



# Chapter 9: Software Refactoring

## 9.2 Refactoring Techniques and Tool Support

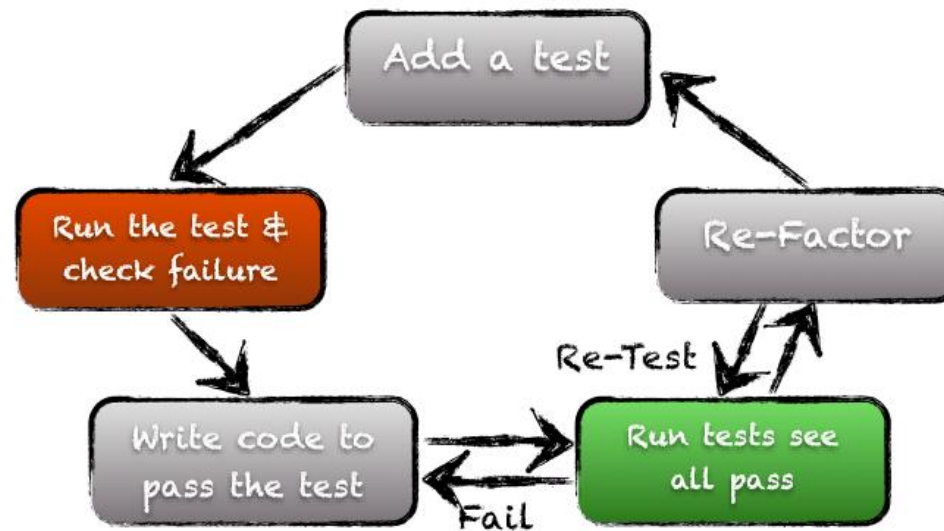
Xu Hanchuan

[xhc@hit.edu.cn](mailto:xhc@hit.edu.cn)

May 20, 2019

# Outline

- Several examples of refactoring
- Catalog of refactoring
- Refactoring patterns
- IDE support for refactoring
- Summary





# 1 Several examples of refactoring



# Smelly Example #1

```
public static int dayOfYear(int month, int dayOfMonth, int year) {
    if (month == 2) {
        dayOfMonth += 31;
    } else if (month == 3) {
        dayOfMonth += 59;
    } else if (month == 4) {
        dayOfMonth += 90;
    } else if (month == 5) {
        dayOfMonth += 31 + 28 + 31 + 30;
    } else if (month == 6) {
        dayOfMonth += 31 + 28 + 31 + 30 + 31;
    } else if (month == 7) {
        dayOfMonth += 31 + 28 + 31 + 30 + 31 + 30;
    } else if (month == 8) {
        dayOfMonth += 31 + 28 + 31 + 30 + 31 + 30 + 31;
    } else if (month == 9) {
        dayOfMonth += 31 + 28 + 31 + 30 + 31 + 30 + 31 + 31;
    } else if (month == 10) {
        dayOfMonth += 31 + 28 + 31 + 30 + 31 + 30 + 31 + 31 + 30;
    } else if (month == 11) {
        dayOfMonth += 31 + 28 + 31 + 30 + 31 + 30 + 31 + 31 + 30 + 31;
    } else if (month == 12) {
        dayOfMonth += 31 + 28 + 31 + 30 + 31 + 30 + 31 + 31 + 30 + 31 + 31;
    }
    return dayOfMonth;
}
```

**Why is this  
code so bad?**

# Don't Repeat Yourself (DRY)

Recall Chapter 5 for  
Reusability and Chapter 7  
for Robustness

- **Duplicated code is a risk to safety.**
  - If you have identical or very similar code in two places, then the fundamental risk is that there's a bug in both copies, and some maintainer fixes the bug in one place but not the other.
- **Avoid duplication like you'd avoid crossing the street without looking.**
  - Copy-and-paste is an enormously tempting programming tool, and you should feel a frisson of danger run down your spine every time you use it.
  - The longer the block you're copying, the riskier it is.
- **Don't Repeat Yourself (DRY), has become a programmer's mantra.**
- **The dayOfYear example is full of identical code. How would you DRY it out?**

# Comments Where Needed

Specifications document assumptions --- Recall Chapter 3 for ADT and Chapter 4 for Understandability

- **Good software developers write comments in their code, and do it judiciously.**
  - Good comments should make code easier to understand, safer from bugs (important assumptions have been documented), and ready for change.
- **One kind of crucial comment is a specification, which appears above a method or above a class and documents the behavior of the method or class.**
  - In Java, this is conventionally written as a Javadoc comment, starting with `/**` and including `@`-syntax, like `@param` and `@return` for methods.

```
/**
 * Compute the hailstone sequence.
 * See http://en.wikipedia.org/wiki/Collatz\_conjecture#Statement\_of\_the\_problem
 * @param n starting number of sequence; requires n > 0.
 * @return the hailstone sequence starting at n and ending with 1.
 *         For example, hailstone(3)=[3,10,5,16,8,4,2,1].
 */
public static List<Integer> hailstoneSequence(int n) {
    ...
}
```

# Comments Where Needed

- Another crucial comment is one that specifies the provenance or source of a piece of code that was copied or adapted from elsewhere. This is vitally important for practicing software developers.

```
// read a web page into a string
// see http://stackoverflow.com/questions/4328711/read-url-to-string-in-few-lines-of-java-code
String mitHomepage = new Scanner(new URL("http://www.mit.edu").openStream(), "UTF-8").useDelimiter("\\A").next();
```

- One reason for documenting sources is to avoid violations of copyright.
  - Small snippets of code on Stack Overflow are typically in the public domain, but code copied from other sources may be proprietary or covered by other kinds of open source licenses, which are more restrictive.
- Another reason for documenting sources is that the code can fall out of date; the Stack Overflow answer from which this code came has evolved significantly in the years since it was first answered.

# Fail Fast

Recall Chapter 7 for Robustness

- **Failing fast means that code should reveal its bugs as early as possible.**
  - The earlier a problem is observed (the closer to its cause), the easier it is to find and fix.
  - Static checking fails faster than dynamic checking, and dynamic checking fails faster than producing a wrong answer that may corrupt subsequent computation.
- **The dayOfYear function doesn't fail fast — if you pass it the arguments in the wrong order, it will quietly return the wrong answer.**
  - In fact, the way dayOfYear is designed, it's highly likely that a non-American will pass the arguments in the wrong order!
  - It needs more checking — either static checking or dynamic checking.



# Avoid Magic Numbers

- There are really only two constants that computer scientists recognize as valid in and of themselves: 0, 1, and maybe 2.
- All other constants are called magic because they appear as if out of thin air with no explanation.
- One way to explain a number is with a comment, but a far better way is to declare the number as a named constant with a good, clear name.
- `dayOfYear()` is full of magic numbers:
  - The months 2, ..., 12
  - The days-of-months 30, 31, 28
  - The mysterious numbers 59 and 90 (Don't hardcode constants that you've computed by hand)

# One Purpose For Each Variable

- **In the `dayOfYear()` example, the parameter `dayOfMonth` is reused to compute a very different value — the return value of the function, which is not the day of the month.**
- **Don't reuse parameters, and don't reuse variables.**
  - Variables are not a scarce resource in programming. Introduce them freely, give them good names, and just stop using them when you stop needing them.
  - You will confuse your reader if a variable that used to mean one thing suddenly starts meaning something different a few lines down.
- **Not only is this an ease-of-understanding question, but it's also a safety-from-bugs and ready-for-change question.**

# One Purpose For Each Variable

- **Method parameters, in particular, should generally be left unmodified.**
  - This is important for being ready-for-change — in the future, some other part of the method may want to know what the original parameters of the method were, so you shouldn't blow them away while you're computing.
  - It's a good idea to use **final** for method parameters, and as many other variables as you can. The **final** keyword says that the variable should never be reassigned, and the Java compiler will check it statically.

```
public static int dayOfYear(final int month, final int dayOfMonth, final int year) {  
    ...  
}
```

# Summary: how to refactoring this code?

```
public static int dayOfYear(int month, int dayOfMonth, int year) {
    if (month == 2) {
        dayOfMonth += 31;
    } else if (month == 3) {
        dayOfMonth += 59;
    } else if (month == 4) {
        dayOfMonth += 90;
    } else if (month == 5) {
        dayOfMonth += 31 + 28 + 31 + 30;
    } else if (month == 6) {
        dayOfMonth += 31 + 28 + 31 + 30 + 31;
    } else if (month == 7) {
        dayOfMonth += 31 + 28 + 31 + 30 + 31 + 30;
    } else if (month == 8) {
        dayOfMonth += 31 + 28 + 31 + 30 + 31 + 30 + 31;
    } else if (month == 9) {
        dayOfMonth += 31 + 28 + 31 + 30 + 31 + 30 + 31 + 31;
    } else if (month == 10) {
        dayOfMonth += 31 + 28 + 31 + 30 + 31 + 30 + 31 + 31 + 30;
    } else if (month == 11) {
        dayOfMonth += 31 + 28 + 31 + 30 + 31 + 30 + 31 + 31 + 30 + 31;
    } else if (month == 12) {
        dayOfMonth += 31 + 28 + 31 + 30 + 31 + 30 + 31 + 31 + 30 + 31 + 31;
    }
    return dayOfMonth;
}
```

## Smelly Example #2

- There was a latent bug in `dayOfYear()`: It didn't handle leap years at all. Suppose we write a leap-year method.

```
public static boolean leap(int y) {
    String tmp = String.valueOf(y);
    if (tmp.charAt(2) == '1' || tmp.charAt(2) == '3' || tmp.charAt(2) == 5 || tmp.charA
t(2) == '7' || tmp.charAt(2) == '9') {
        if (tmp.charAt(3)=='2' || tmp.charAt(3)=='6') return true; /*R1*/
        else
            return false; /*R2*/
    }else{
        if (tmp.charAt(2) == '0' && tmp.charAt(3) == '0') {
            return false; /*R3*/
        }
        if (tmp.charAt(3)=='0' || tmp.charAt(3)=='4' || tmp.charAt(3)=='8')return true; /*R
4*/
    }
    return false; /*R5*/
}
```

What are the bugs hidden in this code?  
What style problems that we've already talked about?

# Use Good Names

Recall Chapter 4 for Understandability

- Good method and variable names are long and self-descriptive.
- Comments can often be avoided entirely by making the code itself more readable, with better names that describe the methods and variables.

```
int tmp = 86400;  
    //tmp is the number of seconds in a day  
-----  
int secondsPerDay = 86400;
```

- In general, variable names like `tmp`, `temp`, and `data` are awful, symptoms of extreme programmer laziness. Every local variable is temporary, and every variable is data, so those names are generally meaningless. Better to use a longer, more descriptive name, so that your code reads clearly all by itself.

# Use Good Names

Recall Chapter 4 for Understandability

- **Follow the lexical naming conventions of the language.**
- **In Java:**
  - `methodsAreNamedWithCamelCaseLikeThis`
  - `variablesAreAlsoCamelCase`
  - `CONSTANTS_ARE_IN_ALL_CAPS_WITH_UNDERSCORES`
  - `ClassesAreCapitalized`
  - `packages.are.lowercase.and.separated.by.dots`
- **Method names are usually verb phrases, like `getDate` or `isUpperCase`, while variable and class names are usually noun phrases.**
- **Choose short words, and be concise, but avoid abbreviations.**
  - `message` is clearer than `msg`, and `word` is so much better than `wd`.

# Use Whitespace to Help the Reader

- **Use consistent indentation.**
- **Put spaces within code lines to make them easy to read.**
- **Never use tab characters for indentation, only space characters.**
  - The reason for this rule is that different tools treat tab characters differently — sometimes expanding them to 4 spaces, sometimes to 2 spaces, sometimes to 8.
  - Always set your programming editor to insert space characters when you press the Tab key.



# Let's refactor this code

```

public static boolean leap(int y) {
    String tmp = String.valueOf(y);
    if (tmp.charAt(2) == '1' || tmp.charAt(2) == '3' || tmp.charAt(2) == 5 || tmp.charA
t(2) == '7' || tmp.charAt(2) == '9') {
        if (tmp.charAt(3) == '2' || tmp.charAt(3) == '6') return true; /*R1*/
        else
            return false; /*R2*/
    } else {
        if (tmp.charAt(2) == '0' && tmp.charAt(3) == '0') {
            return false; /*R3*/
        }
        if (tmp.charAt(3) == '0' || tmp.charAt(3) == '4' || tmp.charAt(3) == '8') return true; /*R
4*/
    }
    return false; /*R5*/
}

```

## Smelly Example #3

```
public static int LONG_WORD_LENGTH = 5;
public static String longestWord;

public static void countLongWords(List<String> words) {
    int n = 0;
    longestWord = "";
    for (String word: words) {
        if (word.length() > LONG_WORD_LENGTH) ++n;
        if (word.length() > longestWord.length()) longestWord = word;
    }
    System.out.println(n);
}
```

# Don't Use Global Variables

- Avoid global variables.
- A global variable is:
  - a *variable* , a name whose meaning can be changed
  - that is *global* , accessible and changeable from anywhere in the program.
- In Java, a global variable is declared **public static**. The **public** modifier makes it accessible anywhere, and **static** means there is a single instance of the variable.
- In general, change global variables into parameters and return values, or put them inside objects that you're calling methods on.

# Methods should return results, not print them

- **countLongWords( ) isn't ready for change.**
  - It sends some of its result to the console, `System.out` .
  - That means that if you want to use it in another context — where the number is needed for some other purpose, like computation rather than human eyes — it would have to be rewritten.
- **In general, only the highest-level parts of a program should interact with the human user or the console.**
- **Lower-level parts should take their input as parameters and return their output as results.**
- **The sole exception here is debugging output, which can of course be printed to the console. But that kind of output shouldn't be a part of your design, only a part of how you debug your design.**

# Let's refactor this code

```
public static int LONG_WORD_LENGTH = 5;
public static String longestWord;

public static void countLongWords(List<String> words) {
    int n = 0;
    longestWord = "";
    for (String word: words) {
        if (word.length() > LONG_WORD_LENGTH) ++n;
        if (word.length() > longestWord.length()) longestWord = word;
    }
    System.out.println(n);
}
```



## 2 Catalog of refactoring




# 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

<https://www.refactoring.com/catalog/>

# Classification of refactoring

- 
- **Composing Methods:** to package code properly for methods that are too long.
  - **Moving Features Between Objects:** to decide where to put responsibilities.
  - **Organizing Data:** to make working with data easier.
  - **Simplifying Conditional Expressions**
  - **Making Method Calls Simpler**
  - **Dealing with Generalization**
  - **Big Refactorings**



# Refactoring patterns

## ■ Composing Methods

- 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

## ■ Moving Features Between Objects

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

# Refactoring patterns

## ■ Organizing Data

- 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

## ■ Simplifying Conditional Expressions

- 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

# Refactoring patterns

## ■ Making Method Calls Simpler

- Rename Method
- Add/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

# Refactoring patterns

## ■ Dealing with Generalization

- Pull Up Field; Method; Constructor Body
- Push Down Method; Push Down Field
- Extract Subclass; Superclass; Interface
- Collapse Hierarchy
- Form Template Method
- Replace Inheritance with Delegation (or vice versa)

## ■ Big Refactorings

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



# 3 Refactoring patterns



# Encapsulate Field

There is a public field.

*Make it private and provide accessors.*

- Un-encapsulated data is a no-no in OO application design.
- Use property get and set procedures to provide public access to private (encapsulated) member variables.

```
public class Course {  
    public List students;  
}
```

```
int classSize =  
    course.students.size();
```



```
public class Course {  
    private List students;  
    public List getStudents() {  
        return students;  
    }  
    public void setStudents(List s) {  
        students = s;  
    }  
}
```

```
int classSize =  
    course.getStudents().size();
```

# 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.*

- Break one class into two, e.g. having the phone details as part of the Customer class is not a realistic OO model, and also breaks the Single Responsibility design principle.
- We can refactor this into two separate classes, each with the appropriate responsibility.

```
public class Customer {  
    private String name;  
    private String workPhoneAreaCode;  
    private String workPhoneNumber;  
}
```



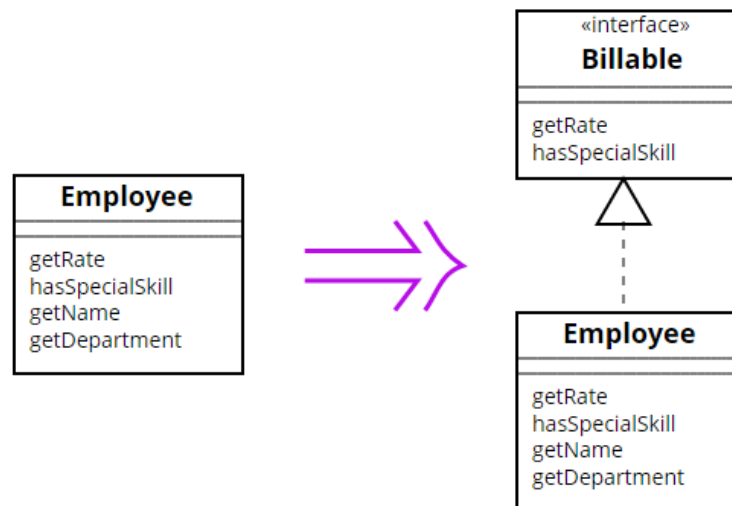
```
public class Customer {  
    private String name;  
    private Phone workPhone;  
}  
  
public class Phone {  
    private String areaCode;  
    private String number;  
}
```

# 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.*

- Extract an interface from a class.
- Some clients may need to know a Customer's name, while others may only need to know that certain objects can be serialized to XML.
- Having toXml() as part of the Customer interface breaks the Interface Segregation design principle which tells us that it's better to have more specialized interfaces than to have one multi-purpose interface.





# Extract Interface

```
public class Customer {
    private String name;

    public String getName(){
        return name;
    }

    public void setName(String string) {
        name = string;
    }

    public String toXML(){
        return "<Customer><Name>" +
            name + "</Name></Customer>";
    }
}
```



```
public class Customer implements SerXML{
    private String name;

    public String getName(){
        return name;
    }

    public void setName(String string) {
        name = string;
    }

    public String toXML(){
        return "<Customer><Name>" +
            name + "</Name></Customer>";
    }
}
```

```
public interface SerXml {
    String toXML();
}
```

# 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.*

- Sometimes we have methods that do too much. The more code in a single method, the harder it is to understand and get right. It also means that logic embedded in that method cannot be reused elsewhere.
- The Extract Method refactoring is one of the most useful for reducing the amount of duplication in code.

```
public class Customer {  
    void int foo() {  
        ...  
        // Compute score  
        score = a*b+c;  
        score *= xfactor;  
    }  
}
```



```
public class Customer {  
    void int foo(){  
        ...  
        score = ComputeScore(a,b,c,xfactor);  
    }  
  
    int ComputeScore(int a, int b,  
                     int c, int x) {  
        return (a*b+c)*x;  
    }  
}
```

# Extract Subclass

A class has features that are used only in some instances.

*Create a subclass for that subset of features.*

- When a class has features (attributes and methods) that would only be useful in specialized instances, we can create a specialization of that class and give it those features.
- This makes the original class less specialized (i.e., more abstract), and good design is about binding to abstractions wherever possible.

```
public class Person {  
    private String name;  
    private String jobTitle;  
}
```



```
public class Person {  
    protected String name;  
}  
  
public class Employee extends Person {  
    private String jobTitle;  
}
```

# Extract Super Class

You have two classes with similar features.

*Create a superclass and move the common features to the superclass.*

- When you find two or more classes that share common features, consider abstracting those shared features into a super-class.
- This makes it easier to bind clients to an abstraction, and removes duplicate code from the original classes.

```
public class Employee {  
    private String name;  
    private String jobTitle;  
}  
  
public class Student {  
    private String name;  
    private Course course;  
}
```



```
public abstract class Person {  
    protected String name;  
}  
  
public class Employee extends Person {  
    private String jobTitle;  
}  
  
public class Student extends Person {  
    private Course course;  
}
```

# Form Template Method - Before

- When you find two methods in subclasses that perform the same steps, but do different things in each step, create methods for those steps with the same signature and move the original method into the base class

```
public abstract class Party { }
```

```
public class Person extends Party  
{  
    private String firstName;  
    private String lastName;  
    private Date dob;  
    private String nationality;
```

```
    public void printNameAndDetails() {  
        System.out.println("Name: " + firstName + " " + lastName);  
        System.out.println("DOB: " + dob.toString() + ", Nationality: " + nationality);  
    }  
}
```

```
public class Company extends Party {  
    private String name;  
    private String companyType;  
    private Date incorporated;  
  
    public void PrintNameAndDetails() {  
        System.out.println("Name: " + name + " " +  
                           companyType);  
        System.out.println("Incorporated: " +  
                           incorporated.toString());  
    }  
}
```

# Form Template Method - Refactored

```
public abstract class Party {
    public void PrintNameAndDetails() {
        printName();
        printDetails();
    }
    public abstract void printName();
    public abstract void printDetails();
}
```

```
public class Person extends Party {
    private String firstName;
    private String lastName;
    private Date dob;
    private String nationality;
```

```
    public void printDetails() {
        System.out.println("DOB: " + dob.toString() + ", Nationality: " + nationality);
    }
    public void printName() {
        System.out.println("Name: " + firstName + " " + lastName);
    }
}
```

```
public class Company extends Party {
    private String name;
    private String companyType;
    private Date incorporated;
    public void printDetails() {
        System.out.println("Incorporated: " +
            incorporated.toString());
    }
    public void printName() {
        System.out.println("Name: " + name + " " +
            companyType);
    }
}
```

# Move Method

- If a method on one class uses (or is used by) another class more than the class on which its defined, move it to the other class.
- The student class now no longer needs to know about the Course interface, and the `isTaking()` method is closer to the data on which it relies - making the design of Course more cohesive and the overall design more loosely coupled

```
public class Student {  
    public boolean isTaking(Course course) {  
        return (course.getStudents().contains(this));  
    }  
}  
  
public class Course {  
    private List students;  
    public List getStudents() {  
        return students;  
    }  
}
```

```
public class Student {  
}  
  
public class Course {  
    private List students;  
    public boolean isTaking(Student student) {  
        return students.contains(student);  
    }  
}
```

# Introduce Null Object

- If relying on null for default behavior, use inheritance instead

```
public class User {  
    Plan getPlan() {  
        return plan;  
    }  
}
```

```
if (user == null)  
    plan = Plan.basic();  
else  
    plan = user.getPlan();
```



```
public class User{  
    Plan getPlan(){  
        return plan;  
    }  
}  
  
public class NullUser extends User {  
    Plan getPlan(){  
        return Plan.basic();  
    }  
}
```



# Replace Error Code with Exception

- A method returns a special code to indicate an error is better accomplished with an Exception.

```
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

- Conversely, if you are catching an exception that could be handled by an if-statement, use that instead.

```
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];  
}
```

# Nested Conditional with Guard

- A method has conditional behavior that does not make clear what the normal path of execution is. Use Guard Clauses for all the special cases.

```
double getPayAmount() {  
    double result;  
    if (isDead) result = deadAmount();  
    else {  
        if (isSeparated) result = separatedAmount();  
        else {  
            if (isRetired) result = retiredAmount();  
            else result = normalPayAmount();  
        }  
    }  
    return result;  
}
```

```
double getPayAmount() {  
    if (isDead) return deadAmount();  
    if (isSeparated) return separatedAmount();  
    if (isRetired) return retiredAmount();  
    return normalPayAmount();  
}
```

# Replace Parameter with Explicit Method

- You have a method that runs different code depending on the values of an enumerated parameter. Create a separate method for each value of the parameter.

```
void setValue (String name, int value) {  
    if (name.equals("height")) {  
        height = value;  
        return;  
    }  
    if (name.equals("width")) {  
        width = value;  
        return;  
    }  
    Assert.shouldNeverReachHere();  
}
```



```
void setHeight(int arg) {  
    height = arg;  
}  
  
void setWidth (int arg) {  
    width = arg;  
}
```

# Replace Temp with Query

- You are using a temporary variable to hold the result of an expression. Extract the expression into a method. Replace all references to the temp with the expression. The new method can then be used in other methods and allows for other refactorings.

```
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;  
}
```



# Rename Variable or Method

- Perhaps one of the simplest, but one of the most useful that bears repeating: If the name of a method or variable does not reveal its purpose then change the name of the method or variable.

```
public class Customer {  
    public double getinvcdtlmt();  
}
```



```
public class Customer {  
    public double getInvoiceCreditLimit();  
}
```

# More on Refactorings

- **Refactoring Catalog**

- <http://www.refactoring.com/catalog>

- **Java Refactoring Tools**

- NetBeans 4+ – Built In
  - JFactor – works with VisualAge and JBuilder
  - RefactorIt – plug-in tool for NetBeans, Forte, JBuilder and JDeveloper. Also works standalone.
  - JRefactory – for jEdit, NetBeans, JBuilder or standalone

- **Visual Studio 2005+**

- Refactoring Built In
    - Encapsulate Field, Extract Method, Extract Interface, Reorder Parameters, Remove Parameter, Promote Local Var to Parameter, more.

# Refactoring Exercise

## ■ Refactor the Trivia Game code

```
import java.util.ArrayList;

public class TriviaData {
    private ArrayList<TriviaQuestion> data;

    public TriviaData() {
        data = new ArrayList<TriviaQuestion>();
    }

    public void addQuestion(String q, String a, int v, int t) {
        TriviaQuestion question = new TriviaQuestion(q,a,v,t);
        data.add(question);
    }

    public void showQuestion(int index) {
        TriviaQuestion q = data.get(index);
        System.out.println("Question " + (index +1) + ".  " + q.value + " points.");
        if (q.type == TriviaQuestion.TRUEFALSE) {
            System.out.println(q.question);
            System.out.println("Enter 'T' for true or 'F' for false.");
        }
        else if (q.type == TriviaQuestion.FREEFORM) {
            System.out.println(q.question);
        }
    }
}
```



# TriviaData.java TriviaQuestion.java

```

    public int numQuestions()
    {
        return data.size();
    }

    public TriviaQuestion getQuestion(int index)
    {
        return data.get(index);
    }
}

public class TriviaQuestion
{
    public static final int TRUEFALSE = 0;
    public static final int FREEFORM = 1;

    public String question;           // Actual question
    public String answer;             // Answer to question
    public int value;                 // Point value of question
    public int type;                 // Question type, TRUEFALSE or FREEFORM

    public TriviaQuestion()
    {
        question = "";
        answer = "";
        value = 0;
        type = FREEFORM;
    }

    public TriviaQuestion(String q, String a, int v, int t)
    {
        question = q;
        answer = a;
        value = v;
        type = t;
    }
}

```

# TriviaGame.java

```
import java.io.*;
import java.util.Scanner;

public class TriviaGame
{
    public TriviaData questions;          // Questions

    public TriviaGame()
    {
        // Load questions
        questions = new TriviaData();
        questions.addQuestion("The possession of more than two sets of chromosomes is termed?",
                               "polyploidy", 3, TriviaQuestion.FREEFORM);
        questions.addQuestion("Erling Kagge skied into the north pole alone on January 7, 1993.",
                               "F", 1, TriviaQuestion.TRUEFALSE);
        questions.addQuestion("1997 British band that produced 'Tub Thumper'",
                               "Chumbawumba", 2, TriviaQuestion.FREEFORM);
        questions.addQuestion("I am the geometric figure most like a lost parrot",
                               "polygon", 2, TriviaQuestion.FREEFORM);
        questions.addQuestion("Generics were introduced to Java starting at version 5.0.",
                               "T", 1, TriviaQuestion.TRUEFALSE);
    }
}
```

# TriviaGame.java

```
// Main game loop
public static void main(String[] args)
{
    int score = 0; // Overall score
    int questionNum = 0; // Which question we're asking
    TriviaGame game = new TriviaGame();
    Scanner keyboard = new Scanner(System.in);
    // Ask a question as long as we haven't asked them all
    while (questionNum < game.questions.numQuestions())
    {
        // Show question
        game.questions.showQuestion(questionNum);
        // Get answer
        String answer = keyboard.nextLine();
        // Validate answer
        TriviaQuestion q = game.questions.getQuestion(questionNum);
        if (q.type == TriviaQuestion.TRUEFALSE)
        {
            if (answer.charAt(0) == q.answer.charAt(0))
            {
                System.out.println("That is correct! You get " + q.value + " points.");
                score += q.value;
            }
            else
            {
                System.out.println("Wrong, the correct answer is " + q.answer);
            }
        }
    }
}
```

# TriviaGame.java

```
else if (q.type == TriviaQuestion.FREEFORM)
{
    if (answer.toLowerCase().equals(q.answer.toLowerCase()))
    {
        System.out.println("That is correct!  You get " + q.value + " points.");
        score += q.value;
    }
    else
    {
        System.out.println("Wrong, the correct answer is " + q.answer);
    }
    System.out.println("Your score is " + score);
    questionNum++;
}
System.out.println("Game over!  Thanks for playing!");
}
```



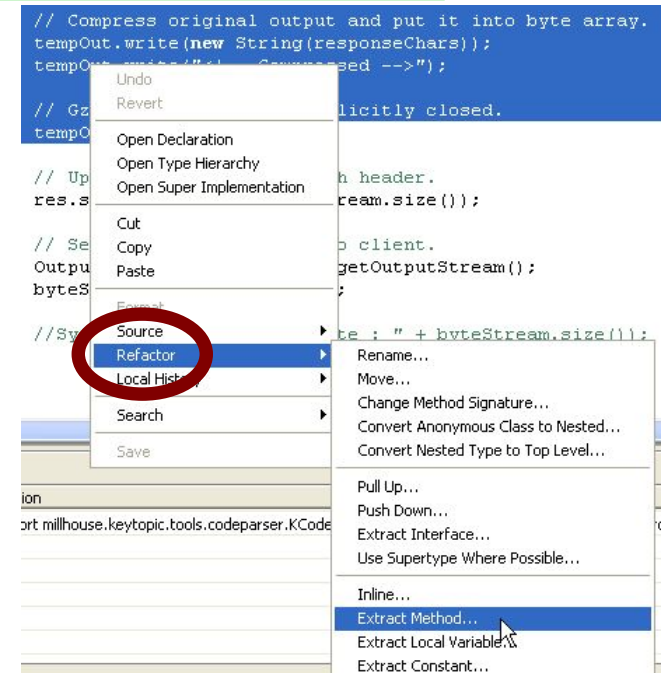
## 4 IDE support for refactoring



# IDE support for refactoring

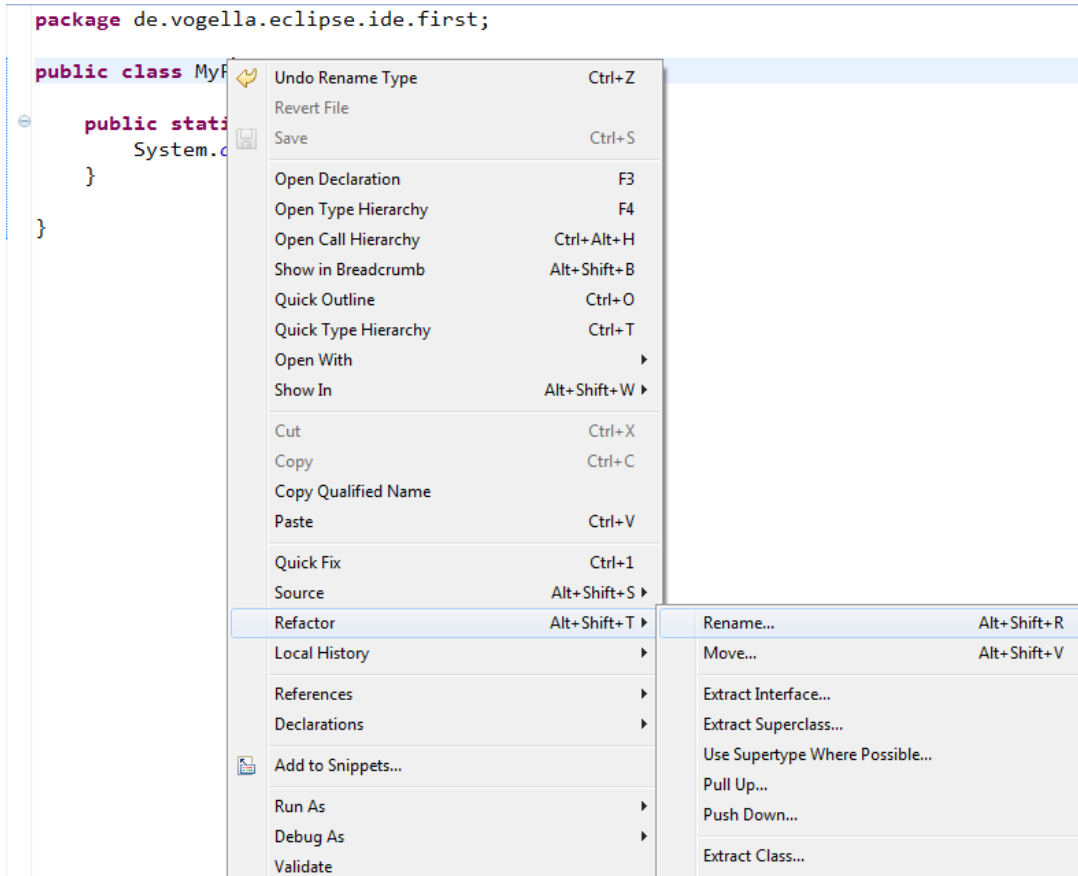
## ■ Eclipse and Visual Studio support:

- Variable / method / class renaming
- Method or constant extraction
- Extraction of redundant code snippets
- Method signature change
- Extraction of an interface from a type
- Method inlining
- Providing warnings about method invocations with inconsistent parameters
- Help with self-documenting code through auto-completion



# Eclipse IDE support for refactoring

- 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...                           |    |

# Eclipse IDE support for refactoring

```

44 void authenticate(Request request, Response response) {
45     if (exists code = request.parameter("code")) {
46         value clientRequest
47         = ClientRequest {
48             uri = gitHubAuth;
49             Parameter("client_id", clientId),
50             Parameter("client_secret", clientSecret),
51             Parameter("code", code).
52
53
54     clientRequest.setHeader("Accept", "application/json");
55     clientRequest.setHeader("Cache-Control", "no-cache");
56
57     value json = clientRequest.execute().contents;
58     assert (is JsonObject result = parse(json),

```

Enter new name for 4 occurrences of 'clientRequest' ... ↩

<http://help.eclipse.org/luna/index.jsp?topic=%2Forg.eclipse.jdt.doc.user%2Freference%2Fref-menu-refactor.htm>

```

public class MyFirstClass {
    public static void main(String[] args) {
        System.out.println("Hello Eclipse!");
        int sum = 0;
        for (int i = 0; i <= 100; i++) {
            sum += i;
        }
        System.out.println(sum);
    }
}

```

Problems @ Javadoc Declaration Console Error Log  
 <terminated> MyFirstClass [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe  
 Hello Eclipse!  
 5050

Extract Method

Method name:

Access modifier: ☐ public ☐ protected ☐ default ☒ private

Parameters:

| Type | Name |
|------|------|
| int  | sum  |
|      |      |
|      |      |
|      |      |
|      |      |

☐ Declare thrown runtime exceptions  
☐ Generate method comment  
☐ Replace additional occurrences of statements with method

Method signature preview:  
**private static int calculateSum(int sum)**

Preview > OK Cancel



# External tools for refactoring



HOME / MARKETPLACE / SEARCH / SEARCH RESULTS / SITE



☆ 6    0

Install

## Compositional Refactoring

This Eclipse bundle provides new Quick Assist actions to support refactorings in a compositional paradigm. In the compositional paradigm, the tool automates small, predictable steps of a refactorin...

Editor, Code Management, J2EE Development Platform, Source Code Analyzer, Application Development Frameworks

Last Updated on Monday, November 7, 2016 - 14:12 by **Mohsen Vakilian**



☆ 4    6

Install

## AutoRefactor

AutoRefactor is an Eclipse plugin to automatically refactor Java code bases. The aim is to fix language/API usage in order to deliver smaller, more maintainable and more expressive code bases.

Source Code Analyzer, Tools, Languages, IDE, Code Management

Last Updated on Monday, November 7, 2016 - 14:17 by **Jean-Noel Rouvignac**



☆ 30    2

Install

## Spartan Refactoring

Automatically find and correct fragments of code to make your Java's source code more efficient, shorter and more readable More info on <https://www.spartan.org.il>

Source Code Analyzer, Tools, Languages

Last Updated on Wednesday, February 22, 2017 - 07:26 by **Daniel Mittelman**



☆ 8    0

Install

## Refactory

Refactory is a set of Eclipse plugins for creating refactorings for arbitrary models. It consists of a SDK to define new generic refactorings which then can be reused for any metamodel you like. Fu...

Modeling Tools, Languages, Modeling, IDE, Tools

Last Updated on Monday, November 7, 2016 - 14:20 by **Jan Reimann**





The end

May 20, 2019