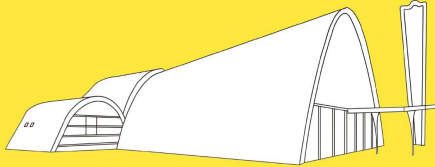


# SOFTWARE ENGINEERING

A Modern Approach



MARCO TULLIO VALENTE

## Chapter 9 - Refactoring

Prof. Marco Tulio Valente

<https://softengbook.org>

CC-BY: This license enables anyone to distribute, remix, adapt, and build upon the material in any medium or format, so long as attribution is given to the author.

# Software Maintenance

- Preventive: bugs not yet reported
- Corrective: bugs reported by users
- Adaptive: customizations, new PL/OS versions, etc
- Evolutionary: new features
- Refactoring: code or design improvements

# Refactoring

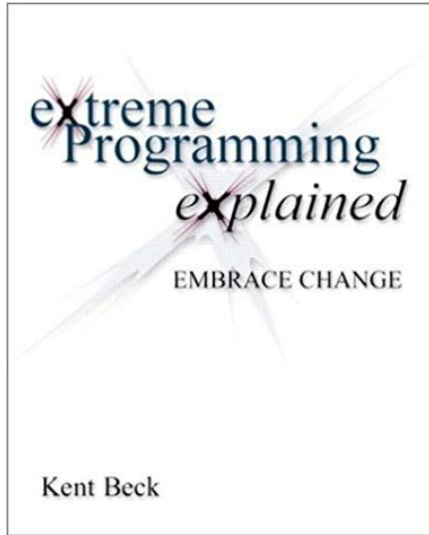
- Code transformations that improve maintainability without affecting external behavior

REFACTORING OBJECT-ORIENTED FRAMEWORKS

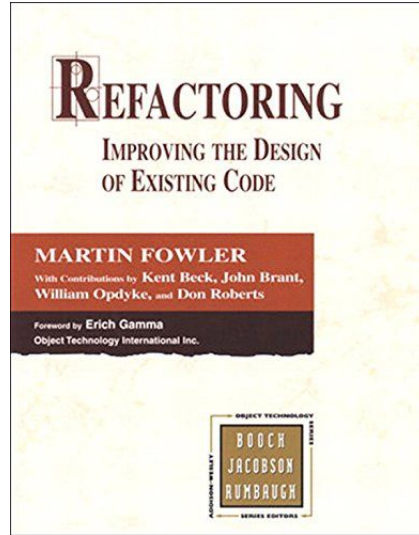
William F. Opdyke, Ph.D.  
Department of Computer Science  
University of Illinois at Urbana-Champaign, 1992  
Ralph E. Johnson, Advisor

This thesis defines a set of program restructuring operations (refactorings) that support the design, evolution and reuse of object-oriented application frameworks.

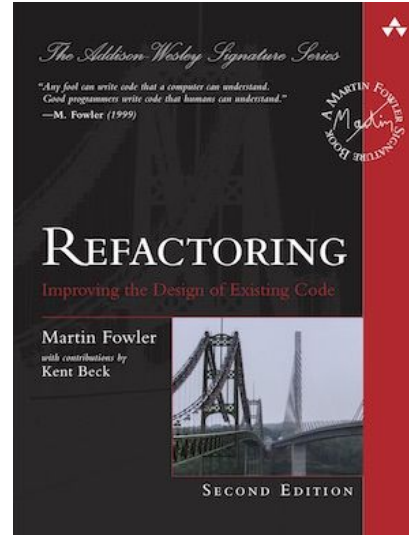
# Idea has become quite popular ...



1999



2000



2018

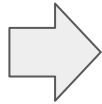
# Catalog of Refactorings

- Extract Method
- Inline Method
- Move Method
- Extract Class
- Renaming
- etc

# Method Extraction

# Method Extraction

```
void f() {  
    ... // A  
    ... // B  
    ... // C  
}
```



```
void g() { // extracted method  
    ... // B  
}  
  
void f () {  
    ... // A  
    g();  
    ... // C  
}
```

A real example ...



```
void onCreate(SQLiteDatabase database) {// before extraction
    // creates table 1
    database.execSQL("CREATE TABLE " +
        CELL_SIGNAL_TABLE + " (" + COLUMN_ID +
        " INTEGER PRIMARY KEY AUTOINCREMENT, " + ...
    database.execSQL("CREATE INDEX cellID_index ON " + ...);
    database.execSQL("CREATE INDEX cellID_timestamp ON " +...);

    // creates table 2
    String SMS_DATABASE_CREATE = "CREATE TABLE " +
        SILENT_SMS_TABLE + " (" + COLUMN_ID +
        " INTEGER PRIMARY KEY AUTOINCREMENT, " + ...
    database.execSQL(SMS_DATABASE_CREATE);
    String ZeroSMS = "INSERT INTO " + SILENT_SMS_TABLE +
        " (Address,Display,Class,ServiceCtr,Message) " +
        "VALUES ('"+ ...
    database.execSQL(ZeroSMS);

    // creates table 3
    String LOC_DATABASE_CREATE = "CREATE TABLE " +
        LOCATION_TABLE + " (" + COLUMN_ID +
        " INTEGER PRIMARY KEY AUTOINCREMENT, " + ...
    database.execSQL(LOC_DATABASE_CREATE);
    // more 200 lines, creating other tables
}
```

## Before

```
public void onCreate(SQLiteDatabase database) {  
    createCellSignalTable(database);  
    createSilentSmsTable(database);  
    createLocationTable(database);  
    createCellTable(database);  
    createOpenCellIDTable(database);  
    createDefaultMCCTable(database);  
    createEventLogTable(database);  
}
```

## After



# Inline Methods (opposite of extraction)

## Before

```
private void writeContentToFile(final byte[] revision) {  
    getVirtualFile().setBinaryContent(revision);  
}  
  
private void write(byte[] revision) {  
    VirtualFile virtualFile = getVirtualFile();  
    ...  
    if (document == null) {  
        writeContentToFile(revision);  
    }  
    ...  
}
```

```
private void writeContentToFile(final byte[] revision) {
    getVirtualFile().setBinaryContent(revision);
}

private void write(byte[] revision) {
    VirtualFile virtualFile = getVirtualFile();
    ...
    if (document == null) {
        writeContentToFile(revision);
    }
    ...
}
```



```
private void write(byte[] revision) {
    VirtualFile virtualFile = getVirtualFile();
    ...
    if (document == null) {
        virtualFile.setBinaryContent(revision); // after inline
    }
    ...
}
```

**After**

# Move Method

```
- System.out.println("Average among the N/3 median times: " + PlatformTestUtil.averageAmongMedians(time, 3) + "ms");
+ System.out.println("Average among the N/3 median times: " + ArrayUtil.averageAmongMedians(time, 3) + "ms");

//System.out.println("JobLauncher.COUNT = " + JobLauncher.COUNT);
//System.out.println("JobLauncher.TINY = " + JobLauncher.TINY_COUNT);

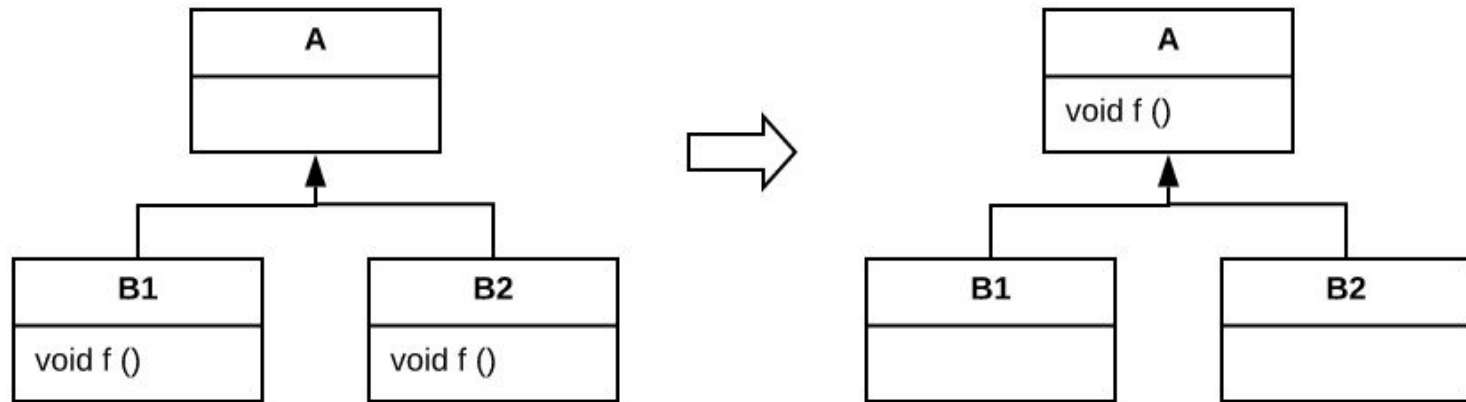
@@ -205,7 +205,7 @@ public int compare(HighlightInfo o1, HighlightInfo o2) {
}
FileEditorManagerEx.getInstanceEx(getProject()).closeAllFiles();

- System.out.println("Average among the N/3 median times: " + PlatformTestUtil.averageAmongMedians(time, 3) + "ms");
+ System.out.println("Average among the N/3 median times: " + ArrayUtil.averageAmongMedians(time, 3) + "ms");
```

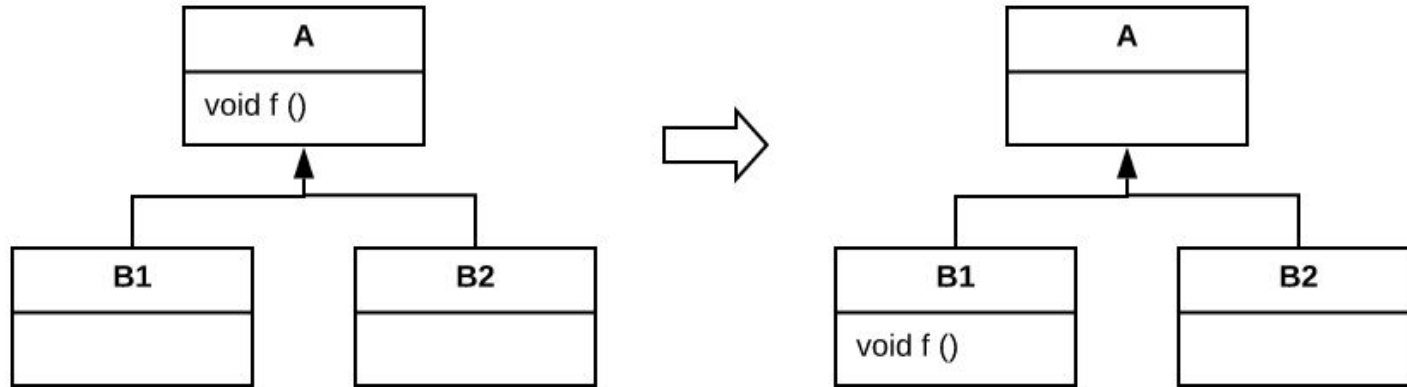


# Specific cases of Move Method (along a class hierarchy)

# Pull Up Method

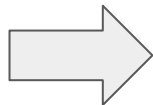


# Push Down Method



# Extract Class

```
class Person {  
    String areaCode;  
    String phone;  
    String alternativeAreaCode;  
    String alternativePhone;  
    ...  
}
```



```
class Phone { // extracted class  
    String areaCode;  
    String number;  
}  
  
class Person {  
    Phone phone;  
    Phone alternativePhone;  
    ...  
}
```

# Renaming

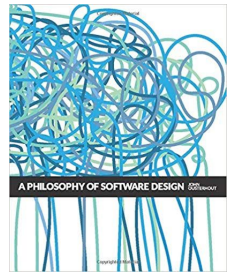
(variable, parameter, method, class, exception, etc)

Giving good names to variables is one of the  
hardest problems in programming!

# Refactoring Practice



# Refactorings & Tests



Developers avoid refactoring without good test suites.

Instead, they try to minimize the number of code changes for each new feature or bug fix...

Which means that complexity accumulates and design mistakes don't get corrected.

-- John Ousterhout

# When to refactor?

1. Opportunistic Refactorings
2. Planned Refactorings

# Opportunistic Refactorings

- Performed in the midst of another task
- Most common type of refactoring

# Planned (or Scheduled) Refactorings

- Correction of a complex design problem
- Sessions just for performing refactorings

# Automated Refactorings

(performed with the help of an IDE)












```

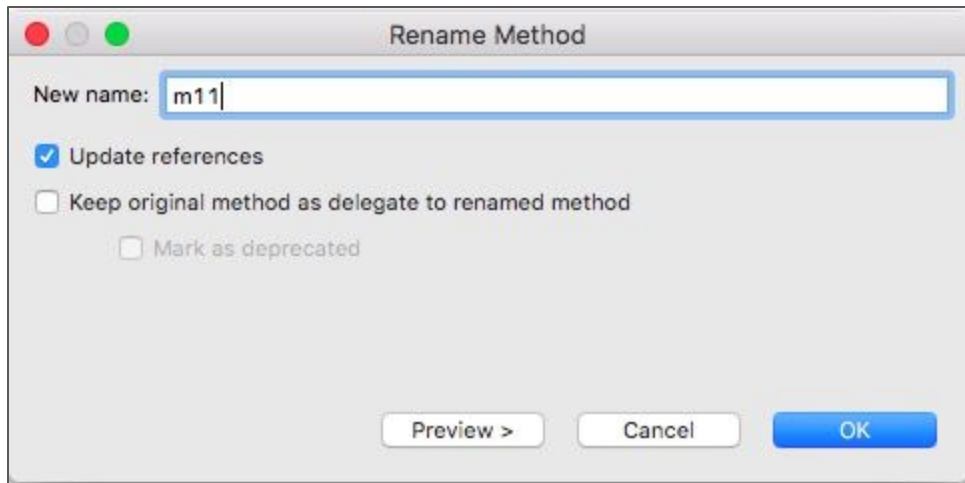
class A {
    void m1() {}
    void m2() {}
    void m3() { m1(); }
}

class B {
    void m4() { new A().m1(); }
}

```



Refactor 	▶	Rename...	 ⌘ R
Local History	▶	Move...	 ⌘ V
References	▶	Change Method Signature...	 ⌘ C
Declarations	▶	Inline...	 ⌘ I
 Add to Snippets...		Move Type to New File...	
 Coverage As	▶	Extract Interface...	
 Run As	▶	Extract Superclass...	
 Debug As	▶	Use Supertype Where Possible...	
Team	▶	Pull Up...	
Compare With	▶	Push Down...	
Replace With	▶	Extract Class...	
<input checked="" type="checkbox"/> Validate		Introduce Parameter Object...	
Preferences...		Introduce Indirection...	
 Remove from Context 		Infer Generic Type Arguments...	



```
class A {  
    void m11() {}  
    void m2() {}  
    void m3() { m11(); }  
}  
  
class B {  
    void m4() { new A().m11(); }  
}
```

# Another Example: Remove Dead Code

- Code that is no longer being used
- More common than we think...



# Case Study: Meta/Facebook

- Internal tool to remove dead code

```
-class PhotoViewLoggingEndpoint {  
-  public function getResponse() {  
-    ...  
-  }  
-  
-  public static function getLogResults() {  
-    ...  
-  }  
-}
```

## CHANGE SUMMARY

Here's why class PhotoViewLoggingEndpoint is unused:

- This class is not syntactically referenced in other code
- Endpoint has not been accessed in production for 1 month
- Endpoint is mapped to path `"/photo/view/"` which does not appear in code
- Method `getLogResults` is dead:
  - This method is not configured in job execution service

# Usage Stats

- Tool was used to analyze hundreds of MLOC
- In 5 years, it helped to delete +100 MLOC, via 370K PR

Original sentence from the article (since the numbers above are very high):

SCARF has grown to analyze hundreds of millions of lines of code; and five years on, it has automatically deleted more than 100 million lines of code in over 370,000 change requests.

# Exercises

1. What is the relationship between the following sentence and the practice of refactoring?

“For each desired change, make the change easy (warning: this may be hard), then make the easy change.”

-- Kent Beck

2. Give the names of refactorings A and B that, if executed in sequence, do not change the system's code.

Thus, refactoring B undoes the transformations made by A.

3. Normally, the application of a refactoring depends on certain preconditions. For example:

(a) When is it not possible to rename a local variable named “a” to “b”?

(b) When is it not possible to move a method “f” from class A to class B?

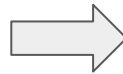
4. (a) What code transformation was performed in this Java program? (b) Is it a refactoring? Justify.

```
class A {
    void f(){ print("hi");}
}

class B extends A {
    ...
}

class C {
    void f(){ print("hello");}
}

main() {
    B b = new B();
    b.f();
}
```



```
class A {
    void f(){ print("hi");}
}

class B extends A {
    void f(){ print("hello");}
}

class C {
    ...
}

main() {
    B b = new B();
    b.f();
}
```

5. (a) What code transformation was performed in this Java program? (b) Is it a refactoring? Justify.

package1/A.java

```
package package1;
public class A {
    void n() { (new B()).m("abc");
}
}
```

package1/A.java

```
package package1;
public class A {
    void n() { (new B()).m("abc");
}
}
```

package1/B.java

```
package package1;
public class B {
    public void m(Object o) {...}
    void m(String s) {...}
}
}
```



package2/B.java

```
package package2;
public class B {
    public void m(Object o) {...}
    void m(String s) {...}
}
}
```



6. Is a change made to improve the performance of a system a refactoring?

Exercise on software design, testability,  
and refactoring

First, study the following code

```
import static javax.swing.JOptionPane.showMessageDialog;

class Dashboard {
    private Stock stock;

    public Dashboard(Stock stock) {
        this.stock = stock;
    }

    public void alert() {
        showMessageDialog(null, "New price " + stock.getName() + " " +
                               stock.getPrice());
    }
}
```



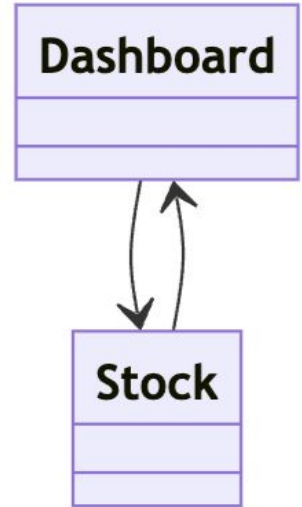
```
class Stock {  
    private String name;  
    private float price;  
    private Dashboard dashboard;  
  
    public Stock(String name, float price) {  
        this.name = name;  
        this.price = price;  
    }  
  
    public void setDashboard(Dashboard dashboard) {  
        this.dashboard = dashboard;  
    }  
  
    public String getName() {  
        return this.name;  
    }  
  
    public float getPrice() {  
        return this.price;  
    }  
}
```

```
// Continuation of the Stock class

    public void updatePrice(float price) {
        this.price = price;
        dashboard.alert();
    }
}

public class Main {
    public static void main(String[] args) {
        Stock stock = new Stock("PETR", 40);
        Dashboard dashboard = new Dashboard(stock);
        stock.setDashboard(dashboard);
        stock.updatePrice(50);
    }
}
```

- In the previous code, there is a circular dependency between Dashboard and Stock
- Circular dependencies are indicators of design and testability issues
- For example, why is it difficult to write a unit test for the updatePrice() method?



# Exercises

1. Refactor the code to remove the circular dependency between the classes.
2. Why is it now easier to write a unit test for `updatePrice()`?

Answer

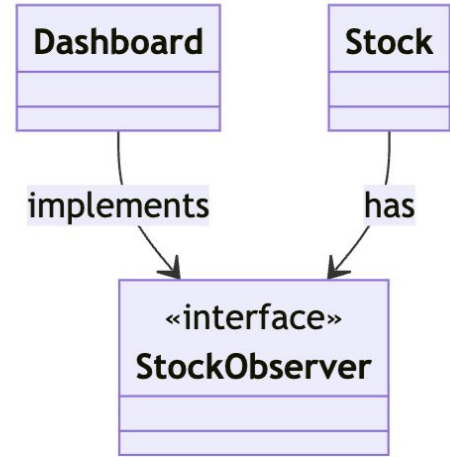


```
interface StockObserver {
    public void alert();
}

class Dashboard implements StockObserver {
    ...
    public void alert() {
        showMessageDialog(null, ...);
    }
}

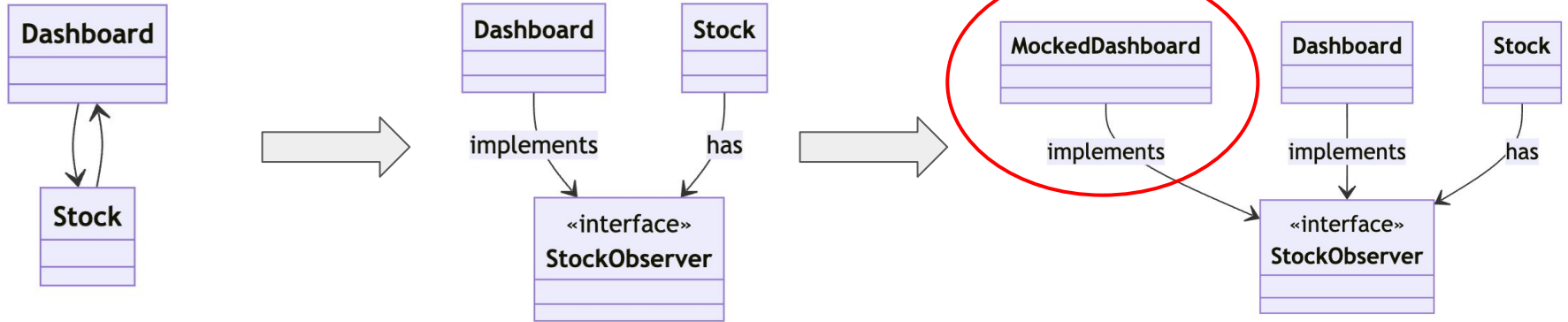
class Stock { ...
    private StockObserver observer;

    public void setObserver(StockObserver observer)
    {
        this.observer = observer;
    }
    ...
    public void updatePrice(float price) {
        this.price = price;
        observer.alert();
    }
}
```



```
public class Main {  
    public static void main(String[] args) {  
        Stock stock = new Stock("PETR", 40);  
        Dashboard dashboard = new Dashboard(stock);  
        stock.setObserver(dashboard);  
        stock.updatePrice(50);  
    }  
}
```

# Summarizing



# Code Smells

# Code (or Bad) Smells

- Indicators of low-quality code
- Code that is hard to maintain, understand, modify or test
- Therefore, candidate for refactoring

## **Chapter 3** **Bad Smells in Code**

*by Kent Beck and Martin Fowler*

*“If it stinks, change it.”*

*— Grandma Beck, discussing child-rearing philosophy*

By now you have a good idea of how refactoring works. But just because you know how doesn't mean you know when. Deciding when to start refactoring—and when to stop—is just as important to refactoring as knowing how to operate the mechanics of it.

# Catalog of Code Smells

- Duplicated Code
- Long Methods
- Large Classes
- Feature Envy
- Long Parameter List
- Global Variables

- Primitive Obsession
- Mutable Objects
- Data Classes
- Comments

# Duplicated Code

# Duplicated Code

- Makes maintenance more difficult
- Therefore, candidate for refactoring



### RANKING OF MOST POPULAR CODE SMELLS/ANTI-PATTERNS

<b>Smell/Anti-Pattern</b>	<b>Points</b>
1. Duplicated code	19.53
2. Long method	9.78
3. Accidental complexity	8.32
4. Large class	7.09
5. Excessive use of literals	3.04
6. Suboptimal information hiding	2.70
7. Lazy class	2.33
8. Feature Envy	2.33
9. Long parameter list	2.31
10. Dead code	2.25
11. Bad (or lack of good) comments	1.50
12. Use deprecated components	1.50
13. Single Responsibility	1.20
14. Complex conditionals	1.12
15. Bad naming	1.12

Aiko Yamashita, Leon Moonen. Do developers care about code smells? An exploratory survey. WCRE 2013.

Duplicated Code  $\Rightarrow$  Clones

# Clone Type 1 (comments and spaces)

```
int factorial(int n) {  
    fat = 1;  
    for (i = 1; i <= n; i++)  
        fat = fat * i;  
    return fat;  
}
```

Original code

```
int factorial(int n) {  
    fat=1;  
    for (i=1; i<=n; i++)  
        fat=fat*i;  
    return fat; // returns factorial  
}
```

# Clone Type 2 (type 1 + different names)

```
int factorial(int n) {  
    fat = 1;  
    for (i = 1; i <= n; i++)  
        fat = fat * i;  
    return fat;  
}
```

Original code

```
int factorial(int n) {  
    f = 1;  
    for (j = 1; j <= n; j++)  
        f = f * j;  
    return f;  
}
```

# Clone Type 3 (type 2 + changes in commands)

```
int factorial(int n) {  
    fat = 1;  
    for (i = 1; i <= n; i++)  
        fat = fat * i;  
    return fat;  
}
```

Original code

```
int factorial(int n) {  
    fat = 1;  
    for (j = 1; j <= n; j++)  
        fat = fat * j;  
    System.out.println(fat); // new command  
    return fat;  
}
```

# Clone Type 4 (equivalent algorithms)

```
int factorial(int n) {  
    fat = 1;  
    for (i = 1; i <= n; i++)  
        fat = fat * i;  
    return fat;  
}
```

Original code

```
int factorial(int n) {  
    if (n == 0)  
        return 1;  
    else return n*factorial(n-1);  
}
```

# Don't DRY Your Code Prematurely

<https://testing.googleblog.com/2024/05/dont-dry-your-code-prematurely.html>

DRY = Don't Repeat Yourself

While functions may look the same, they may also serve different requirements that evolve differently over time.



# Don't DRY Your Code Prematurely

```
# Repetitive but allows for clear, entity-specific  
# logic and future changes.
```



```
{ def set_task_deadline(task_deadline):  
    if task_deadline <= datetime.now():  
        raise ValueError("Date must be in the future")
```




```
{ def set_payment_deadline(payment_deadline):  
    if payment_deadline <= datetime.now():  
        raise ValueError("Date must be in the future")
```

```
set_task_deadline(datetime(2024, 3, 12))  
set_payment_deadline(datetime(2024, 3, 18))
```

# Don't DRY Your Code Prematurely

```
# Premature DRY abstraction assuming uniform rules,  
# limiting entity-specific changes.
```

```
class DeadlineSetter:  
    def __init__(self, entity_type):  
        self.entity_type = entity_type
```

 {

```
    def set_deadline(self, deadline):  
        if deadline <= datetime.now():  
            raise ValueError("Date must be in the future")
```

```
task = DeadlineSetter("task")  
task.set_deadline(datetime(2024, 3, 12))
```

```
payment = DeadlineSetter("payment")  
payment.set_deadline(datetime(2024, 3, 18))
```

# Feature Envy

# Feature Envy

- Method that "envies" data and methods of another class
- Uses more methods and data from another class
- Therefore, it is a candidate to be moved to it

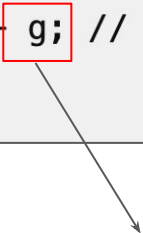
```
public class DrawingEditorProxy
    extends AbstractBean implements DrawingEditor {
    ...
    void fireAreaInvalidated2 (AbstractTool abt , Double r ){
        Point p1 = abt.getView().drawingToView (...);
        Point p2 = abt.getView().drawingToView (...);
        Rectangle r=new Rectangle(p1.x,p1.y,p2.x-p1.x p2.y-p1.y);
        abt.fireAreaInvalidated (r);
    }
    ...
}
```

# Global Variables

# Global Variables

- Poor coupling: makes understanding a method more difficult

```
void f(...) {  
    // computes a certain value x  
    return x + g; // where g is a global variable  
}
```



To understand what `f` returns, we need to know the value of `g`  
This value may vary between calls to `f`

# Primitive Obsession



# Primitive Obsession

- Zip Code, Currency, Date, Time, Color, Email, etc should not be primitive types
- But rather have their own type with methods
- For example, methods for validation

# Mutable Objects

# Mutable vs Immutable Objects

- Mutable: state can change
- Immutable: once created, state does not change

Exercise: (a) What will be printed by the following Java program? Justify. (b) Are Strings in Java immutable or not?

```
class Main {  
    public static void main(String[] args) {  
        String s1 = "Hello";  
        String s2 = s1.toUpperCase();  
        System.out.println(s1);  
        System.out.println(s2);  
    }  
}
```

Exercise: (a) What will be printed by the following C++ program? (b) Are Strings in C++ immutable or not?

```
#include <iostream>
#include <string>

int main() {
    std::string s = "ball";
    s[0] = 'c';
    std::cout << s;
}
```

# Why are immutable objects "good"?

- They give more "security" to the object creator
  - You can pass the object to other methods and be sure that they will not change its state
- They are not subject to concurrency issues
  - No need for synchronizations, locks, mutex, etc

# How to interpret this code smell

- Whenever possible:
  - Implement immutable objects
  - Especially, simple objects (ZIP, Date, Time, etc)
- On the other hand:
  - In imperative languages it is natural to have a number of mutable objects

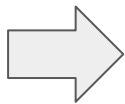
# Comments



Don't comment bad code, rewrite it  
-- B. Kernighan & P. J. Plauger

```
void f() {  
    // task1  
    ...  
    // task2  
    ...  
    // taskn  
    ...  
}
```

```
void f() {  
    // task1  
    ...  
    // task2  
    ...  
    // taskn  
    ...  
}
```



```
void task1 { ... }  
void task2 { ... }  
void taskn { ... }  
  
void f {  
    task1();  
    task2();  
    ...  
    taskn();  
}
```

# "Noise" Comments (add nothing ...)



# Example #1

```
/** The day of the month. */  
private int dayOfMonth;
```

```
// this function sends an email  
void sendEmail() {  
    ...  
}  
  
// this class holds data for an employee  
public class Employee {  
    ...  
}
```

## Example #2

```
// Add a horizontal scroll bar
hScrollBar = new JScrollBar(JScrollBar.HORIZONTAL);
add(hScrollBar, BorderLayout.SOUTH);

// Add a vertical scroll bar
vScrollBar = new JScrollBar(JScrollBar.VERTICAL);
add(vScrollBar, BorderLayout.EAST);

// Initialize the caret-position related values
caretX    = 0;
caretY    = 0;
caretMemX = null;
```

Comments just repeat what is already clear in the code

Source: A Philosophy of Software Design (chapter 13)

Certainly, not every comment is a code smell

# When is it useful to comment?

- Complex code

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



# When is it useful to comment?

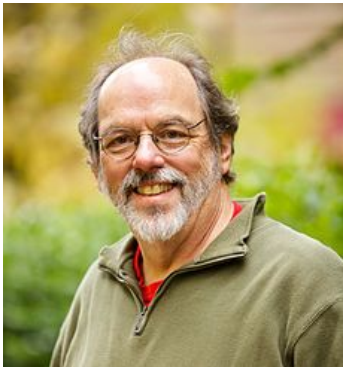
- API Documentation

```
/**
 * Registers the text to display in a tool tip. The text
 * displays when the cursor lingers over the component.
 *
 * @param text  the string to display.  If the text is null,
 *              the tool tip is turned off for this component.
 */
public void setToolTipText(String text) {
```

# Before wrapping up: Technical Debt

# Technical Debt

- Metaphor to explain the importance of SE practices
- Proposed by Ward Cunningham (1992)
- Non-optimal design solutions that make maintenance and evolution difficult



# Examples of Technical Debt

- Lack of tests
- Lack of compliance with architectural patterns
- High coupling and low cohesion
- Code smells
- Lack of documentation
- Code that does not follow predefined layout
- etc

# Exercises

1. Consider the following Price class. One advantage is that it may have a method (not shown) to convert the value to other currencies. (a) Why are objects of this class mutable? (b) Reimplement the class so that its objects are immutable.

```
class Price {  
    ...  
    private double value = 0.0;  
  
    void increment(double amount) {  
        this.value+= amount;  
    }  
    ...  
}
```

# Answer in Java

```
final class Price {           // final: forbids subclasses
    ...
    private final double value; // initialized once
                                   // (usually, in the constructor)

    public Price(double value) {
        this.value = value;
    }

    Price increment(double amount) {
        return new Price(this.value + amount);
    }
    ...
}
```

# Answer in Java, using records

```
public record Price(double value) {  
  
    public Price increment(double amount) {  
        return new Price(value + amount);  
    }  
  
}
```

- Records: simple and compact syntax for implementing immutable objects, available from Java 14



2. In the next three slides, we show the code of a function from an open-source system called FitNesse, which is also used in one of the examples from the Clean Code book.

(a) What code smell exists in this function?

(b) What is the main refactoring that eliminates this smell?

Note: if you prefer, the function's code is [here](#).

```
public static String testableHtml(
    PageData pageData,
    boolean includeSuiteSetup
) throws Exception {
    WikiPage wikiPage = pageData.getWikiPage();
    StringBuffer buffer = new StringBuffer();
    if (pageData.hasAttribute("Test")) {
        if (includeSuiteSetup) {
            WikiPage suiteSetup = PageCrawlerImpl.getInheritedPage(
                SuiteResponder.SUITE_SETUP_NAME, wikiPage
            );
            if (suiteSetup != null) {
                WikiPagePath pagePath =
                    suiteSetup.getPageCrawler().getFullPath(suiteSetup);
                String pagePathName = PathParser.render(pagePath);
                buffer.append("!include -setup .")
                    .append(pagePathName)
                    .append("\n");
            }
        }
    }
}
```

```
WikiPage setup = PageCrawlerImpl.getInheritedPage("SetUp", wikiPage);
if (setup != null) {
    WikiPagePath setupPath = wikiPage.getPageCrawler().getFullPath(setup);
    String setupPathName = PathParser.render(setupPath);
    buffer.append("!include -setup .")
        .append(setupPathName)
        .append("\n");
}
}
buffer.append(pageData.getContent());
if (pageData.hasAttribute("Test")) {
    WikiPage teardown = PageCrawlerImpl.getInheritedPage("TearDown", wikiPage);
    if (teardown != null) {
        WikiPagePath tearDownPath =
            wikiPage.getPageCrawler().getFullPath(teardown);
        String tearDownPathName = PathParser.render(tearDownPath);
        buffer.append("!include -teardown .")
            .append(tearDownPathName)
            .append("\n");
    }
}
```

```
if (includeSuiteSetup) {
    WikiPage suiteTeardown = PageCrawlerImpl.getInheritedPage(
        SuiteResponder.SUITE_TEARDOWN_NAME,
        wikiPage
    );
    if (suiteTeardown != null) {
        WikiPagePath pagePath =
            suiteTeardown.getPageCrawler().getFullPath(suiteTeardown);
        String pagePathName = PathParser.render(pagePath);
        buffer.append("!include -teardown .")
            .append(pagePathName)
            .append("\n");
    }
}
pageData.setContent(buffer.toString());
return pageData.getHtml();
```

```
}
```

**End**