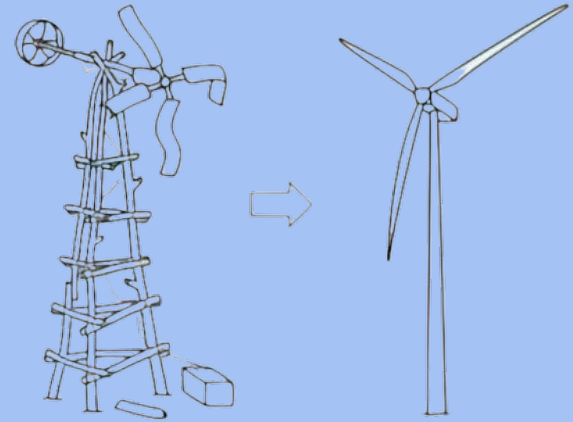


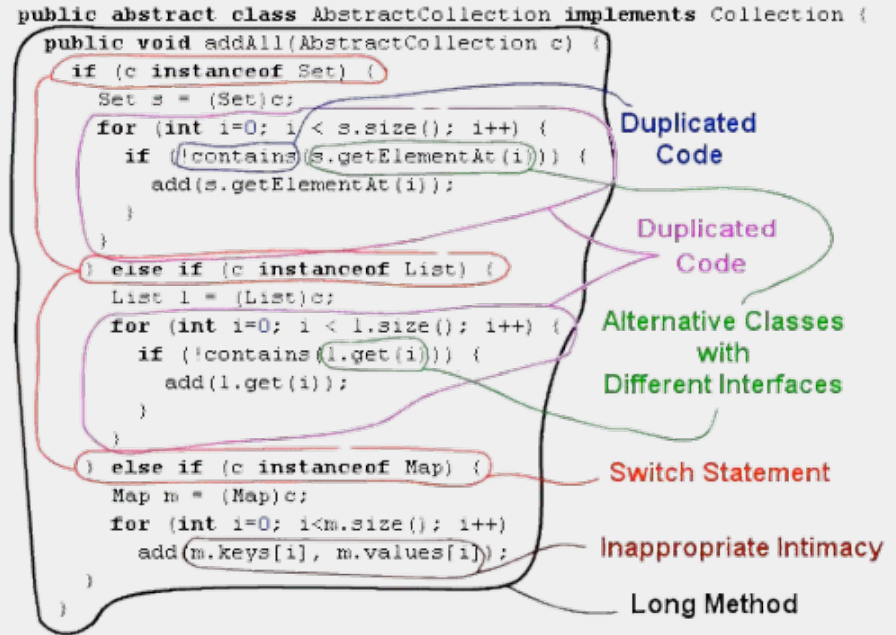
Code Smell & Refactoring



What is Code Smell?

A code smell is a design that duplicates, complicates, bloats or tightly coupled code.

Code smell is any symptom in the source code of a program that possibly indicates a deeper problem. Code smells are usually not bugs—they are not technically incorrect and do not currently prevent the program from functioning. Instead, they indicate weaknesses in design that may be slowing down development or increasing the risk of bugs or failures in the future.



What is Refactoring?

"**Refactoring** is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure." – Martin Fowler



A short history of Code Smells

- If it stinks, change it!
- Kent Beck coined the term code smell to signify something in code that needed to be changed.



Common Code Smells

- Inappropriate naming
- Comments
- Dead code
- Duplicated code
- Primitive obsession
- Large class
- Lazy class
- Middle man
- Data clumps
- Data class
- Long method
- Long parameter list
- Switch statements
- Speculative generality
- Oddball solution
- Feature envy
- Refused bequest
- Black sheep
- Contrived complexity
- Divergent change
- Shotgun Surgery

Inappropriate Naming

- Names given to variables (fields), methods or class should be clear and meaningful.
- A variable, field, class name should say exactly what it is.
- **Which is better?**

```
private double s; OR private double salary;
```

- A method should say exactly what it does.
- **Which is better?**

```
public double calc(double s); OR
```

```
public double calculateFederalTaxes (double salary);
```

Refactoring Techniques:

- ❑ Rename Variables, Fields, Method, Class

Practice: Inappropriate Naming

Identify and refactor inappropriate names for variables, methods, and classes to enhance the clarity and readability of the code.

```
public class M
{
    public double M2(int x, int y)
    {
        return x - y
    }

    private int Calc(int a, int b)
    {
        return a + b;
    }
}
```

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 it.
 - Refactoring Techniques
 - ☐ Extract Method
 - ☐ Rename Method



Comments (Cont'd)

```
void List::add(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];
            delete[] m_elements;
            m_elements = elements2;
        }
        m_elements[m_size++] = element;
    }
}
```

Comments (Cont'd)

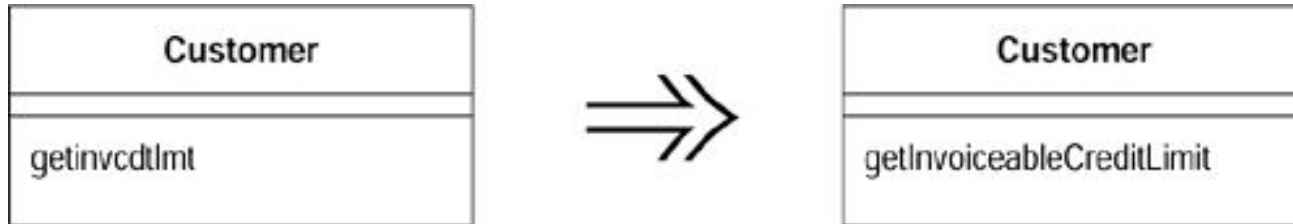
```
void List::add(string element)
{
    if (m_readOnly)
        return;
    if (shouldGrow())
        grow();
    storeElement(element);
}

bool List::shouldGrow()
{
    return (m_size + 1) > capacity();
}
```

```
void List::grow()
{
    m_capacity += 10;
    string *newElements = new
string[m_capacity];
    for(int i = 0;i < m_size;i++)
        newElements[i] = m_elements[i];
    delete [] m_elements;
    m_elements = newElements;
}


void List::storeElement(string element)
{
    m_elements[m_size++] = element;
}
```

Refactoring Technique: Rename Method



Refactoring Technique: 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);  
}
```

Practice: Comments

Refactor the code by removing the comments and making the code self-explanatory through better naming and structure.

```
public class ShoppingCart
{
    private List<Item> items = new List<Item>();
    private int maxCapacity = 10;

    public void AddItem(Item item)
    {
        // Check if the cart has space
        if (items.Count >= maxCapacity)
        {
            // If full, expand the cart
            ExpandCart();
        }

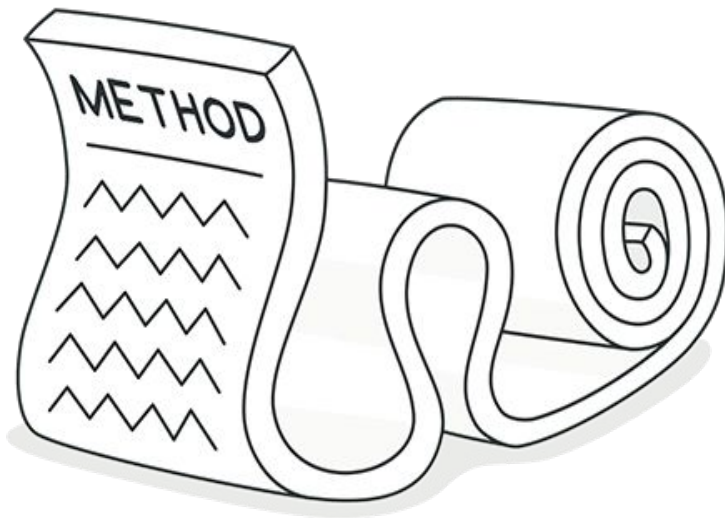
        // Add item to the cart
        items.Add(item);
    }

    private void ExpandCart()
    {
        // Increase max capacity
        maxCapacity += 10;
        Console.WriteLine("Cart capacity expanded.");
    }
}
```

Long Method

A method is long when it is:

- Hard to read
- Too hard to quickly comprehend
- Difficult to test
- Trying to do too much
- Low reusability
- Tendency for bugs
- Hard to change



Long Method (Cont'd)

No. of lines

- 10–20 lines: Often considered an ideal
- More than 30 lines: Method to be getting too long and likely contains more than one responsibility
- More than 50 lines: Strong code smell, indicating it is handling too many responsibilities or has too much complexity

Long Method (Cont'd)

No. of Parameters

- 3–4 parameters is nice
- more than 3–4 parameters is typically considered a code smell. is relying too heavily on external data

No. of Local Variables

- 5–7 local variables is fine
- More than 7 variable indicates that the method might be doing too much. It often leads to increased complexity, making the code harder to follow and maintain

3–4 parameters & 5–7 local variables are common thresholds for identifying a long method

Long Method (Cont'd)

The ideal goal is to keep the method's signature and logic concise, readable, and focused on a single responsibility

Long Method (Cont'd)

Cyclomatic Complexity

- A high cyclomatic complexity (e.g., greater than 10) suggests the method is too complex and difficult to test.

Long Method (Cont'd)

Return Statements and Control Flow

- Multiple return statements or early exits from a method can indicate a method trying to do too much. A clear and singular flow of control (with one return point) is often preferable

Long Method (Cont'd)

Nesting Levels (Indentation Depth)

- Deep nesting (e.g., more than 2-3 levels of if, for, while, or switch) can make the code harder to understand and maintain.

Long Method (Cont'd)

To define a long method, consider:

- Line count (general length).
- Number of parameters (avoid more than 3-4).
- Number of local variables (avoid more than 5-7).
- Cyclomatic complexity (keep under 10).
- Nesting levels (avoid deep indentation).
- Control flow complexity (multiple returns, etc.).
- Single Responsibility Principle (SRP) (ensure the method has one responsibility).
- Duplication (refactor repeated logic).

Long Method (Cont'd)

This “smell” remains unnoticed until the method turns into an ugly, oversized beast.

Refactoring Techniques:

- Extract Method
- Replace Temp with Query
- Introduce Parameter Object
- Preserve Whole Object
- Replace Method with Method Object.
- Decompose Conditional

Refactoring Technique: Extract Method

```
private String toStringHelper(StringBuffer
result) {
    result.append("<");
    result.append(name);
    result.append(attributes.toString());
    result.append(">");
    if (!value.equals(""))
        result.append(value);
    Iterator it = children().iterator();
    while (it.hasNext()) {
        TagNode node = (TagNode)it.next();
        node.toStringHelper(result);
    }
    result.append("</");
    result.append(name);
    result.append(">");
    return result.toString();
}
```

```
private String toStringHelper(StringBuffer result) {
    writeOpenTagTo(result);
    writeValueTo(result);
    writeChildrenTo(result);
    writeEndTagTo(result);
    return result.toString();
}

private void writeOpenTagTo(StringBuffer result) {
    result.append("<");
    result.append(name);
    result.append(attributes.toString());
    result.append(">");
}

private void writeValueTo(StringBuffer result) {
    if (!value.equals(""))
        result.append(value);
}

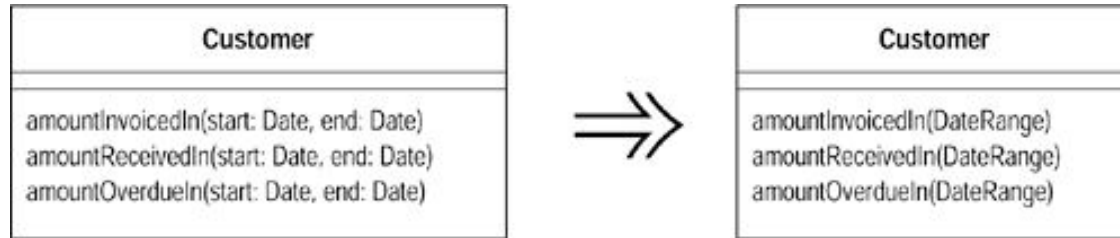
private void writeChildrenTo(StringBuffer result) {
    Iterator it = children().iterator();
    while (it.hasNext()) {
        TagNode node =
            (TagNode)it.next();
        node.toStringHelper(result);
    }
}
```

Refactoring Technique: Replace Temp with Query

```
double basePrice = _quantity *  
_itemPrice;  
if(basePrice > 1000)  
{  
    return basePrice * 0.95;  
}  
else  
{  
    return basePrice * 0.98;  
}
```

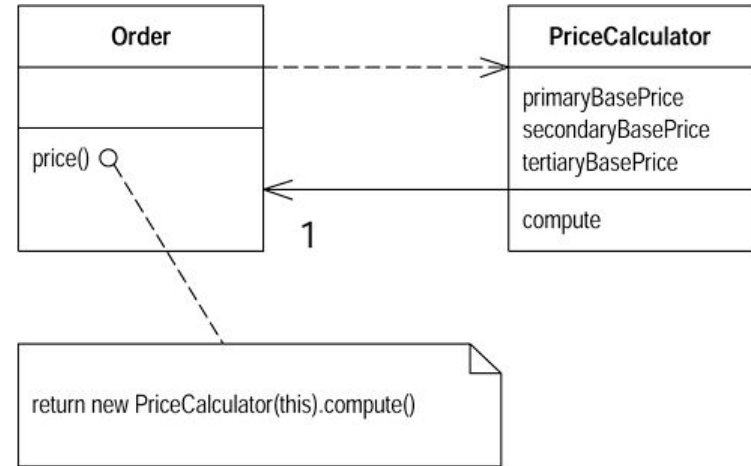
```
if(getBasePrice() > 1000) {  
    return getBasePrice() * 0.95;  
}  
else {  
    return getBasePrice() * 0.98;  
}  
  
double getBasePrice() {  
    return _quantity * _itemPrice;  
}
```


Refactoring Technique: Introduce Parameter Object



Refactoring Technique: Replace Method with Method Object

```
class Order
{
    double Price() {
        double primaryBasePrice;
        double secondaryBasePrice;
        double tertiaryBasePrice;
        // long computation;
        ...
    }
}
```



Refactoring Technique: Decompose Conditional

You have a complicated conditional (if-then-else) statement. Extract methods from the condition, then part, and else parts.

```
if (date.before (SUMMER_START) || date.after (SUMMER_END))  
    charge = quantity * _winterRate + _winterServiceCharge;  
else c  
    harge = quantity * _summerRate;
```

```
-----  
-----  
if (notSummer(date))  
    charge = winterCharge(quantity);  
else  
    charge = summerCharge (quantity);
```

Practice: Long Method

You are given a method in a `PurchaseOrder` class that performs several operations, including calculations, condition checks, and updates. The method is long and difficult to follow. Your task is to refactor the code using appropriate techniques to make it cleaner and easier to maintain.

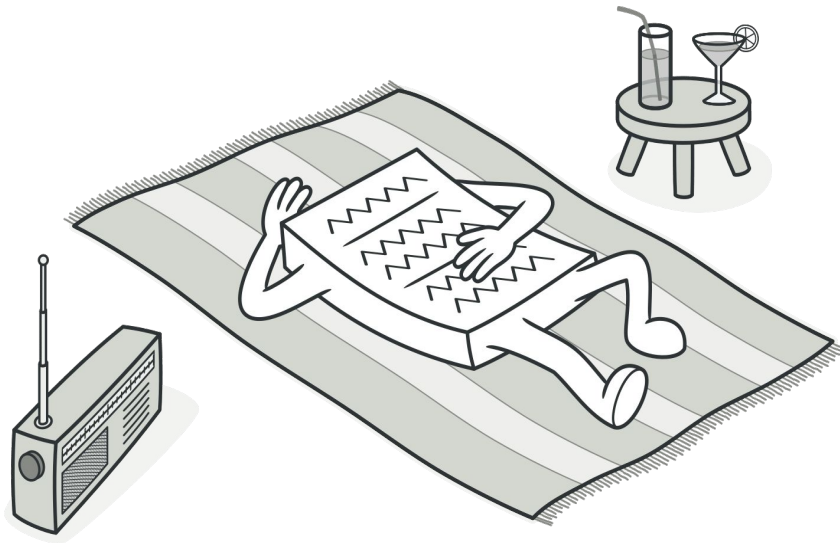
bit.ly/3YrrkvB

Code Smell: Lazy Class

A class that isn't doing enough to carry its weight. We let the class die with dignity

Refactoring Techniques:

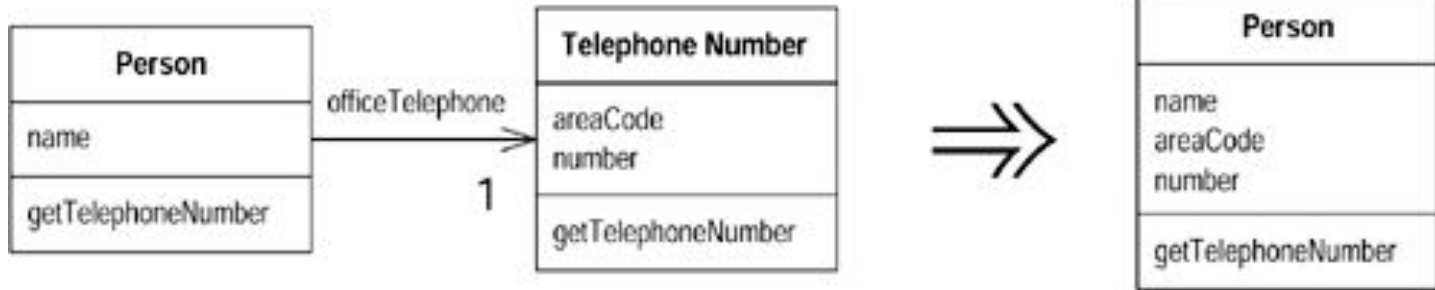
- Inline Class
- Collapse Hierarchy



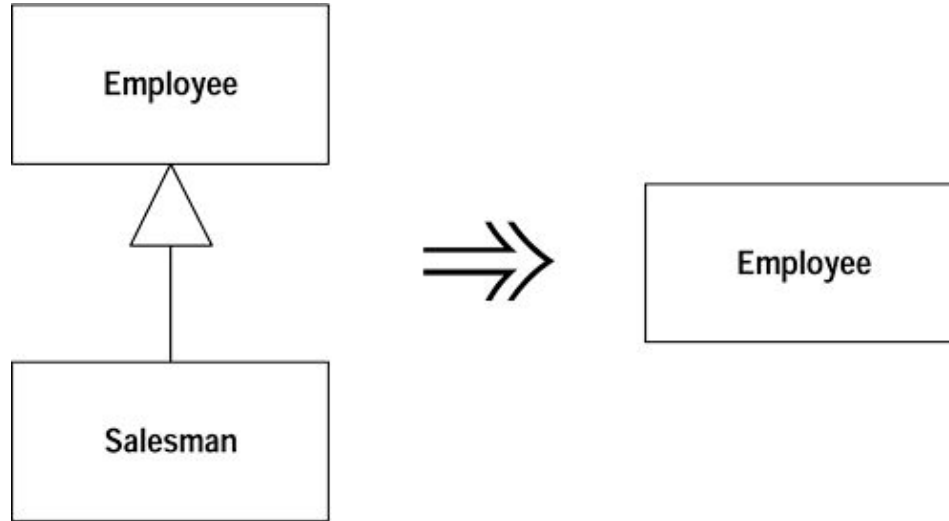
Code Smell: Lazy Class (Cont'd)

```
public class Letter {  
    private final String content;  
  
    public Letter(String content) {  
        this.content = content;  
    }  
  
    public String getContent() {  
        return content;  
    }  
}
```

Refactoring Technique: Inline Class



Refactoring Technique: Collapse Hierarchy



Code Smell: Middle man

An unnecessary layer that just passes calls along without adding any real value

Refactoring technique:
Remove middle man



Code Smell: Middle man (Cont'd)

```
public class CustomerData {  
    public string GetName(int id) {  
        return "John Doe";  
    }  
    public string GetAddress(int id) {  
        return "123 Elm St.";  
    }  
}
```

```
public class CusManager {  
    private CustomerData cusData;  
  
    public CusManager() {  
        this.cusData = new CustomerData();  
    }  
  
    public string GetName(int id) {  
        return cusData.GetName(id);    }  
  
    public string GetAddress(int id) {  
        return cusData.GetAddress(id);  
    }  
}
```

Code Smell: Middle man (Cont'd)

```
public class Client {  
    public void DisplayCustomer(int id) {  
        CusManager manager = new CusManager();  
        Console.WriteLine(manager.GetName(id));  
        Console.WriteLine(manager.GetAddress(id));  
    }  
}
```

Code Smell: Oddball Solution

- A particular problem is solved in multiple different ways throughout a codebase, rather than using a consistent approach.
- Choosing different tools or libraries



Code Smell: Oddball Solution (Cont'd)

```
public class RegistrationForm {  
    public bool ValidateEmail(string email) {  
        //Using regular expression for email validation  
        Regex regex = new Regex(@"^[^@\s]+@[^@\s]+\.[^@\s]+$");  
        return regex.IsMatch(email);  
    }  
}
```

Code Smell: Oddball Solution (Cont'd)

```
public class ContactForm {  
    public bool ValidateEmail(string email) {  
        // Using a simple string operation for email validation  
        return email.Contains("@") && email.Contains(".");  
    }  
}
```

Code Smell: Oddball Solution (Cont'd)

```
public class NewsletterSubscription {  
    public bool ValidateEmail(string email) {  
        // Using System.Net.Mail to validate email  
        try { var addr = new System.Net.Mail.MailAddress(email);  
            return addr.Address == email; }  
        catch {  
            return false;  
        }  
    }  
}
```

Code Smell: Oddball Solution (Cont'd)

Refactoring technique:

- Extract class
- Extract Method to Utility

Practice:

bit.ly/3YuWand

Code Smell: Black Sheep



Code Smell: Black Sheep

- A subclass that significantly differs from others in the hierarchy.
- A method in a class that is noticeably different from other methods in the same class.
- Lack of Single Responsibility Principle (SRP)

Refactoring Techniques -

- Move Method
- Extract Class

Code Smell: Black Sheep



```
public class FileManager
{
    public string FilePath { get; set; }

    public FileManager(string filePath)
    {
        FilePath = filePath;
    }

    public void ReadFile()
    {
        Console.WriteLine($"Reading the file: {FilePath}");
    }

    public void WriteFile(string content)
    {
        Console.WriteLine($"Writing to the file: {FilePath}");
    }

    public void SendEmail(string to, string subject, string body)
    {
        Console.WriteLine($"Sending an email to {to}: Subject - {subject}");
    }
}
```

Code Smell: Black Sheep



```
public class FileManager
{
    public string FilePath { get; set; }

    public FileManager(string filePath)
    {
        FilePath = filePath;
    }

    public void ReadFile()
    {
        Console.WriteLine($"Reading the file: {FilePath}");
    }

    public void WriteFile(string content)
    {
        Console.WriteLine($"Writing to the file: {FilePath}");
    }
}
```

Code Smell: Black Sheep



```
public class EmailSender
{
    public void SendEmail(string to, string subject, string body)
    {
        Console.WriteLine($"Sending an email to {to}: Subject - {subject}");
    }
}
```

Code Smell: Dead Code

- Code that is no longer used in a system or related system is Dead Code.
- Increased Complexity.
- Accidental Changes.
- More Dead Code

- Remedy



Code Smell: Dead Code (Cont'd)

One of the following constructors is never called by a client. It is dead code.

```
public class Loan {  
public Loan(double commitment, int riskRating, Date maturity, Date expiry) {  
    this(commitment, 0.00, riskRating, maturity, expiry);  
}  
  
public Loan(double commitment, double outstanding, int customerRating, Date  
maturity, Date expiry) {  
    this(null, commitment, outstanding, customerRating, maturity, expiry);  
}  
  
public Loan(CapitalStrategy capitalStrategy, double commitment, int riskRating,  
Date maturity, Date expiry) {  
    this(capitalStrategy, commitment, 0.00, riskRating, maturity, expiry);  
}  
...  
}
```

Code Smell: Feature Envy (Cont'd)

According to OOP principles, data and the logic that manipulates it should be encapsulated within the same class. So, a method that seems more interested in another class's data and behavior than its own indicates a design issue.

When a method frequently reaches out to other classes to access their data or functionality, it's a sign of **Feature Envy**.



Code Smell: Feature Envy (Cont'd)

Refactoring Techniques -

- Move Field
- Move Method
- Extract Class

Practice:

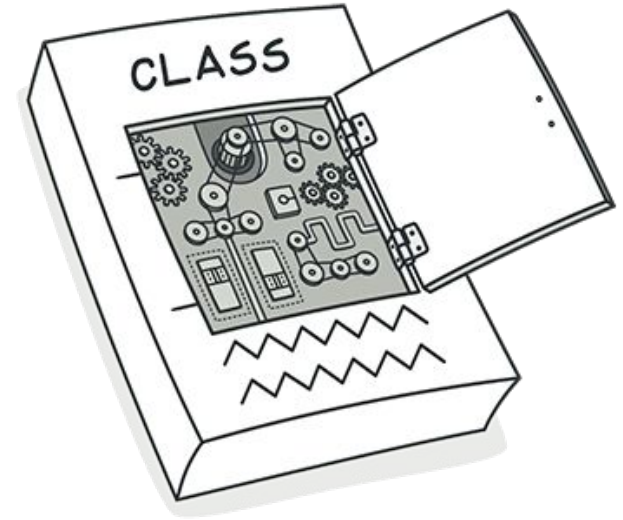
bit.ly/3YEIqPv

Code Smell: Divergent Change

- Divergent Change is when many changes are made to a single class

Refactoring Techniques:

- Split up the behavior of the class via Extract Class.
- If different classes have the same behavior, you may want to combine the classes through inheritance (Extract Superclass and Extract Subclass).

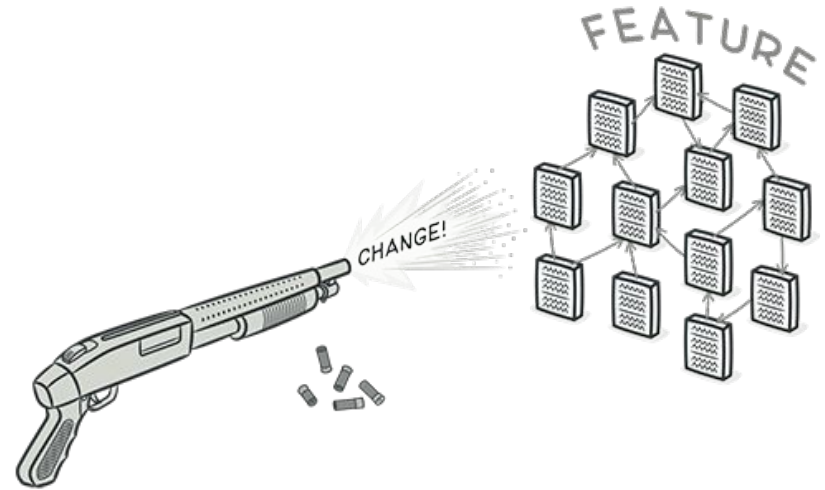


Code Smell: Shotgun Surgery

Making any modifications requires that you make many small changes to many different classes.

Refactoring Techniques:

- Use Move Method and Move Field to move existing class behaviors into a single class. If there's no class appropriate for this, create a new one.
- If moving code to the same class leaves the original classes almost empty, try to get rid of these now-redundant classes via Inline Class.

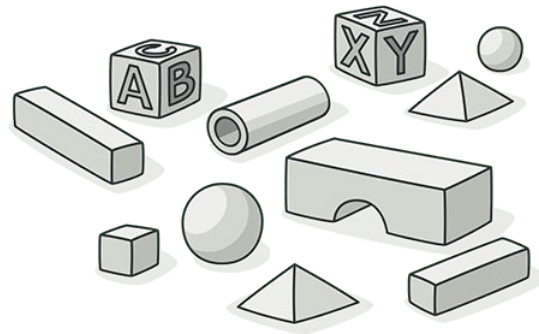


Code Smell: Primitive Obsession

- Use of primitives instead of small objects for simple tasks (such as currency, ranges, special strings for phone numbers, etc.)
- Use of constants for coding information (such as a constant `USER_ADMIN_ROLE = 1` for referring to users with administrator rights.)
- Use of string constants as field names for use in data arrays.

Refactoring Techniques:

- Replace Data Value with Object
- Introduce Parameter Object or Preserve Whole Object.
- Replace Array with Object
- Think in OO Way



Code Smell: Speculative Generality

- You get this smell when people say "Oh, I think we will need the ability to do that someday" and thus want all sorts of hooks and special cases to handle things that aren't required.
- This odor exists when you have generic or abstract code that isn't actually needed today. Such code often exists to support future behavior, which may or may not be necessary in the future.

Refactoring Techniques:

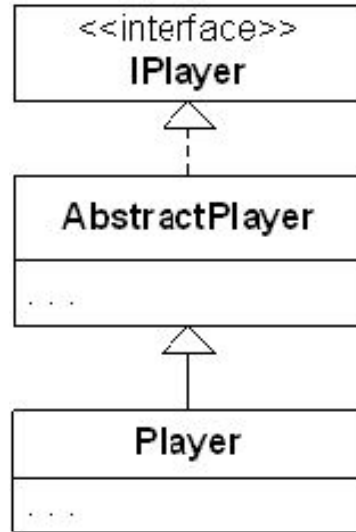
- Collapse Hierarchy
- Inline Class
- Remove Parameter
- Rename Method



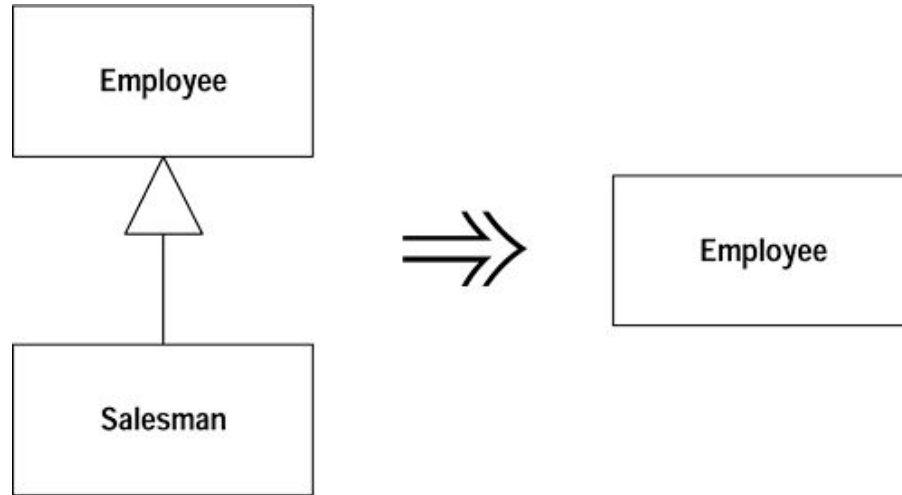
Code Smell: Speculative Generality (Cont'd)

```
public class Customer {
    private String name;
    private String address;
    private String salutation;
    private String otherDetails;
    private MailingSystem mailingSystem;
    public Customer(String name, String salutation, String add, String details,
MailingSystem mSys) {
        this.name = name;
        this.address = add;
        this.salutation = salutation;
        this.otherDetails = details;
        this.mailingSystem = mSys;
    }
    String getName() {
        return name;
    }
    MailingSystem getMailingSystem() {
        return mailingSystem;
    }
}
```

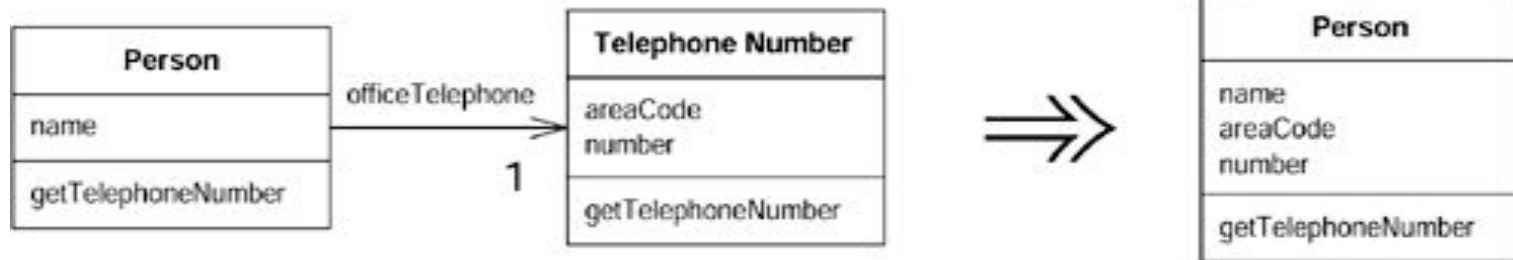
Code Smell: Speculative Generality (Cont'd)



Refactoring Technique: Collapse Hierarchy



Refactoring Technique: Inline Class



Refactoring Technique: Remove Parameter



Code Smell: Refused Bequest

This rather potent odor results when subclasses inherit code that they don't want. In some cases, a subclass may "refuse the bequest" by providing a do nothing implementation of an inherited method.

Refactoring Techniques

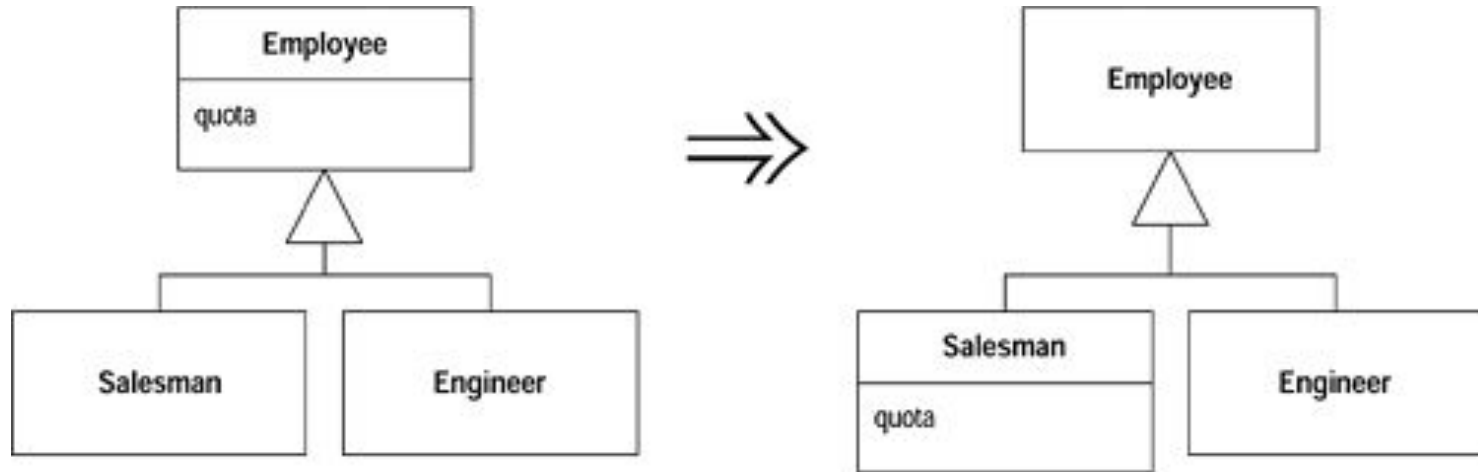
- Push Down Field/Method
- Replace Inheritance with Delegation



Code Smell: Refused Bequest (Cont'd)

```
public abstract class AbstractCollection...  
    public abstract void add(Object element);  
  
public class Map extends AbstractCollection...  
    // Do nothing because user must input key and value  
    public void add(Object element) {  
    }  
}
```

Refactoring Technique: Push Down Method



Duplicate Code

- Duplicated Code
- The *most pervasive and pungent smell* in software coding
- There is obvious or blatant duplication such as copy and paste
- There are subtle or non-obvious duplications
- Such as parallel inheritance hierarchies.
- Similar algorithms

Refactoring Techniques:

- Extract Method
- Pull Up Field
- Form Template Method
- Substitute Algorithm



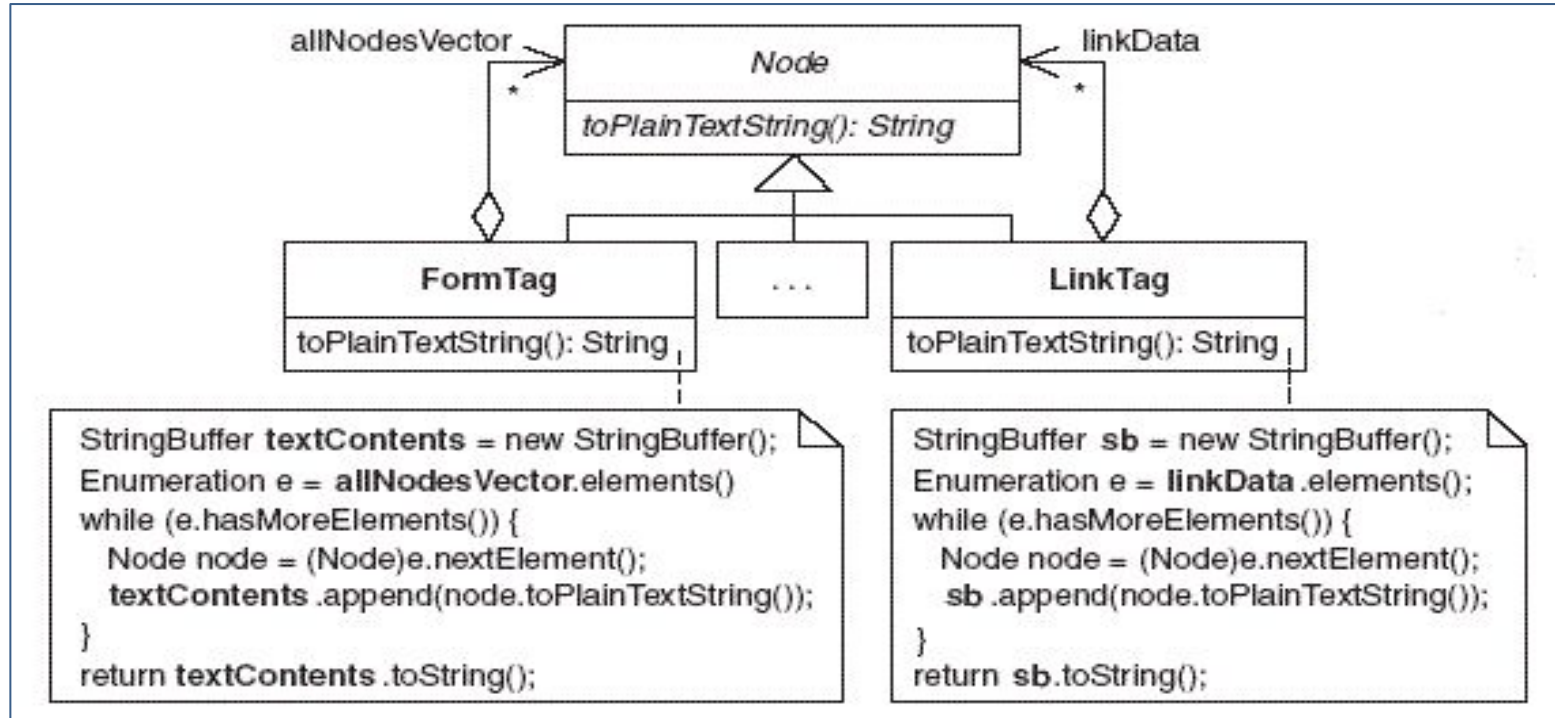
Duplicate Code (Cont'd)

Ctrl+C Ctrl+V Pattern

```
public static MailTemplate getStaticTemplate(Languages language) {
    MailTemplate mailTemplate = null;
    if(language.equals(Languages.English)) {
        mailTemplate = new EnglishLanguageTemplate();
    } else if(language.equals(Languages.French)) {
        mailTemplate = new FrenchLanguageTemplate();
    } else if(language.equals(Languages.Chinese)) {
        mailTemplate = new ChineseLanguageTemplate();
    } else {
        throw new IllegalArgumentException("Invalid language type specified");
    }
    return mailTemplate;
}

public static MailTemplate getDynamicTemplate(Languages language, String content) {
    MailTemplate mailTemplate = null;
    if(language.equals(Languages.English)) {
        mailTemplate = new EnglishLanguageTemplate(content);
    } else if(language.equals(Languages.French)) {
        mailTemplate = new FrenchLanguageTemplate(content);
    } else if(language.equals(Languages.Chinese)) {
        mailTemplate = new ChineseLanguageTemplate(content);
    } else {
        throw new IllegalArgumentException("Invalid language type specified");
    }
    return mailTemplate;
}
```


Duplicate Code (Cont'd)



Duplicate Code (Cont'd)

```
public int addCustomer( int userId, Customer
newCustomer) {
```

```
    List<Customer> customerList = customers.get(userId);
```

```
    int customerId = (int) Math.random() * 1000;
```

```
    // TODO: Logic to find/generate customer id.
```

```
    newCustomer.setId(customerId);
```

```
    if (customerList == null) {
```

```
        customerList = new LinkedList<Customer>();
```

```
        customers.put(userId, customerList);
```

```
    }
```

```
    customerList.add(newCustomer);
```

```
    return customerId;
```

```
}
```

```
public int addTemplate( int userId, Template
newTemplate) {
```

```
    List<Template> templateList =
```

```
    templates.get(userId);
```

```
    int templateId = (int) Math.random() * 1000;
```

```
    // TODO: Logic to find/generate template id.
```

```
    newTemplate.setId(templateId);
```

```
    if (templateList == null) {
```

```
        templateList = new
```

```
        LinkedList<Template>();
```

```
        templates.put(userId,
```

```
        templateList);
```

```
    }
```

```
    templateList.add(newTemplate);
```

Duplicate Code (Cont'd)

Levels of duplication

- Literal
- Semantic
- Data Duplication
- Conceptual

Duplicate Code (Cont'd)

Literal Duplication

Exact copies of code are repeated in multiple locations

Example: Copy-pasting the same block of code without modifications across files or classes

Duplicate Code (Cont'd)

Semantic Duplication

Code segments perform the same task or have the same logic but are written differently, often with slight variations in syntax or implementation.

Example: Two methods that calculate a discount, but one uses a for loop while the other uses a while loop.

Duplicate Code (Cont'd)

Duplication of data structures, constants, or configuration data across the codebase.

Data Duplication

Example: Hardcoding the same value (like tax rates or file paths) in multiple places.



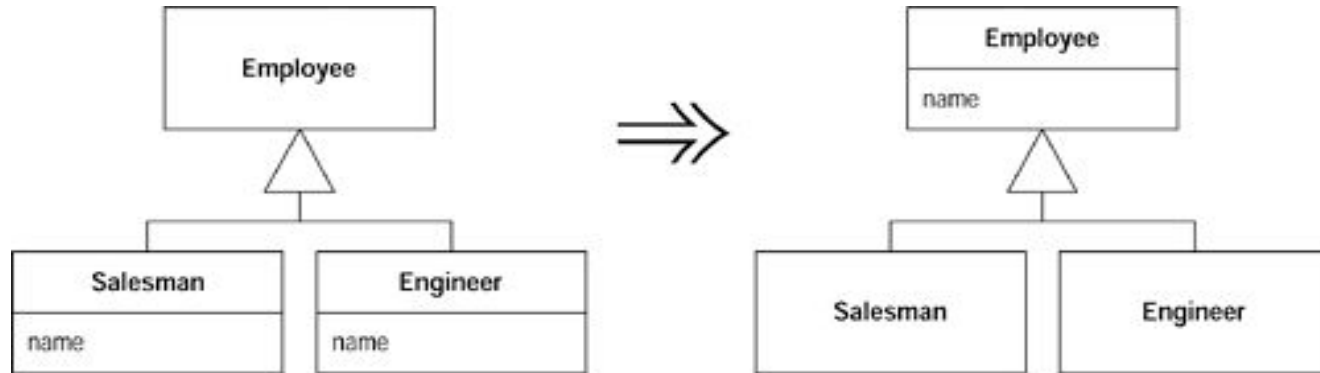
Duplicate Code (Cont'd)

Conceptual Duplication

High-level design or architectural duplication where the same idea, concept, or responsibility is implemented multiple times in different places.

Example: Multiple modules independently implementing their own logging mechanisms instead of using a shared logging service.

Refactoring Technique: Pull Up Field



Refactoring Technique: Substitute Algorithm

```
String foundPerson(String[] people){  
    for (int i = 0; i < people.length; i++) {  
        if (people[i].equals ("Don")){  
            return "Don";  
        }  
        if (people[i].equals ("John")){  
            return "John";  
        }  
        if (people[i].equals ("Kent")){  
            return "Kent";  
        }  
    }  
    return "";  
}
```

```
String foundPerson(String[] people){  
    List candidates = Arrays.asList(new String[]  
    {"Don", "John", "Kent"});  
    for (String person : people)  
        if (candidates.contains(person))  
            return person;  
    return "";
```

Existing algorithm or piece of code can be replaced with a better or simpler one that achieves the same outcome.

This technique involves replacing the current algorithm with a new one that is clearer, more efficient, or easier to understand and maintain.

Code Smell: Switch Smell

- This smell exists when the same switch statement (or “if...else if...else if” statement) is duplicated across a system.
- Such duplicated code reveals a lack of object-orientation and a missed opportunity to rely on the elegance of polymorphism.

Refactoring Techniques

- Replace Type Code with Polymorphism
- Replace Type Code with State / Strategy
- Replace Parameter with Explicit Methods
- Introduce Null Object.



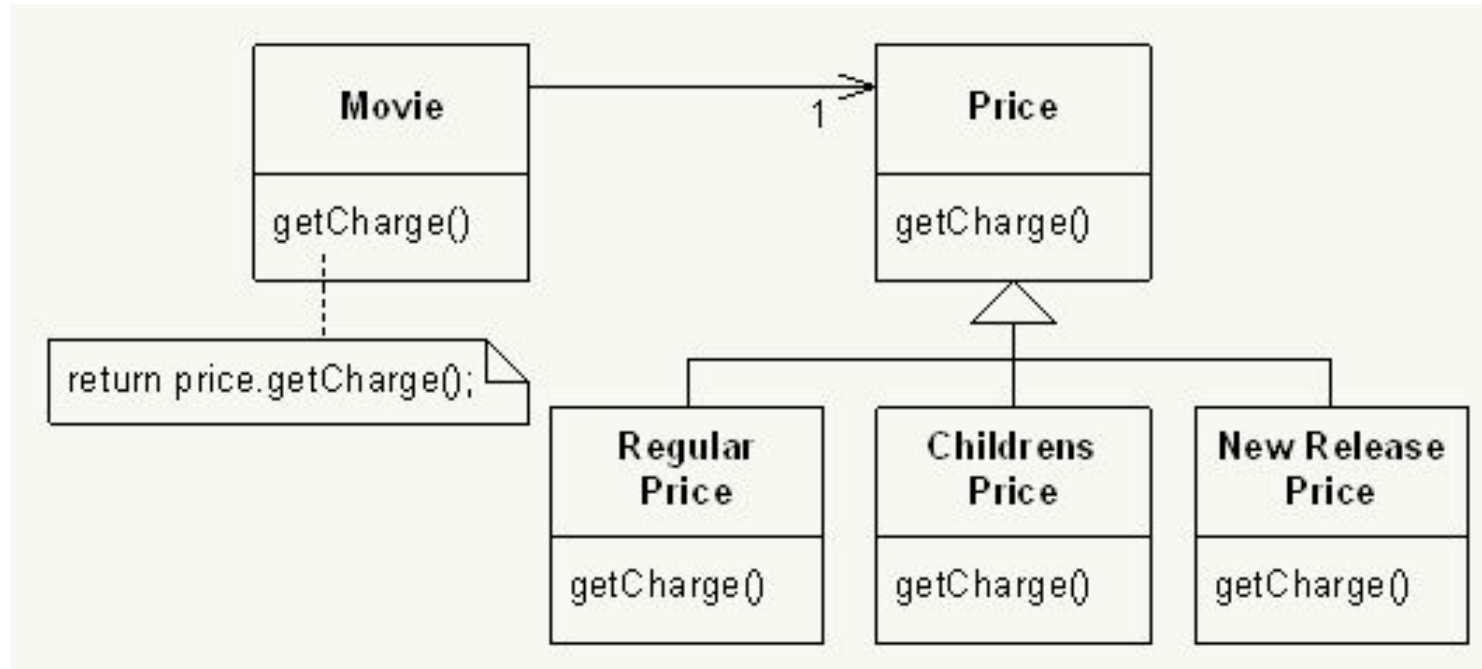
Code Smell: Switch Smell (Cont'd)

```
if("=".equalsIgnoreCase(operator.trim())){  
    for(int i=0;i<ruleCriteria.getCriteriaSize();i++){  
        if(ruleCriteria.getCriteria(i).equalsIgnoreCase("Name"))  
            criteriaCompareStrategy[i] = new Integer(nameFlag);  
        else if(ruleCriteria.getCriteria(i).equalsIgnoreCase("City"))  
            criteriaCompareStrategy[i]=new Integer(cityFlag);  
        else if(ruleCriteria.getCriteria(i).equalsIgnoreCase("Address"))  
            criteriaCompareStrategy[i]=new Integer(addressFlag);  
        else if(ruleCriteria.getCriteria(i).equalsIgnoreCase("Age"))  
            criteriaCompareStrategy[i]=new Integer(ageFlag);  
        else if(ruleCriteria.getCriteria(i).equalsIgnoreCase("Income"))  
            criteriaCompareStrategy[i]=new Integer(incomeFlag);  
        else if(ruleCriteria.getCriteria(i).equalsIgnoreCase("TotalPurchase"))  
            criteriaCompareStrategy[i]=new Integer(spendingFlag);  
    }  
}
```

Code Smell: Switch Smell (Cont'd)

```
while (rentals.hasMoreElements()) {  
    double thisAmount = 0;  
    Rental each = (Rental)rentals.nextElement();  
  
    //determine amounts for each line  
    switch (each.getMovie().getPriceCode()) {  
        case Movie.REGULAR:  
            thisAmount += 2;  
            if (each.getDaysRented() > 2)  
                thisAmount += (each.getDaysRented() - 2) * 1.5;  
            break;  
        case Movie.NEW_RELEASE:  
            thisAmount += each.getDaysRented() * 3;  
            break;  
        case Movie.CHILDRENS:  
            thisAmount += 1.5;  
            if (each.getDaysRented() > 3)  
                thisAmount += (each.getDaysRented() - 3) * 1.5;  
            break;  
    }  
}
```

Refactoring: Replace Type Code with Polymorphism



Refactoring: Replace Parameter with Method

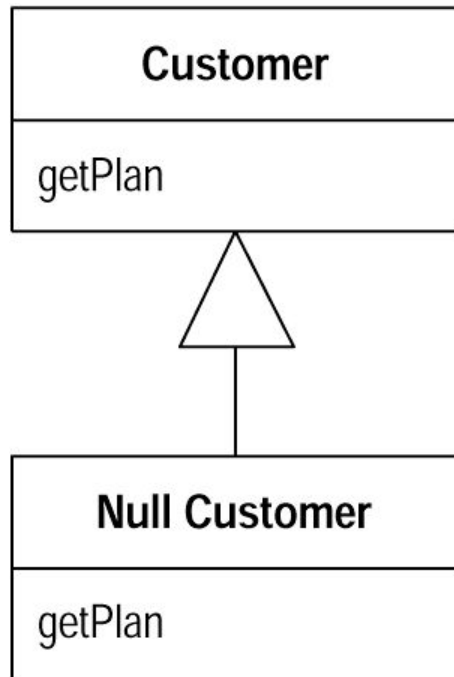
```
void setValue (String name, int value) {  
    if (name.equals("height"))  
        this.height = value;  
    if (name.equals("width"))  
        this.width = value;  
}
```

```
void setHeight(int h) {  
    this.height = h;  
}  
void setWidth (int w) {  
    this.width = w;  
}
```

Refactoring: Introduce Null Object

```
Customer customer = site.getCustomer();  
BillingPlan plan;  
if (customer == null)  
    plan = BillingPlan.basic();  
else  
    plan = customer.getPlan();
```

```
Customer customer = site.getCustomer();  
BillingPlan plan = customer.getPlan();  
  
public BillingPlan getPlan(){  
    return BillingPlan.basic();  
}
```



Refactoring: Large Class

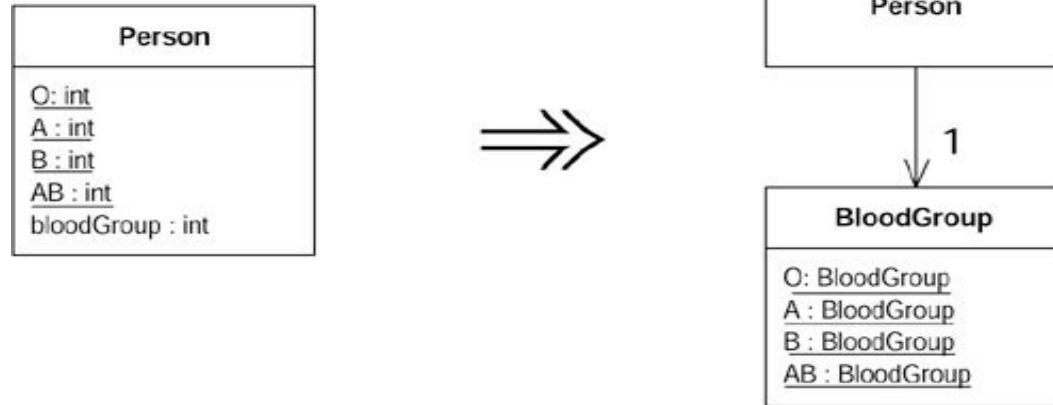
- Like people, classes suffer when they take on too many responsibilities.
- GOD Objects
- Fowler and Beck note that the presence of too many instance variables usually indicates that a class is trying to do too much. In general, large classes typically contain too many responsibilities.
- Remedies
 - Extract Class
 - Replace Type Code with Class/Subclass
 - Replace Type Code with State/Strategy
 - Replace Conditional with Polymorphism
 - Extract Interface



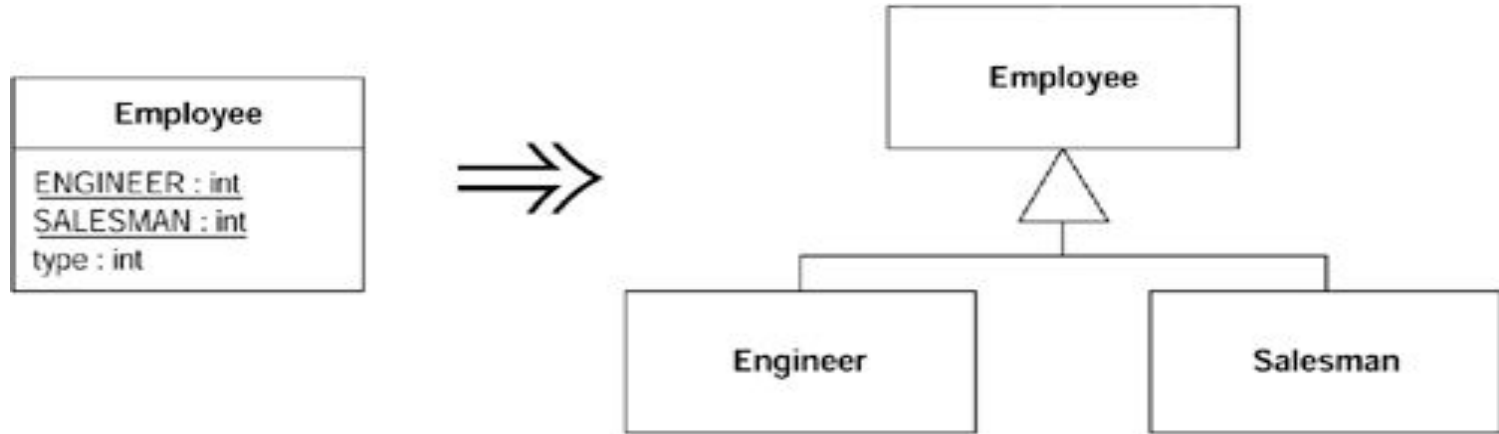
Refactoring: Extract Class



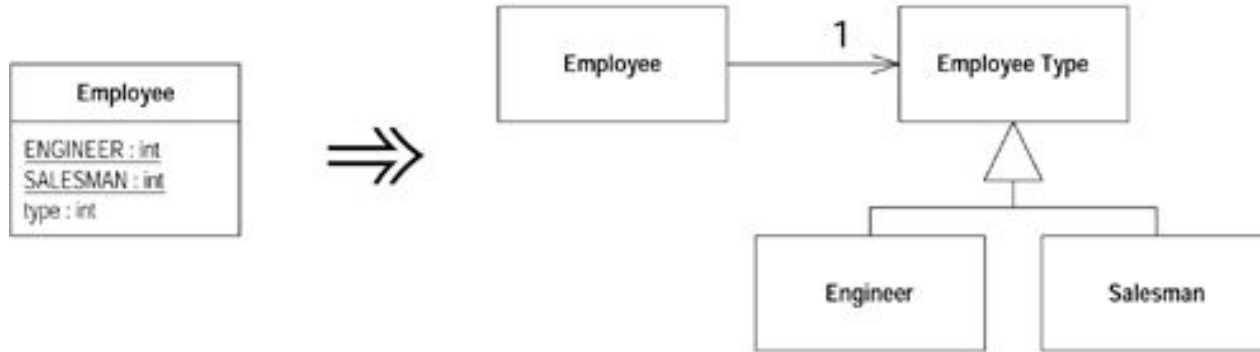
Refactoring: Replace Type Code with Class



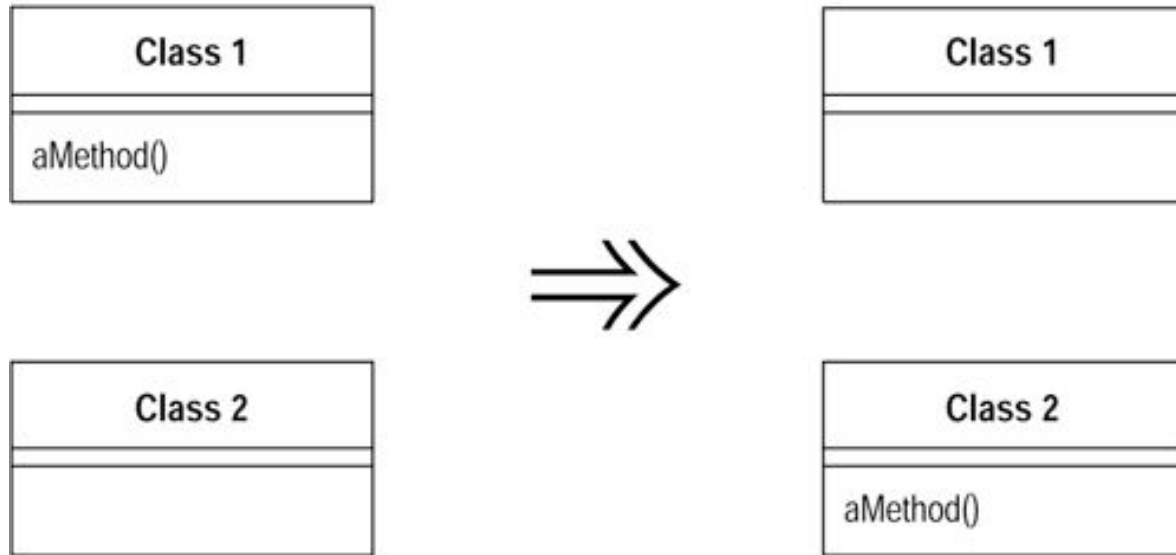
Refactoring: Replace Type Code with Subclasses



Refactoring: Replace Type Code with State/Strategy



Refactoring: Move Method/Field



Data clumps

- Groups of data that frequently appear together in multiple parts of a program

Remedy: Extract class

Data clumps (Cont'd)

```
public class AppointmentScheduler {  
    public void ScheduleAppointment(string firstName,  
        string lastName, string dateOfBirth, string address,  
        DateTime appointmentDate) {  
        // Code to schedule an appointment  
    }  
}
```

```
public class PatientRecords {  
    public void CreateRecord(string firstName,  
        string lastName, string dateOfBirth, string  
        address) {  
        // Code to create a new patient record  
    }  
}
```

```
public class BillingSystem {  
    public void ProcessPayment(string firstName, string lastName, string  
        dateOfBirth, string address, decimal amount) {  
        // Code to process payment  
    }  
}
```

Contrived complexity

Forced usage of overly complicated design patterns where simpler design would suffice.

"Any intelligent fool can make things bigger, more complex, and more violent. It takes a touch of genius...and a lot of courage to move in the opposite direction." ~ E.F. Schumacher



How to start identifying code smells and refactoring

Start with the following three code smells, refactor them without approval from anyone:

- **Long method**
- **Inappropriate naming**
- **Duplicate code**

Thank you

