nagarro

05

# Code Smell

# Code smell - Definition

*„A code smell is a surface indication that usually corresponds to a deeper problem in the system.”*

Kent Beck and Martin Fowler

# What Is a Code Smell?

- Warning signs about potential problems in the source code
- Not all of them indicate a problem, but most are worthy of a look and decision
- Helps to identify source code to refactor

# Exercise – Which Code Smells Did You Refactor?

The trainer deals out cards  with different Code Smells. Discuss in least two groups, which smells did you find and refactor in the previous hands-on.

- Make notes why you believe that this smell occurred.
- Are there any refactorings and/or smells missing?

- If you don’t know or understand the meaning of a smell:
  1. Ask the team
  2. Look at https://blog.codinghorror.com/code-smells/
  3. Ask the trainer

- Gather in a circle in front of the board. Each team sticks their notes on the board and presents their findings. Discuss about your most important insights.

# Code Smell Examples

## Within classes

- Comments
- Conditional Complexity
- Dead Code
- Duplicated Code
- Inconsistent Names
- Large Class
- Long Method
- Long Parameter List
- Speculative Generality
- Temporary Field
- Uncommunicative Name
- ...

## Between classes

- Data Class
- Feature Envy
- Inappropriate Intimacy
- Primitive Obsession
- Shotgun Surgery
- ...

# Code Smell Comments

```
public void InvokeProgram (...) {

   ...
   // Run the program.
   if (program.Run() != true) {
      // Report failure.
      LOGGER.Error("Program failed!");

      // Log Error messages.
      List<ErrorMessage> errorMessageList = program.GetErrorMessageList();

      foreach (ErrorMessage msg in errorMessageList) {
         // Show each errorMessage.
         LOGGER.Error(msg.GetText());
         // Load additional error message information.
         msg.Load();
         //Show help text.
         LOGGER.Error(msg.GetHelp());
      }
      ...
   }
   ...
}
```

# Code Smell Comments

```
public void InvokeProgram (...) {

   ...
   // Run the program.
   if (program.Run() != true) {
      // Report failure.
      LOGGER.Error("Program failed!"

      // Log Error messages.
      List<ErrorMessage> errorMessa

      foreach (ErrorMessage msg in e
         // Show each errorMessage.
         LOGGER.Error(msg.GetText()
         // Load additional error me
         msg.Load();
         //Show help text.
         LOGGER.Error(msg.GetHelp()
      }
      ...
   }
   ...
}
```

```
public void InvokeProgramm (...) {

   ...
   if (program.Run() != true) {
      LOGGER.Error("Program failed!");
      LogErrorMessages(program.GetErrorMessageList());
      ...
   }
   ...
}

private void LogErrorMessages(List<ErrorMessage> errorMessageList){
   foreach (ErrorMessage msg in errorMessageList) {
      msg.Load();
      LOGGER.Error("Error message: {}{}Help: {}",
          msg.GetText(),
          Environment.NewLine,
          msg.GetHelp());
   }
}
```

# Code Smell Duplicated Code

```
public XmlElement CreateAddressElement(XmlDocument xmlDocument, Address address) {

  XmlElement element = xmlDocument.CreateElement("Address");
  XmlAttribute tempAttribute = null;

  tempAttribute = xmlDocument.CreateAttribute("Number");
  tempAttribute.SetValue(address.GetNumber());
  element.SetAttributeNode(tempAttribute);

  tempAttribute = xmlDocument.CreateAttribute("Street");
  tempAttribute.SetValue(address.GetStreet());
  element.SetAttributeNode(tempAttribute);

  tempAttribute = xmlDocument.CreateAttribute("City");
  tempAttribute.SetValue(address.GetCity());
  element.SetAttributeNode(tempAttribute);
  ...
  return element;
}
```

# Code Smell Duplicated Code

```
public XmlEle
  XmlElement
  XmlAttribut
  tempAttribu
  tempAttribu
  element.set
  tempAttribu
  tempAttribu
  element.set
  tempAttribu
  tempAttribu
  element.SetAttributeNode(tempAttribute);
  ...
  return element;
}
```

```
public XmlElement CreateAddressElement(XmlDocument xmlDocument, Address address) {

    XmlElement element = xmlDocument.CreateElement("Address");
    element.SetAttributeNode(CreateAttribute(xmlDocument, "Number", address.GetNumber());
    element.SetAttributeNode(CreateAttribute(xmlDocument, "Street", address.GetStreet());
    element.SetAttributeNode(CreateAttribute(xmlDocument, "City", address.GetCity());
    ...
    return element;
}

private XmlAttribute CreateAttribute(XmlDocument xmlDocument, String name, String value) {
    XmlAttribute attribute = xmlDocument.CreateAttribute(name);
    attribute.SetValue(value);
    return attribute;
}
```

# Code Smells Within Classes

| Code Smell | Description |
|---|---|
| Combinatorial Explosion | You have lots of code that does almost the same thing.. but with tiny variations in data or behavior. This can be difficult to refactor-- perhaps using generics or an interpreter? |
| Comments | There's a fine line between comments that illuminate and comments that obscure. Are the comments necessary? Do they explain "why" and not "what"? Can you refactor the code so the comments aren't required? And remember, you're writing comments for people, not machines. |
| Conditional Complexity | Watch out for large conditional logic blocks, particularly blocks that tend to grow larger or change significantly over time. Consider alternative object-oriented approaches such as decorator, strategy, or state. |
| Dead Code | Ruthlessly delete code that isn't being used. That's why we have source control systems! |
| Duplicated Code | Duplicated code is the bane of software development. Stamp out duplication whenever possible. You should always be on the lookout for more subtle cases of near-duplication, too. Don't Repeat Yourself! |
| Inconsistent Names | Pick a set of standard terminology and stick to it throughout your methods. For example, if you have Open(), you should probably have Close(). |
| Large Class | Large classes, like long methods, are difficult to read, understand, and troubleshoot. Does the class contain too many responsibilities? Can the large class be restructured or broken into smaller classes? |
| Long Method | All other things being equal, a shorter method is easier to read, easier to understand, and easier to troubleshoot. Refactor long methods into smaller methods if you can. |

https://blog.codinghorror.com/code-smells/

# Code Smells Within Classes (cont.)

| Code Smell | Description |
| --- | --- |
| Long Parameter List | The more parameters a method has, the more complex it is. Limit the number of parameters you need in a given method, or use an object to combine the parameters.<br>if you really need multiple solutions to the same problem. |
| Oddball Solution | There should only be one way of solving the same problem in your code. If you find an oddball solution, it could be a case of poorly duplicated code-- or it could be an argument for the adapter model, if you really need multiple solutions to the same problem. |
| Speculative Generality | Write code to solve today's problems, and worry about tomorrow's problems when they actually materialize. Everyone loses in the "what if.." school of design. You (Probably) Aren't Gonna Need It. |
| Temporary Field | Watch out for objects that contain a lot of optional or unnecessary fields. If you're passing an object as a parameter to a method, make sure that you're using all of it and not cherry-picking single fields. |
| Type Embedded in Name | Avoid placing types in method names; it's not only redundant, but it forces you to change the name if the type changes. |

https://blog.codinghorror.com/code-smells/

# Code Smells Between Classes

| Code Smell | Description |
| --- | --- |
| Alternative Classes with Different Interfaces | If two classes are similar on the inside, but different on the outside, perhaps they can be modified to share a common interface. |
| Data Class | Avoid classes that passively store data. Classes should contain data and methods to operate on that data, too. |
| Data Clumps | If you always see the same data hanging around together, maybe it belongs together. Consider rolling the related data up into a larger class. |
| Divergent Change | If, over time, you make changes to a class that touch completely different parts of the class, it may contain too much unrelated functionality. Consider isolating the parts that changed in another class. |
| Feature Envy | Methods that make extensive use of another class may belong in another class. Consider moving this method to the class it is so envious of. |
| Inappropriate Intimacy | Watch out for classes that spend too much time together, or classes that interface in inappropriate ways. Classes should know as little as possible about each other. |
| Incomplete Library Class | We need a method that's missing from the library, but we're unwilling or unable to change the library to include the method. The method ends up tacked on to some other class. If you can't modify the library, consider isolating the method. |
| Indecent Exposure | Beware of classes that unnecessarily expose their internals. Aggressively refactor classes to minimize their public surface. You should have a compelling reason for every item you make public. If you don't, hide it. |

https://blog.codinghorror.com/code-smells/

# Code Smells Between Classes (cont.)

| Code Smell | Description |
|---|---|
| Lazy Class | Classes should pull their weight. Every additional class increases the complexity of a project. If you have a class that isn't doing enough to pay for itself, can it be collapsed or combined into another class? |
| Middle Man | If a class is delegating all its work, why does it exist? Cut out the middleman. Beware classes that are merely wrappers over other classes or existing functionality in the framework. |
| Message Chains | Watch out for long sequences of method calls or temporary variables to get routine data. Intermediaries are dependencies in disguise. |
| Parallel Inheritance Hierarchies | Every time you make a subclass of one class, you must also make a subclass of another. Consider folding the hierarchy into a single class. |
| Primitive Obsession | Don't use a gaggle of primitive data type variables as a poor man's substitute for a class. If your data type is sufficiently complex, write a class to represent it. |
| Refused Bequest | If you inherit from a class, but never use any of the inherited functionality, should you really be using inheritance? |
| Solution Sprawl | If it takes five classes to do anything useful, you might have solution sprawl. Consider simplifying and consolidating your design. |
| Shotgun Surgery | If a change in one class requires cascading changes in several related classes, consider refactoring so that the changes are limited to a single class. |

https://blog.codinghorror.com/code-smells/

# Appendix: References


- Refactoring: Improving the Design of Existing Code; Martin Fowler and Kent Beck; Addison Wesley; 1st edition; 28th June 1999
- Refactoring Workbook; William C. Wake; Pearson Education; 1st edition, 4th September 2003
- Refactoring to Patterns; Joshua Kerievsky; Addison Wesley; 1st edition, 5th August 2004
- https://blog.codinghorror.com/code-smells/