

Refactoring Assistants

John Businge

john.businge@unlv.edu

Refactoring: change the internal structure of a code without compromising its external behaviour

Refactorings can be looked at in two ways:

1. How to identify refactoring targets
2. How to detect applied refactorings

How to identify refactoring targets

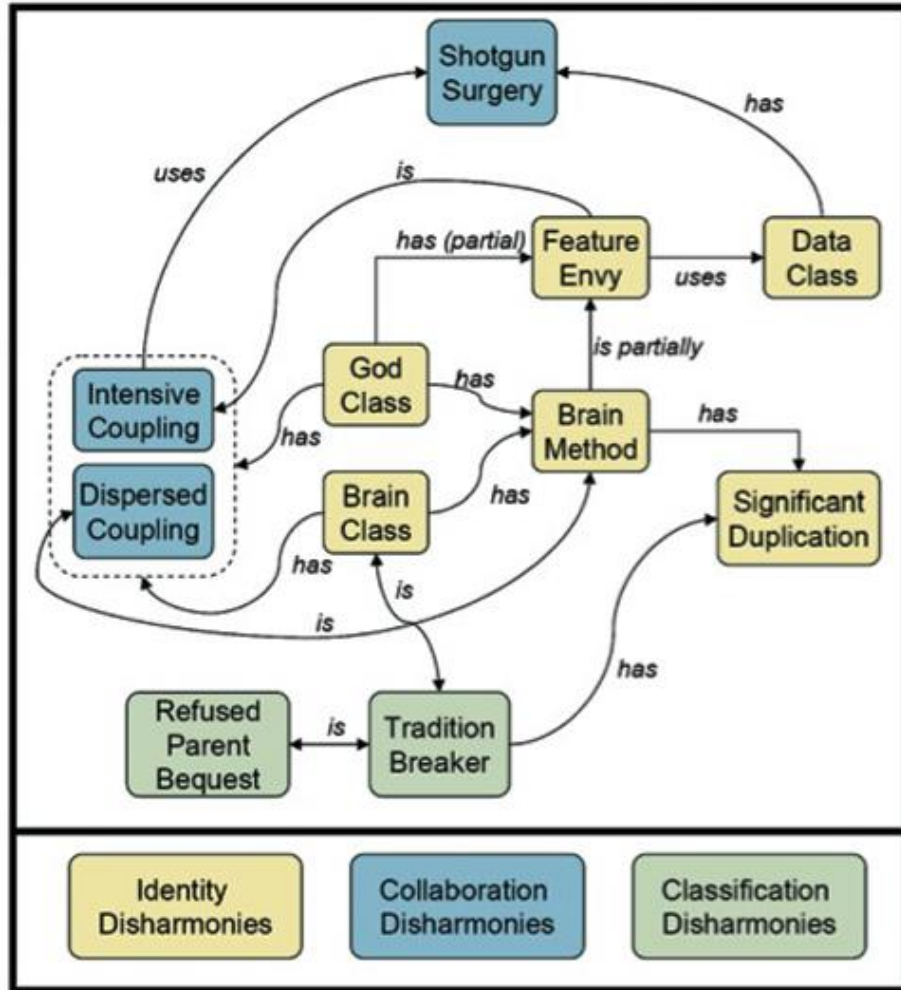
“I wrote the original edition in 2000 when Refactoring was a little-known technique.” – Martin Fowler

- Refactoring is a very common practice that helps developers to complete maintenance tasks (i.e., implement new features and fix bugs) and eliminate various design and code smells
- There are more than 80 types of refactorings
- Some of the common refactorings:
 - Moving a class, renaming an attribute, extracting a method

Strategic Refactoring

- Strategic Refactoring is to apply refactoring for a particular design reason/goal
- Support a new feature/correction
- Solving a specific design problem
- “Refactor to Understand” (OORP, p.127)
- In this Reengineering Course, refactoring without reason/goal is meaningless.
- Please remember the pattern “Keep it Simple” (OORP, p.37) when planning refactoring activities.

Bad/Code Smells



Disharmonies and their correlations

- Code smells are the result of inexperience multiplied by tight deadlines, mismanagement, and nasty shortcuts taken during the development process.
- Code smells are a prime candidate for refactoring
- SonarQube is a nice tool for Smell detection
- In CodeScene, Only the paid version shows Smells

Code Smell Example: God Class

- A God Class is a class that is big on size and/or responsibilities, controlling too many objects.
- Refactoring solution: Extract/Split Class
- It is often possible to “split” a god class into two or more classes with a more clear and logical design

Code Smell Example: God Class

EmployeeManager
+hireEmployee(Employee employee) +terminateEmployee(int employeeld) +editEmployee(Employee employee) +addVacationTime(int employeeld, int days) +useVacationTime(int employeeld, int days) +addAddress(int employeeld, Address address) +removeAddress(int employeeld, int idAddress) +giveBonus(int employeeld, int bonus) +assignEquipment(int employeeld, Equipment equip) +giveRaise(int employeeld, int amount) +dockPay(int employeeld, int amount) +addSchedule(int employeeld, Schedule schedule) +addPhoneNumber(int employeeld, string phone)

Code Smell Example: God Class

EmployeeManager

+hireEmployee(Employee employee)
+terminateEmployee(int employeeId)
+editEmployee(Employee employee)

ScheduleManager

+addEmployeeSchedule(int employeeId, Schedule sch)

VacationManager

+addVacationTime(int employeeId, int days)
+useVacationTime(int employeeId, int days)

PaymentManager

+giveBonus(int employeeId, int amount)
+giveRaise(int employeeId, int amount)
+dockPay(int employeeId, int amount)

EmployeeContactManager

+addAddress(int employeeId, Address address)
+removeAddress(int employeeId, int addressId)
+addPhoneNumber(int employeeId, string phone)

EquipmentManager

+assignEquipment(int employeeId, Equipment eq)

Guidelines on How to Refactor

- (1) Identify where (and when) to refactor
- (2) Consider which refactoring(s) to apply
- (3) Assure behavior preservation on the refactored artifact
- (4) Perform the refactoring(s)
- (5) Assess the effect of the refactoring on quality
- (6) Maintain the system's consistency among the refactored code and other software artifacts

How to detect Applied Refactorings

Refactoring is noise in evolution analysis

- **Bug-inducing analysis (SZZ):** flag refactoring edits as bug-introducing changes
- **Tracing requirements to code:** miss traceability links due to refactoring
- **Regression testing:** unnecessary execution of tests for refactored code with no behavioral changes
- **Code review/merging:** refactoring edits tangled with the actual changes intended by developers

There are many refactoring detection tools

- Demeyer et al. [OOPSLA'00]
 - UMLDiff + JDevAn [Xing & Stroulia ASE'05]
 - RefactoringCrawler [Dig et al. ECOOP'06]
 - Weißgerber and Diehl [ASE'06]
 - Ref-Finder [Kim et al. ICSM'10, FSE'10]
 - RefDiff [Silva & Valente, MSR'17]
 - RefactoringMiner (SOA tool) [Tsantalis et al. TSE'20]
- (RefactoringMiner has the highest average precision (99.6%) and recall (94%) among all competitive tools)

RefactoringMiner – Detecting refactorings in commits

- A library/API written in Java that can detect refactorings applied in the history of a Java project.
- <https://github.com/tsantalis/RefactoringMiner>

RefactoringMiner approach in a nutshell

AST-based statement matching algorithm

- **Input:** code fragments T1 from parent commit and T2 from child commit
- **Output:**
 - M set of matched statement pairs
 - U_{T1} set of unmatched statements from T1
 - U_{T2} set of unmatched statements from T2
- Code changes due to **refactoring mechanics:** *abstraction, argumentization*
- Code changes due to **overlapping refactorings** or **bug fixes:** *syntax-aware AST node replacements*

```
private static Address[] createAddresses(int count) {  
    Address[] addresses = new Address[count];  
    for (int i = 0; i < count; i++) {  
        try {  
            addresses[i] =  
                new Address("127.0.0.1", PORTS.incrementAndGet());  
        }  
        catch (UnknownHostException e) {  
            e.printStackTrace();  
        }  
    }  
    return addresses;  
}
```

Before

After

Extract Method detection rule

```

private static Address[] createAddresses(int count) {
    Address[] addresses = new Address[count];
    for (int i = 0; i < count; i++) {
        try {
            addresses[i] =
                new Address("127.0.0.1", PORTS.incrementAndGet());
        }
        catch (UnknownHostException e) {
            e.printStackTrace();
        }
    }
    return addresses;
}

```

```

private static List<Address> createAddresses(int count) {
    List<Address> addresses = new ArrayList<Address>(count);
    for (int i = 0; i < count; i++) {
        try {
            addresses[i] =
                new Address("127.0.0.1", PORTS.incrementAndGet());
        }
        catch (UnknownHostException e) {
            e.printStackTrace();
        }
    }
    return addresses;
}

```

Before

After

Extract Method detection rule

```

private static Address[] createAddresses(int count) {
    Address[] addresses = new Address[count];
    for (int i = 0; i < count; i++) {
        try {
            addresses[i] =
                new Address("127.0.0.1", PORTS.incrementAndGet());
        }
        catch (UnknownHostException e) {
            e.printStackTrace();
        }
    }
    return addresses;
}

```

```

private static List<Address> createAddresses(AtomicInteger ports, int count){
    List<Address> addresses = new ArrayList<Address>(count);
    for (int i = 0; i < count; i++) {
        try {
            addresses[i] =
                new Address("127.0.0.1", ports.incrementAndGet());
        }
        catch (UnknownHostException e) {
            e.printStackTrace();
        }
    }
    return addresses;
}

```

Before

After

Extract Method detection rule


```

private static Address[] createAddresses(int count) {
    Address[] addresses = new Address[count];
    for (int i = 0; i < count; i++) {
        try {
            addresses[i] =
                new Address("127.0.0.1", PORTS.incrementAndGet());
        }
        catch (UnknownHostException e) {
            e.printStackTrace();
        }
    }
    return addresses;
}

```

```

private static List<Address> createAddresses(AtomicInteger ports, int count){
    List<Address> addresses = new ArrayList<Address>(count);
    for (int i = 0; i < count; i++) {
        try {
            addresses[i] =
                new Address("127.0.0.1", ports.incrementAndGet());
        }
        catch (UnknownHostException e) {
            e.printStackTrace();
        }
    }
    return addresses;
}

```

Before

After

Extract Method detection rule

```
private static Address[] createAddresses(int count) {
    Address[] addresses = new Address[count];
    for (int i = 0; i < count; i++) {
        try {
            addresses[i] =
                new Address("127.0.0.1", PORTS.incrementAndGet());
        }
        catch (UnknownHostException e) {
            e.printStackTrace();
        }
    }
    return addresses;
}
```

```
private static List<Address> createAddresses(AtomicInteger ports, int count){
    List<Address> addresses = new ArrayList<Address>(count);
    for (int i = 0; i < count; i++) {
        addresses.add(createAddress("127.0.0.1", ports.incrementAndGet()));
    }
    return addresses;
}
```

```
protected static Address createAddress(String host, int port) {
    try {
        return new Address(host, port);
    }
    catch (UnknownHostException e) {
        e.printStackTrace();
    }
    return null;
}
```

Before

After

Extract Method detection rule

```
private static Address[] createAddresses(int count) {
    Address[] addresses = new Address[count];
    for (int i = 0; i < count; i++) {
        try {
            addresses[i] =
                new Address("127.0.0.1", PORTS.incrementAndGet());
        }
        catch (UnknownHostException e) {
            e.printStackTrace();
        }
    }
    return addresses;
}
```

```
private static List<Address> createAddresses(AtomicInteger ports, int count){
    List<Address> addresses = new ArrayList<Address>(count);
    for (int i = 0; i < count; i++) {
        addresses.add(createAddress("127.0.0.1", ports.incrementAndGet()));
    }
    return addresses;
}

protected static Address createAddress(String host, int port) {
    try {
        return new Address(host, port); # appears as a return statement in the extracted method.
    }
    catch (UnknownHostException e) {
        e.printStackTrace();
    }
    return null;
}
```

(1) Abstraction

Before

After

Extract Method detection rule

```
private static Address[] createAddresses(int count) {
    Address[] addresses = new Address[count];
    for (int i = 0; i < count; i++) {
        try {
            addresses[i] =
                new Address("127.0.0.1", PORTS.incrementAndGet());
        }
        catch (UnknownHostException e) {
            e.printStackTrace();
        }
    }
    return addresses;
}
```

```
private static List<Address> createAddresses(AtomicInteger ports, int count){
    List<Address> addresses = new ArrayList<Address>(count);
    for (int i = 0; i < count; i++) {
```

```
        addresses.add(createAddress("127.0.0.1", ports.incrementAndGet()));
```

(2) Argumentization

```
    }
    return addresses;
}

protected static Address createAddress(String host, int port) {
    try {
        return new Address(host, port);
    }
    catch (UnknownHostException e) {
        e.printStackTrace();
    }
    return null;
}
```

Before

After

Extract Method detection rule

```
private static Address[] createAddresses(int count) {
    Address[] addresses = new Address[count];
    for (int i = 0; i < count; i++) {
        try {
            addresses[i] =
                new Address("127.0.0.1", PORTS.incrementAndGet());
        }
        catch (UnknownHostException e) {
            e.printStackTrace();
        }
    }
    return addresses;
}
```

```
private static List<Address> createAddresses(AtomicInteger ports, int count){
    List<Address> addresses = new ArrayList<Address>(count);
    for (int i = 0; i < count; i++) {
        addresses.add(createAddress("127.0.0.1", ports.incrementAndGet()));
    }
    return addresses;
}
```

(2) Argumentization

```
protected static Address createAddress(String host, int port) {
    try {
        return new Address("127.0.0.1", ports.incrementAndGet());
    }
    catch (UnknownHostException e) {
        e.printStackTrace();
    }
    return null;
}
```

Before

After

Extract Method detection rule

```
private static Address[] createAddresses(int count) {
    Address[] addresses = new Address[count];
    for (int i = 0; i < count; i++) {
        try {
            addresses[i] =
                new Address("127.0.0.1", PORTS.incrementAndGet());
        }
        catch (UnknownHostException e) {
            e.printStackTrace();
        }
    }
    return addresses;
}
```

```
private static List<Address> createAddresses(AtomicInteger ports, int count){
    List<Address> addresses = new ArrayList<Address>(count);
    for (int i = 0; i < count; i++) {
        addresses.add(createAddress("127.0.0.1", ports.incrementAndGet()));
    }
    return addresses;
}
```

(3) AST Node Replacements

```
protected static Address createAddress(String host, int port) {
    try {
        return new Address("127.0.0.1", ports.incrementAndGet());
    }
    catch (UnknownHostException e) {
        e.printStackTrace();
    }
    return null;
}
```

Before

After

Extract Method detection rule

```
private static Address[] createAddresses(int count) {
    Address[] addresses = new Address[count];
    for (int i = 0; i < count; i++) {
        try {
            addresses[i] =
                new Address("127.0.0.1", PORTS.incrementAndGet());
        }
        catch (UnknownHostException e) {
            e.printStackTrace();
        }
    }
    return addresses;
}
```

```
private static List<Address> createAddresses(AtomicInteger ports, int count){
    List<Address> addresses = new ArrayList<Address>(count);
    for (int i = 0; i < count; i++) {
        addresses.add(createAddress("127.0.0.1", ports.incrementAndGet()));
    }
    return addresses;
}

protected static Address createAddress(String host, int port) {
    try {
        return new Address("127.0.0.1", PORTS.incrementAndGet());
    }
    catch (UnknownHostException e) {
        e.printStackTrace();
    }
    return null;
}
```

Before

After

Extract Method detection rule

```
private static Address[] createAddresses(int count) {
    Address[] addresses = new Address[count];
    for (int i = 0; i < count; i++) {
        try {
            addresses[i] =
                new Address("127.0.0.1", PORTS.incrementAndGet());
        }
        catch (UnknownHostException e) {
            e.printStackTrace();
        }
    }
    return addresses;
}
```

textual similarity = 100%

```
private static List<Address> createAddresses(AtomicInteger ports, int count){
    List<Address> addresses = new ArrayList<Address>(count);
    for (int i = 0; i < count; i++) {
        addresses.add(createAddress("127.0.0.1", ports.incrementAndGet()));
    }
    return addresses;
}
```

```
protected static Address createAddress(String host, int port) {
    try {
        return new Address("127.0.0.1", PORTS.incrementAndGet());
    }
    catch (UnknownHostException e) {
        e.printStackTrace();
    }
    return null;
}
```

Before

After

Extract Method detection rule


```

private static Address[] createAddresses(int count) {
A Address[] addresses = new Address[count];
B for (int i = 0; i < count; i++) {
C   try {
D       addresses[i] =
           new Address("127.0.0.1", PORTS.incrementAndGet());
       }
E   catch (UnknownHostException e) {
F       e.printStackTrace();
       }
       }
G return addresses;
}

```

M = {(C, 4) (D, 5) (E, 6) (F, 7)}
U_{T1} = {A, B, G}
U_{T2} = {8}

Before

```

private static List<Address> createAddresses(AtomicInteger ports, int count){
1 List<Address> addresses = new ArrayList<Address>(count);
2 for (int i = 0; i < count; i++) {
3   addresses.add(createAddress("127.0.0.1", ports.incrementAndGet()));
       }
9 return addresses;
}

protected static Address createAddress(String host, int port) {
4 try {
5   return new Address("127.0.0.1", PORTS.incrementAndGet());
       }
6 catch (UnknownHostException e) {
7   e.printStackTrace();
       }
8 return null;
}

```

After

Extract Method detection rule

Extract Method detection rule

$(M, U_{T_1}, U_{T_2}) = \text{statement-matching}(\text{createAddresses}, \text{createAddress})$

$M = \{(C, 4) (D, 5) (E, 6) (F, 7)\}$ $U_{T_1} = \{A, B, G\}$ $U_{T_2} = \{8\}$

`createAddress` is a **newly added** method in child commit ✓

`createAddresses` in parent commit **does not call** `createAddress` ✓

`createAddresses` in child commit **calls** `createAddress` ✓

$|M| > |U_{T_2}|$ ✓

\Rightarrow `createAddress` has been extracted from `createAddresses`

The Project

- Intermediate Report
 - What refactorings are you planning to implement in the project
 - Reasons why the refactorings are important for your goal
 - Describe the planned refactoring activities
- Final Report
 - Same as the intermediate Report, but the refactorings must be “completed” by then
 - Commits relating to the refactorings should be clearly labelled.