Contact Us

FEEDS

Atom
RSS

TOPICS

8th Light University
AWS
Android
Ansible
Apprentice Blog of the Week
Apprenticeship
Architecture
Business
Clojure
ClojureScript
Coding
Communications
Community
Consulting
Craftsmanship
Design
DevOps
Elixir
Inspiration
Java
JavaScript
Learning
Microservices
Mobbing
Pairing
Principles
Process
Quality

Common Code Smells

Georgina Mcfadyen / 19 Jan 2017 Coding Design Learning Craftsmanship

Developers are typically trained to look out for and guard against logical errors that have been accidentally introduced to their code. Such errors will range from forgotten edge cases that have not been handled to logical bugs that cause entire systems to crash. But what about the more subtle issues that don't affect the way the system works? For example, the design issues that make the system hard to maintain, and increase the chance of bugs in the future?

> A code smell is a surface indication that usually corresponds to a deeper problem in the system.
>
> —*Martin Fowler*

Code Smells are signals that your code should be refactored in order to improve extendability, readability, and supportability.

Below describes some of the most common code smells that, when caught early, should not be too difficult to address:

Long Methods

The majority of a programmer's time is spent reading code rather than writing code. Apart from the difficulty of having to keep a lot of complex logic in mind whilst reading through a long method, it is usually a sign that the method

React
Ruby
Testing
Tools
UX Design
Web Development
iOS

has too many responsibilities. Long methods make code hard to maintain and debug. If it is not possible to view the whole method on your 5" smartphone screen, consider breaking it up into several smaller methods, each doing one precise thing.

Refuse Bequest

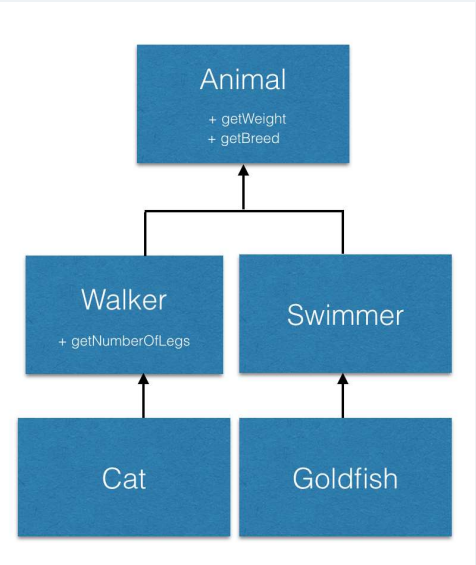
If a class inherits from a base class but doesn't use any of the inherited fields or methods, developers should ask themselves if inheritance really is the right model. Signs of this code smell may be that the inherited methods go unused, or are overridden with empty method bodies. For example:

```
class Animal {
  private int numberOfLegs;

  ...

    public getNumberOfLegs() {
      return numberOfLegs;
    }
}

class Cat extends Animal {
  public getNumberOfLegs() {
    super.getNumberOfLegs();
  }
}

class GoldFish extends Animal {

  ...

    public getNumberOfLegs() {
      throw NotImplementedException();
    }
}
```

Inheritance should be used when a class wants to reuse the code in its superclass. If the classes diverge and the

subclass no longer needs that functionality, the hierarchy should be broken and delegation considered instead.

Alternatively, to keep some inheritance, remove the unused fields and methods from the subclass and create a new layer that the objects can inherit from. For example:



Data Clumps

Where multiple method calls take the same set of parameters, it may be a sign that those parameters are related. To keep the group of parameters together, it can be useful to combine them together in a class. This can help aid organisation of code.

For example:

```
connect(String host, int port, String username);
```

would become:

```
class DatabaseCredentials {
  ...
    public String getHost() {
      return host;
    }
```

```
  public int getPort() {
    return port;
  }

  public String getUsername() {
    return username;
  }
}

...

connect(DatabaseCredentials databaseCredentials);
```

Duplicate Code

It's been known for a developer to fix a bug, only for the same symptoms to then resurface in a slightly different part of the system. This can be the result of code duplication, and a bug being fixed in one occurrence of the imperfect code but not in the duplicated versions. This poses an overhead in terms of maintenance. When developers are not aware of the duplication, they only know to fix the occurrence they have come across.

Be on the lookout for repeated code blocks and extract them out into a single place—don't repeat yourself!

Middle Man

When a class exists just to delegate to another, a developer should ask themselves what its real purpose is. Sometimes this is the result of a refactoring task, where logic has been moved out of a class gradually, leaving an almost empty shell. For example:

```
class Pet {
  private Animal animal;

  public String getName() {
    return animal.getName();
  }
```

```
  public String getBreed() {
    return animal.getBreed();
  }

  public Owner getOwner() {
    return animal.getOwner();
  }
}
```

For every class that exists, there is an overhead of maintenance. Make sure the class justifies its existence, and if it doesn't, go ahead and remove it.

Primitive Obsession

Primitive types give little in terms of domain context. A `String id` field could ultimately contain any sort of value. Where primitives have a domain meaning, wrap them in a small class to represent the idea. Often the class is expanded to include methods to add to the class.

For example:

Rather than

```
String id = product.getId();
if (isValid(id)) {
  ...
}
```

use

```
ProductId id = product.getId();
if (id.isValid()) {
  ...
}

class ProductId {

  public bool isValid() {
    ...
```

```
    }
  }
```

Comments

Can comments be trusted? Where comments are re-iterating what can be read by a developer, they may provide little value, especially when they have not been updated and no longer reflect the intent of the current code.

Rather than adding a comment to clarify a piece of code, think about whether the code can be refactored such that the comment is no longer needed.

It may be possible to provide a more descriptive name that provides the same clarity as the comment, meaning the comment can disappear, resulting in more intuitive and readable code.

Conclusion

Just because the code isn't difficult to maintain or extend now, be on the lookout for code smells so that small refactoring can be made early to prevent larger problems in the future.

Whilst only a handful of examples have been described in this article, become familiar with the different categories of code smells, and see which ones are most prominent in the project you are working on. Share the solutions and refactorings amongst the development team so that going forward your code becomes less smelly and more sleek.

Georgina McFadyen is a Software Crafter at 8th Light London, who enjoys the variety of developing applications using craftsmanship techniques.

Follow @gemcfadyen

RELATED POSTS

Diversity, Inclusion, and 8th Light Paul Pagel
The Principles of Chess and Software Ian Carroll
Does the open/closed principle apply to Elm? Rob Looby
Project Build Protocol Jeff Ramnani
Managing Cloud Resources with CloudFormation Jeff Ramnani
A Conversation with Jessi Chartier from App Camp for Girls Becca Nelson
How JSON decoding works in Elm—Part 2 Kofi Gumbs
Hedy Lamarr: More Than the Most Beautiful Woman in the World Elizabeth Engelman
How JSON decoding works in Elm—Part 1 Kofi Gumbs
API Assumption Tests Christoph Gockel

MORE POSTS BY THIS AUTHOR

Using an Elixir Umbrella
Working Effectively with Offshore Teams
A Good Craftsman Never Blames His Tools
TDD - From the Inside Out or the Outside In?

Interested in 8th Light's services? Let's talk.

Contact Us

Contact

Privacy Policy

Apprenticeship

Blog

Twitter

Sitemap

Chicago

London

Los Angeles

New York