CSE 470
Software Engineering
Refactoring and Code Smells

Imran Zahid
Lecturer
Computer Science and Engineering, BRAC University

Implementation Phase

Implementation

- No design decisions are taken during the implementation phase.
- The design is converted into corresponding code depending on the programming language, and the libraries and frameworks used.
- As a result, the implementation phase largely depends on the stack being used and so, is not discussed in the course.
- However some steps are common to any and all languages, which will be discussed.

BRAC UNIVERSITY
Inspiring Excellence

Refactoring

What is Refactoring?

- A series of small steps, each of which changes the program's internal structure without changing its external behavior - Martin Fowler
- Verify no change in external behavior by
  - Testing
  - Using various tools - e.g. an IDE
  - Formal code analysis by tools
  - Being very, very careful!

What if you hear...

- ✗ We'll just refactor the code to support logging
- ✗ Can you refactor the code so that it authenticates against LDAP instead of Database?
- ✓ We have too much duplicate code, we need to refactor the code to eliminate duplication
- ✓ This class is too big, we need to refactor it
- ✗ Caching?

Why do we Refactor?

- Helps us deliver more business value faster
- Improves the design of our software
- Minimizes technical debt
- Keeps development at speed
- Makes the software easier to understand
- Write for people, not the compiler
- To help find bugs

Technical Debt

The cost of additional rework caused by choosing quick and easy solutions over better long-term approaches in software development.

- Short-term compromises in code or design for quick delivery.
- Accumulates as rework required later.
- Results in higher maintenance costs and slower future progress.
- Managed by refactoring and addressing during planning.

Readability

Which code segment is easier to read?

Sample 1

```
if (date.Before(Summer_Start) || date.After(Summer_End)):
     charge = quantity   winterRate + winterServiceCharge
else
     charge = quantity   summerRate
```

Sample 2

```
if (IsSummer(date)):
     charge = SummerCharge(quantity)
else:
     charge = WinterCharge(quantity)
```

When should you refactor?

- To prepare to add new functionality
  - refactor existing code until you understand it
  - refactor the design to make it simple to add
- To find bugs
  - refactor to understand the code
- For code reviews
  - immediate effect of code review
  - allows for higher level suggestions

Two Sides of Implementation

Adding Functionality

- Adds new capabilities to the
- system
- Adds new tests
- Gets the tests working

Refactoring

- Does not add any new features
- Does not add tests (but may change some)
- Restructures the code to remove redundancy

How do we Refactor?

- We looks for Code Smells
  - Things that we suspect are not quite right or will cause us severe pain if we do not fix
- We follow the guiding principles
  - First do no harm
  - Baby steps

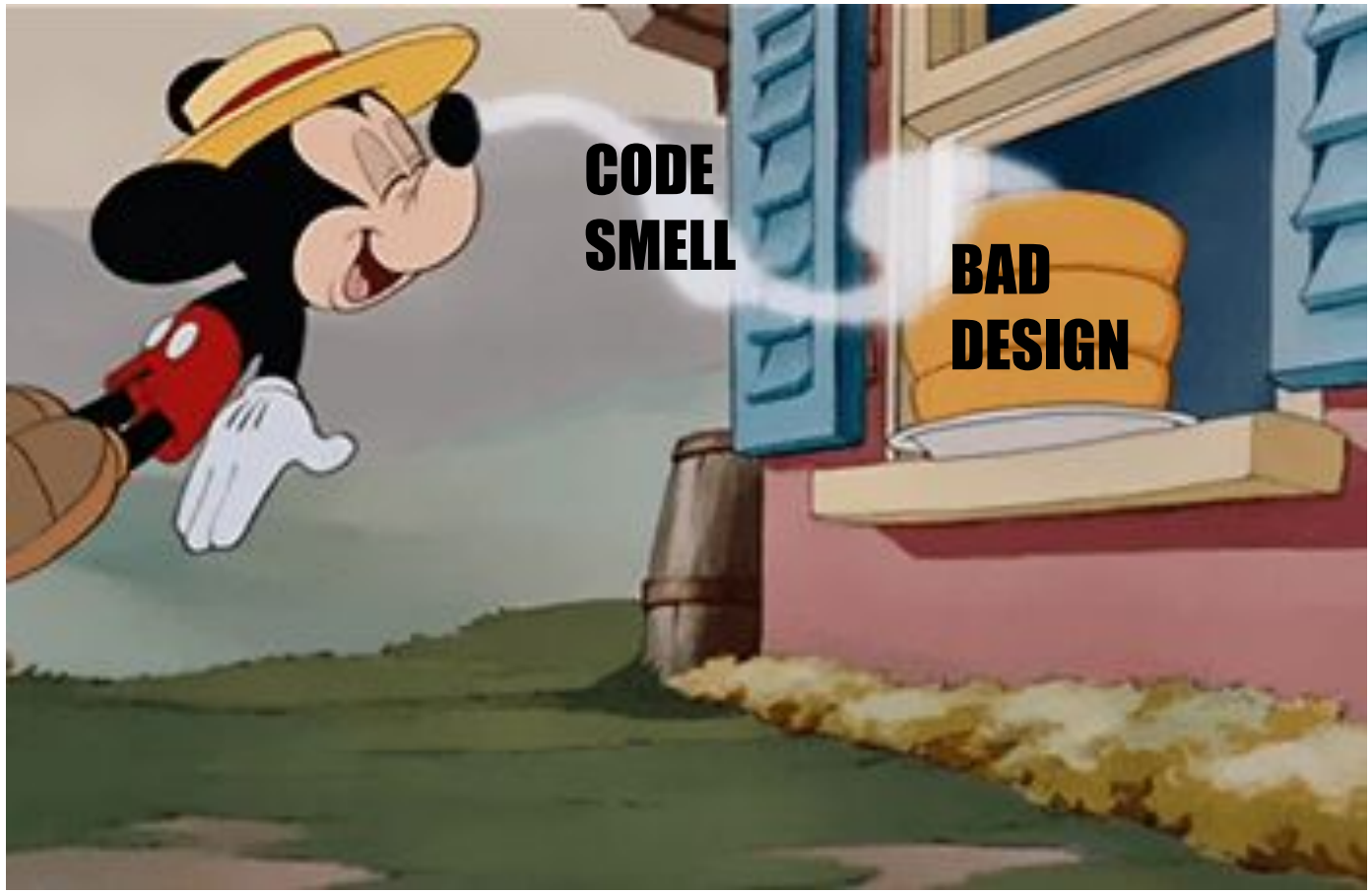
BRAC UNIVERSITY
Inspiring Excellence

Code Smells

Code Smells

- Code Smells identify frequently occurring design problems in a way that is more specific or targeted than general design guidelines (like “loosely coupled code” or “duplication-free code”). - Joshua K
- A code smell is a design that duplicates, complicates, bloats or tightly coupled code
- If it stinks, change it!
- Kent Beck coined the term code smell to signify something in code that needed to be changed.

Code Smells



Common Code Smells

- Inappropriate Naming
- Comments
- Dead Code
- Duplicated code
- Primitive Obsession
- Large Class
- Lazy Class
- Alternative Class with
- Different Interface
- Long Method
- Long Parameter List
- Switch Statements
- Speculative Generality
- Oddball Solution
- Feature Envy
- Refused Bequest
- Black Sheep
- Train Wreck

Code Smell - Inappropriate Naming

- Names given to variables (fields) and methods should be clear and meaningful.
- A variable name should say exactly what it is.
- Which is better?
  - private string s; OR private string salary;
- A method should say exactly what it does.
  - Which is better?
  - public double calc (double s)
  - public double calculateFederalTaxes (double salary)

Code Smell - Comments

- Comments are often used as deodorant.
- Comments represent a failure to express an idea in the code. Try to make your code self-documenting or intention-revealing.
- When you feel like writing a comment, first try to refactor so that the comment becomes unnecessary.
- Remedies:
  - Extract Method
  - Rename Method
  - Introduce Assertion

Example - “Grow the Array”

```
public class MyList {
    int INITIAL_CAPACITY = 10;
    bool m_readOnly;
    int m_size = 0;
    int m_capacity;
    string[] m_elements;

    public MyList() {
        m_elements = new string[INITIAL_CAPACITY];
        m_capacity = INITIAL_CAPACITY;
    }

    int GetCapacity() {
        return m_capacity;
    }
```

```
    void AddToList(string element) {
        if (!m_readOnly) {
            int newSize = m_size + 1;
            if (newSize > GetCapacity()) {
                // grow the array
                m_capacity += INITIAL_CAPACITY;
                string[] elements2 = new string[m_capacity];
                for (int i = 0; i < m_size; i++) {
                    elements2[i] = m_elements[i];
                }
                m_elements = elements2;
            }
            m_elements[m_size++] = element;
        }
    }
}
```

Example - Fixed

```
void AddToList(string element) {
    if (m_readOnly)
        return;
    if (ShouldGrow())
    {
        Grow();
    }
    StoreElement(element);
}
```

```
private bool ShouldGrow() {
    return (m_size + 1) > GetCapacity();
}
```

```
private void Grow() {
    m_capacity += INITIAL_CAPACITY;
    string[] elements2 = new string[m_capacity];
    for (int i = 0; i < m_size; i++)
        elements2[i] = m_elements[i];

    m_elements = elements2;
}
```

```
private void StoreElement(string element) {
    m_elements[m_size++] = element;
}
```

Code Smell - Comments

Document the “Why,” Not the “What”

Use comments to explain why a decision was made, not what the code does.

Example:

// We use a recursive algorithm here because it handles edge cases effectively

Comments: Rename Method

- Renaming a method is an effective solution to a comment code smell when the comment exists to explain what the method does.
- In this case, a clearer method name can eliminate the need for the comment by making the code self-explanatory.

Comments: Rename Method

Comments: Rename Method

```
// This method calculates the total price including tax
public double calc(double price, double tax) {
    return price + (price * tax);
}
```

```
public double calculateTotalPrice(double price, double tax) {
    return price + (price * tax);
}
```

Comments: Extract Method

- Extracting a method is a solution to a comment code smell when a comment is used to explain a specific block of code within a larger method.
- This often indicates that the code block performs a distinct task, and extracting it into a separate, well-named method can make the code self-explanatory, eliminating the need for the comment.

Comments: Extract Method

```
void PrintOwning(double amount){
      PrintBanner();
      // print details
      System.Console.Out.WriteLine(“name:” + name);
      System.Console.Out.WriteLine(“amount:” + amount);
}
```

Comments: Extract Method

```
void PrintOwning(double amount){
      PrintBanner();
      // print details
      System.Console.Out.WriteLine("name:" + name);
      System.Console.Out.WriteLine("amount:" + amount);
}
```

```
void PrintOwning(double amount){
      PrintBanner();
      PrintDetails(amount);
}
void PrintDetails(double amount){
      System.Console.Out.WriteLine("name:" + name);
      System.Console.Out.WriteLine("amount:" + amount);
}
```

Comments: Introduce an Assertion

- Introducing an assertion is a solution to a comment code smell when a comment is being used to specify a precondition, postcondition, or invariant in the code.
- Assertions can programmatically enforce these conditions, making the code self-documenting and ensuring that the conditions are always checked during execution.

Comments: Introduce Assertion

```
double getExpenseLimit() {
      // should have either expense limit or a primary project
      return (_expenseLimit != NULL_EXPENSE) ? _expenseLimit :  _primaryProject.GetMemberExpenseLimit();
}
```

Comments: Introduce Assertion

```
double getExpenseLimit() {
      // should have either expense limit or a primary project
      return (_expenseLimit != NULL_EXPENSE) ? _expenseLimit :  _primaryProject.GetMemberExpenseLimit();
}
```

```
      double getExpenseLimit() {
            Assert(_expenseLimit != NULL_EXPENSE || _primaryProject != null, “Both Expense Limit and
      Primary Project must not be null”);
            return (_expenseLimit != NULL_EXPENSE) ? _expenseLimit :
      _primaryProject.GetMemberExpenseLimit();
      }
```

Thank you