the disability amount
```

# CONSOLIDATE DUPLICATE CONDITIONAL FRAGMENTS

- ❑ The same fragment of code is in all branches of a conditional expression.
- ❑ Move it outside of the expression.

```
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();
```



# REPLACE ARRAY WITH OBJECT

- ❑ You have an array in which certain elements mean different things.
- ❑ Replace the array with an object that has a field for each element.

```
String[] row = new String[3];  
row [0] = "Liverpool";  
row [1] = "15";
```

```
Performance row = new Performance();  
row.setName("Liverpool");  
row.setWins("15");
```

# REPLACE TYPE CODE WITH POLYMORPHISM

- ❑ You have a **type code** that **affects** the **behavior** of a **class**.
- ❑ Replace the type code with classes: one for each type code variant

Mountain Bike
type_code
off_road_ability price



Rigid Mountain Bike
off_road_ability price

Front Suspension Mountain Bike
off_road_ability price

Full Suspension Mountain Bike
off_road_ability price

# OBSTACLES TO REFACTORING

## ☐ Complexity

- ☐ Changing design is hard
- ☐ Understanding code is hard

## ☐ Possibility to introduce errors

- ☐ Run tests if possible
- ☐ Build tests

## ☐ Cultural Issues

- ☐ “We pay you to add new features, not to improve the code!”

## ☐ Performance issue

- ☐ Refactoring may slow down the execution

## ☐ Normally only 10% of your system consumes 90% of the resources so just focus on 10 %.

- ☐ Refactorings help to localize the part that need change
- ☐ Refactorings help to concentrate the optimizations

## ☐ Development is always under time pressure

- ☐ Refactoring takes time
- ☐ Refactoring better after delivery

# SUMMARY

- ❑ “The process of changing a software system in such a way that it does not alter the external behavior of the code, yet improves its internal structure” [Fowler]
- ❑ Refactor to
  - ❑ **Improve** the software design
  - ❑ Make the software **easier** to **understand**
  - ❑ Help **find bugs**
- ❑ A catalog of refactoring exists: Extract Method, Move Method, Replace Temp with Query, etc...
- ❑ Refactoring has some obstacles

# NEXT COURSE

**Anti-patterns**