PEMROGRAMAN LANJUT

Code Smells: Dispensable

Oleh

Tri Hadiah Muliawati

Politeknik Elektronika Negeri Surabaya

2021

Politeknik Elektronika Negeri Surabaya

Departemen Teknik Informatika dan Komputer

Designs Smells

1. Rigidity – hard to change
   The system is hard to change because every change forces many other changes to other parts of the system
2. Fragility – easy to break
   Changes cause the system to break in places that have no conceptual relationship to the part that was changed
3. Immobility – hard to reuse
   It is hard to disentangle the system into components that can be reused in other systems
4. Viscosity – hard to do the right thing
   Doing the things right is harder than doing the things wrong
5. Needless Complexity – overdesign
   The design contains infrastructure that adds no direct benefit
6. Needless Repetition – mouse abuse
   The design contains repeating structures that could be unified under a single abstraction
7. Opacity – disorganized expression
   It is hard to read and understand. It does not express its intent well

Code Smells

- Dispensable
- Object Oriented Abuser
- Bloater
- Change Preventer
- Coupler

Dispensable

something pointless and unneeded whose absence would make the code cleaner, more efficient and easier to understand

Dispensable

- Comments
- Duplicate Code
- Speculative Generality
- Dead Code
- Data Class
- Lazy Class

Comments

Comments

- The older a comment is, and the farther away it is from the code it describes, the more likely it is to be just plain wrong. Since, programmers can’t realistically maintain them.
- Don’t use a comment when you can use a function or a variable

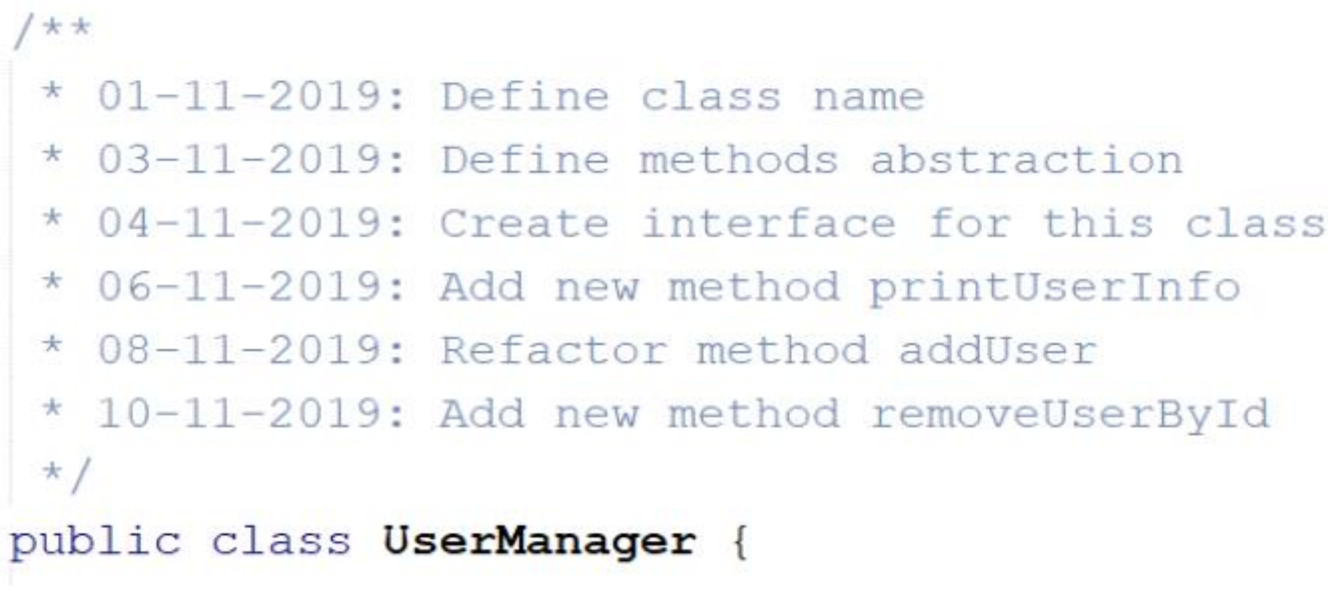
Comments

- Misleading comments
- Mandated comments

```
/**
 *
 * @param title The title of the CD
 * @param author The author of the CD
 * @param tracks The number of tracks on the CD
 * @param durationInMinutes The duration of the CD in minutes
 */
public void addCD(String title, String author,
                  int tracks, int durationInMinutes) {
  CD cd = new CD();
  cd.title = title;
  cd.author = author;
  cd.tracks = tracks;
  cd.duration = duration;
  cdList.add(cd);
}
```

Comments

- Journal comments



```
/**
 * 01-11-2019: Define class name
 * 03-11-2019: Define methods abstraction
 * 04-11-2019: Create interface for this class
 * 06-11-2019: Add new method printUserInfo
 * 08-11-2019: Refactor method addUser
 * 10-11-2019: Add new method removeUserById
 */
public class UserManager {
```

Comments

- Scary noise comments

```
/** The name. */
private String name;

/** The version. */
private String version;

/** The licenceName. */
private String licenceName;

/** The version. */
private String info;
```

- Position markers
- Commented-out code
- Attribution and bylines

```
/* Added by Rick */
```

- Closing brace comments

```
catch (IOException e) {
  System.err.println("Error:" + e.getMessage());
} //catch
```

Refactoring

- **Extract variable**
  If a comment is intended to explain a complex expression, the expression should be split into understandable subexpressions.

```
void renderBanner() {
  if ((platform.toUpperCase().indexOf("MAC") > -1) &&
       (browser.toUpperCase().indexOf("IE") > -1) &&
        wasInitialized() && resize > 0 )
  {
    // do something
  }
}
```

```
void renderBanner() {
  final boolean isMacOs = platform.toUpperCase().indexOf("MAC") > -1;
  final boolean isIE = browser.toUpperCase().indexOf("IE") > -1;
  final boolean wasResized = resize > 0;

  if (isMacOs && isIE && wasInitialized() && wasResized) {
    // do something
```

Refactoring

- **Extract method**
  If a comment explains a section of code, this section can be turned into a separate method. In addition, extract method can also be used when   the extracted expression is used in other places in your code.

```
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);
}
```

Refactoring

- **Rename method**
  If a method has already been extracted, but comments are still necessary to explain what the method does, give the method a self-explanatory name

Good Comments

However, some comments are necessary:

- Java docs in Public APIs
- Legal Comments

```
// Copyright (C) 2003,2004,2005 by Object Mentor, Inc. All rights reserved.
// Released under the terms of the GNU General Public License version 2 or later.
```

- Informative Comments

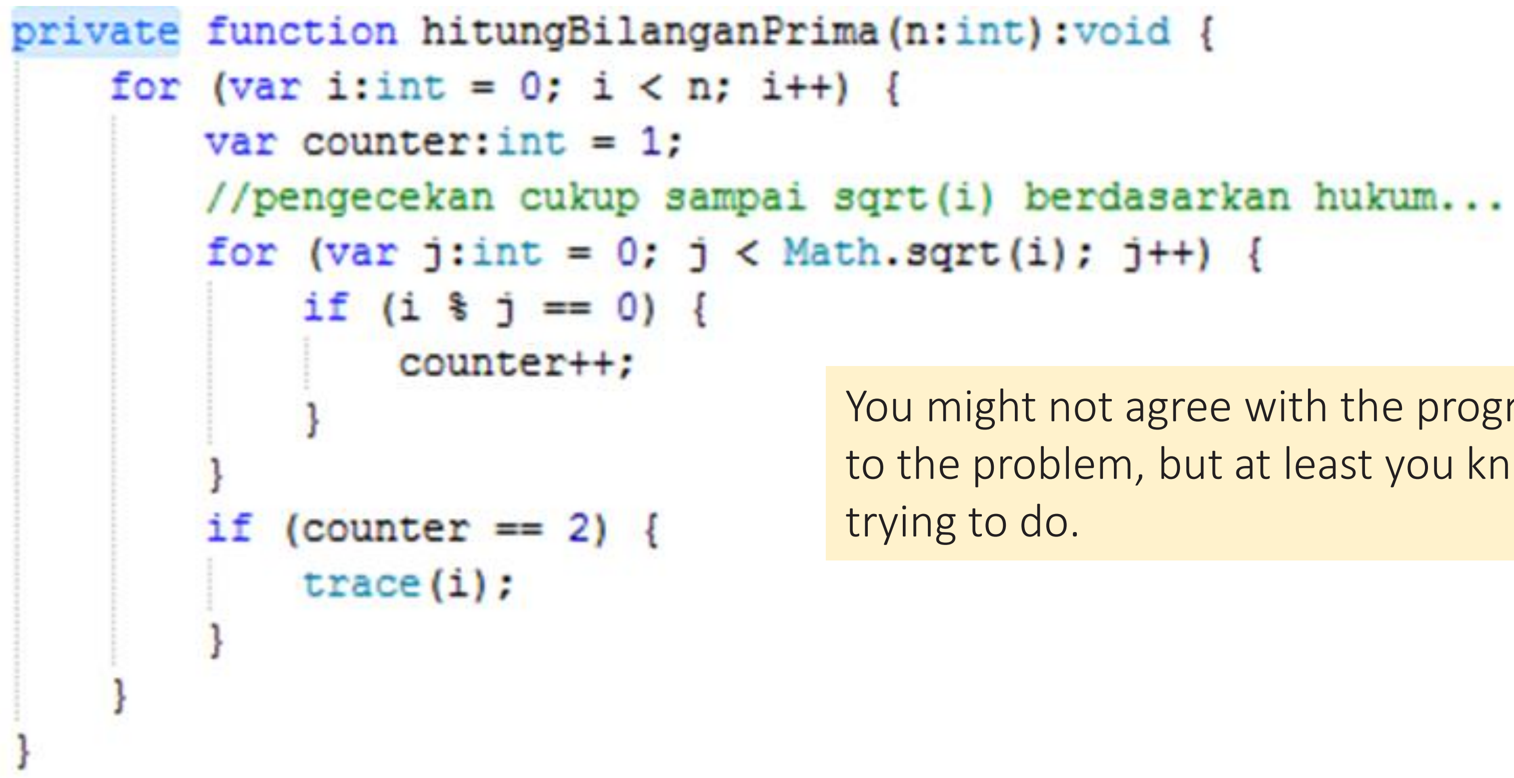
```
// format matched kk:mm:ss EEE, MMM dd, yyyy
Pattern timeMatcher = Pattern.compile(
  "\\d*:\\d*:\\d* \\w*, \\w* \\d*, \\d*");
```

it might have been better, and clearer, if this code had been moved to a special class that converted the formats of dates and times. Then the comment would likely have been superfluous.

Good Comments

However, some comments are necessary:

- Explanation of Intent



```
private function hitungBilanganPrima(n:int):void {
    for (var i:int = 0; i < n; i++) {
        var counter:int = 1;
        //pengecekan cukup sampai sqrt(i) berdasarkan hukum...
        for (var j:int = 0; j < Math.sqrt(i); j++) {
            if (i % j == 0) {
                counter++;
            }
        }
        if (counter == 2) {
            trace(i);
        }
    }
}
```

You might not agree with the programmer’s solution to the problem, but at least you know what he was trying to do.

Duplicate Code

Duplicate Code

- Two code fragments look almost identical
- It is aligned with one of programming principles, which is DRY (Do not Repeat Yourself)

Refactoring

- **Extract method**

1. If the same code is found in two or more methods in the same class.
2. If the same code is found in two subclasses of the same level. followed by **Pull Up Field** for the fields used in the method that you're pulling up

Refactoring

- **Pull up constructor body**
  If the same code is found in two subclasses of the same level and the duplicate code is inside a constructor.

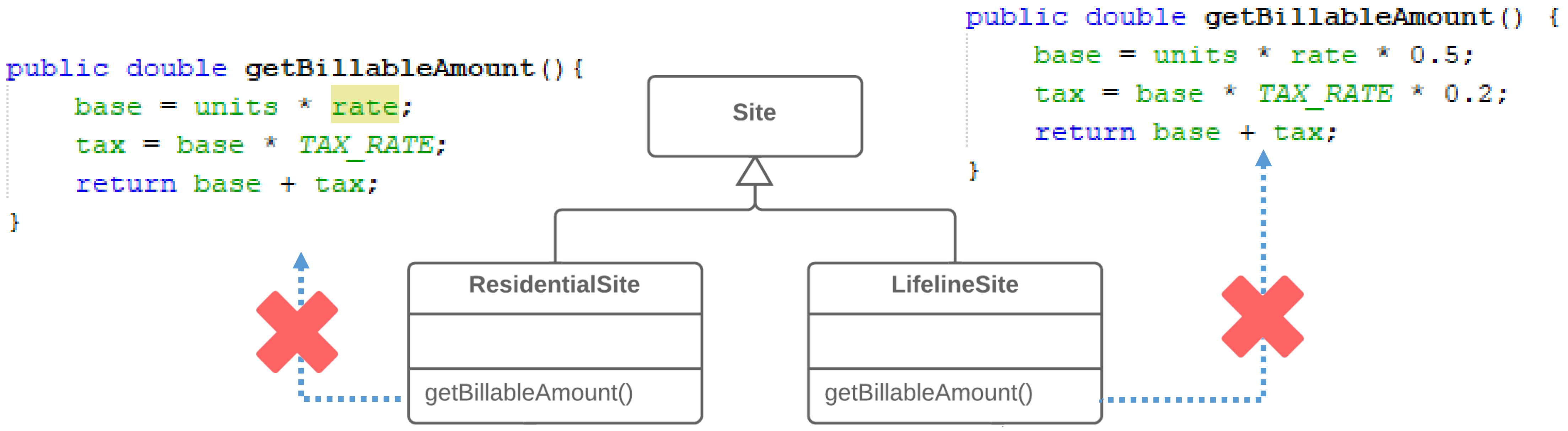
```java
class Manager extends Employee {
  public Manager(String name, String id, int grade) {
    this.name = name;
    this.id = id;
    this.grade = grade;
  }
  // ...
}
```

```java
class Manager extends Employee {

  public Manager(String name, String id, int grade) {
    super(name, id);
    this.grade = grade;
  }
  // ...
}
```

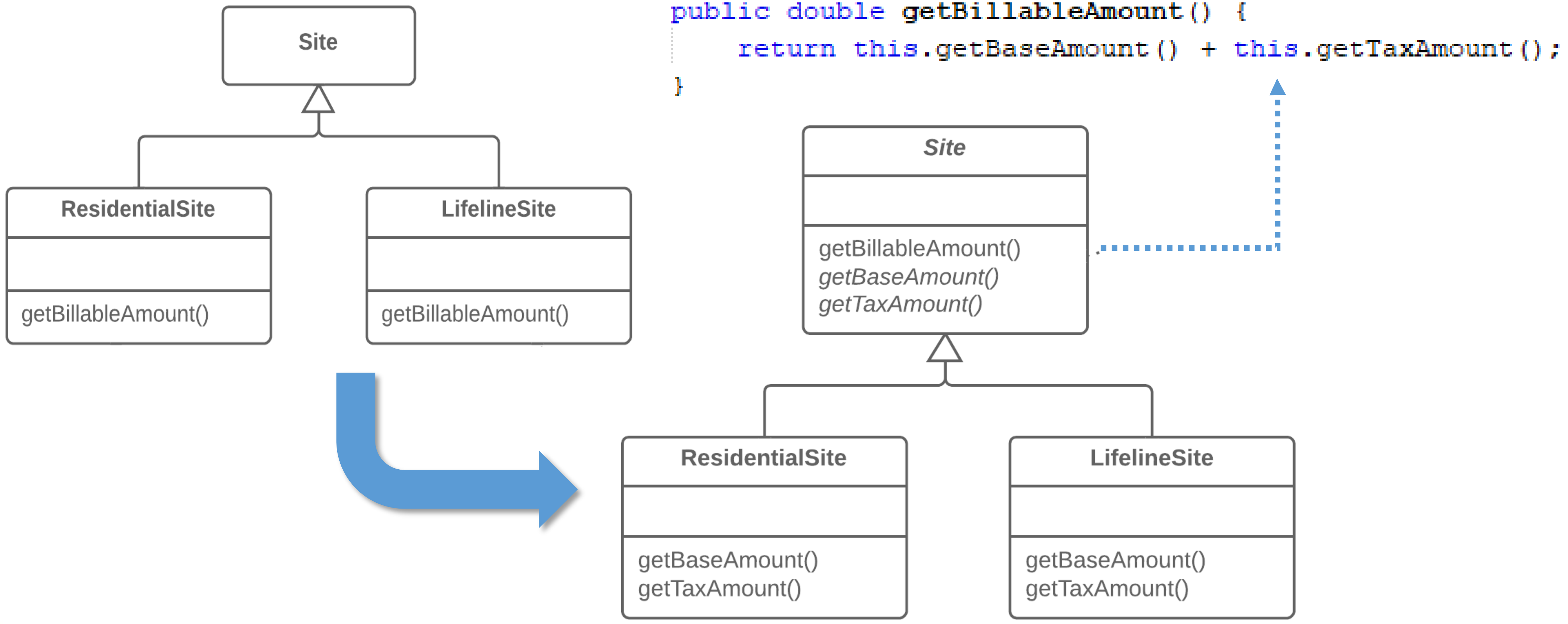
Refactoring

- **Form template method**
  If the same code is found in two subclasses of the same level and the duplicate code is similar but not completely identical.



```
public double getBillableAmount(){
    base = units * rate;
    tax = base * TAX_RATE;
    return base + tax;
}
```

```
public double getBillableAmount() {
    base = units * rate * 0.5;
    tax = base * TAX_RATE * 0.2;
    return base + tax;
}
```



```
public double getBillableAmount() {
    return this.getBaseAmount() + this.getTaxAmount();
}
```

Refactoring

- **Substitute algorithm**
  If the same code is found in two subclasses of the same level and If two methods do the same thing but use different algorithms, select the best algorithm. It is aligned with one of programming principles, which is KISS (Keep It Simple Stu***)

Refactoring

- **Extract superclass**
  If duplicate code is found in two different classes and those classes aren’t part of a hierarchy.
- **Extract class**
  If duplicate code is found in two different classes, but it’s difficult or impossible to create a superclass. Use the new component in the other.

Refactoring

- **Consolidate conditional expression**
  If a large number of conditional expressions are present and perform the same code (differing only in their conditions)

```
double disabilityAmount() {
  if (seniority < 2) {
    return 0;
  }
  if (monthsDisabled > 12) {
    return 0;
  }
  if (isPartTime) {
    return 0;
  }
  // Compute the disability amount.
  // ...
}
```

```
double disabilityAmount() {
  if (isNotEligableForDisability()) {
    return 0;
  }
  // Compute the disability amount.
  // ...
}
```

Refactoring

- **Consolidate duplicate conditional fragment**
  If Identical code can be found in all branches of a conditional

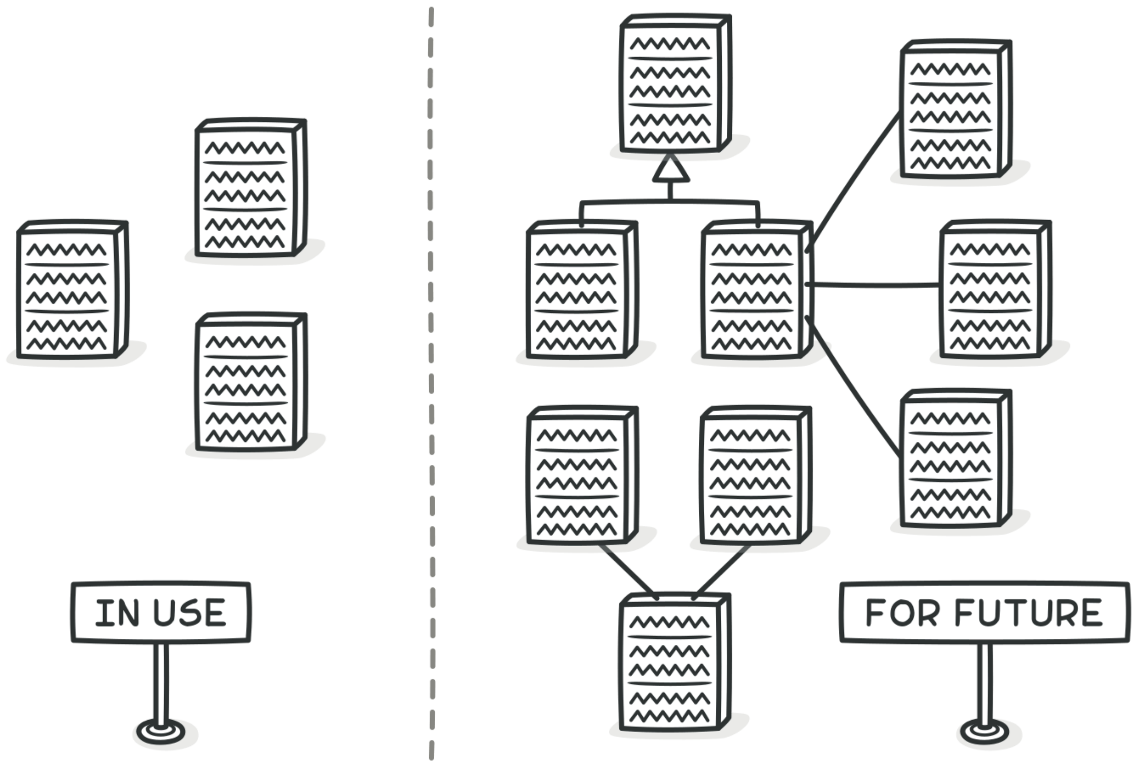
```
if (isSpecialDeal()) {
  total = price * 0.95;
  send();
}
else {
  total = price * 0.98;
  send();
}
```

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

Speculative Generality

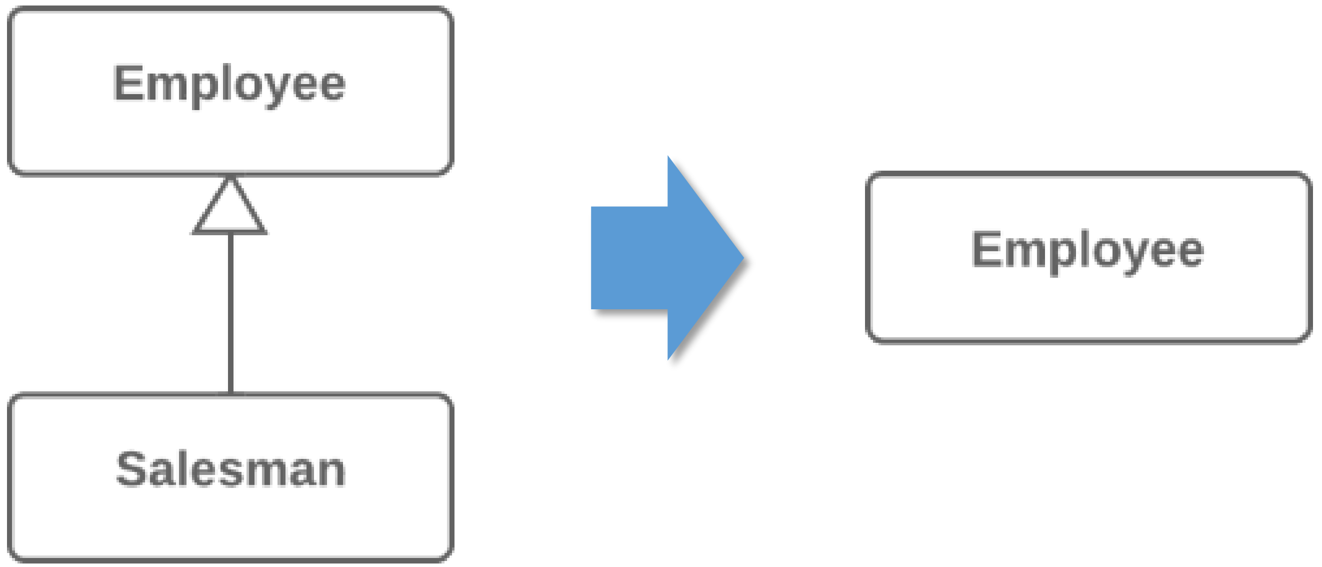
Speculative Generality

- If there is an unused class, method, field or parameter



Refactoring

- **Collapse Hierarchy**
  if a subclass is practically the same as its superclass.



Refactoring

- **Inline method**
  When a method body is more obvious than the method itself.

```java
class PizzaDelivery {
  // ...
  int getRating() {
    return moreThanFiveLateDeliveries() ? 2 : 1;
  }
  boolean moreThanFiveLateDeliveries() {
    return numberOfLateDeliveries > 5;
  }
}
```

```java
class PizzaDelivery {
  // ...
  int getRating() {
    return numberOfLateDeliveries > 5 ? 2 : 1;
  }
}
```

Refactoring

- **Remove parameter**
  When parameter isn’t used in the body of a method.

Dead Code

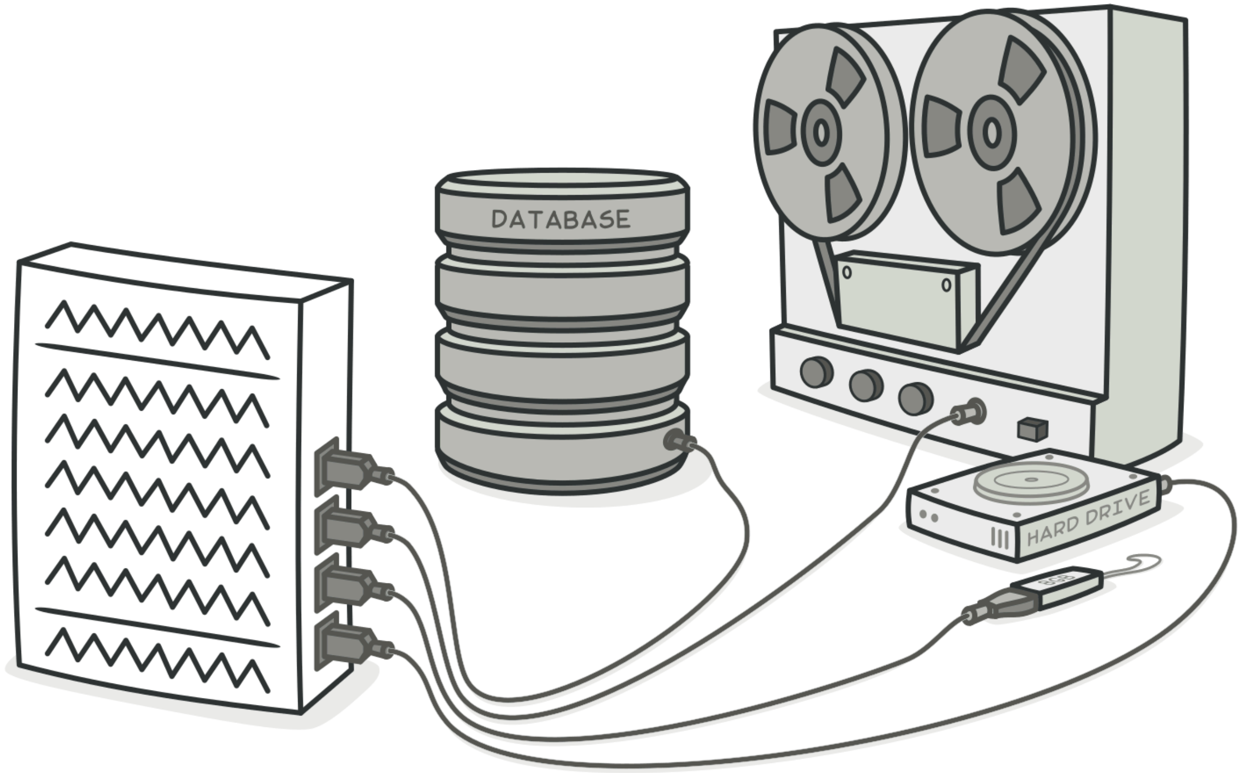
Dead Code

- A variable, parameter, field, method or class is no longer used (usually because it's obsolete)

Data Class

Data Class

- A data class refers to a class that contains only fields.
- These are simply containers for data used by other classes. These classes don’t contain any additional functionality and can’t independently operate on the data that they own.



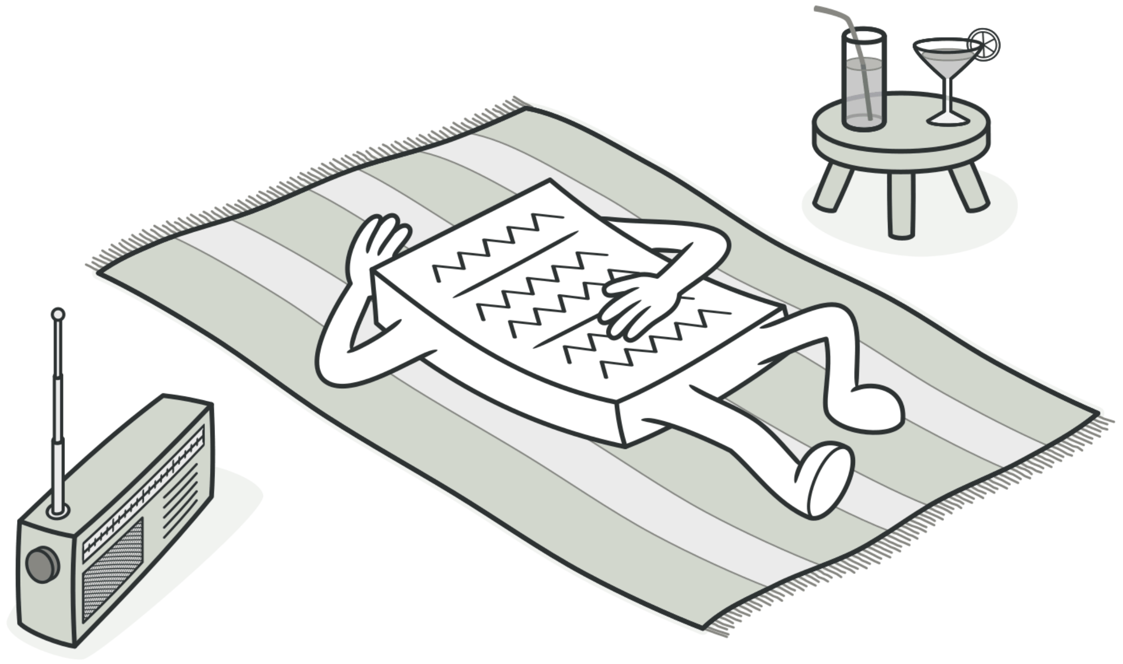
Refactoring

- **Encapsulate field and encapsulate collection**
  to hide them from direct access and require that access be performed via getters and setters only
- **Move method**
  Review the client code that uses the class. In case there is any functionality that would be better located in the data class itself.

Lazy Class

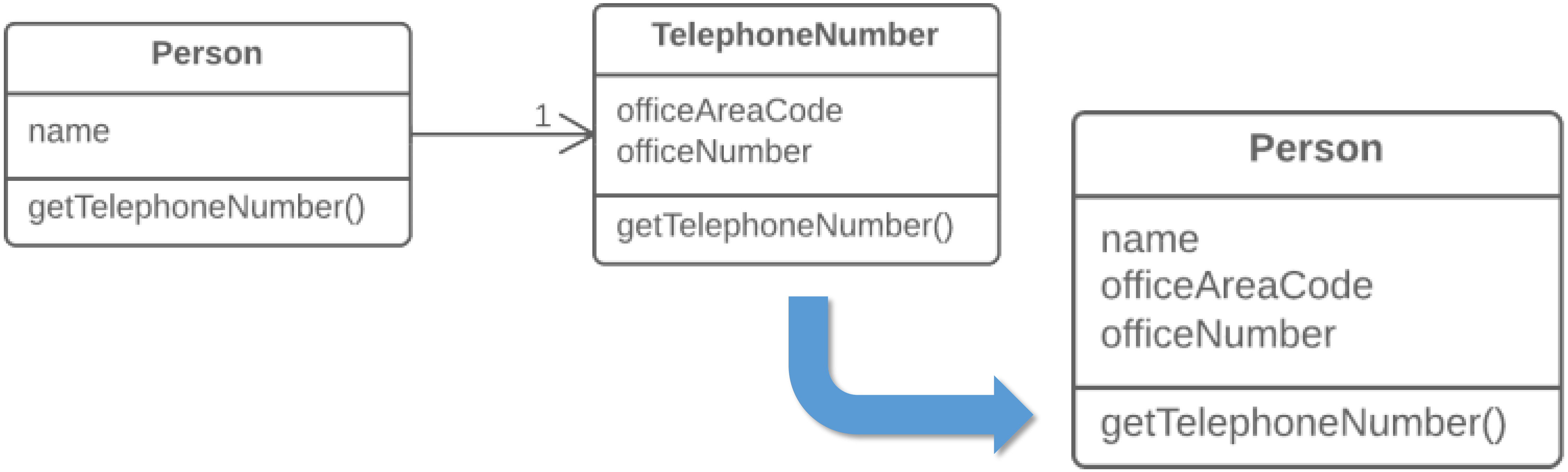
Lazy Class

- if a class doesn’t do enough to earn your attention, it should be deleted



Refactoring

- **Collapse hierarchy**
- **Inline class**



References

- Martin, Robert C. Clean Code: A Handbook of Agile Software Craftsmanship. Pearson. 2008.
- Fowler, Martin. Refactoring: Improving the Design of Existing Code. Addison-Wesley Professional, 1999.
- https://refactoring.guru/
- Rasyid Institute. Modul Workshop Clean Code. 2019.

bridge to the future

http://www.eepis-its.edu