Code Smells & Refactoring

Thomas Fritz

Isabella Chesney

Many thanks to Reid Holmes and Elisa Baniassad

Agenda

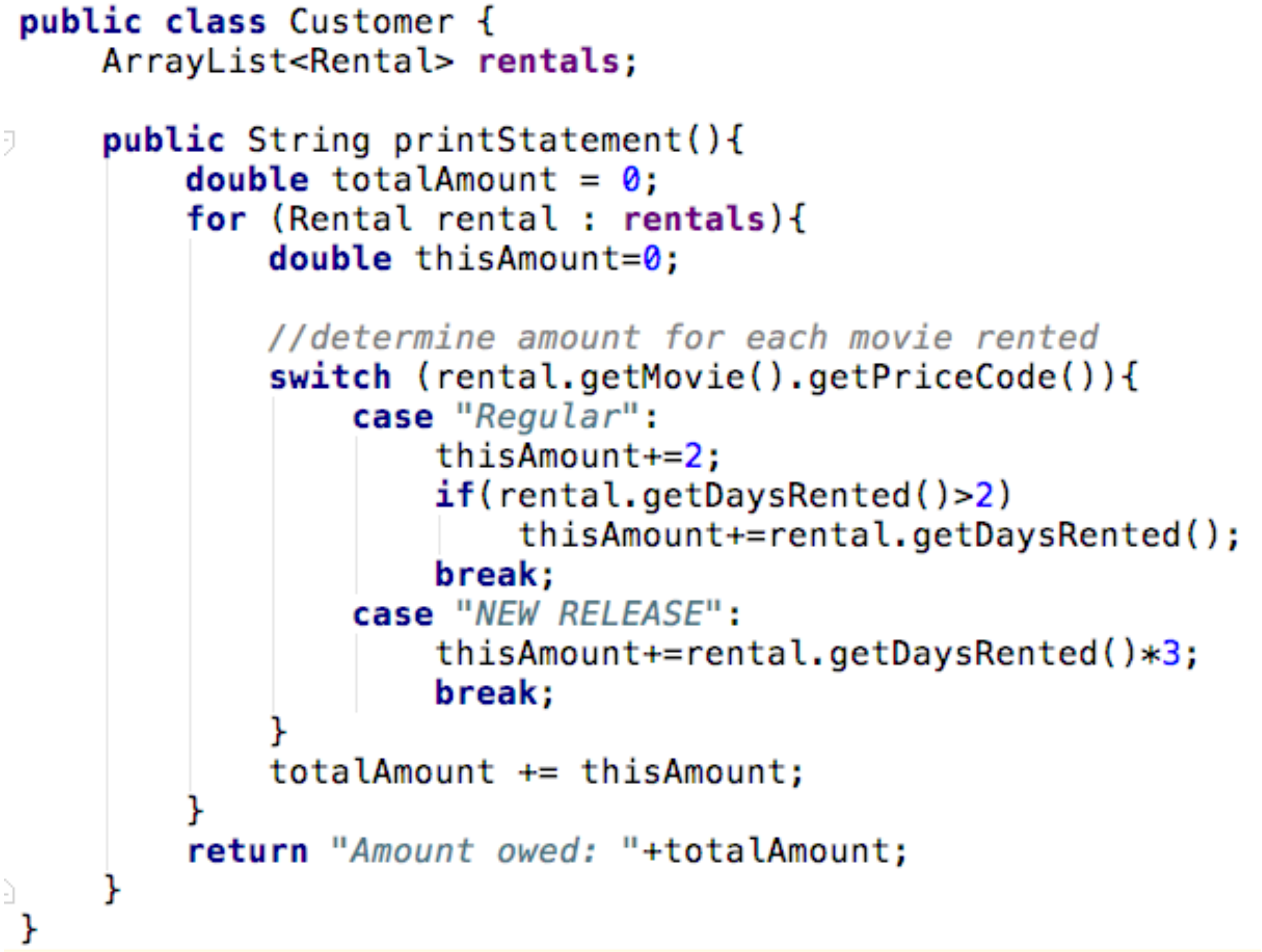
1. Code Smells and Broken Code
2. Refactoring
3. Code Smell and Refactoring Combinations
4. Quiz

Refactoring in the real world

Getting “smelly”

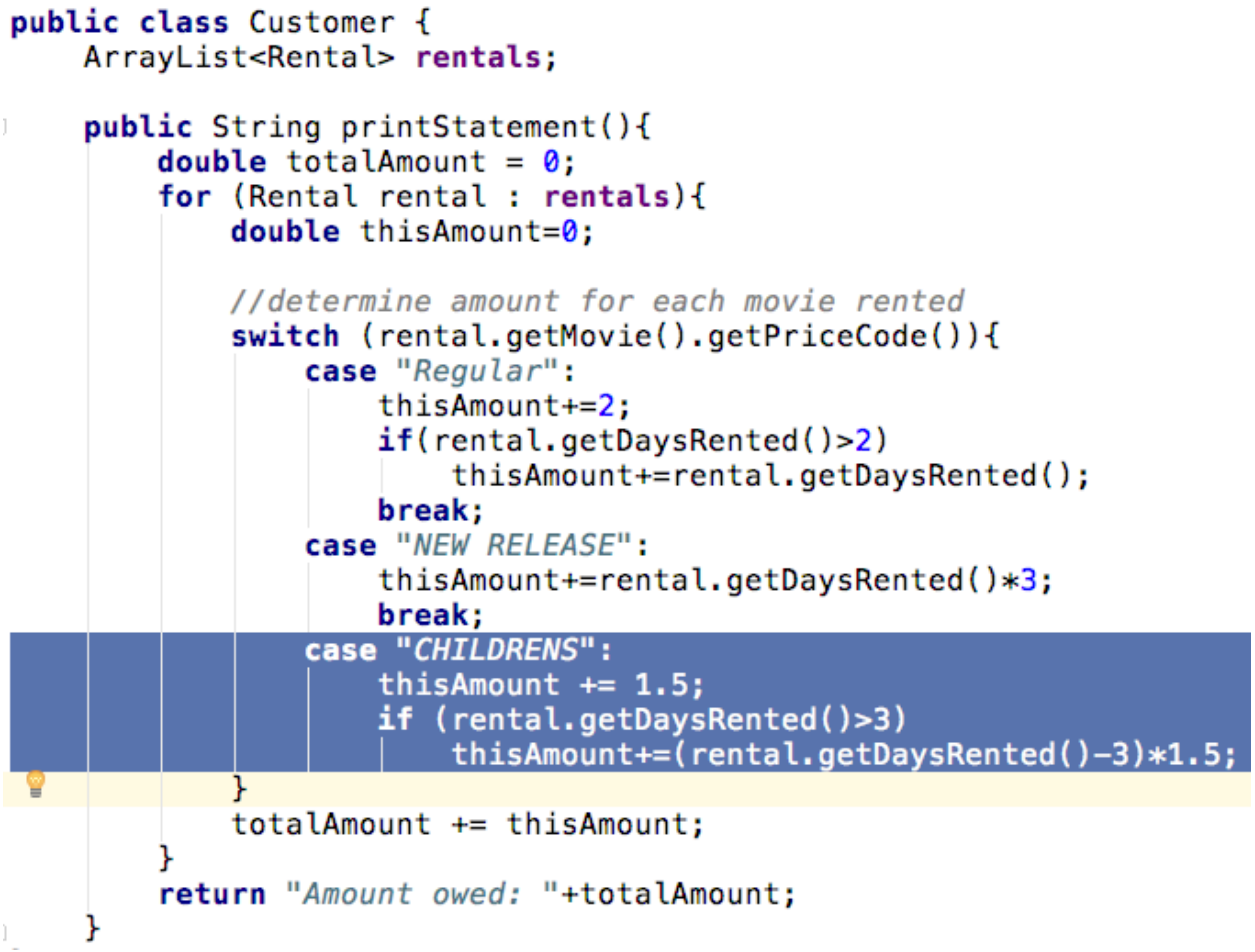
Refactoring

Code @start



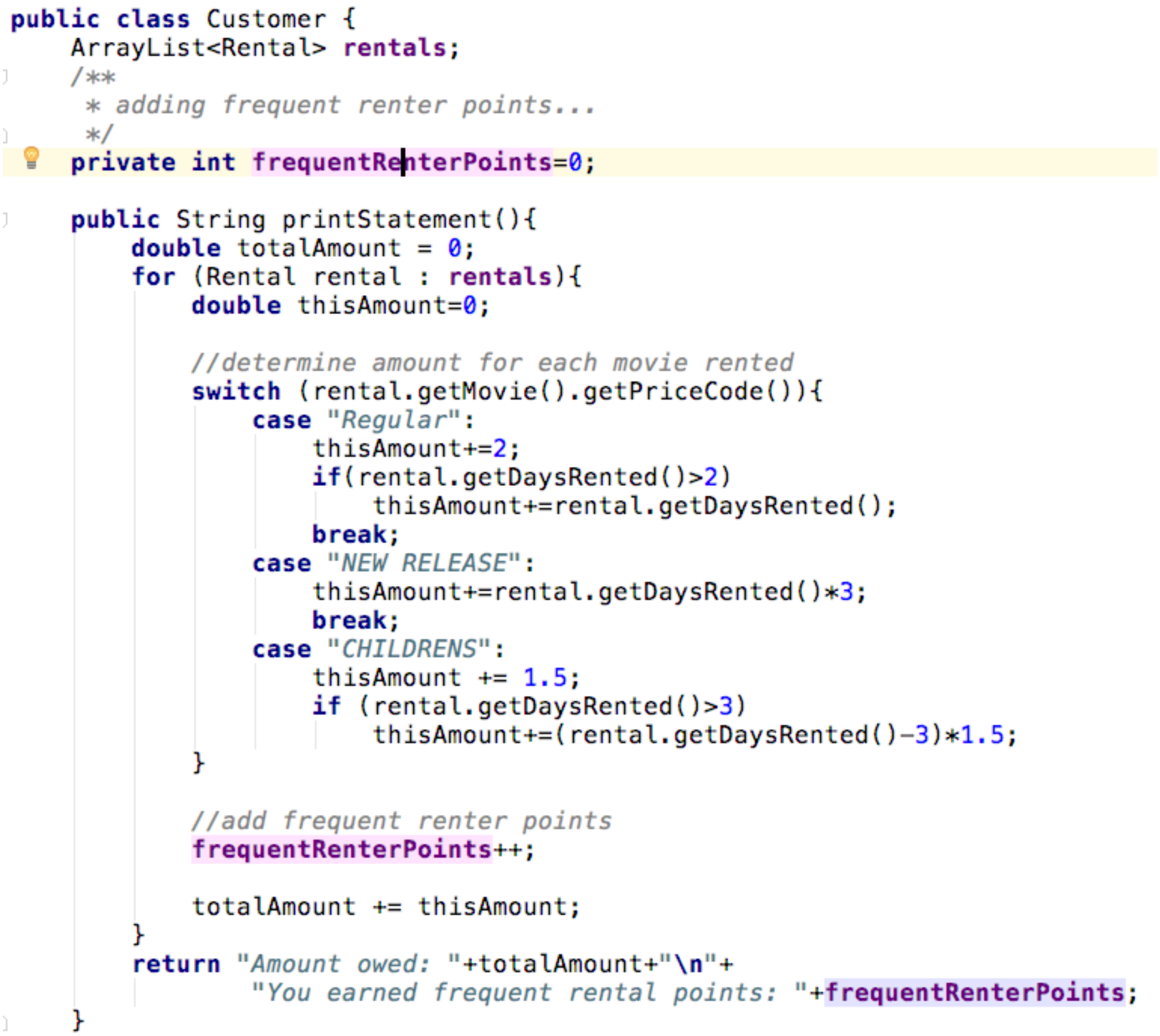
```java
public class Customer {
    ArrayList<Rental> rentals;

    public String printStatement(){
        double totalAmount = 0;
        for (Rental rental : rentals){
            double thisAmount=0;

            //determine amount for each movie rented
            switch (rental.getMovie().getPriceCode()){
                case "Regular":
                    thisAmount+=2;
                    if(rental.getDaysRented()>2)
                        thisAmount+=rental.getDaysRented();
                    break;
                case "NEW RELEASE":
                    thisAmount+=rental.getDaysRented()*3;
                    break;
            }
            totalAmount += thisAmount;
        }
        return "Amount owed: "+totalAmount;
    }
}
```

Evolution



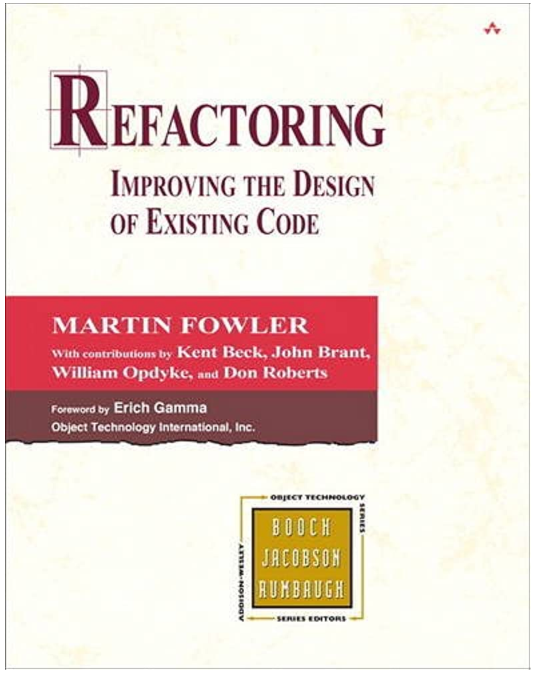
```
public class Customer {
    ArrayList<Rental> rentals;

    public String printStatement(){
        double totalAmount = 0;
        for (Rental rental : rentals){
            double thisAmount=0;

            //determine amount for each movie rented
            switch (rental.getMovie().getPriceCode()){
                case "Regular":
                    thisAmount+=2;
                    if(rental.getDaysRented()>2)
                        thisAmount+=rental.getDaysRented();
                    break;
                case "NEW RELEASE":
                    thisAmount+=rental.getDaysRented()*3;
                    break;
                case "CHILDRENS":
                    thisAmount += 1.5;
                    if (rental.getDaysRented()>3)
                        thisAmount+=(rental.getDaysRented()-3)*1.5;
            }
            totalAmount += thisAmount;
        }
        return "Amount owed: "+totalAmount;
    }
```

Evolution cont’d



```java
public class Customer {
    ArrayList<Rental> rentals;
    /**
     * adding frequent renter points...
     */
    private int frequentRenterPoints=0;

    public String printStatement(){
        double totalAmount = 0;
        for (Rental rental : rentals){
            double thisAmount=0;

            //determine amount for each movie rented
            switch (rental.getMovie().getPriceCode()){
                case "Regular":
                    thisAmount+=2;
                    if(rental.getDaysRented()>2)
                        thisAmount+=rental.getDaysRented();
                    break;
                case "NEW RELEASE":
                    thisAmount+=rental.getDaysRented()*3;
                    break;
                case "CHILDRENS":
                    thisAmount += 1.5;
                    if (rental.getDaysRented()>3)
                        thisAmount+=(rental.getDaysRented()-3)*1.5;
            }

            //add frequent renter points
            frequentRenterPoints++;

            totalAmount += thisAmount;
        }
        return "Amount owed: "+totalAmount+"\n"+
                "You earned frequent rental points: "+frequentRenterPoints;
    }
```

Introduction



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

- The term “Code Smell” was introduced in 1999 in M. Fowler’s Book *Refactoring: Improving the Design of Existing Code*

- By precicely defining what the “symptoms” were, it was possible to define solutions for each code smell -> refactoring

Examinables

By the end of this class you should be able to..

- Explain what code smells are and when refactoring is needed
- Know the most common code smells and how to refactor them
- Know how to use your IDE for efficient refactoring

Code Smells and Broken Code

Learning Objectives

Be able to:

- Recognize and name code smells
- Know the categories of different code smells

Class Exercise – Code Smells Intro

https://bit.ly/3VKFoxe

- In Groups, ca. 5min
- **Task: What could be done better in the following code snippets? (5min)**

  Think about it and write down your ideas into the shared document!

```
1. q = ((p<=1) ? (p ? 0 : 1) : (p==-4) ? 2 : (p+1));
```

```
2. while (*a++ = *b--);
```

```
3. int main(int argc, char **argv) { return ((((((argc % 3) << 4) |
((argv[1][0] == '-') << 3)) | ((argv[2][0] == '-') << 2)) |
((argv[3][0] == '-') << 1)) | ((argv[4][0] == '-') << 0)); }
```

Broken Code

Every module has three functions:

- To execute according to its purpose
- To afford change
- To communicate its purpose to the readers

If it does not do one or more of these, it is broken

Code Smells

- "**Code smell**, also known as **bad smell**, in computer programming code, refers to any symptom in the source code of a program that possibly **indicates** a deeper problem.” (Robert C. Martin)

Code smells

- A recognisable indicator that something structural may be wrong in the code

- **Not** bugs or errors, but indicators of code that could be improved in some way

Cause of Code Smells

- Perfect code really doesn’t exist
- Bad code is typically a product of more than just one developer
- External conditions can affect the quality of your work
- Approaching deadlines, stress, ... (we’ve all been there)

Overview of (some) Code Smells

Bloaters

- Long Method
- Large Class
- Primitive Obsession

OO-Abusers

- Switch Statements
- Refused Requests
- Message Chain

Dispensables

- Duplicate Code
- Comments
- Dead Code

Couplers

- Feature Envy
- Inappropriate Intimacy

Change-Preventers

- Divergent Change
- Shotgun Surgery

Within-Class Smells

- Duplicate Code
    - Increases the risk of inconsistencies
    - Makes the code harder to modify

- Long Parameter List
    - Reduces readability
    - Indicates violation of the single responsibility principle

- Long Method / Large Class
    - Long methods and large classes are more difficult to understand
    - Harder to maintain and modify
    - Indicate violation of the single responsibility principle

- Dead Code
    - Unused parameters and variables
    - Section of code that is never reached or they are computed but never used

Within-Class Smells

- Switch Statements
  - Often overlooked as code smell
  - Misused in large application, where the same switch statement with the same cases appears in multiple places within the code base
  - If you add a new case somewhere, you'll have to add it everywhere else
  - Or you have one large switch statement that is responsible for too much

- Magic numbers
  - Your code uses a number that has a certain meaning
  - Problem: Meaning will be forgotten

- Comments
  - Not necessarily bad themselves but may indicate areas where the code is not as clear as it could be (deodorant for other smells)

Between-Class Smells

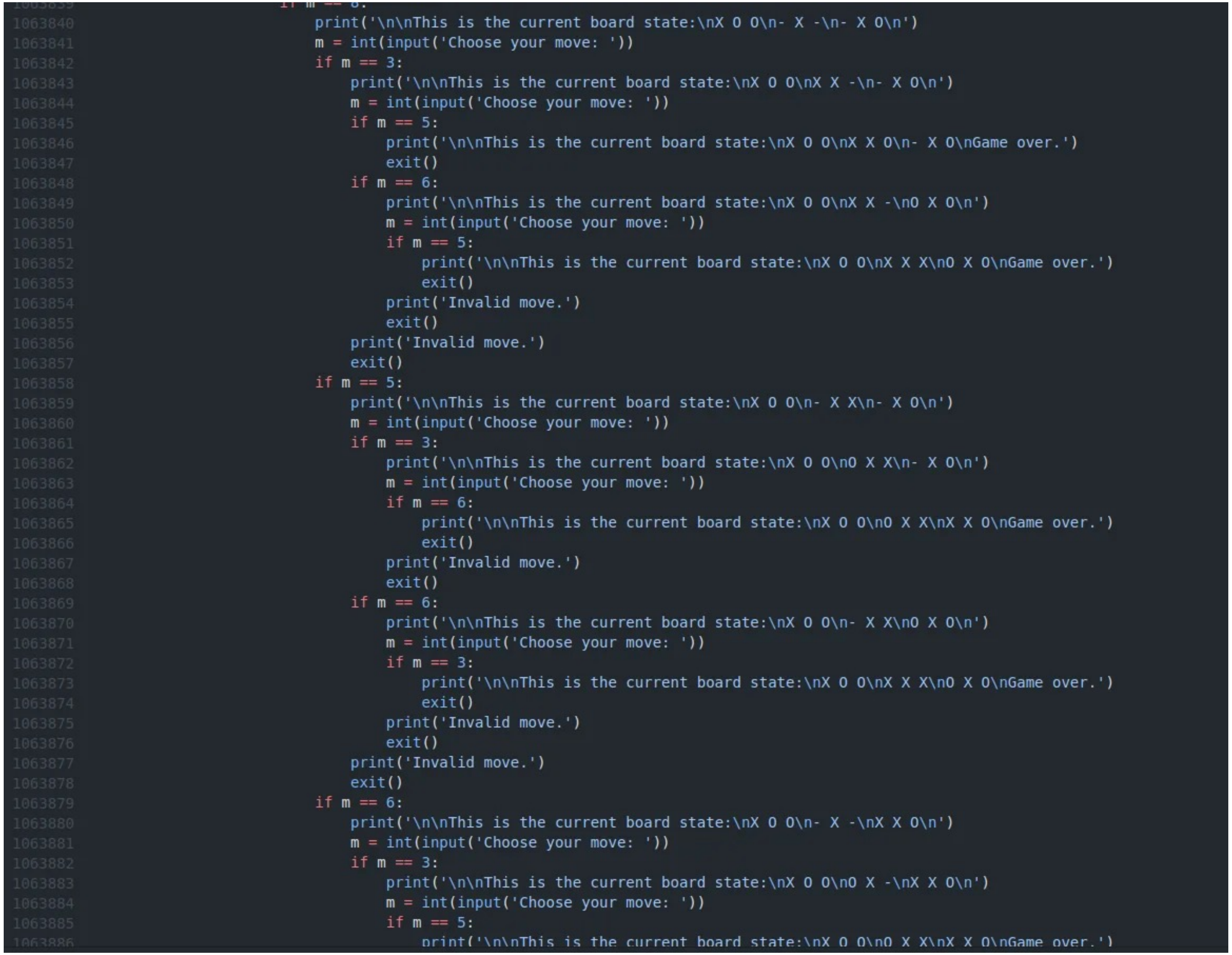
- Shotgun Surgery
  - If a change in one class requires cascading changes in several classes
  - Indicates high level of coupling between the classes

- Message Chains
  - `person.getDepartment().getManager()`
  - A client asks an object for another object and then asks that object for another object etc.
  - Violates encapsulation, increases coupling

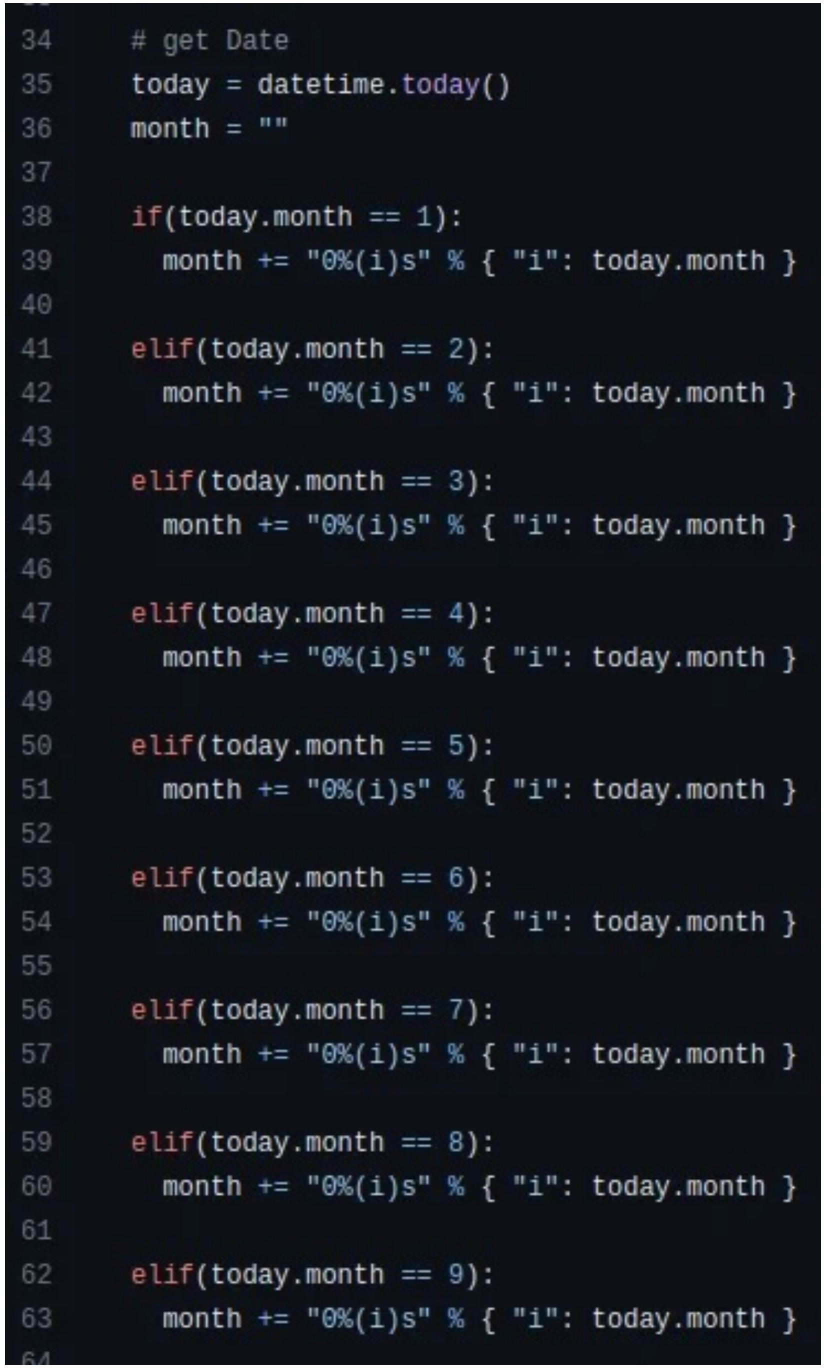
Comprehensive List of Smells

- Alternative Classes with Different Interfaces
- Comments
- Conditional Complexity
- Data Class
- Data Clumps
- Divergent Change
- Duplicated Code
- Feature Envy
- Inappropriate Intimacy
- Incomplete Library Class
- Large Class
- Lazy Class
- Long Method
- Long Parameter List
- Message Chains
- Middle Man
- Parallel Inheritance Hierarchies
- Primitive Obsession
- Refused Bequest
- Shotgun Surgery
- Speculative Generality
- Switch Statements
- Temporary Field
- Type Conditionals
- …

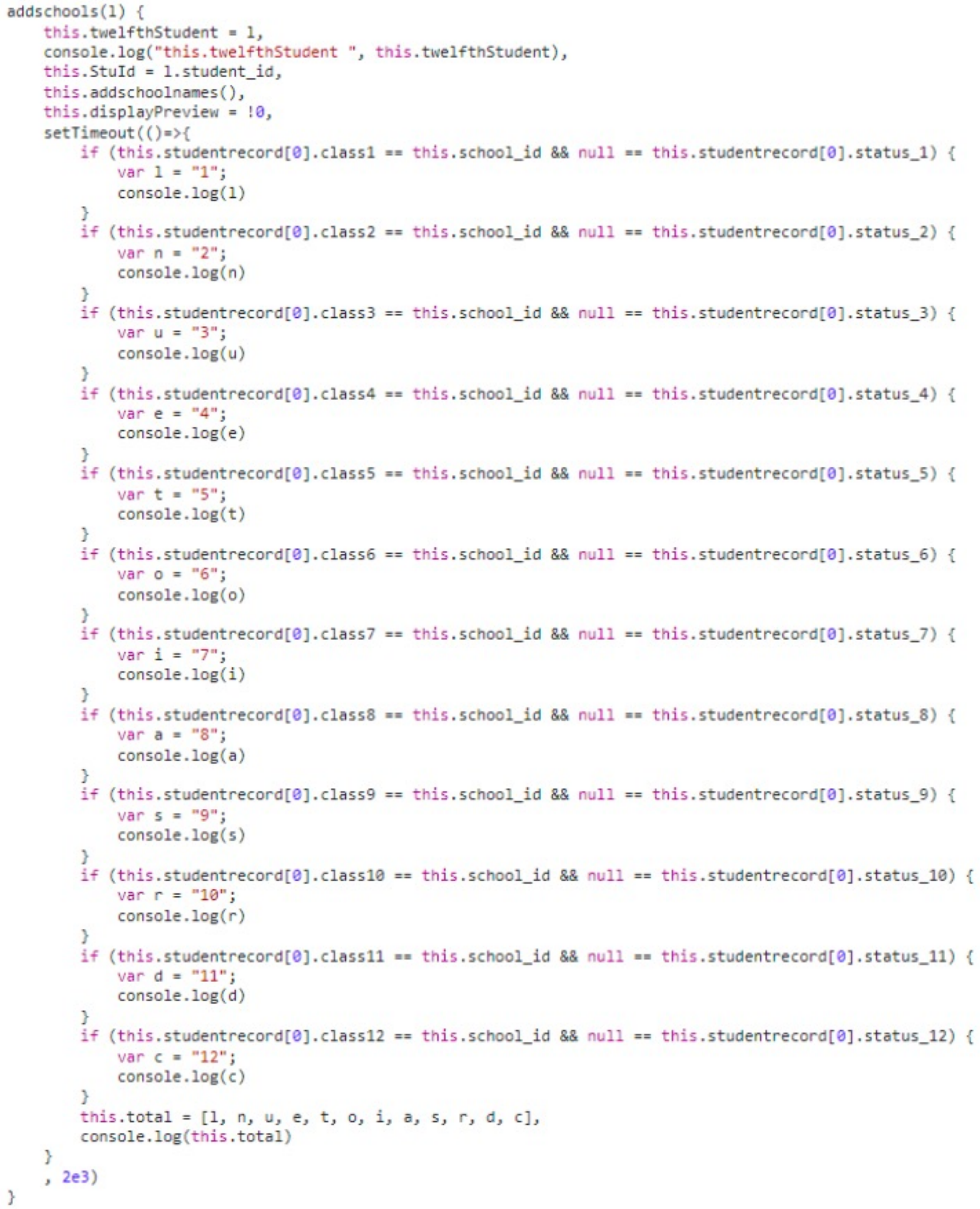
http://www.codinghorror.com/blog/2006/05/code-smells.html
http://sourcemaking.com/refactoring/bad-smells-in-code



```
if m == 8:
    print('\n\nThis is the current board state:\nX O O\n- X -\n- X O\n')
    m = int(input('Choose your move: '))
    if m == 3:
        print('\n\nThis is the current board state:\nX O O\nX X -\n- X O\n')
        m = int(input('Choose your move: '))
        if m == 5:
            print('\n\nThis is the current board state:\nX O O\nX X O\n- X O\nGame over.')
            exit()
        if m == 6:
            print('\n\nThis is the current board state:\nX O O\nX X -\nO X O\n')
            m = int(input('Choose your move: '))
            if m == 5:
                print('\n\nThis is the current board state:\nX O O\nX X X\nO X O\nGame over.')
                exit()
            print('Invalid move.')
            exit()
        print('Invalid move.')
        exit()
    if m == 5:
        print('\n\nThis is the current board state:\nX O O\n- X X\n- X O\n')
        m = int(input('Choose your move: '))
        if m == 3:
            print('\n\nThis is the current board state:\nX O O\nO X X\n- X O\n')
            m = int(input('Choose your move: '))
            if m == 6:
                print('\n\nThis is the current board state:\nX O O\nO X X\nX X O\nGame over.')
                exit()
            print('Invalid move.')
            exit()
        if m == 6:
            print('\n\nThis is the current board state:\nX O O\n- X X\nO X O\n')
            m = int(input('Choose your move: '))
            if m == 3:
                print('\n\nThis is the current board state:\nX O O\nX X X\nO X O\nGame over.')
                exit()
            print('Invalid move.')
            exit()
        print('Invalid move.')
        exit()
    if m == 6:
        print('\n\nThis is the current board state:\nX O O\n- X -\nX X O\n')
        m = int(input('Choose your move: '))
        if m == 3:
            print('\n\nThis is the current board state:\nX O O\nO X -\nX X O\n')
            m = int(input('Choose your move: '))
            if m == 5:
                print('\n\nThis is the current board state:\nX O O\nO X X\nX X O\nGame over.')
```



```
    # get Date
    today = datetime.today()
    month = ""

    if(today.month == 1):
      month += "0%(i)s" % { "i": today.month }

    elif(today.month == 2):
      month += "0%(i)s" % { "i": today.month }

    elif(today.month == 3):
      month += "0%(i)s" % { "i": today.month }

    elif(today.month == 4):
      month += "0%(i)s" % { "i": today.month }

    elif(today.month == 5):
      month += "0%(i)s" % { "i": today.month }

    elif(today.month == 6):
      month += "0%(i)s" % { "i": today.month }

    elif(today.month == 7):
      month += "0%(i)s" % { "i": today.month }

    elif(today.month == 8):
      month += "0%(i)s" % { "i": today.month }

    elif(today.month == 9):
      month += "0%(i)s" % { "i": today.month }
```



```
addschools(l) {
    this.twelfthStudent = l,
    console.log("this.twelfthStudent ", this.twelfthStudent),
    this.StuId = l.student_id,
    this.addschoolnames(),
    this.displayPreview = !0,
    setTimeout(()=>{
        if (this.studentrecord[0].class1 == this.school_id && null == this.studentrecord[0].status_1) {
            var l = "1";
            console.log(l)
        }
        if (this.studentrecord[0].class2 == this.school_id && null == this.studentrecord[0].status_2) {
            var n = "2";
            console.log(n)
        }
        if (this.studentrecord[0].class3 == this.school_id && null == this.studentrecord[0].status_3) {
            var u = "3";
            console.log(u)
        }
        if (this.studentrecord[0].class4 == this.school_id && null == this.studentrecord[0].status_4) {
            var e = "4";
            console.log(e)
        }
        if (this.studentrecord[0].class5 == this.school_id && null == this.studentrecord[0].status_5) {
            var t = "5";
            console.log(t)
        }
        if (this.studentrecord[0].class6 == this.school_id && null == this.studentrecord[0].status_6) {
            var o = "6";
            console.log(o)
        }
        if (this.studentrecord[0].class7 == this.school_id && null == this.studentrecord[0].status_7) {
            var i = "7";
            console.log(i)
        }
        if (this.studentrecord[0].class8 == this.school_id && null == this.studentrecord[0].status_8) {
            var a = "8";
            console.log(a)
        }
        if (this.studentrecord[0].class9 == this.school_id && null == this.studentrecord[0].status_9) {
            var s = "9";
            console.log(s)
        }
        if (this.studentrecord[0].class10 == this.school_id && null == this.studentrecord[0].status_10) {
            var r = "10";
            console.log(r)
        }
        if (this.studentrecord[0].class11 == this.school_id && null == this.studentrecord[0].status_11) {
            var d = "11";
            console.log(d)
        }
        if (this.studentrecord[0].class12 == this.school_id && null == this.studentrecord[0].status_12) {
            var c = "12";
            console.log(c)
        }
        this.total = [l, n, u, e, t, o, i, a, s, r, d, c],
        console.log(this.total)
    }
    , 2e3)
}
```

Class Exercise – Code Smells

- ca. 5min
- **Task: Which smells do you detect in this code?**

```
private String PaymentStatusToString(String address, boolean status, String costumer,
int amount, int account, String date){
    if (bank.getBranch().getPayment().check(costumer, amount, account, date) == true) {
        if (status != null) {
            return convert.ToBoolean(status) == true ? "Success" : "Failed";
        } else if (status == null){
            return "Failed";
        } else {
            return null;
        }
    } else {
//if payment didn't go through, intervention case nr 3 is triggered
        bank.getBranch().getPayment().intervene(3);
       return null;
    }
```

Refactoring

Learning Objectives

Be able to:

- Know how and when to remove code smells

Refactoring

- Process of improving the internal structure of code without changing its external behaviour

- Has the goal of improving code quality and making code more adaptable to changing requirements

When to remove a code smell

So, when *should* you change a running system?

- If it is bad enough to legitimate the effort
- When it affects the ability to make changes
- When the code is no longer understandable

When to refactor

If you’ve decided to refactor: When is the right time to do so?

- NOT:
  - 2 weeks every 6 months
  - When tests are failing
  - When you should just rewrite the code
  - When you have impending deadlines

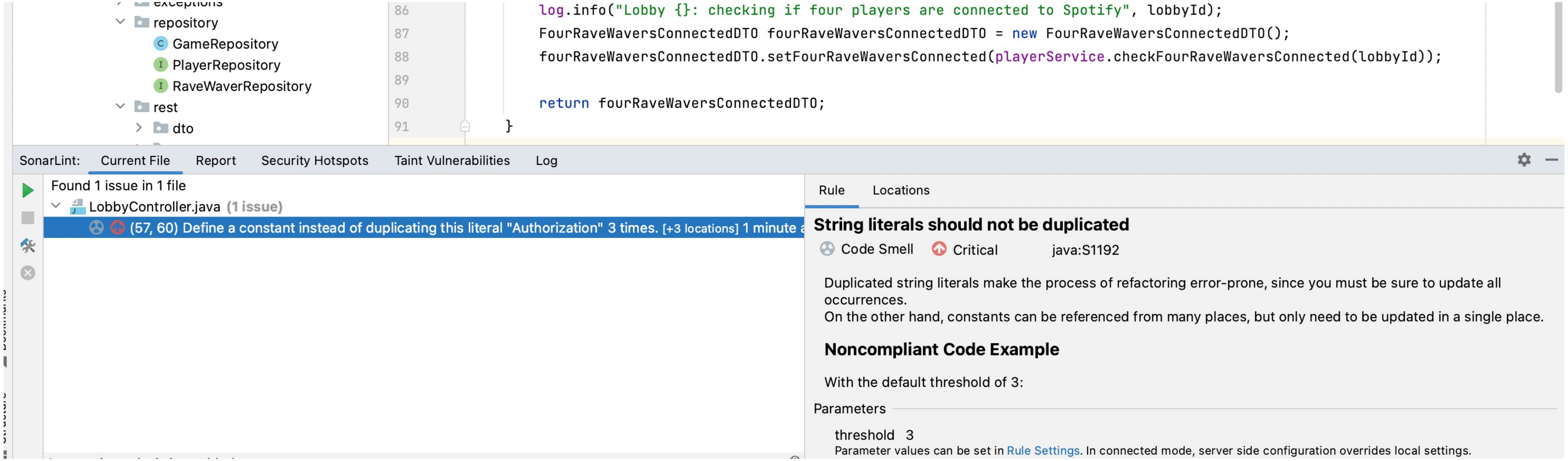
When to refactor

If you’ve decided to refactor: When is the right time to do so?

- Do it as you develop – Opportunistic Refactoring
- Leave it better than you found it (Boy Scout rule)
- When you want to add a new functionality
  - Before to start with clean code
  - And/or afterwards to clean-up
- When you do a code review

Code Linting

- Linting is the automated checking of your source code for programmatic and stylistic errors using a Linter
- Many IDEs have a built in linter that complies with the standards of the respective language (E.g., Eclipse)



```
        log.info("Lobby {}: checking if four players are connected to Spotify", lobbyId);
        FourRaveWaversConnectedDTO fourRaveWaversConnectedDTO = new FourRaveWaversConnectedDTO();
        fourRaveWaversConnectedDTO.setFourRaveWaversConnected(playerService.checkFourRaveWaversConnected(lobbyId));

        return fourRaveWaversConnectedDTO;
    }
```

Found 1 issue in 1 file

LobbyController.java (1 issue)

(57, 60) Define a constant instead of duplicating this literal "Authorization" 3 times. [+3 locations] 1 minute a

String literals should not be duplicated

Code Smell Critical java:S1192

Duplicated string literals make the process of refactoring error-prone, since you must be sure to update all occurrences.
On the other hand, constants can be referenced from many places, but only need to be updated in a single place.

Noncompliant Code Example

With the default threshold of 3:

Parameters

threshold 3

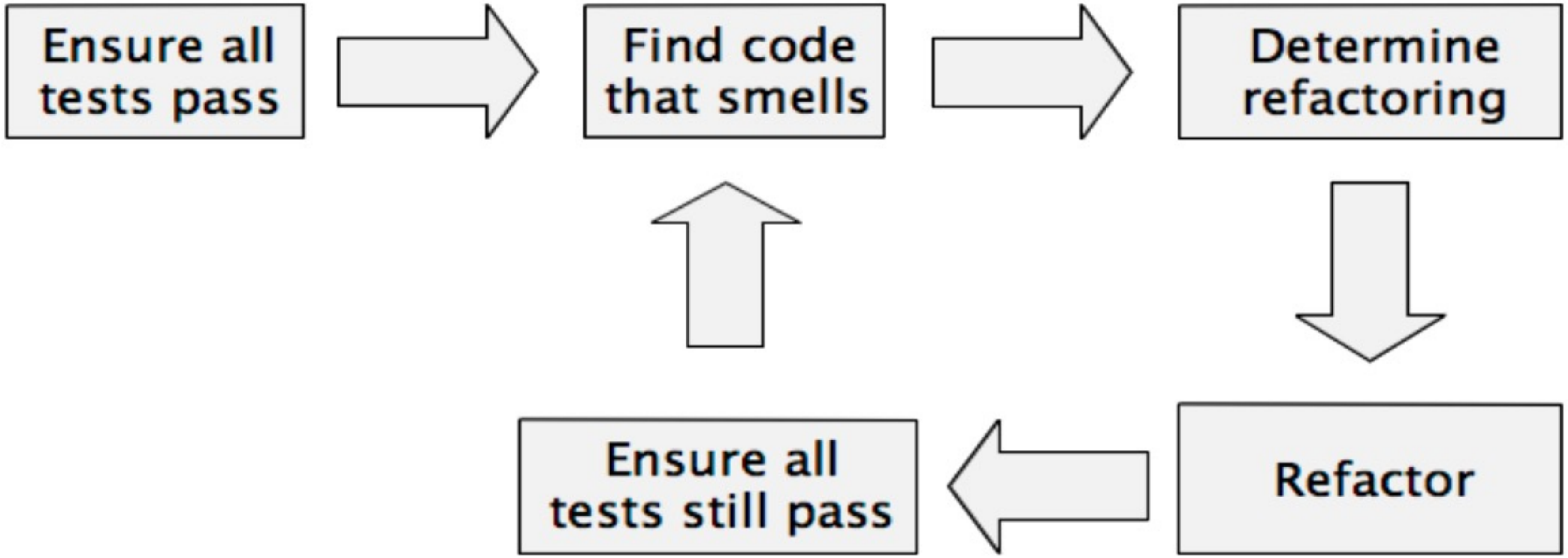
Parameter values can be set in Rule Settings. In connected mode, server side configuration overrides local settings.

Class Exercise – Code Linting https://bit.ly/3B9NTsm

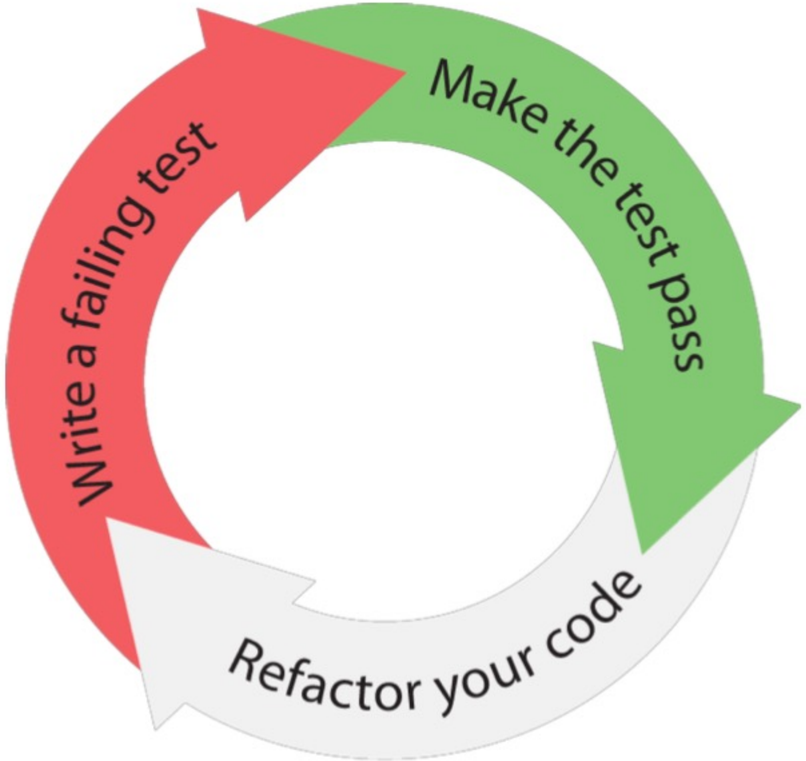
- 15 min
- SonarLint - https://www.sonarlint.org/
- Provided by SonarSource - Swiss Company founded in 2008
- **Task: Download SonarLint for your IDE (supports: JetBrains/Eclipse/VS Code) or download the plugin directly in your IDE and run it on your own code (SoPra, other projects). Write down what smells you were able to detect.**
- Otherwise, you can run it on the following codebase:

  https://github.com/matrix-org/matrix-js-sdk

The process of refactoring



TDD and refactoring



What if no tests are available?

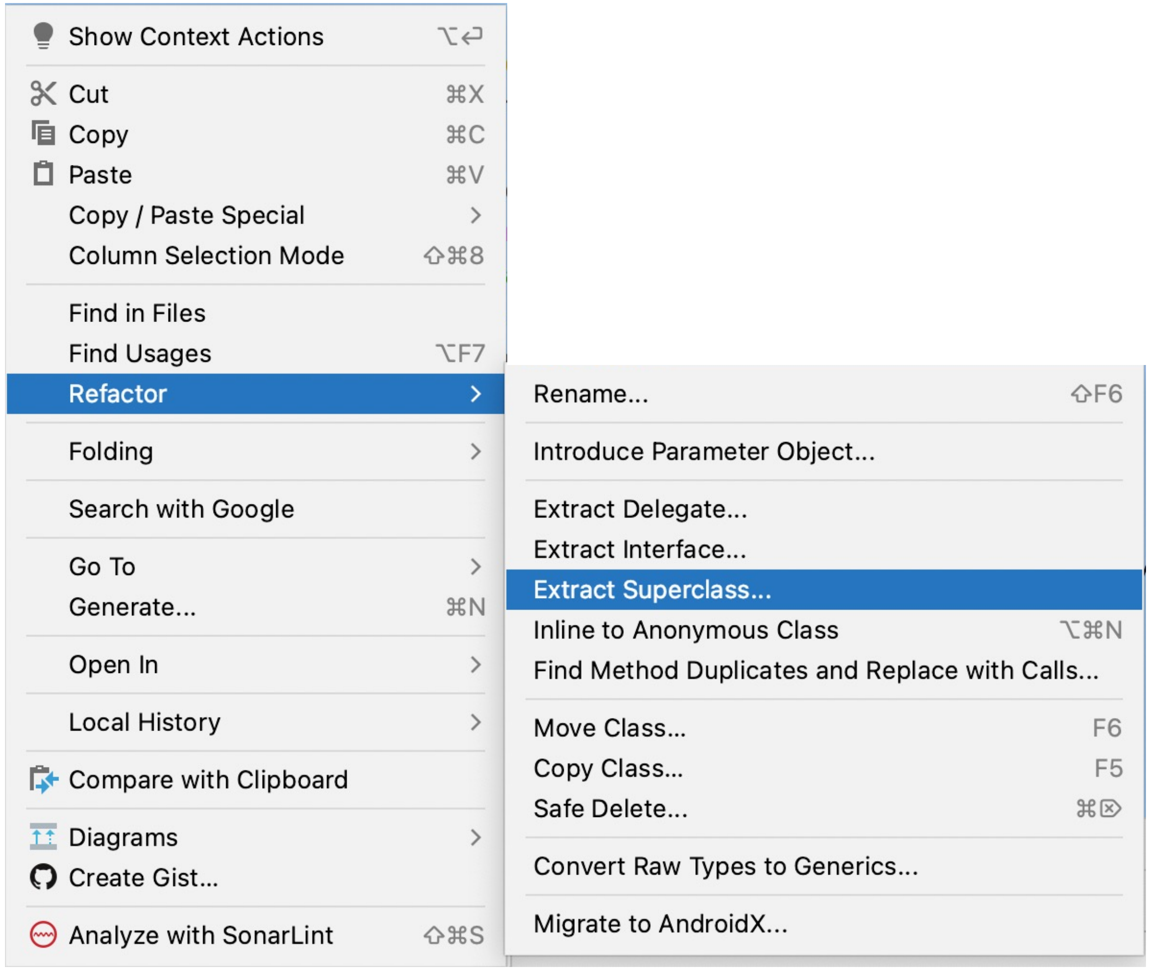
- Sometimes you’ll have to refactor code that is complex and untested
- Without tests you don’t know whether your refactoring is correct or not
    - Write tests before you refactor!

How to refactor

- Manually by hand
- By the book (following a specific process manually)
  smellstorefactorings.pdf
- Automatic, using IDE support

Refactoring in Practice

- IDE supports refactoring
- Much less error-prone than doing it by hand



Refactoring truths

- *Most* of the time your intuition is good
- Doing it *by the book* is hard
  - Use IDE tools
- Unit tests are key
  - Run unit tests
  - Refactor
  - Run unit tests...

Code Smell and Refactoring Combinations

Learning Objectives

Be able to:

- Know how to refactor specific code smells

Code Smell: Duplicate Code

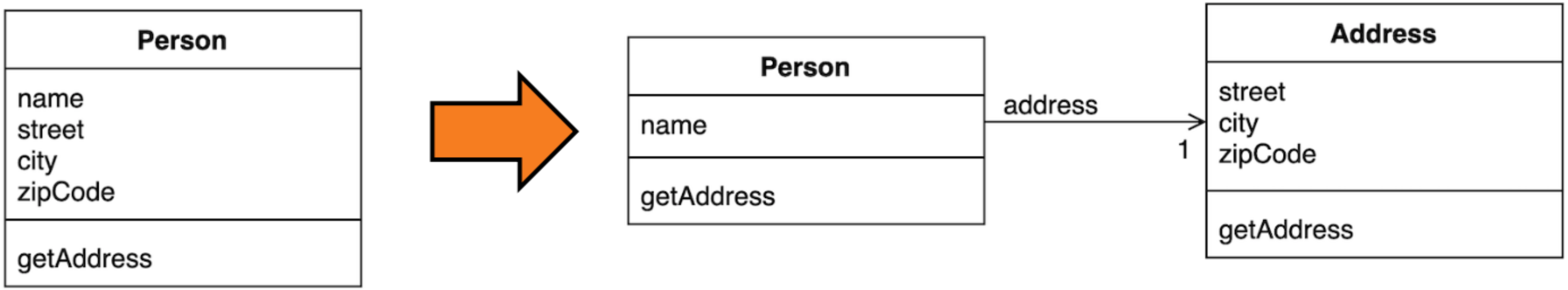
- How to approach refactoring:
  - Extract Method
  - Extract Class
  - Pull Up Method
  - Pull Up Field
- Rule of three!

Refactoring: Extract Method

- Pull code out into a separate method when the original method is long or complex
- Name the new method (and maybe rename the old method) to make their distinction clear
- Each method should have just one task (single responsibility principle)

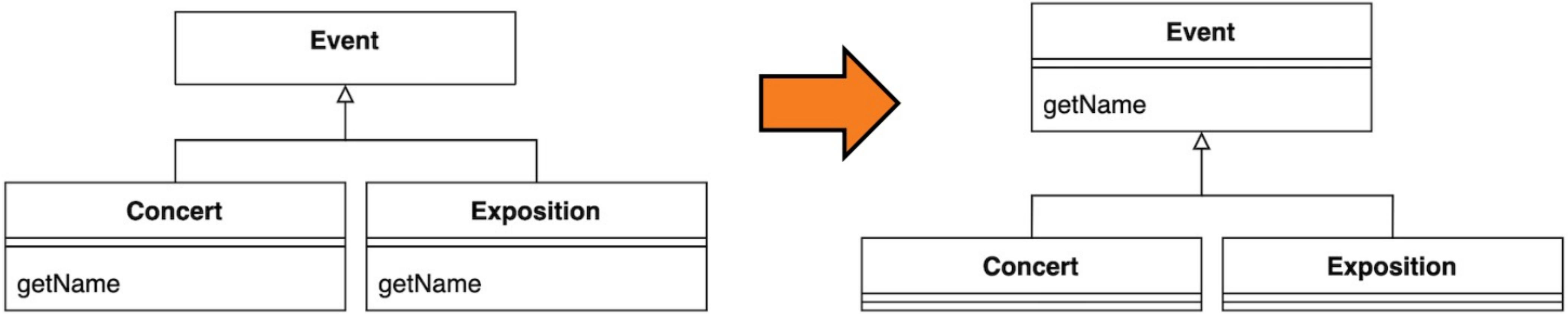
Refactoring: Extract Class

- One class is doing work that should be done by two classes.
- Create a new class and move the relevant fields and methods from the old class into the new class.



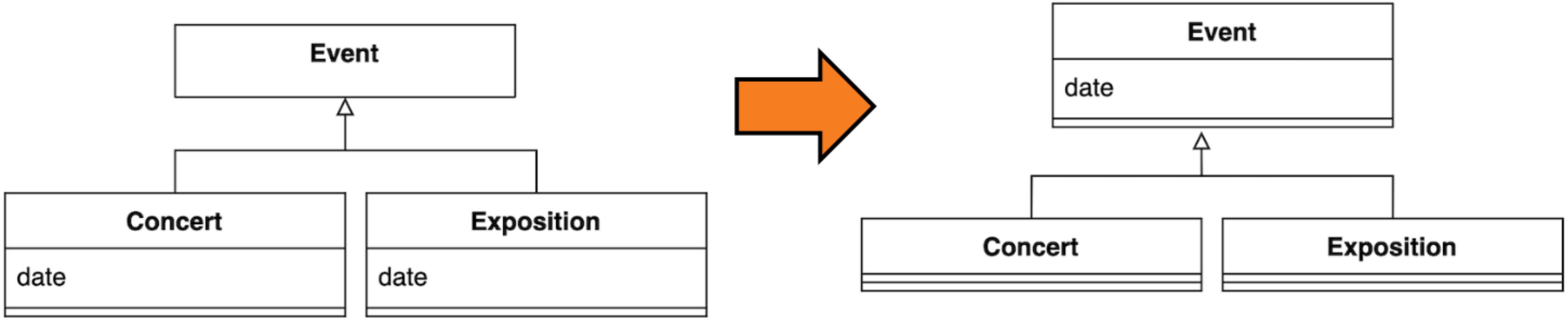
Refactoring: Pull Up Method

If there are identical methods in more than one subclass, move the method to the superclass



Refactoring: Pull Up Field

- If there are identical fields in more than one subclass, move the field to the superclass

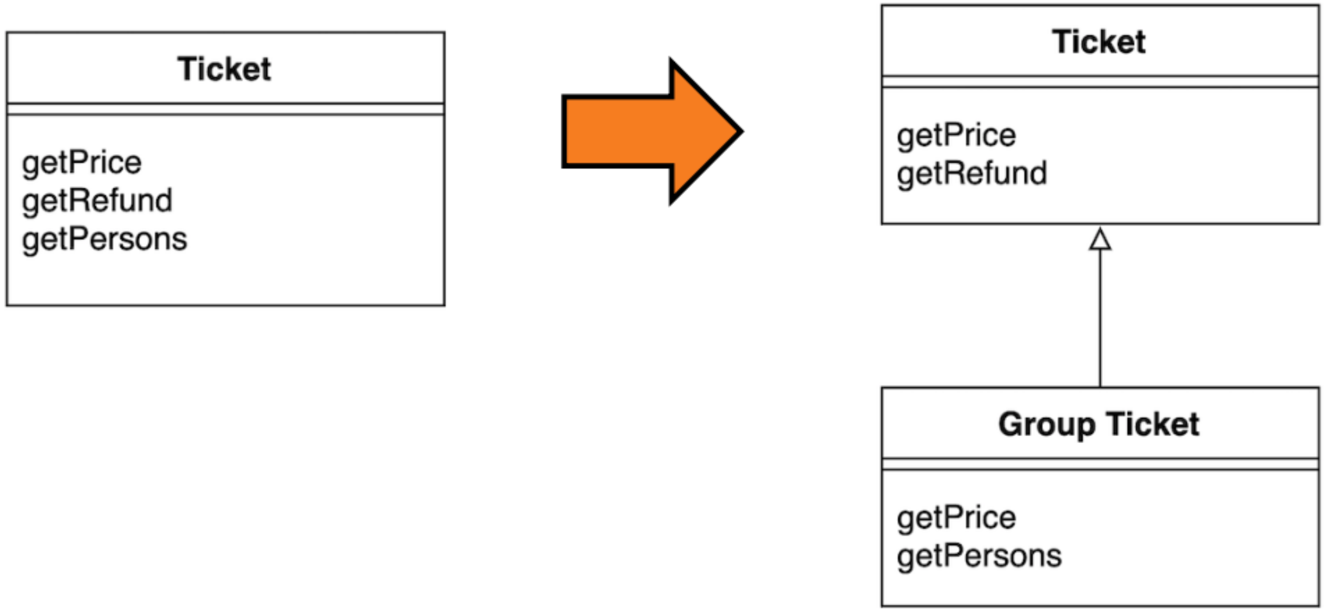


Code Smell: Large Class

- A class with too many instance variables or too much code
- How to approach refactoring:
  - Extract Class
  - Extract Subclass
  - Extract Interface
  - Replace Data Value with Object

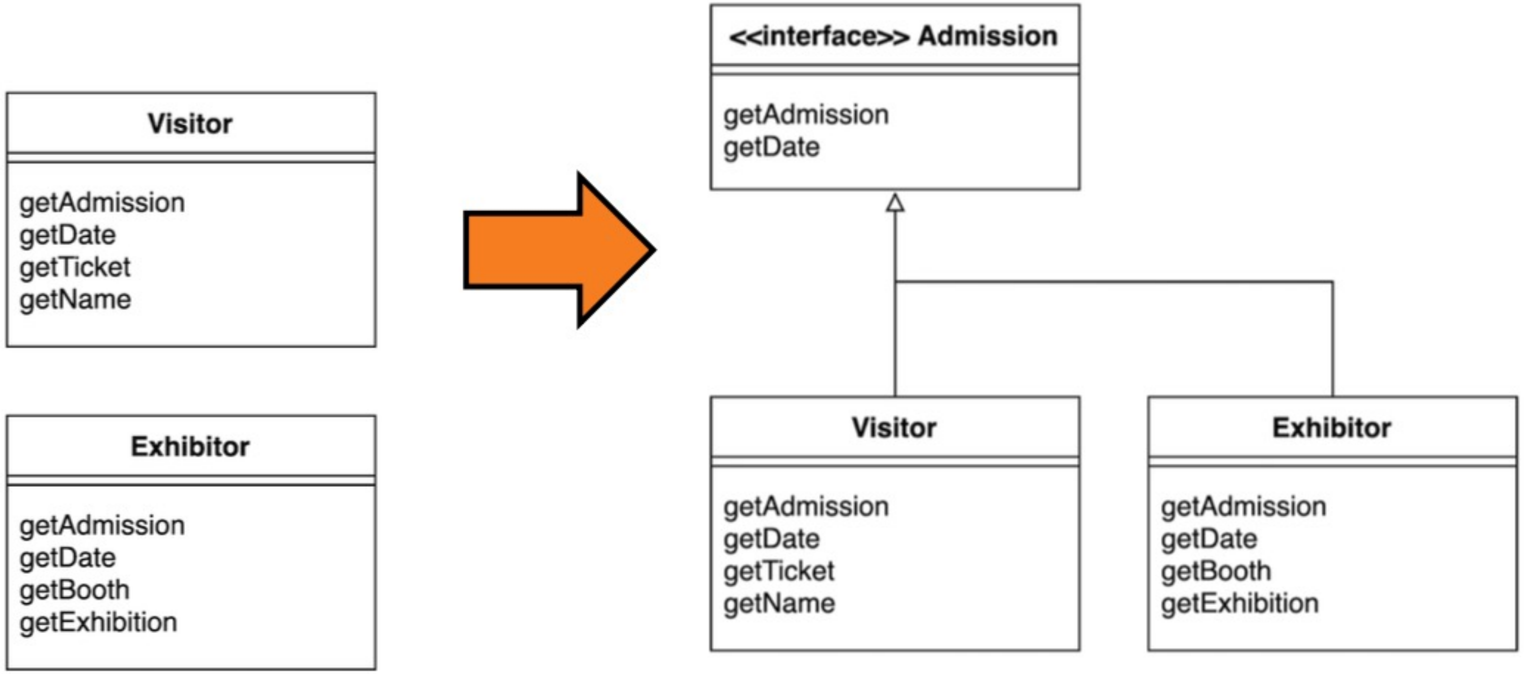
Refactoring: Extract Subclass

If a class has features that are used only in some instances, you can create a subclass for that subset of features.



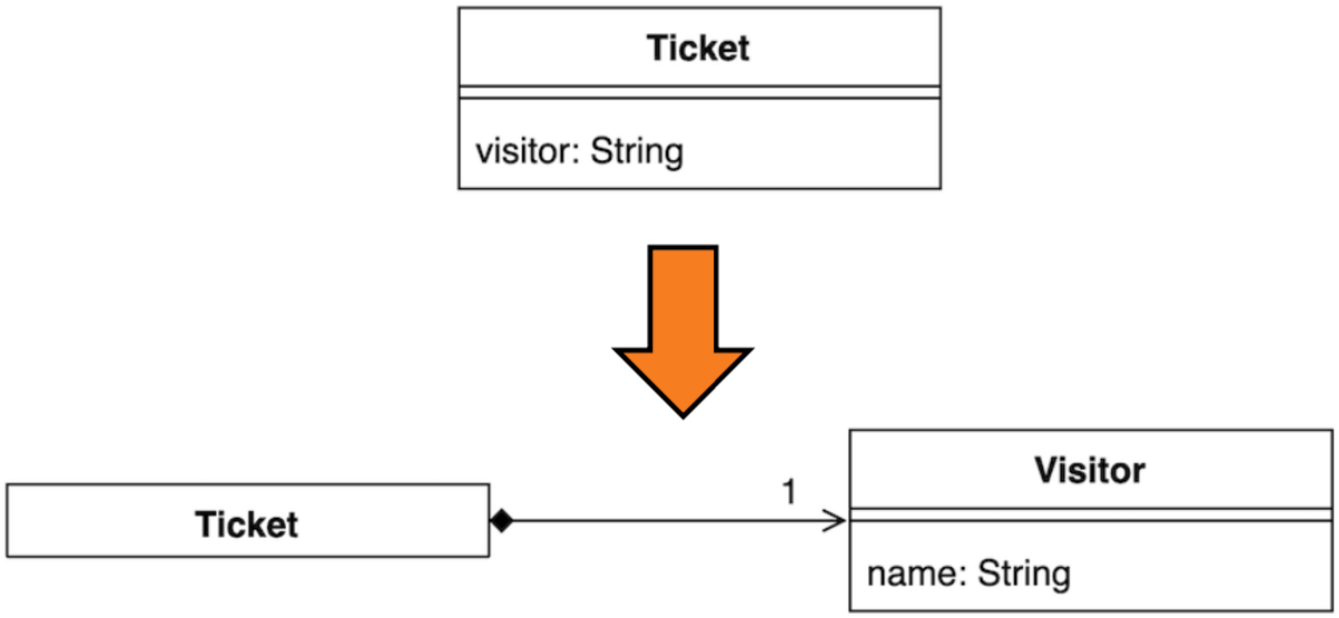
Refactoring: Extract Interface

If several clients use the same subset of a class's API/interface, or two classes have part of their interfaces in common, you can extract the subset into an interface.



Replace Data Value with Object

If you have some data that needs additional data or behavior, you can turn the data into an object



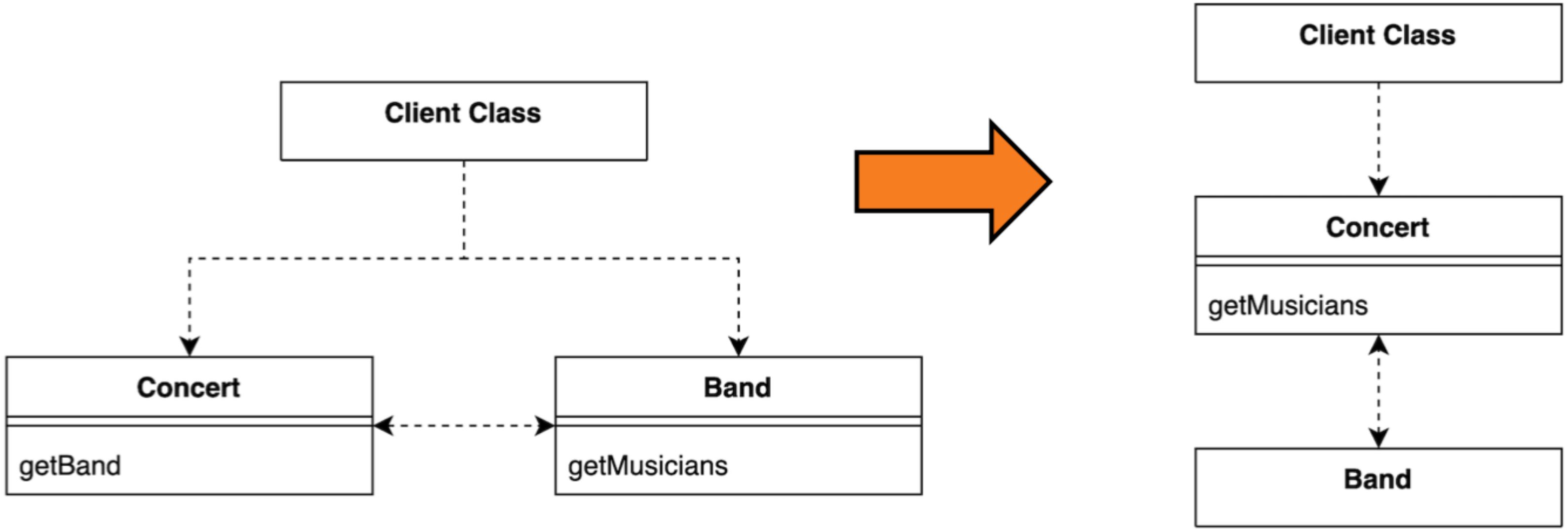
Code Smell: Message Chain

- A client has to use one object to get another, and then use that one to get yet another, etc.

- How to approach refactoring:
  - Hide Delegate

Refactoring: Hide Delegate

- Add a new method that takes over the task from the delegate

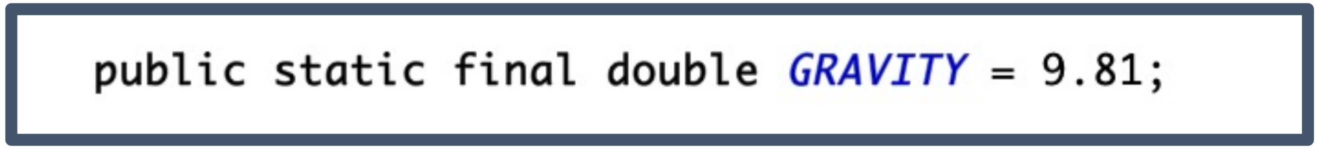


`concert.getBand().getMusicians()` → `concert.getMusicians()`

Code Smell: Magic Numbers

- Any use of an actual number right in the code
- How to approach refactoring:
  - Either change the number to a constant
  - Or change it to a variable

```
double potentialEnergy(double mass, double height) {
   return mass * 9.81 * height;
}
```



```
public static final double GRAVITY = 9.81;
```

Additional Refactorings

- Extract Superclass
  - Similar features for some classes
  - Create superclass with common features

- Push Down Method/Field
  - Opposite of Pull Up
  - Move methods/fields that are only relevant for some subclasses

- Rename
  - Consider renaming if a class/field/method is not clearly conveying its intent

- Move
  - Field/method is more often used in another class than in the one it is defined
  - Consider moving

Class Exercise – Refactoring https://bit.ly/3NWZ5jH

- 20 min
- Go to Olat > Course Materials > download smellyCode.zip > unzip locally
- https://lms.uzh.ch/smellyCode.zip (direct link)
- Go through the code, determine the smells and refactor them
- Write down what code smells you found and how you refactored them in the Google forms link above

Resources

- Cleaner code on GitHub, change branch to «cleanCode» https://github.com/jeschm/smellyCode/tree/smellyCode
- IntelliJ Cheat Sheet https://resources.jetbrains.com/storage/products/intellij-idea/docs/IntelliJIDEA_ReferenceCard.pdf
- Book by Martin Fowler "Refactoring: improving the design of existing code»
- List of refactorings: www.refactoring.com/catalog
- Smells to refactorings: https://www.industriallogic.com/blog/smells-to-refactorings-cheatsheet/
- Builder pattern: https://howtodoinjava.com/design-patterns/creational/builder-pattern-in-java/

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

Martin Fowler

Refactoring: Improving the design of existing code