CEN206 Object-Oriented Programming

Code Smells and Refactoring Fundamentals

Author: Asst. Prof. Dr. Ugur CORUH

List of Figures

List of Tables

CEN206 Object-Oriented Programming

Week-12 (Code Smells and Refactoring Fundamentals)

Spring Semester, 2025-2026 Download DOC-PDF[1], DOC-DOCX[2], SLIDE[3]

Code Smells and Refactoring Fundamentals

Outline

- **Module A:** What is Refactoring? – Definition, Clean Code, Technical Debt, When to Refactor, How to Refactor
- **Module B:** Bloaters – Long Method, Large Class, Primitive Obsession, Long Parameter List, Data Clumps
- **Module C:** Object-Orientation Abusers – Switch Statements, Temporary Field, Refused Bequest, Alternative Classes with Different Interfaces
- **Module D:** Change Preventers – Divergent Change, Shotgun Surgery, Parallel Inheritance Hierarchies
- **Module E:** Dispensables – Comments, Duplicate Code, Lazy Class, Data Class, Dead Code, Speculative Generality
- **Module F:** Couplers – Feature Envy, Inappropriate Intimacy, Message Chains, Middle Man

[1] ce204-week-12.en.md_doc.pdf
[2] ce204-week-12.en.md_word.docx
[3] ce204-week-12.en.md_slide.pdf

Outline (Continued)

All 22 Code Smells Covered

Category	Smells
Bloaters (5)	Long Method, Large Class, Primitive Obsession, Long Parameter List, Data Clumps
OO Abusers (4)	Switch Statements, Temporary Field, Refused Bequest, Alternative Classes with Different Interfaces
Change Preventers (3)	Divergent Change, Shotgun Surgery, Parallel Inheritance Hierarchies
Dispensables (6)	Comments, Duplicate Code, Lazy Class, Data Class, Dead Code, Speculative Generality
Couplers (4)	Feature Envy, Inappropriate Intimacy, Message Chains, Middle Man

Source: Refactoring.Guru – Code Smells[4]

Code Smell Categories (Overview)

Module A: What is Refactoring?

Definition, Clean Code, Technical Debt, When to Refactor, How to Refactor

Module A: What is Refactoring? – Definition

Refactoring is a systematic process of improving the internal structure of code **without changing its external behavior**.

- The main purpose of refactoring is to **fight technical debt**
- It transforms messy, poorly designed code into clean, well-designed code
- Refactoring is **not** about fixing bugs or adding new features
- It is about making code easier to understand, maintain, and extend
- Think of it as applying a series of **small behavior-preserving transformations**

> "Refactoring is a controlled technique for improving the design of an existing code base. Its essence is applying a series of small behavior-preserving transformations, each of which is too small to be worth doing. However the cumulative effect of each of these transformations is quite significant." – Martin Fowler

Reference: RefactoringGuru – What is Refactoring[5]

[4] https://refactoring.guru/refactoring/smells
[5] https://refactoring.guru/refactoring/what-is-refactoring

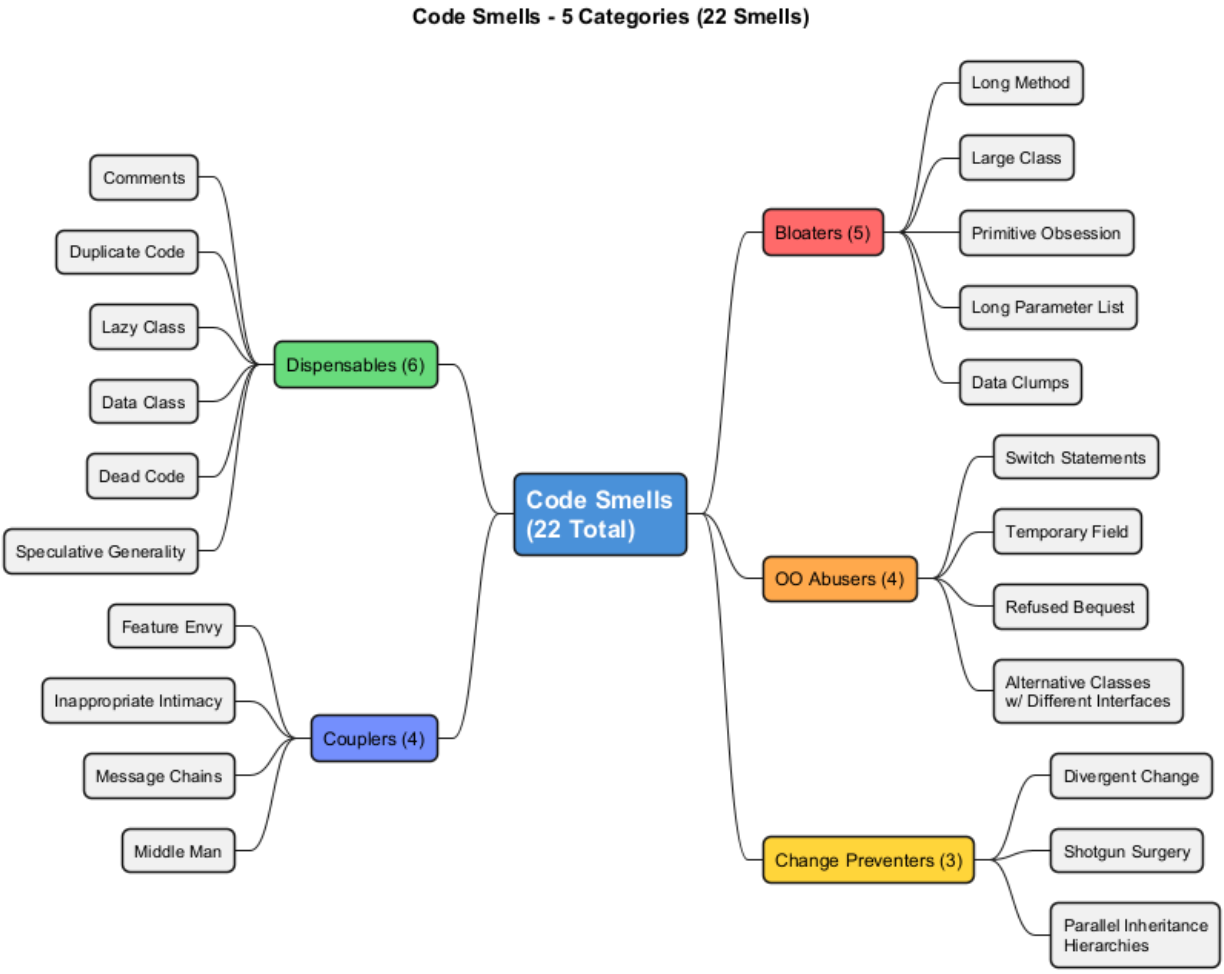


Figure 1: center

Module A: Dirty Code vs. Clean Code

What is Dirty Code?

Dirty code is code that is hard to read, hard to change, and hard to maintain. It accumulates over time due to shortcuts, rushed deadlines, and lack of discipline.

What is Clean Code?

Clean code is the opposite – it is obvious, maintainable, and a pleasure to work with.

> "The main purpose of refactoring is to fight technical debt. It transforms a mess into clean code and simple design."

The Relationship

```
Dirty Code  --[refactoring]-->  Clean Code
     |                               |
     v                               v
  Hard to maintain            Easy to maintain
  Expensive to change         Cheap to change
  Full of hidden bugs         Bugs are visible
```

Module A: Clean Code Characteristics

Clean code has **four key characteristics** that distinguish it from dirty code:

1. Clean Code is Obvious for Other Programmers

- Poor variable names, bloated classes, and magic numbers make code sloppy and difficult to grasp
- Good code reads almost like well-written prose
- Other developers can understand it without extensive documentation

2. Clean Code Does Not Contain Duplication

- Each time you make a change in duplicated code, you must remember to make the same change in every instance
- Increases cognitive load and slows down progress
- Violates the DRY (Don't Repeat Yourself) principle

> "Each time you have to make a change in a duplicate code, you have to remember to make the same change to every instance." – RefactoringGuru

---

Module A: Clean Code Characteristics (Continued)

3. Clean Code Contains a Minimal Number of Classes and Moving Parts

- "Less code is less stuff to keep in your head"
- "Less code is less maintenance"
- Less code means fewer bugs
- Code is a **liability**, not an asset – minimize it

4. Clean Code Passes All Tests

- Code that passes zero percent of tests is dirty code
- Code that is 95% clean but does not pass tests is still not clean enough
- Tests serve as a safety net for refactoring

The Bottom Line

> **"Clean code is easier and cheaper to maintain!"**

- The majority of a software project's lifecycle is spent on maintenance
- Clean code reduces the cost of ongoing development
- New team members can ramp up faster

Reference: RefactoringGuru – Clean Code[6]

Module A: Technical Debt – The Metaphor

What is Technical Debt?

Technical debt is a metaphor coined by **Ward Cunningham**. It compares quick-and-dirty coding decisions to financial debt:

- **Taking on debt:** You ship faster now by writing quick, dirty code
- **Interest payments:** Every time you work with dirty code, you spend extra time understanding and modifying it
- **Bankruptcy:** Eventually, the codebase becomes so messy that further development is nearly impossible

> "You can temporarily speed up without writing tests for new features, but this will gradually slow your progress every day until you eventually pay off the debt by writing tests." – RefactoringGuru

The Debt Spiral

```
Fast & Dirty Code  -->  Technical Debt Accumulates
       |                         |
       v                         v
  Ship Faster Now        Development Slows Down
       |                         |
       v                         v
  More Shortcuts         More Bugs, More Complexity
       |                         |
       v                         v
  Even More Debt    -->  Project May Become Unmaintainable
```

Reference: RefactoringGuru – Technical Debt[7]

Technical Debt Cycle (Diagram)

Module A: Causes of Technical Debt

#	Cause	Description
1	**Business pressure**	Rushing feature releases forces temporary patches into production code
2	**Lack of understanding of consequences**	Management may not recognize how debt compounds over time

[6] https://refactoring.guru/refactoring/clean-code
[7] https://refactoring.guru/refactoring/technical-debt

#	Cause	Description
3	**Failing to combat strict coherence of components**	Monolithic architectures create tight coupling, making isolated changes impossible
4	**Lack of tests**	Missing test coverage encourages risky workarounds that reach production untested
5	**Lack of documentation**	New team members struggle to onboard, departing experts create knowledge gaps
6	**Lack of interaction between team members**	Distributed knowledge leads to outdated understanding
7	**Long-term simultaneous development in branches**	Isolated branch work accumulates debt that compounds during merging
8	**Delayed refactoring**	Postponing architectural updates creates widespread dependent code requiring future rework
9	**Lack of compliance monitoring**	Inconsistent standards and insufficient developer skill both contribute to debt accumulation

Key Insight

- Everyone incurs technical debt from time to time
- The key is not to let it get out of hand
- **Refactoring is the primary method of paying down technical debt**

Module A: When to Refactor – The Rule of Three

The **Rule of Three** is a simple heuristic for deciding when to refactor:

1. First Time – Just Do It

- When you are doing something for the first time, just get it done
- Even if the code is not perfect, move forward

2. Second Time – Wince at the Duplication

- When you do something similar for the second time, cringe at having to repeat yourself
- But do it anyway (for now)

3. Third Time – Refactor

- When you do something for the **third time**, it is time to refactor
- Three strikes and you refactor!

> "When you're doing something for the first time, just do it. The second time, you wince at the duplication but do it anyway. The third time, you refactor." – RefactoringGuru

Reference: RefactoringGuru – When to Refactor[8]

---

[8] https://refactoring.guru/refactoring/when

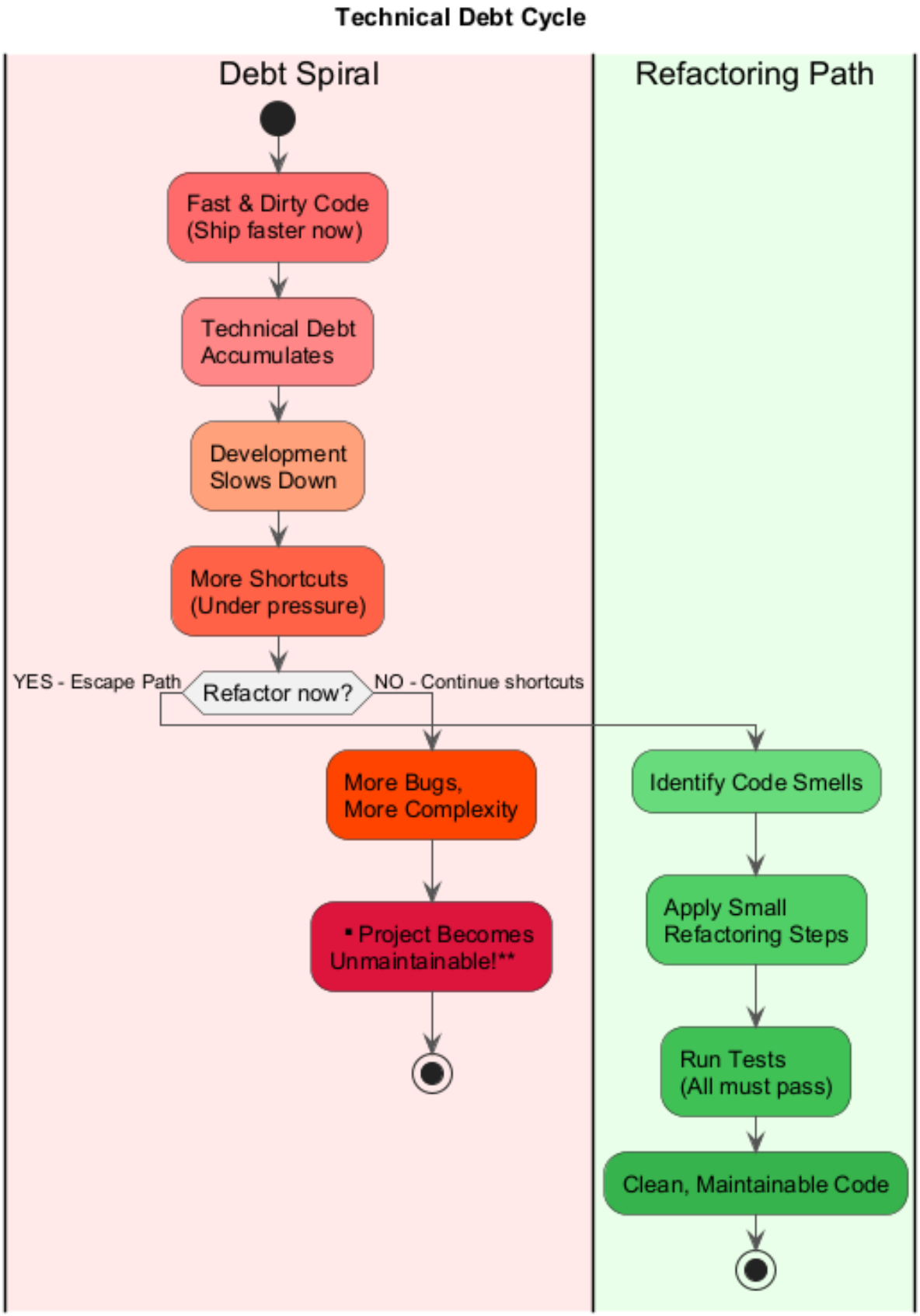


Figure 2: center

Module A: When to Refactor – Specific Situations

When Adding a Feature

Refactoring helps when adding features in two ways:

1. **Understanding Other People's Code** – Refactoring the code to make it clearer helps you understand what is going on. "It's much easier to make changes in clean code."
2. **Facilitating the Addition** – Sometimes the existing code structure makes it very difficult to add a new feature. Refactoring first can make the new feature much easier to implement.

When Fixing a Bug

- "Bugs in code behave just like those in real life: they live in the darkest, dirtiest places in the code."
- By cleaning problematic code first, errors become easier to identify and resolve
- Managers understand that bugs need fixing, so you can often bundle refactoring with bug fixes

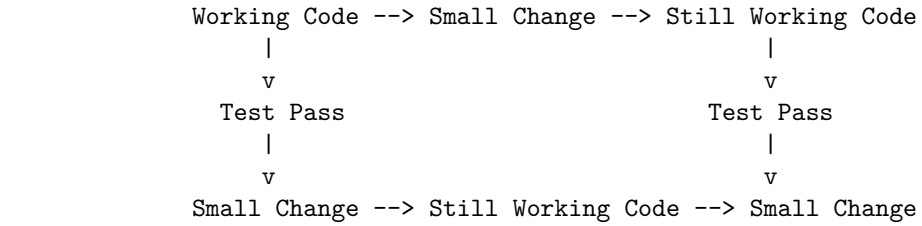
During Code Review

- Code review is the **last chance** to tidy up code before it becomes available to the public
- Best performed in a pair: the reviewer proposes changes and the author implements them immediately
- Start with simple refactorings, tackle complex ones as follow-up tasks

---

Module A: How to Refactor – The Core Principle

> "Refactoring should be done as a series of small changes, each of which makes the existing code slightly better while still leaving the program in working order." – RefactoringGuru



- Each individual refactoring is so small it is very unlikely to go wrong
- The system remains fully functional after each small step
- If something breaks, you know exactly which small change caused it

Reference: RefactoringGuru – How to Refactor[9]

Module A: The Refactoring Checklist

Before, during, and after refactoring, verify these three conditions:

1. The Code Should Become Cleaner

- If the code remains just as unclean after refactoring, you are wasting time
- Sometimes this happens when you mix multiple refactorings into one large change
- For severely problematic code, consider complete rewrites – but only after writing tests and allocating sufficient time

2. New Functionality Should Not Be Created During Refactoring

- "Don't mix refactoring and direct development of new features"
- Keep refactoring and new features as **separate commits** for clarity and traceability

[9] https://refactoring.guru/refactoring/how-to

3. All Existing Tests Must Pass

- After refactoring, all existing tests should still pass
- **Scenario 1:** You made an error during refactoring – fix the small error
- **Scenario 2:** The tests were too low-level (testing private methods rather than behavior) – refactor the tests themselves or write higher-level BDD-style tests

---

Module A: Safe Refactoring Workflow

Step-by-Step Process

1. **Ensure you have a comprehensive test suite** for the code you want to refactor
2. **Make one small change** at a time
3. **Run all tests** after each change
4. **If tests pass**, move on to the next change
5. **If tests fail**, undo the change immediately and try a different approach
6. **Commit frequently** so you can easily revert if something goes wrong
7. **Review the final result** – is the code cleaner?

Tools That Help

Tool	Purpose
Version control (Git)	Easy rollback of bad changes
Automated tests	Verify behavior after each change
IDE refactoring tools	Automated rename, extract method, etc.
Continuous Integration	Catches regressions early

Refactoring Workflow (Diagram)

Module A: Takeaway

Key Points – What is Refactoring?

- **Refactoring** = improving code structure without changing external behavior
- **Clean code** is obvious, duplication-free, minimal, tested, and maintainable
- **Dirty code** creates **technical debt** that slows future development
- Technical debt comes from business pressure, lack of tests, poor documentation, tightly coupled components, and more
- Use the **Rule of Three**: first time do it, second time cringe, third time refactor
- Refactor when adding features, fixing bugs, and during code reviews
- Make **small, incremental changes** and keep all tests passing
- Never mix refactoring and feature development in the same commit

> "Any fool can write code that a computer can understand. Good programmers write code that humans can understand." – Martin Fowler

---

Module B: Bloaters

Code, Methods, and Classes That Have Grown So Large They Are Hard to Work With

---

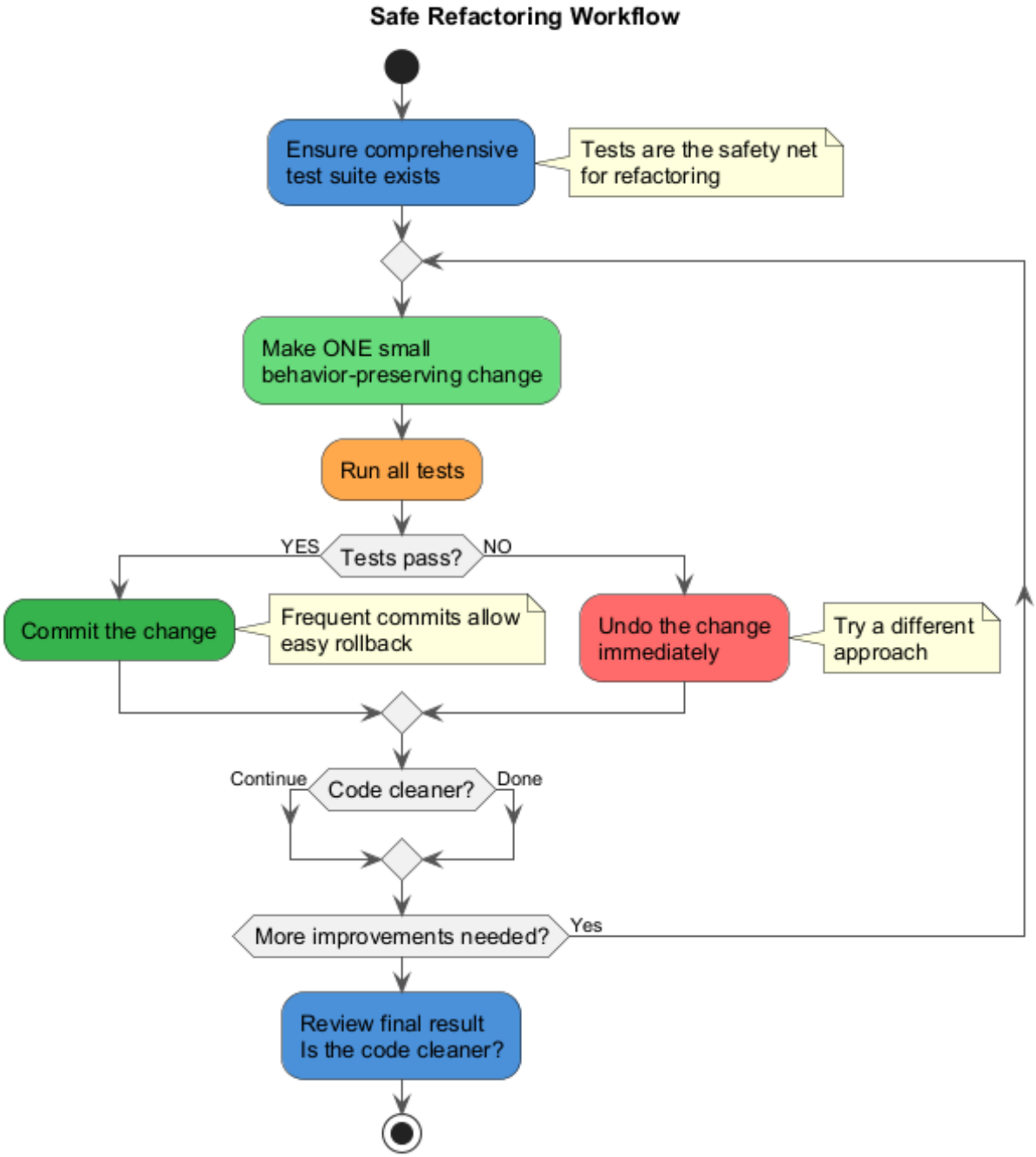


Figure 3: center

Module B: Bloaters – Overview

Bloaters are code, methods, and classes that have increased to such gargantuan proportions that they become hard to work with. Usually these smells do not crop up right away; rather, they accumulate over time as the program evolves (and especially when nobody makes an effort to eradicate them).

The 5 Bloater Smells

#	Smell	Key Issue
1	**Long Method**	Method contains too many lines of code
2	**Large Class**	Class contains too many fields/methods/lines
3	**Primitive Obsession**	Overuse of primitives instead of small objects
4	**Long Parameter List**	More than three or four parameters in a method
5	**Data Clumps**	Groups of data that always appear together

Reference: RefactoringGuru – Bloaters[10]

Module B: Bloater #1 – Long Method

What Is It?

A method contains **too many lines of code**. Generally, any method longer than **ten lines** should make you start asking questions.

Signs and Symptoms

- The method has grown beyond 10-20 lines
- You need comments to explain what different sections of the method do
- The method name does not fully describe everything the method does
- There are deeply nested control structures

Reasons for the Problem

- It is mentally easier to **add** to an existing method than to create a new one
- "I'll just add these two lines here…" – and it keeps growing
- Since it is harder to create a method than to add to one, most code is written as additions to existing methods
- The method originally started small but grew over time

Reference: RefactoringGuru – Long Method[11]

Long Method - Before (Code Smell)

Long Method - After (Refactored)

---

[10] https://refactoring.guru/refactoring/smells/bloaters
[11] https://refactoring.guru/smells/long-method

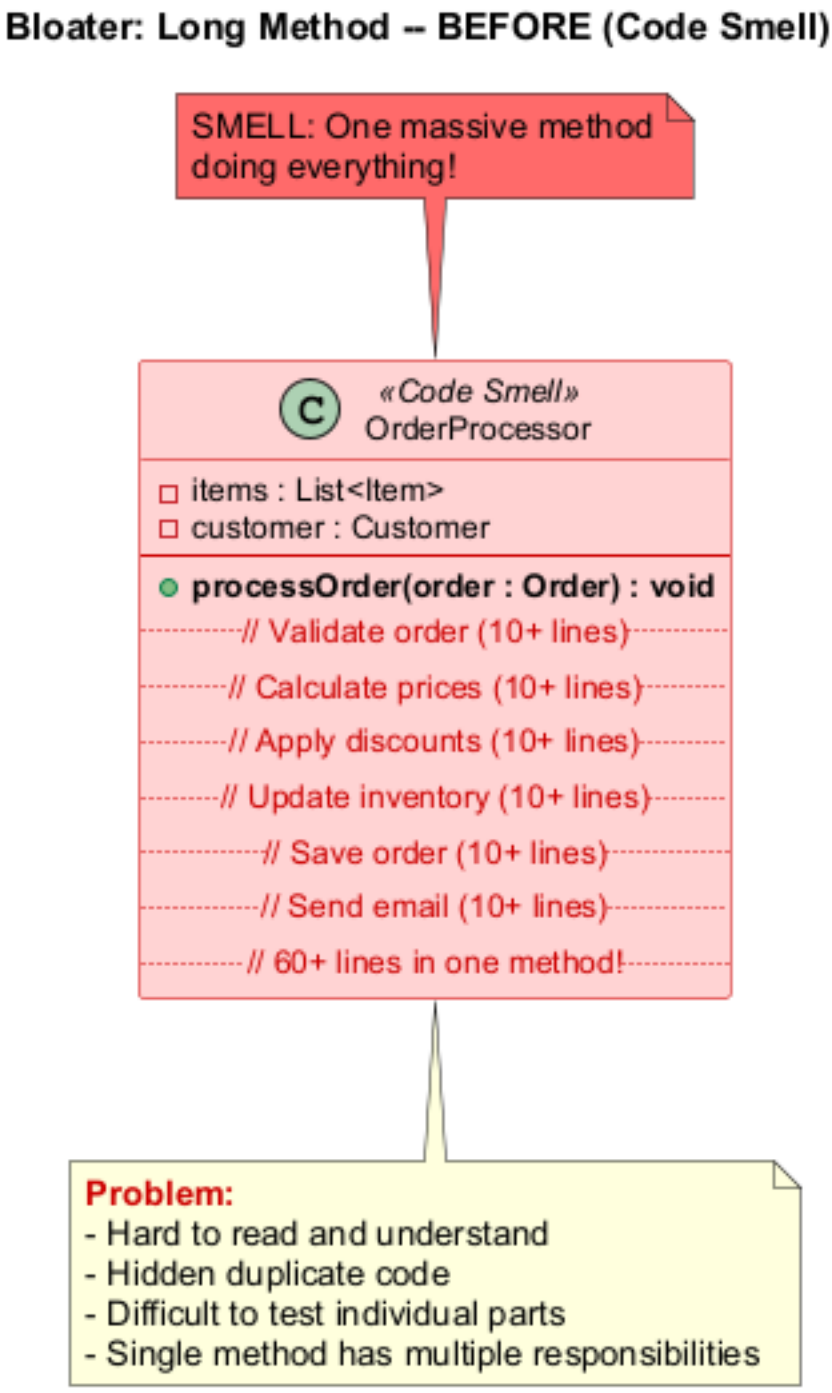


Figure 4: center

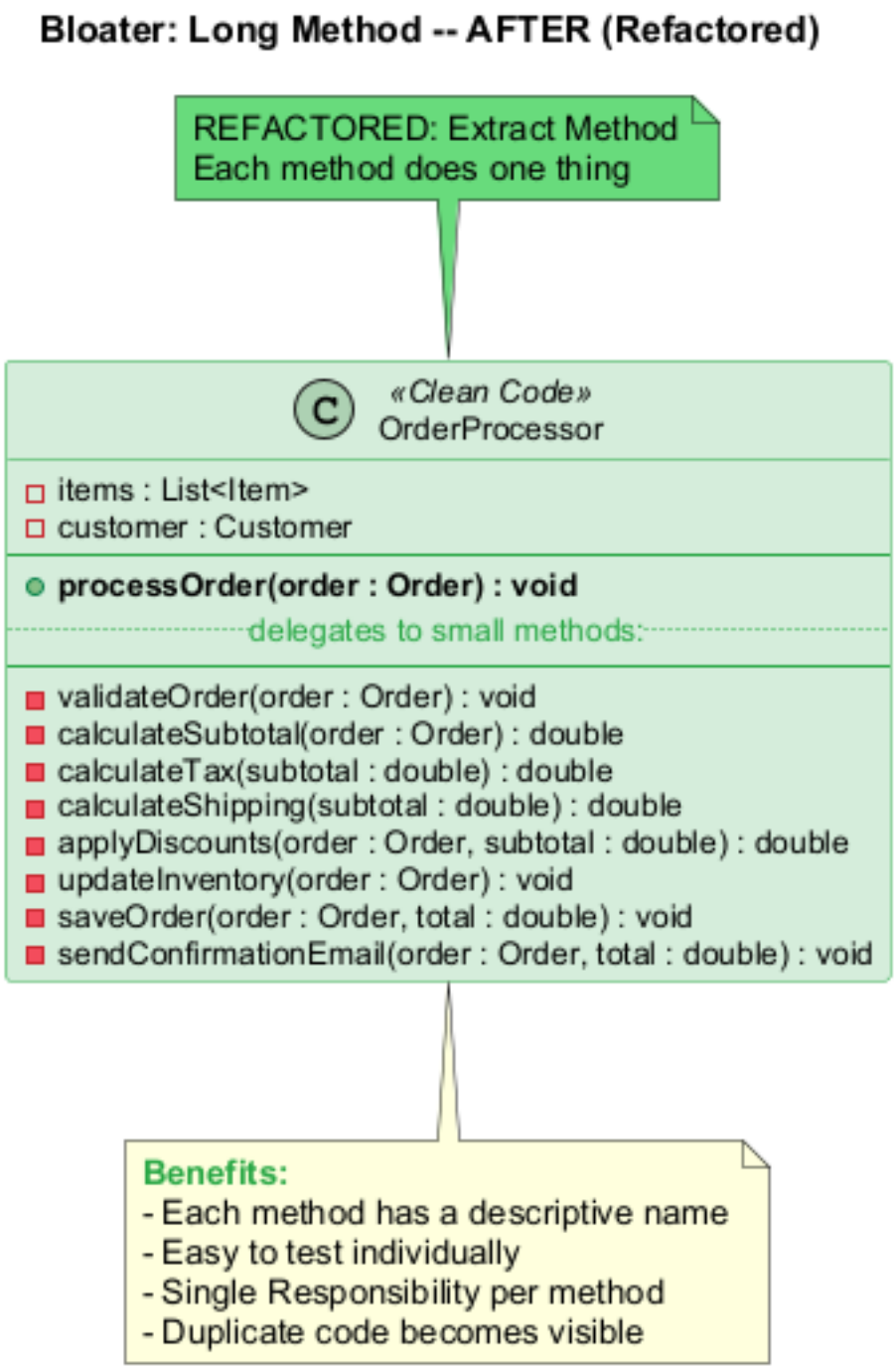


Figure 5: center

Module B: Bloater #1 – Long Method (Java Example – Before)

```java
// SMELL: Long Method -- this method does too many things
public void processOrder(Order order) {
    // Validate order (10+ lines)
    if (order == null) throw new IllegalArgumentException("Order is null");
    if (order.getItems().isEmpty())
        throw new IllegalStateException("Order has no items");
    for (Item item : order.getItems()) {
        if (item.getQuantity() <= 0)
            throw new IllegalStateException("Invalid quantity");
        if (!inventory.isAvailable(item.getProductId(), item.getQuantity()))
            throw new IllegalStateException("Product not available");
    }

    // Calculate prices (10+ lines)
    double subtotal = 0;
    for (Item item : order.getItems()) {
        subtotal += item.getPrice() * item.getQuantity();
    }
    double tax = subtotal * 0.08;
    double shipping = subtotal > 100 ? 0 : 9.99;

    // Apply discounts (10+ lines)
    if (order.getCustomer().isLoyalMember()) {
        subtotal *= 0.90;
    }
    double total = subtotal + tax + shipping;

    // Update inventory, save order, send email... (20+ more lines)
}
```

Module B: Bloater #1 – Long Method (Java Example – After)

```java
// REFACTORED: Extract Method applied
public void processOrder(Order order) {
    validateOrder(order);
    double subtotal = calculateSubtotal(order);
    double tax = calculateTax(subtotal);
    double shipping = calculateShipping(subtotal);
    subtotal = applyDiscounts(order, subtotal);
    double total = subtotal + tax + shipping;
    updateInventory(order);
    saveOrder(order, total);
    sendConfirmationEmail(order, total);
}

private void validateOrder(Order order) {
    if (order == null) throw new IllegalArgumentException("Order is null");
    if (order.getItems().isEmpty())
        throw new IllegalStateException("Order has no items");
    for (Item item : order.getItems()) {
        validateItem(item);
    }
}
```

```java
private double calculateSubtotal(Order order) {
    return order.getItems().stream()
        .mapToDouble(item -> item.getPrice() * item.getQuantity())
        .sum();
}
```

Module B: Bloater #1 – Long Method (Treatment & Payoff)

Treatment (Refactoring Techniques)

Technique	When to Use
Extract Method	Take a fragment of code and turn it into its own method with a descriptive name
Replace Temp with Query	Move an expression into a new method and return the result instead of storing it in a local variable
Introduce Parameter Object	Replace repeating groups of parameters with an object
Preserve Whole Object	Pass the whole object instead of individual values from it
Replace Method with Method Object	Transform the entire method into a separate class when local variables prevent extraction
Decompose Conditional	Extract complex conditional logic into well-named methods

Payoff

- Among all types of object-oriented code, classes with **short methods live longest**
- The longer a method is, the harder it is to understand and maintain
- Long methods are a perfect hiding place for **unwanted duplicate code**
- Now that you have clear code, you are more likely to find **truly effective** performance optimizations

---

Module B: Bloater #2 – Large Class

What Is It?

A class contains **many fields, methods, and lines of code**. Classes become bloated over time as programs grow.

Signs and Symptoms

- The class has too many fields (more than 5-10 that represent different concepts)
- The class has too many methods (covering multiple responsibilities)
- You cannot describe what the class does in one sentence without using "and"

Reasons for the Problem

- Classes usually start small but over time they get bloated as the program grows
- Programmers find it mentally less taxing to **place a new feature in an existing class** than to create a new class for the feature
- "This class already handles orders… I'll just add shipping logic here too"

Reference: RefactoringGuru – Large Class[12]

---

[12] https://refactoring.guru/smells/large-class

Large Class - Before (Code Smell)

Large Class - After (Refactored)

Module B: Bloater #2 – Large Class (Java Example – Before)

```java
// SMELL: Large Class -- Order handles too many responsibilities
public class Order {
    // Customer fields
    private String customerName, customerEmail, customerPhone;
    // Shipping address fields
    private String shippingStreet, shippingCity, shippingState, shippingZip;
    // Billing address fields
    private String billingStreet, billingCity, billingState, billingZip;
    // Order fields
    private List<Item> items;
    private double subtotal, tax, shipping, discount, total;
    private Date orderDate, shipDate, deliveryDate;
    private String status, trackingNumber, paymentMethod;

    // Methods for ordering, shipping, billing, notifications, reporting...
    // 50+ methods covering multiple responsibilities
    public void addItem(Item item) { /* ... */ }
    public void calculateTotal() { /* ... */ }
    public void shipOrder() { /* ... */ }
    public void sendConfirmation() { /* ... */ }
    public void generateInvoice() { /* ... */ }
    public void updateCustomerProfile() { /* ... */ }
}
```

Module B: Bloater #2 – Large Class (Java Example – After)

```java
// REFACTORED: Extract Class -- each class has a single responsibility
public class Order {
    private Customer customer;
    private ShippingAddress shippingAddress;
    private BillingAddress billingAddress;
    private List<Item> items;
    private OrderStatus status;

    public void addItem(Item item) { /* ... */ }
    public double calculateTotal() { /* ... */ }
}

public class Customer {
    private String name, email, phone;
    public void updateProfile() { /* ... */ }
}

public class ShippingAddress {
    private String street, city, state, zip;
}
```

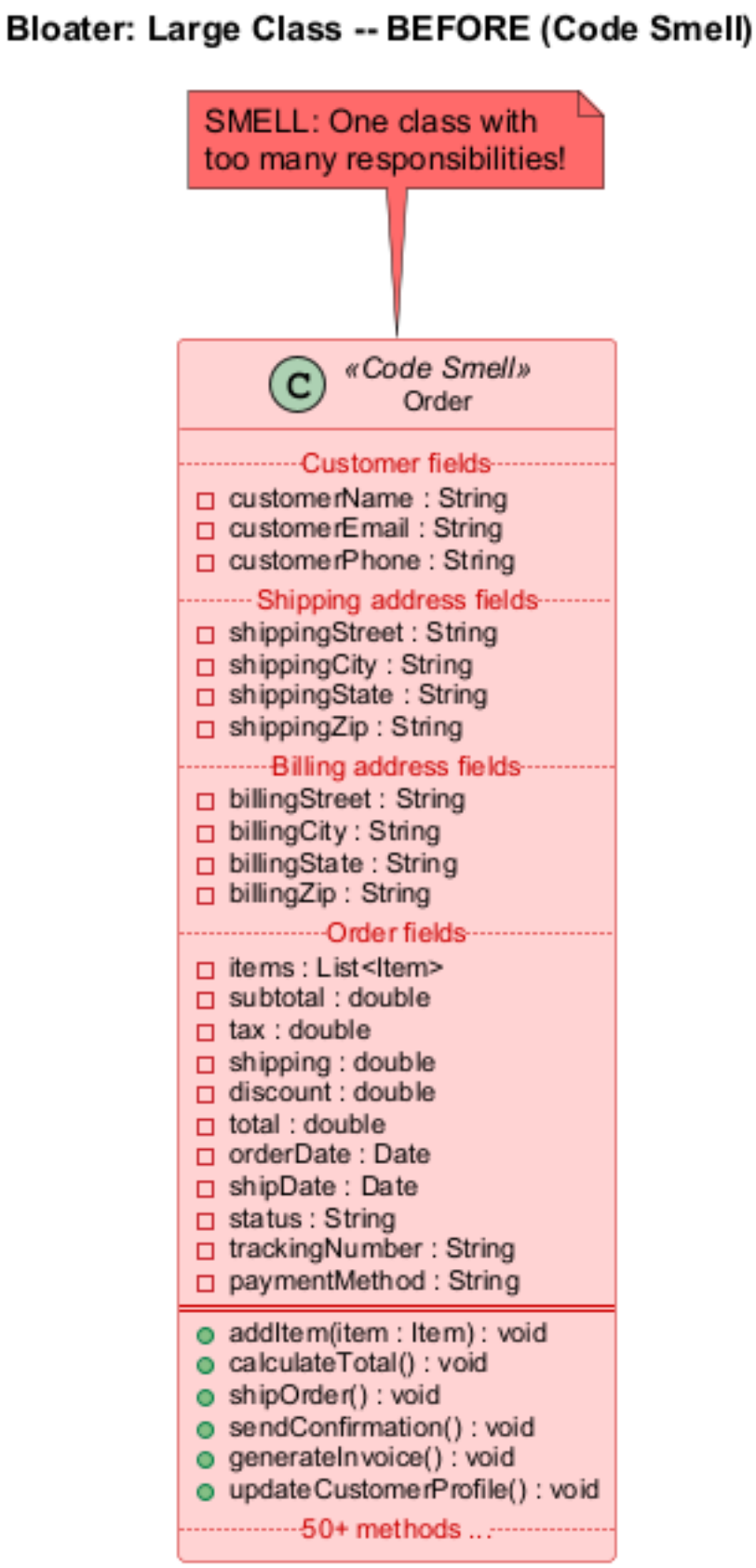


Figure 6: center

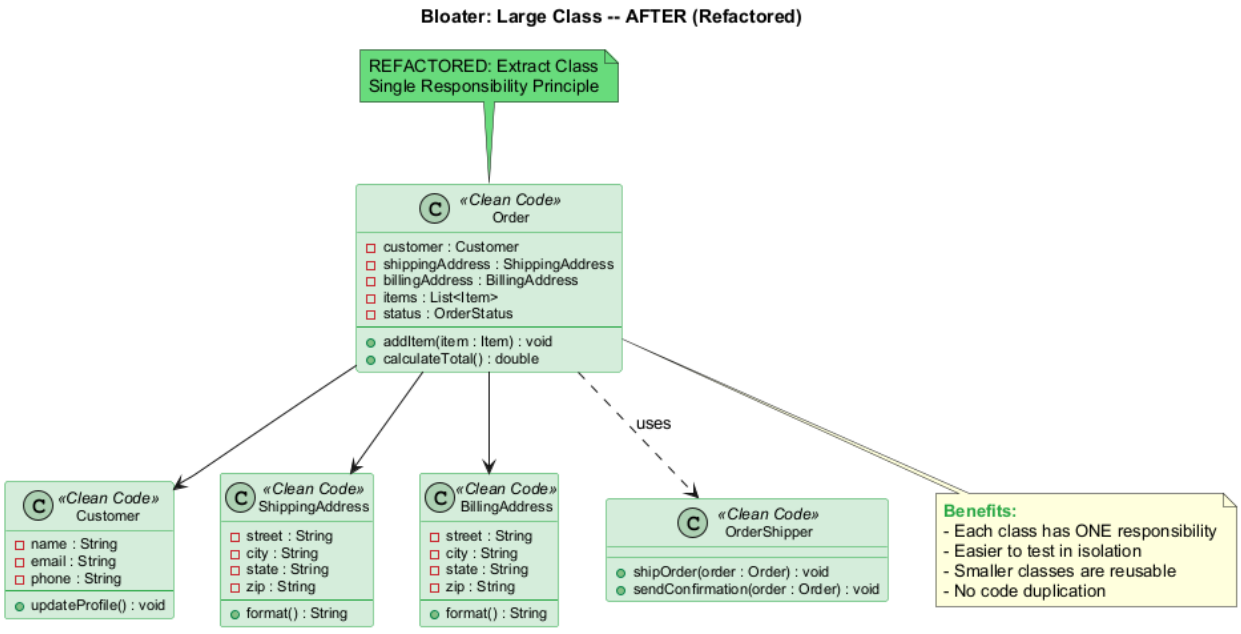


Figure 7: center

```
public class BillingAddress {
    private String street, city, state, zip;
}

public class OrderShipper {
    public void shipOrder(Order order) { /* ... */ }
    public void sendConfirmation(Order order) { /* ... */ }
}
```

Module B: Bloater #2 – Large Class (Treatment & Payoff)

Treatment (Refactoring Techniques)

Technique	When to Use
Extract Class	When part of a class's behavior can be spun off into a separate class
Extract Subclass	When part of a class's behavior is only relevant to some instances
Extract Interface	When you need to identify what operations and behaviors are available to the client
Duplicate Observed Data	If a large class is responsible for GUI, move data and behavior to a separate domain object

Payoff

- Developers do not have to remember a large number of attributes for a class
- Splitting large classes frequently **eliminates code duplication** and redundant functionality
- Each class has **one clear purpose** (Single Responsibility Principle)
- Smaller classes are easier to **test in isolation**
- Smaller, focused classes can be **reused** in other contexts

Module B: Bloater #3 – Primitive Obsession

What Is It?

The overuse of **primitive types** (int, String, double, boolean) instead of creating small objects for domain concepts.

Signs and Symptoms

- 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 `USER_ADMIN_ROLE = 1`)
- Use of **string constants as field names** for use in data arrays

Reasons for the Problem

- Primitives are created "just to store a value" and eventually grow in usage
- Creating a field is easier than creating a whole new class
- "I'll just use a String for the phone number…"
- Primitives are often born from laziness: it seems simpler to use an `int` or `String` than to create a new class

Reference: RefactoringGuru – Primitive Obsession[13]

Module B: Bloater #3 – Primitive Obsession (Java Example – Before)

```java
// SMELL: Primitive Obsession -- primitives everywhere
public class Employee {
    private String name;
    private String phoneNumber;        // Should be PhoneNumber object
    private int currencyAmount;        // Cents? Dollars? Euros?
    private String zipCode;            // Should validate format
    private int temperatureInFahrenheit;
    private String startDate;          // String for a date? Really?
    private int employeeType;          // 1=Engineer, 2=Salesman, 3=Manager
    private String socialSecurityNumber; // No validation!

    // Type code used in conditionals
    public double calculatePay() {
        if (employeeType == 1) {
            return baseSalary;
        } else if (employeeType == 2) {
            return baseSalary + commission;
        } else if (employeeType == 3) {
            return baseSalary + bonus;
        }
        return 0;
    }
}
```

[13] https://refactoring.guru/smells/primitive-obsession

Module B: Bloater #3 – Primitive Obsession (Java Example – After)

```java
// REFACTORED: Replace Data Value with Object, Replace Type Code with Subclasses
public class PhoneNumber {
    private String areaCode;
    private String number;
    public PhoneNumber(String areaCode, String number) {
        validate(areaCode, number);
        this.areaCode = areaCode;
        this.number = number;
    }
    private void validate(String areaCode, String number) { /* ... */ }
    public String format() { return "(" + areaCode + ") " + number; }
}

public class Money {
    private long amountInCents;
    private Currency currency;
    public Money(long amountInCents, Currency currency) {
        this.amountInCents = amountInCents;
        this.currency = currency;
    }
    public Money add(Money other) { /* currency-safe addition */ }
}

// Type code replaced with polymorphism
abstract class Employee {
    abstract double calculatePay();
}
class Engineer extends Employee {
    double calculatePay() { return getBaseSalary(); }
}
```

Module B: Bloater #3 – Primitive Obsession (Treatment & Payoff)

Treatment (Refactoring Techniques)

Technique	When to Use
Replace Data Value with Object	Group related primitive fields and relocate associated behavior
Introduce Parameter Object	When primitives appear grouped in method parameters
Preserve Whole Object	When you extract multiple values from an object and pass them individually
Replace Type Code with Class	When type code values do not affect behavior
Replace Type Code with Subclasses	When type code affects behavior
Replace Type Code with State/Strategy	When you cannot use subclasses
Replace Array with Object	When arrays contain mixed data types

Payoff

- Code becomes **more flexible** thanks to use of objects instead of primitives
- Better **understandability** and organization of code
- Easier to find **duplicate code** that processes the same data
- Enables **validation** inside the value object (e.g., phone number format)

Module B: Bloater #4 – Long Parameter List

What Is It?

A method has **more than three or four parameters**, making it hard to understand and use correctly.

Signs and Symptoms

- Method signatures that are too long to fit on one line
- Parameters that are hard to distinguish from each other
- Callers frequently pass wrong arguments in wrong positions

Reasons for the Problem

- A long parameter list may happen after **several types of algorithms are merged** in a single method
- A long list may have been created to **control which algorithm will be run** and how
- Long parameter lists can also result from efforts to **make classes more independent** of each other (instead of passing an object, individual values are passed)
- Long lists become "contradictory and hard to use as they grow longer"

Reference: RefactoringGuru – Long Parameter List[14]

Module B: Bloater #4 – Long Parameter List (Java Example – Before)

```java
// SMELL: Long Parameter List -- 13 parameters!
public void createUser(String firstName, String lastName,
    String email, String phone, String street,
    String city, String state, String zipCode,
    String country, int age, String gender,
    boolean isActive, String role) {
    // ...
}

// Another example -- parameter values that could be calculated
public double calculateFinalPrice(
    int quantity, double itemPrice,
    double seasonalDiscount, double memberDiscount,
    double taxRate, double shippingCost,
    boolean applyPromo) {
    double base = quantity * itemPrice;
    double discount = base * (seasonalDiscount + memberDiscount);
    double tax = (base - discount) * taxRate;
    return base - discount + tax + shippingCost;
}
```

Module B: Bloater #4 – Long Parameter List (Java Example – After)

```java
// REFACTORED: Introduce Parameter Object + Replace Parameter with Method Call
public class UserInfo {
    private String firstName, lastName, email, phone;
    private int age;
    private String gender;
```

[14] https://refactoring.guru/smells/long-parameter-list

```
    private boolean isActive;
    private String role;
    // constructor, getters...
}

public class Address {
    private String street, city, state, zipCode, country;
    // constructor, getters...
}

public void createUser(UserInfo userInfo, Address address) {
    // Clean, clear, and easy to understand
}

// Replace Parameter with Method Call
public double calculateFinalPrice(int quantity, double itemPrice) {
    double base = quantity * itemPrice;
    double discount = base * getSeasonalDiscount() + getMemberDiscount();
    double tax = (base - discount) * getTaxRate();
    return base - discount + tax + getShippingCost();
    // Method internally obtains values it needs
}
```

Module B: Bloater #4 – Long Parameter List (Treatment & Payoff)

Treatment (Refactoring Techniques)

Technique	When to Use
Replace Parameter with Method Call	When a parameter value can be obtained by calling a method of another object that is already available
Preserve Whole Object	Pass the whole object instead of extracting its individual fields
Introduce Parameter Object	Create a new class to group related parameters that often appear together

Payoff

- More readable, **shorter code**
- Refactoring may reveal previously unnoticed **duplicate code**
- Method signatures become self-documenting

When to Ignore

- Do not create unwanted **dependencies between classes** just to reduce parameter count
- If the only connection between parameters is that they appear together in this method, introducing a parameter object may be premature

---

Module B: Bloater #5 – Data Clumps

What Is It?

Identical groups of variables appear repeatedly across different parts of the code. These clumps should be turned into their own classes.

Signs and Symptoms

- The same group of fields appears in multiple classes (e.g., street, city, zipCode)
- The same group of parameters appears in multiple method signatures
- Removing one value from the group makes the rest meaningless

Reasons for the Problem

- Data clumps often appear when **copy-pasting** code
- Groups of data are passed around together but never formalized as a concept
- A good test: **delete one of the data values** – do the rest still make sense? If not, it is a clump

Reference: RefactoringGuru – Data Clumps[15]

Module B: Bloater #5 – Data Clumps (Java Example – Before)

```
// SMELL: Data Clumps -- same three fields repeated everywhere
class Customer {
    private String street;
    private String city;
    private String zipCode;
    // other fields...
}

class Order {
    private String shippingStreet;
    private String shippingCity;
    private String shippingZipCode;
    // other fields...
}

class Warehouse {
    private String street;
    private String city;
    private String zipCode;
    // other fields...
}

// Same data clump in method signatures
void deliverOrder(String street, String city, String zipCode,
                  String recipientName) {
    // ...
}
```

Module B: Bloater #5 – Data Clumps (Java Example – After)

```
// REFACTORED: Extract Class -- Address encapsulates the data clump
public class Address {
    private String street;
    private String city;
    private String zipCode;

    public Address(String street, String city, String zipCode) {
        this.street = street;
```

[15] https://refactoring.guru/smells/data-clumps

```
        this.city = city;
        this.zipCode = zipCode;
    }

    public String format() {
        return street + ", " + city + " " + zipCode;
    }
    // getters, equals, hashCode...
}

class Customer { private Address address; }
class Order { private Address shippingAddress; }
class Warehouse { private Address address; }

// Clean method signature using Introduce Parameter Object
void deliverOrder(Address deliveryAddress, String recipientName) {
    // ...
}
```

Module B: Bloater #5 – Data Clumps (Treatment & Payoff)

Treatment (Refactoring Techniques)

Technique	When to Use
Extract Class	Move the group of fields into a new class
Introduce Parameter Object	For method signatures with data clumps
Preserve Whole Object	Pass the whole object instead of individual fields

Payoff

- Improves **understanding and organization** of code
- Reduces **code size** by eliminating repeated field groups
- Operations associated with the data can be moved into the new class (e.g., address formatting, validation)

When to Ignore

- Passing complete objects as parameters may create **unwanted dependencies** between classes
- If the data group only appears together in two places, the overhead of a new class may not be worth it

---

Module B: Takeaway

Key Points – Bloaters

Smell	Key Takeaway
Long Method	If a method is longer than ~10 lines, consider extracting smaller methods
Large Class	If a class has more than one responsibility, split it up
Primitive Obsession	Replace primitives with small value objects for domain concepts
Long Parameter List	More than 3-4 parameters? Introduce parameter objects or pass whole objects

Smell	Key Takeaway
Data Clumps	Groups of data that always appear together belong in their own class

Common Thread

All bloaters grow **gradually over time**. No one writes a 200-line method on purpose on day one. Vigilance and regular refactoring prevent bloaters from taking hold.

> "If you feel the need to comment something inside a method, you should take that code and put it into a new method. Even a single line can and should be broken out into a separate method if it requires explanation." – Martin Fowler

---

Module C: Object-Orientation Abusers

Incomplete or Incorrect Application of Object-Oriented Programming Principles

---

Module C: Object-Orientation Abusers – Overview

Object-Orientation Abusers are cases where object-oriented programming principles are applied incompletely or incorrectly.

The 4 OO Abuser Smells

#	Smell	Key Issue
1	**Switch Statements**	Complex `switch` or `if` chains that could use polymorphism
2	**Temporary Field**	Fields that get values only under certain circumstances
3	**Refused Bequest**	Subclass uses only some of its parent's methods and properties
4	**Alternative Classes with Different Interfaces**	Two classes perform identical functions but have different method names

All these smells represent incomplete or incorrect applications of OOP principles.

Reference: RefactoringGuru – OO Abusers[16]

Module C: OO Abuser #1 – Switch Statements

What Is It?

You have a complex `switch` operator or sequence of `if` statements that performs different actions based on a type or condition.

Signs and Symptoms

- A complex `switch` or `if-else` chain that tests the same condition in multiple places
- The same `switch` structure is likely **duplicated** everywhere the type is handled
- Adding a new type requires modifying **every** `switch` statement

[16] https://refactoring.guru/refactoring/smells/oo-abusers

Reasons for the Problem

- Relatively rare use of `switch` and `case` is one of the hallmarks of object-oriented code
- Often a `switch` is scattered in different places in the program
- When a new condition is added, you have to find all the `switch` code and modify it
- As a rule of thumb, **when you see a `switch` you should think of polymorphism**

Reference: RefactoringGuru – Switch Statements[17]

Switch Statements - Before (Code Smell)

Switch Statements - After (Refactored)

Module C: OO Abuser #1 – Switch Statements (Java Example – Before)

```java
// SMELL: Switch Statements -- duplicated conditional logic
public class PayrollCalculator {

    public double calculatePay(Employee employee) {
        switch (employee.getType()) {
            case ENGINEER:
                return employee.getMonthlySalary();
            case SALESMAN:
                return employee.getMonthlySalary()
                       + employee.getCommission();
            case MANAGER:
                return employee.getMonthlySalary()
                       + employee.getBonus();
            default:
                throw new RuntimeException("Invalid employee type");
        }
    }

    // Same switch duplicated in another method!
    public int calculateVacationDays(Employee employee) {
        switch (employee.getType()) {
            case ENGINEER:  return 15;
            case SALESMAN:  return 12;
            case MANAGER:   return 20;
            default: throw new RuntimeException("Invalid type");
        }
    }
}
```

Module C: OO Abuser #1 – Switch Statements (Java Example – After)

```java
// REFACTORED: Replace Conditional with Polymorphism
public abstract class Employee {
    private double monthlySalary;
```

[17] https://refactoring.guru/smells/switch-statements

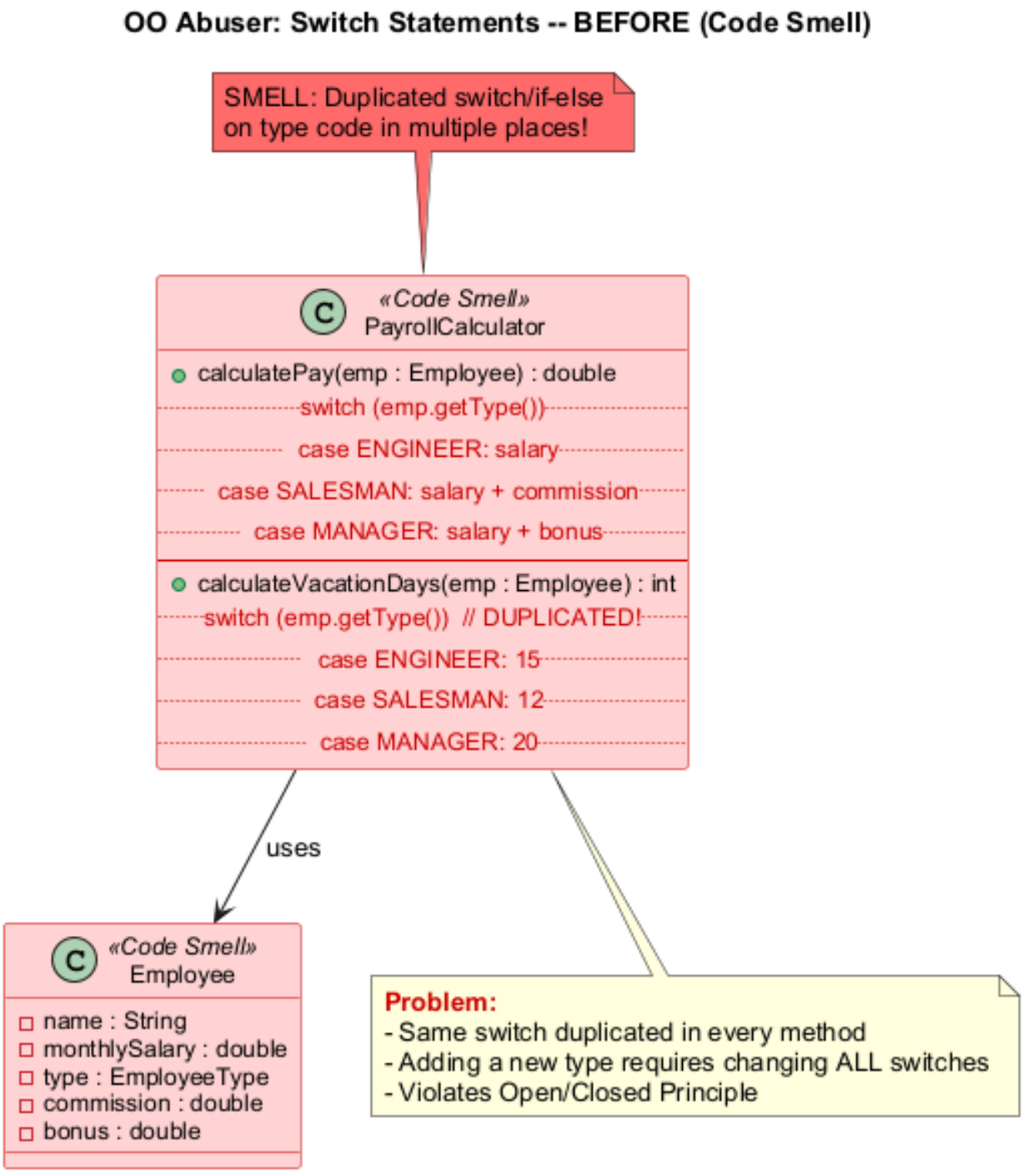


Figure 8: center

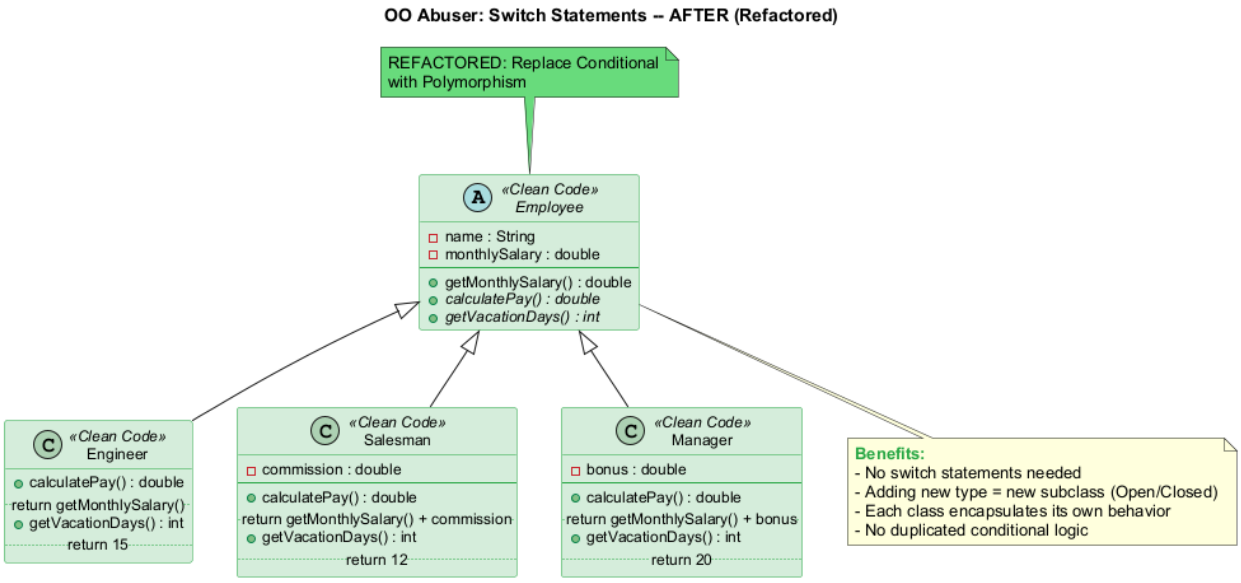


Figure 9: center

```
    public abstract double calculatePay();
    public abstract int getVacationDays();

    public double getMonthlySalary() { return monthlySalary; }
}

public class Engineer extends Employee {
    public double calculatePay() { return getMonthlySalary(); }
    public int getVacationDays() { return 15; }
}

public class Salesman extends Employee {
    private double commission;
    public double calculatePay() {
        return getMonthlySalary() + commission;
    }
    public int getVacationDays() { return 12; }
}

public class Manager extends Employee {
    private double bonus;
    public double calculatePay() {
        return getMonthlySalary() + bonus;
    }
    public int getVacationDays() { return 20; }
}
```

Module C: OO Abuser #1 – Switch Statements (Treatment & Payoff)

Treatment (Refactoring Techniques)

Technique	When to Use
Replace Type Code with Subclasses	Create subclasses for each type
Replace Conditional with Polymorphism	Use method overriding instead of conditionals
Replace Type Code with State/Strategy	When subclasses are not possible
Replace Parameter with Explicit Methods	When conditions call the same method with different parameters
Introduce Null Object	For null conditions

Payoff

- Improved **code organization** through polymorphism
- **Easier to add new types** – just add a new subclass (Open/Closed Principle)
- Eliminates **duplicated conditional logic** scattered throughout the codebase

When to Ignore

- When a `switch` performs **simple actions** and appears in only one place
- Simple factory methods (Factory Method or Abstract Factory patterns) are often acceptable
- If the switch is unlikely to change, polymorphism may be overkill

---

Module C: OO Abuser #2 – Temporary Field

What Is It?

Temporary fields get their values (and thus are needed by objects) only **under certain circumstances**. Outside of these circumstances, they are empty or contain irrelevant data.

Signs and Symptoms

- Object fields that are only populated and used during specific algorithm execution
- Fields that are `null` or empty most of the time
- You expect to see data in fields that are **almost always empty**

Reasons for the Problem

- Often temporary fields are created for use in an algorithm that **requires a large amount of inputs**
- Instead of creating many parameters in the method, the programmer decides to create **fields** for this data in the class
- These fields are used only in the algorithm and are useless the rest of the time
- This code is hard to understand because you expect object fields to consistently contain meaningful data

Reference: RefactoringGuru – Temporary Field[18]

Module C: OO Abuser #2 – Temporary Field (Java Example – Before)

```java
// SMELL: Temporary Field -- fields meaningful only during checkout
public class Order {
    private List<Item> items;
    private Customer customer;

    // These fields are only used during checkout!
    private double taxRate;          // Only set during checkout
```

[18] https://refactoring.guru/smells/temporary-field

```
    private String promoCode;        // Only set when applying discount
    private double shippingWeight;   // Only relevant for physical items
    private String shippingMethod;   // Empty until shipping is selected

    public double checkout() {
        // Uses taxRate, promoCode, shippingWeight, shippingMethod
        // These are confusing when order is not in checkout phase
        double subtotal = calculateSubtotal();
        double discount = promoCode != null ? applyPromo(promoCode) : 0;
        double tax = subtotal * taxRate;
        double shipping = calculateShipping(shippingWeight, shippingMethod);
        return subtotal - discount + tax + shipping;
    }
}
```

Module C: OO Abuser #2 – Temporary Field (Java Example – After)

```
// REFACTORED: Extract Class -- separate checkout context
public class Order {
    private List<Item> items;
    private Customer customer;

    public double calculateSubtotal() {
        return items.stream()
            .mapToDouble(item -> item.getPrice() * item.getQuantity())
            .sum();
    }
}

public class CheckoutContext {
    private double taxRate;
    private String promoCode;
    private double shippingWeight;
    private String shippingMethod;

    public CheckoutContext(double taxRate, String promoCode,
                           double shippingWeight, String shippingMethod) {
        this.taxRate = taxRate;
        this.promoCode = promoCode;
        this.shippingWeight = shippingWeight;
        this.shippingMethod = shippingMethod;
    }

    public double calculateFinalPrice(Order order) {
        double subtotal = order.calculateSubtotal();
        double discount = promoCode != null ? applyPromo(promoCode, subtotal) : 0;
        double tax = subtotal * taxRate;
        double shipping = calculateShipping(shippingWeight, shippingMethod);
        return subtotal - discount + tax + shipping;
    }
}
```

Module C: OO Abuser #2 – Temporary Field (Treatment & Payoff)

Treatment (Refactoring Techniques)

Technique	When to Use
Extract Class	Move temporary fields and related operations into a separate class (creates a method object)
Introduce Null Object	Replace conditional code checking for temporary field existence with a Null Object
Replace Method with Method Object	If the temporary fields exist for a complex algorithm, extract the entire algorithm into a separate class

Payoff

- **Better code clarity and organization** – no more confusion about when fields have meaningful values
- All fields in a class are always meaningful
- Code becomes easier to reason about

When to Ignore

- If the temporary state is truly transient and clearly documented
- If the algorithm is simple enough that the temporary fields do not cause confusion

---

Module C: OO Abuser #3 – Refused Bequest

What Is It?

If a subclass uses **only some** of the methods and properties inherited from its parents, the hierarchy is off-kilter. The unneeded methods may simply go unused or be **redefined to throw exceptions**.

Signs and Symptoms

- A subclass overrides parent methods to throw `UnsupportedOperationException`
- A subclass leaves most inherited methods unused
- The inheritance relationship does not model a true "is-a" relationship

Reasons for the Problem

- Someone was motivated to create inheritance between classes only by the **desire to reuse code** in a superclass
- But the superclass and subclass are **completely different** in nature
- Like making Dog extend Chair because both have four legs

Reference: RefactoringGuru – Refused Bequest[19]

Module C: OO Abuser #3 – Refused Bequest (Java Example – Before)

```java
// SMELL: Refused Bequest -- Dog inherits fly() it cannot use
public class Animal {
    public void walk() { /* walking logic */ }
    public void fly()  { /* flying logic */ }
    public void swim() { /* swimming logic */ }
    public void eat()  { /* eating logic */ }
```

[19] https://refactoring.guru/smells/refused-bequest

```
}

public class Dog extends Animal {
    // Dog can walk, swim, and eat -- but NOT fly
    @Override
    public void fly() {
        throw new UnsupportedOperationException("Dogs can't fly!");
    }
}

public class Fish extends Animal {
    // Fish can swim and eat -- but NOT walk or fly
    @Override
    public void walk() {
        throw new UnsupportedOperationException("Fish can't walk!");
    }
    @Override
    public void fly() {
        throw new UnsupportedOperationException("Fish can't fly!");
    }
}
```

Module C: OO Abuser #3 – Refused Bequest (Java Example – After)

```
// REFACTORED: Extract Superclass + Replace Inheritance with Delegation
public abstract class Animal {
    public abstract void eat();
}

public interface Walkable { void walk(); }
public interface Swimmable { void swim(); }
public interface Flyable { void fly(); }

public class Dog extends Animal implements Walkable, Swimmable {
    public void eat()  { /* eating logic */ }
    public void walk() { /* walking logic */ }
    public void swim() { /* swimming logic */ }
    // No fly() method -- dogs don't fly!
}

public class Eagle extends Animal implements Walkable, Flyable {
    public void eat()  { /* eating logic */ }
    public void walk() { /* walking logic */ }
    public void fly()  { /* flying logic */ }
}

public class Fish extends Animal implements Swimmable {
    public void eat()  { /* eating logic */ }
    public void swim() { /* swimming logic */ }
}
```

Module C: OO Abuser #3 – Refused Bequest (Treatment & Payoff)

Treatment (Refactoring Techniques)

Technique	When to Use
Replace Inheritance with Delegation	When inheritance does not make logical sense, eliminate the parent-child relationship and use composition instead
Extract Superclass	If inheritance is appropriate, create a new intermediary superclass that both classes can extend without refused methods

Payoff

- Improves code **clarity and organization**
- Eliminates confusing or meaningless inheritance hierarchies
- Adheres to the **Liskov Substitution Principle** (a subtype should be substitutable for its parent type)
- No more methods that throw `UnsupportedOperationException`

When to Ignore

- If the refused bequest is minimal and does not cause confusion
- If reuse through inheritance provides significant benefits and the refused methods are clearly documented

---

Module C: OO Abuser #4 – Alternative Classes with Different Interfaces

What Is It?

Two classes perform **identical functions** but have **different method names**. They should share a common interface.

Signs and Symptoms

- Two classes that do the same thing but with different method names
- Client code uses `if-else` to choose between the two classes
- The classes cannot be used interchangeably

Reasons for the Problem

- The developer who created one class probably **did not know** that a functionally equivalent class already existed
- Or they created it independently and did not think to unify the interfaces

Reference: RefactoringGuru – Alternative Classes with Different Interfaces[20]

Module C: OO Abuser #4 – Alternative Classes (Java Example – Before)

```java
// SMELL: Alternative Classes with Different Interfaces
public class EmailNotifier {
    public void sendEmail(String to, String subject, String body) {
        // sends email notification
    }
}

public class SMSNotifier {
```

[20] https://refactoring.guru/smells/alternative-classes-with-different-interfaces

```
    public void dispatchSMS(String phoneNumber, String message) {
        // sends SMS notification
    }
}

// Client code has to know the specific interface of each
public class NotificationService {
    public void notifyUser(User user, String message) {
        if (user.prefersEmail()) {
            EmailNotifier emailer = new EmailNotifier();
            emailer.sendEmail(user.getEmail(), "Notification", message);
        } else {
            SMSNotifier sms = new SMSNotifier();
            sms.dispatchSMS(user.getPhone(), message);
        }
    }
}
```

Module C: OO Abuser #4 – Alternative Classes (Java Example – After)

```
// REFACTORED: Extract Superclass / common interface
public interface Notifier {
    void send(String recipient, String message);
}

public class EmailNotifier implements Notifier {
    @Override
    public void send(String recipient, String message) {
        // sends email notification to 'recipient' (email address)
    }
}

public class SMSNotifier implements Notifier {
    @Override
    public void send(String recipient, String message) {
        // sends SMS notification to 'recipient' (phone number)
    }
}

// Client code uses the common interface -- polymorphism!
public class NotificationService {
    public void notifyUser(User user, String message) {
        Notifier notifier = user.prefersEmail()
            ? new EmailNotifier()
            : new SMSNotifier();
        notifier.send(user.getContactInfo(), message);
    }
}
```

Module C: OO Abuser #4 – Alternative Classes (Treatment & Payoff)

Treatment (Refactoring Techniques)

Technique	When to Use
Rename Method	Make method names consistent across the alternative classes
Move Method / Add Parameter / Parameterize Method	Align method signatures and implementations
Extract Superclass	If only part of the functionality is duplicated, extract common parts into a parent class
Delete redundant class	Once unified, one of the duplicate classes can be removed

Payoff

- Eliminates unnecessary **duplicated code**, making the code lighter
- Code becomes **more readable** and understandable through a unified interface
- Enables **polymorphism** – clients can use the common interface interchangeably

When to Ignore

- Sometimes merging classes is impossible, such as when the alternative classes are in **different libraries** each with their own versioning

---

Module C: Takeaway

Key Points – Object-Orientation Abusers

Smell	Key Takeaway
Switch Statements	Replace with polymorphism; think subclasses instead of conditionals
Temporary Field	Extract into a separate class; fields should always be meaningful
Refused Bequest	Replace inheritance with delegation if the "is-a" relationship is wrong
Alt. Classes w/ Different Interfaces	Unify interfaces; extract superclass or common interface

Common Thread

All OO Abusers stem from **misunderstanding or misapplying** OOP principles. The fix is almost always: use polymorphism correctly, prefer composition over inheritance when appropriate, and design clean interfaces.

> "When you see a switch statement, think polymorphism." – RefactoringGuru

---

Module D: Change Preventers

Smells That Make Changing Code Disproportionately Difficult

Module D: Change Preventers – Overview

Change Preventers are smells that mean if you need to change something in one place in your code, you have to make many changes in other places too. Program development becomes much more expensive and complex as a result.

The 3 Change Preventer Smells

#	Smell	Key Issue
1	**Divergent Change**	One class is commonly changed in different ways for different reasons
2	**Shotgun Surgery**	A single change requires edits in many different classes
3	**Parallel Inheritance Hierarchies**	Creating a subclass in one hierarchy forces creation in another

> These smells are **opposites** in some ways: Divergent Change is one class suffering many changes, while Shotgun Surgery is one change affecting many classes.

Reference: RefactoringGuru – Change Preventers[21]

Module D: Change Preventer #1 – Divergent Change

What Is It?

You find yourself having to change **many unrelated methods** when you make changes to a class. For example, when adding a new product type, you have to change the methods for finding, displaying, and ordering products.

Signs and Symptoms

- A class requires frequent modifications for different, unrelated reasons
- One class has accumulated responsibilities that should belong to different classes
- The class violates the **Single Responsibility Principle**

Reasons for the Problem

- These scattered modifications typically stem from **poor program structure** or copy-paste programming
- A class has gradually absorbed multiple responsibilities over time
- "This is the Order class – I'll just add the display logic here too"

Reference: RefactoringGuru – Divergent Change[22]

Divergent Change (Diagram)

Module D: Change Preventer #1 – Divergent Change (Java Example – Before)

```java
// SMELL: Divergent Change -- one class, many reasons to change
public class Order {
    private List<Item> items;
    private Customer customer;

    // Changes when product types change
    public void addProduct(Product p) { /* ... */ }
```

[21] https://refactoring.guru/refactoring/smells/change-preventers
[22] https://refactoring.guru/smells/divergent-change

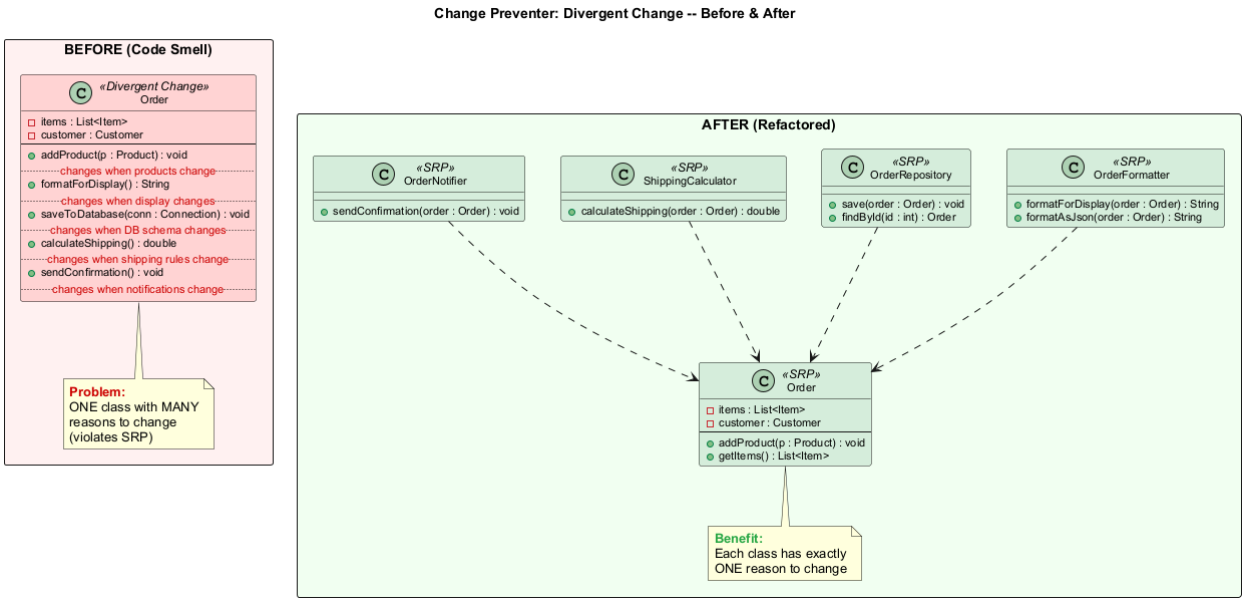


Figure 10: center

```
    // Changes when display format changes
    public String formatForDisplay() {
        return "Order: " + items.toString(); // HTML? JSON? XML?
    }

    // Changes when database schema changes
    public void saveToDatabase(Connection conn) {
        String sql = "INSERT INTO orders ...";
        // JDBC code here
    }

    // Changes when shipping rules change
    public double calculateShipping() {
        if (getTotalWeight() > 50) return 15.99;
        return 5.99;
    }

    // Changes when notification requirements change
    public void sendConfirmation() {
        // email sending logic
    }
}
```

Module D: Change Preventer #1 – Divergent Change (Java Example – After)

```
// REFACTORED: Extract Class -- each class has a single reason to change
public class Order {
    private List<Item> items;
    private Customer customer;
    public void addProduct(Product p) { /* ... */ }
    public List<Item> getItems() { return items; }
}
```

```java
public class OrderFormatter {
    public String formatForDisplay(Order order) {
        return "Order: " + order.getItems().toString();
    }
    public String formatAsJson(Order order) { /* ... */ }
}

public class OrderRepository {
    public void save(Order order) { /* database logic */ }
    public Order findById(int id) { /* database logic */ }
}

public class ShippingCalculator {
    public double calculateShipping(Order order) { /* shipping rules */ }
}

public class OrderNotifier {
    public void sendConfirmation(Order order) { /* email logic */ }
}
```

Module D: Change Preventer #1 – Divergent Change (Treatment & Payoff)

Treatment (Refactoring Techniques)

Technique	When to Use
Extract Class	Split up the behavior of the class into separate classes, each with a single responsibility
Extract Superclass	Combine classes sharing identical behavior through inheritance
Extract Subclass	Use inheritance to organize related but distinct behaviors

Payoff

- Improves **code organization** – each class changes for exactly one reason
- Reduces **code duplication** across the codebase
- Simplifies **support and maintenance** – changes are localized
- Follows the **Single Responsibility Principle** (SRP)

Divergent Change vs. Shotgun Surgery

Divergent Change	Shotgun Surgery
Many **different changes** in one class	One change affects **many classes**
One class, many reasons to change	One reason to change, many classes

---

Module D: Change Preventer #2 – Shotgun Surgery

What Is It?

Making any modification requires you to make many **small changes to many different classes**.

Signs and Symptoms

- A single logical change (adding a field, changing a rule) requires touching 5+ classes
- You often forget to make a change in one of the scattered locations
- The scattered classes all deal with the same concern

Reasons for the Problem

- A **single responsibility has been split** up among a large number of classes
- This can happen after overzealous application of Divergent Change refactoring – splitting a class too aggressively
- Incremental development where related code ended up in different places

Reference: RefactoringGuru – Shotgun Surgery[23]

Shotgun Surgery (Diagram)

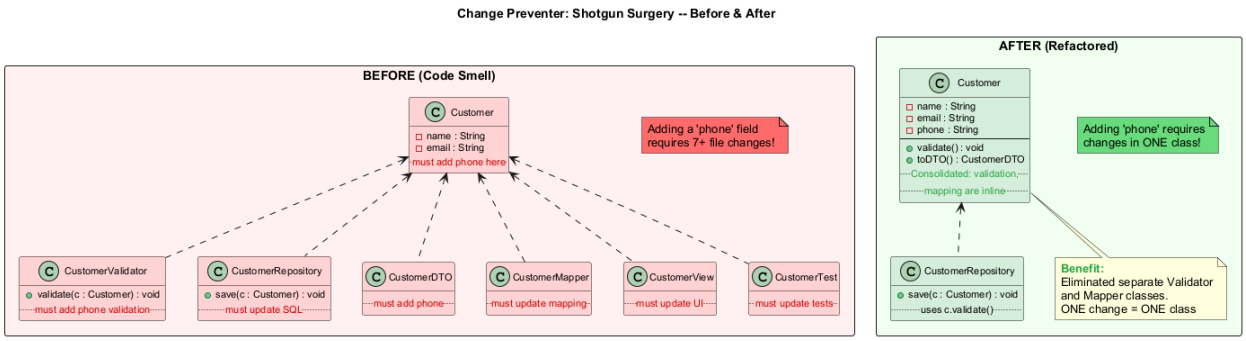


Figure 11: center

Module D: Change Preventer #2 – Shotgun Surgery (Java Example – Before)

```java
// SMELL: Shotgun Surgery -- adding a "phone" field requires 7+ file changes!
public class Customer {
    private String name;
    private String email;
    // Adding "phone" requires changes in ALL of these classes:
}

public class CustomerValidator {
    public void validate(Customer c) {
        if (c.getName() == null) throw new ValidationException("Name required");
        if (c.getEmail() == null) throw new ValidationException("Email required");
        // Must add phone validation here too
    }
}

public class CustomerRepository {
    public void save(Customer c) {
        String sql = "INSERT INTO customers (name, email) VALUES (?, ?)";
        // Must update SQL to include phone
    }
```

[23] https://refactoring.guru/smells/shotgun-surgery

```
}

public class CustomerDTO { /* Must add phone */ }
public class CustomerMapper { /* Must update mapping */ }
public class CustomerView { /* Must update UI */ }
public class CustomerTest { /* Must update tests */ }
// 7+ files changed for ONE small addition!
```

Module D: Change Preventer #2 – Shotgun Surgery (Java Example – After)

```
// REFACTORED: Consolidate responsibility -- Move Method + Move Field
public class Customer {
    private String name;
    private String email;
    private String phone; // New field -- only one class to change!

    public void validate() {
        if (name == null) throw new ValidationException("Name required");
        if (email == null) throw new ValidationException("Email required");
        if (phone == null) throw new ValidationException("Phone required");
    }

    public CustomerDTO toDTO() {
        return new CustomerDTO(name, email, phone);
    }
}

public class CustomerRepository {
    public void save(Customer c) {
        c.validate(); // Customer validates itself
        // Single place for persistence logic
    }
}

// Eliminated: CustomerValidator (inlined into Customer)
// Eliminated: CustomerMapper (inlined into Customer.toDTO())
```

Module D: Change Preventer #2 – Shotgun Surgery (Treatment & Payoff)

Treatment (Refactoring Techniques)

Technique	When to Use
Move Method and **Move Field**	Move existing methods and fields from scattered classes into a single class
Create new class	If no existing class seems like a good fit for the consolidated responsibility
Inline Class	If moving code to a related class empties the donor class, eliminate the now-redundant class

Payoff

- Better **code organization** – related logic lives together
- Less **code duplication** across the codebase

- **Easier maintenance** – changes are localized to fewer files
- Reduces the risk of **missing a change** in one of many scattered locations

When to Ignore

- Some frameworks (Spring, Jakarta EE) intentionally separate concerns into layers (Controller, Service, Repository)
- In these cases, shotgun surgery may be an acceptable tradeoff for architectural clarity

---

Module D: Change Preventer #3 – Parallel Inheritance Hierarchies

What Is It?

Whenever you create a subclass for a class, you find yourself needing to create a subclass for **another class** as well.

Signs and Symptoms

- Two class hierarchies grow in lockstep
- Adding a class in one hierarchy always requires adding a class in the other
- Class prefixes from one hierarchy mirror prefixes in another (e.g., `Circle`/`CircleRenderer`)

Reasons for the Problem

- Initially manageable class structures become increasingly difficult to modify as new classes are added
- The parallel may have been so small nobody noticed at first
- This smell is actually a **special case of Shotgun Surgery**

Reference: RefactoringGuru – Parallel Inheritance Hierarchies[24]

Module D: Change Preventer #3 – Parallel Inheritance (Java Example – Before)

```java
// SMELL: Parallel Inheritance Hierarchies
// Every time you add a new shape, you must also add a new renderer
public abstract class Shape {
    abstract double area();
}
public class Circle extends Shape {
    double area() { return Math.PI * radius * radius; }
}
public class Rectangle extends Shape {
    double area() { return width * height; }
}
public class Triangle extends Shape {
    double area() { return 0.5 * base * height; }
}

// Parallel hierarchy -- grows in lockstep!
public abstract class ShapeRenderer {
    abstract void render(Shape shape);
}
public class CircleRenderer extends ShapeRenderer {
    void render(Shape shape) { /* draw circle */ }
}
public class RectangleRenderer extends ShapeRenderer {
```

[24] https://refactoring.guru/smells/parallel-inheritance-hierarchies

```
    void render(Shape shape) { /* draw rectangle */ }
}
public class TriangleRenderer extends ShapeRenderer {
    void render(Shape shape) { /* draw triangle */ }
}
// Add Pentagon? Must add PentagonRenderer too!
```

Module D: Change Preventer #3 – Parallel Inheritance (Java Example – After)

```
// REFACTORED: Move Method + Move Field -- merge hierarchies
public abstract class Shape {
    abstract double area();
    abstract void render(); // Rendering logic moved INTO the shape
}

public class Circle extends Shape {
    private double radius;

    double area() { return Math.PI * radius * radius; }

    void render() {
        // Circle rendering logic (formerly in CircleRenderer)
        System.out.println("Drawing circle with radius " + radius);
    }
}

public class Rectangle extends Shape {
    private double width, height;

    double area() { return width * height; }

    void render() {
        // Rectangle rendering logic (formerly in RectangleRenderer)
        System.out.println("Drawing rectangle " + width + "x" + height);
    }
}
// No more parallel ShapeRenderer hierarchy!
// Adding Pentagon requires changes in ONE hierarchy only
```

Module D: Change Preventer #3 – Parallel Inheritance (Treatment & Payoff)

Treatment (Refactoring Techniques)

Technique	When to Use
Move Method and **Move Field**	De-duplicate the parallel hierarchy by moving behavior from one hierarchy into the other
Create cross-references	Establish references where one hierarchy's instances point to another hierarchy's instances, then eliminate the redundant hierarchy

Payoff

- Reduces **code duplication** across related hierarchies

- Improves overall **code organization** and maintainability
- Adding a new type requires changes in **one hierarchy** instead of two

When to Ignore

- Sometimes having parallel class hierarchies is a way of **avoiding even bigger design problems**
- If they provide clear separation of concerns (e.g., rendering vs. business logic), they may be justified
- If refactoring attempts produce uglier code, it may be wiser to accept the current structure

---

Module D: Takeaway

Key Points – Change Preventers

Smell	Key Takeaway
Divergent Change	One class with multiple reasons to change – split it into focused classes (SRP)
Shotgun Surgery	One change touching many classes – consolidate the scattered responsibility
Parallel Inheritance Hierarchies	Two hierarchies growing in lockstep – merge them when possible

Common Thread

Change Preventers make software **rigid** – they resist change by requiring disproportionate effort for simple modifications. The fixes involve reorganizing responsibilities so that changes are **localized**.

> "The art of refactoring is knowing which smells indicate an opportunity for improvement and which are acceptable trade-offs." – Martin Fowler

---

Module E: Dispensables

Something Pointless and Unneeded Whose Absence Would Make Code Cleaner

---

Module E: Dispensables – Overview

A **dispensable** is something pointless and unneeded whose absence would make the code cleaner, more efficient, and easier to understand.

The 6 Dispensable Smells

#	Smell	Key Issue
1	**Comments**	Method is filled with explanatory comments
2	**Duplicate Code**	Two code fragments look almost identical
3	**Lazy Class**	A class that does too little to earn its keep
4	**Data Class**	A class with only fields and getters/setters
5	**Dead Code**	Code that is no longer used

#	Smell	Key Issue
6	**Speculative Generality**	Unused classes, methods, or parameters created "just in case"

Reference: RefactoringGuru – Dispensables[25]

Module E: Dispensable #1 – Comments

What Is It?

A method is filled with **explanatory comments** that attempt to explain what the code does. The best comment is a good name for a method or class.

Signs and Symptoms

- Large comment blocks explaining what the next few lines do
- Comments that restate the code in English
- Comments used as "deodorant" trying to mask the smell of bad code
- Outdated comments that no longer match the code

Reasons for the Problem

- Comments are usually created with the **best of intentions** when the author realizes code is not intuitive
- The comment acts as a deodorant trying to mask the smell of bad code
- If you feel that a code fragment cannot be understood without comments, try to **change the code structure** so that comments become unnecessary

Reference: RefactoringGuru – Comments[26]

Module E: Dispensable #1 – Comments (Java Example – Before)

```java
// SMELL: Comments -- code needs excessive explanation
public class PricingEngine {

    public double calculateDiscount(Customer customer, double price) {
        // Check if the customer is eligible for a premium discount
        // by verifying their membership status and purchase history.
        // A customer qualifies if they have been a member for more
        // than 2 years AND have made purchases totaling at least $1000
        // AND their account status is active.
        if ((customer.getMembershipYears() > 2) &&
            (customer.getTotalPurchases() >= 1000.0) &&
            (customer.getStatus().equals("ACTIVE"))) {
            // Apply the premium discount rate of 15%
            // This was changed from 10% in Q3 2024
            return price * 0.15;
        }
        // Apply standard discount rate of 5% for non-premium customers
        return price * 0.05;
    }
}
```

[25] https://refactoring.guru/refactoring/smells/dispensables
[26] https://refactoring.guru/smells/comments

Module E: Dispensable #1 – Comments (Java Example – After)

```java
// REFACTORED: Extract Method + Extract Variable -- self-documenting code
public class PricingEngine {

    private static final double PREMIUM_DISCOUNT_RATE = 0.15;
    private static final double STANDARD_DISCOUNT_RATE = 0.05;

    public double calculateDiscount(Customer customer, double price) {
        if (customer.isEligibleForPremiumDiscount()) {
            return price * PREMIUM_DISCOUNT_RATE;
        }
        return price * STANDARD_DISCOUNT_RATE;
    }
}

// In Customer class:
public class Customer {
    private static final int MIN_MEMBERSHIP_YEARS = 2;
    private static final double MIN_PURCHASE_AMOUNT = 1000.0;

    public boolean isEligibleForPremiumDiscount() {
        return getMembershipYears() > MIN_MEMBERSHIP_YEARS
            && getTotalPurchases() >= MIN_PURCHASE_AMOUNT
            && isActive();
    }

    public boolean isActive() {
        return "ACTIVE".equals(status);
    }
}
```

Module E: Dispensable #1 – Comments (Treatment & Payoff)

Treatment (Refactoring Techniques)

Technique	When to Use
Extract Variable	When a comment explains a complex expression, extract it into a well-named variable
Extract Method	When a comment explains a section of code, extract that section into a method with a descriptive name
Rename Method	When a method has been extracted but the name does not explain it well enough
Introduce Assertion	When a comment documents a necessary system state or rule

Payoff

- Code becomes **more self-explanatory** and intuitive
- Eliminates the risk of **outdated or misleading comments**
- Reduces maintenance burden (no need to update comments when code changes)

When to Ignore

- **Explaining WHY** something is done: `// Using insertion sort because array is nearly sorted`
- **Warning of consequences**: `// WARNING: Changing this order breaks the payment pipeline`
- **TODO comments**: `// TODO: Replace with async call after API v2 migration`
- **Javadoc for public APIs**: Documenting parameters, return values, exceptions

---

Module E: Dispensable #2 – Duplicate Code

What Is It?

Two code fragments look **almost identical**. This is one of the most common code smells.

Signs and Symptoms

- The same code structure appears in multiple places with minor variations
- Copy-pasted code with small modifications
- Two methods that do essentially the same thing with different variable names

Reasons for the Problem

- **Multiple programmers** working simultaneously write similar solutions independently
- **Copy-paste programming** under time pressure and deadline rushing
- Subtle duplication where code *looks* different but performs identical operations
- Developer laziness or inability to refactor existing "almost right" code

Reference: RefactoringGuru – Duplicate Code[27]

Module E: Dispensable #2 – Duplicate Code (Java Example – Before)

```java
// SMELL: Duplicate Code -- same logic in two methods
public class ReportGenerator {

    public void printOwing() {
        // Print banner (DUPLICATED)
        System.out.println("**********************");
        System.out.println("*** Customer Owes ***");
        System.out.println("**********************");

        // Calculate outstanding (DUPLICATED)
        double outstanding = 0.0;
        for (Order order : orders) {
            outstanding += order.getAmount();
        }
        System.out.println("Amount: " + outstanding);
    }

    public void printStatement() {
        // Print banner (DUPLICATED)
        System.out.println("**********************");
        System.out.println("*** Customer Owes ***");
        System.out.println("**********************");

        // Calculate outstanding (DUPLICATED)
```

[27] https://refactoring.guru/smells/duplicate-code

```
        double outstanding = 0.0;
        for (Order order : orders) {
            outstanding += order.getAmount();
        }
        // ... additional statement logic
    }
}
```

Module E: Dispensable #2 – Duplicate Code (Java Example – After)

```
// REFACTORED: Extract Method -- duplicated code consolidated
public class ReportGenerator {

    private void printBanner() {
        System.out.println("**********************");
        System.out.println("*** Customer Owes ***");
        System.out.println("**********************");
    }

    private double calculateOutstanding() {
        return orders.stream()
            .mapToDouble(Order::getAmount)
            .sum();
    }

    public void printOwing() {
        printBanner();
        double outstanding = calculateOutstanding();
        System.out.println("Amount: " + outstanding);
    }

    public void printStatement() {
        printBanner();
        double outstanding = calculateOutstanding();
        // ... additional statement logic
    }
}
```

Module E: Dispensable #2 – Duplicate Code (Treatment & Payoff)

Treatment (Refactoring Techniques)

Context	Technique
Same code in two methods of the same class	**Extract Method** – create a shared method
Same code in two subclasses	**Extract Method** + **Pull Up Method/Field** to the superclass; use **Form Template Method** if similar but not identical
Same code in two unrelated classes	**Extract Class** or **Extract Superclass** depending on hierarchy
Conditional duplicates	**Consolidate Conditional Expression** + **Extract Method**

Payoff

- "Merging duplicate code **simplifies** the structure of your code and makes it shorter"
- Bug fixes only need to be applied **once** instead of in multiple places
- Code that is easier to understand and cheaper to maintain

When to Ignore

- In very rare cases, merging two identical code fragments can make the code **less intuitive**
- In **test code**, sometimes it is acceptable to keep tests independent and duplicated for clarity

---

Module E: Dispensable #3 – Lazy Class

What Is It?

A class that does **too little** to justify its existence. Understanding and maintaining classes always costs time and money. So if a class does not do enough to earn your attention, it should be deleted.

Signs and Symptoms

- The class has very few methods or fields
- The class does nothing more than wrap a single value
- Other classes do all the real work with this class's data

Reasons for the Problem

- A class may have become ridiculously small after **refactoring** reduced its original functionality
- Or it was **designed to support future development** that never happened
- Or it was created from an Extract Class refactoring that was **too aggressive**

Reference: RefactoringGuru – Lazy Class[28]

Module E: Dispensable #3 – Lazy Class (Java Example – Before)

```java
// SMELL: Lazy Class -- does almost nothing
public class PhoneNumber {
    private String number;

    public String getNumber() {
        return number;
    }

    public void setNumber(String number) {
        this.number = number;
    }
    // That's it. No validation, no formatting, no logic at all.
}

public class Customer {
    private String name;
    private PhoneNumber phone; // Wrapper adds no value

    public String getPhoneNumber() {
        return phone.getNumber();
    }
```

[28] https://refactoring.guru/smells/lazy-class

```
    public void setPhoneNumber(String number) {
        phone.setNumber(number);
    }
}
```

Module E: Dispensable #3 – Lazy Class (Java Example – After)

```
// REFACTORED: Inline Class -- merge lazy class into its user
public class Customer {
    private String name;
    private String phone; // Simple field, no wrapper needed

    public String getPhone() { return phone; }
    public void setPhone(String phone) { this.phone = phone; }
}

// ALTERNATIVE: If PhoneNumber SHOULD exist, give it real behavior
public class PhoneNumber {
    private String areaCode;
    private String number;

    public PhoneNumber(String rawNumber) {
        validate(rawNumber);
        parse(rawNumber);
    }

    private void validate(String rawNumber) {
        if (!rawNumber.matches("\\d{3}-\\d{4}"))
            throw new IllegalArgumentException("Invalid phone number");
    }

    public String format() {
        return "(" + areaCode + ") " + number;
    }
    // Now PhoneNumber earns its keep!
}
```

Module E: Dispensable #3 – Lazy Class (Treatment & Payoff)

Treatment (Refactoring Techniques)

Technique	When to Use
Inline Class	Merge the lazy class into the class that uses it
Collapse Hierarchy	If a subclass is essentially the same as its superclass, merge them

Payoff

- Reduced **code size** and fewer classes to understand
- Easier **maintenance** – less cognitive load
- Simpler class hierarchy and navigation

When to Ignore

- Sometimes a Lazy Class is created to **delineate intentions for future development**
- If the class represents a **domain concept** that may grow, keeping it may be wise
- If the class provides **type safety** (e.g., `EmailAddress` vs. `String`), it may be worth keeping even if simple
- Maintain a balance between clarity and simplicity

Module E: Dispensable #4 – Data Class

What Is It?

A **data class** is a class that contains only fields and crude methods for accessing them (getters and setters). These are simply containers for data used by other classes. They do not contain any additional functionality and cannot independently operate on the data they own.

Signs and Symptoms

- A class with only private fields, getters, and setters
- No methods that perform logic or operations on the data
- All behavior that manipulates this class's data lives in **other** classes

Reasons for the Problem

- It is normal for a newly created class to contain only a few public fields
- But the true power of objects is that they can contain **behavior** along with data
- Data classes indicate that **behavior** related to this data lives somewhere else – which violates encapsulation

Reference: RefactoringGuru – Data Class[29]

Module E: Dispensable #4 – Data Class (Java Example – Before)

```java
// SMELL: Data Class -- just data, no behavior
public class Employee {
    private String name;
    private String department;
    private double salary;

    public String getName() { return name; }
    public void setName(String name) { this.name = name; }
    public String getDepartment() { return department; }
    public void setDepartment(String dept) { this.department = dept; }
    public double getSalary() { return salary; }
    public void setSalary(double salary) { this.salary = salary; }
    // No behavior! Other classes do everything with this data.
}

// Behavior lives in client code
public class PayrollSystem {
    public double calculateRaise(Employee emp) {
        return emp.getSalary() * 0.10; // Feature Envy!
    }
    public double calculateBonus(Employee emp) {
        return emp.getSalary() * 0.05; // Feature Envy!
```

[29] https://refactoring.guru/smells/data-class

```
    }
    public boolean isHighEarner(Employee emp) {
        return emp.getSalary() > 100000; // Feature Envy!
    }
}
```

Module E: Dispensable #4 – Data Class (Java Example – After)

```
// REFACTORED: Move Method -- behavior moved into the data class
public class Employee {
    private String name;
    private String department;
    private double salary;

    public Employee(String name, String department, double salary) {
        this.name = name;
        this.department = department;
        this.salary = salary;
    }

    // Behavior now lives WITH the data
    public double calculateRaise() {
        return salary * 0.10;
    }

    public double calculateBonus() {
        return salary * 0.05;
    }

    public boolean isHighEarner() {
        return salary > 100000;
    }

    // Remove unnecessary setter -- salary should not be changed arbitrarily
    public String getName() { return name; }
    public String getDepartment() { return department; }
    public double getSalary() { return salary; }
    // No setSalary() -- use calculateRaise() and apply it instead
}
```

Module E: Dispensable #4 – Data Class (Treatment & Payoff)

Treatment (Refactoring Techniques)

Step	Technique
Step 1: Encapsulate	**Encapsulate Field** – make public fields private, add getters/setters. **Encapsulate Collection** – return unmodifiable views.
Step 2: Move behavior	**Move Method** / **Extract Method** – find code in client classes that manipulates this class's data and move it into the data class

Step 3: Remove access	**Remove Setting Method** – for fields that should not be changed after creation. **Hide Method** – for getters that only the class itself needs.

Payoff

- Improves **understanding and organization** of code – behavior and data are together
- Helps discover **duplicated code** in client classes that operated on the same data
- Classes become **self-contained** and capable of performing their own operations

When to Ignore

- Data classes are acceptable as **DTOs (Data Transfer Objects)** when they serve purely as transport containers between layers
- In **functional programming** style, separating data from behavior may be intentional

---

Module E: Dispensable #5 – Dead Code

What Is It?

A variable, parameter, field, method, or class is **no longer used** (usually because it is obsolete).

Signs and Symptoms

- Unused variables, parameters, or fields
- Methods that are never called
- Classes that are never instantiated
- Code branches that can never be reached (e.g., `if (false) { ... }`)
- Commented-out code left behind "just in case"

Reasons for the Problem

- Requirements changed and old code was **never cleaned up**
- Code was **commented out** "temporarily" and never removed
- Dead code often accumulates after **library upgrades** or **feature removals**
- Fear of removing code that "might be needed later"
- Complex conditionals where certain branches became inaccessible

Reference: RefactoringGuru – Dead Code[30]

Module E: Dispensable #5 – Dead Code (Java Example – Before)

```java
// SMELL: Dead Code -- various forms of unused code
public class UserService {
    // Dead field -- never read anywhere
    private int legacyUserCount;

    // Dead method -- never called (migration happened 3 years ago)
    public void migrateOldAccounts() {
        // This migration was completed in 2022
    }

    // Dead parameter -- 'includeHistory' is completely ignored
    public User getUser(int userId, boolean includeHistory) {
```

[30] https://refactoring.guru/smells/dead-code

```
        return userRepository.findById(userId);
    }

    // Dead code branch -- condition is always false
    public void processUser(User user) {
        if (false) {
            // Someone left this here "just in case"
            sendLegacyNotification(user);
        }
        // Normal processing
        processNormally(user);
    }

    // Commented-out code
    // public void oldProcessUser(User user) {
    //     // old implementation
    // }
}
```

Module E: Dispensable #5 – Dead Code (Java Example – After)

```
// REFACTORED: Delete unused code -- Git remembers everything
public class UserService {
    // legacyUserCount -- DELETED
    // migrateOldAccounts() -- DELETED
    // commented-out code -- DELETED

    // Dead parameter removed
    public User getUser(int userId) {
        return userRepository.findById(userId);
    }

    // Dead code branch removed
    public void processUser(User user) {
        processNormally(user);
    }
}

// If you ever need migrateOldAccounts() again,
// check Git history:
//   git log --all --full-history -- "**/UserService.java"
//   git show <commit-hash>:path/to/UserService.java
```

Module E: Dispensable #5 – Dead Code (Treatment & Payoff)

Treatment (Refactoring Techniques)

Technique	When to Use
Delete unused code	The simplest and most effective treatment – just delete it
Remove Parameter	Eliminate unused parameters from method signatures
Inline Class or **Collapse Hierarchy**	For unnecessary classes

Technique	When to Use
Use IDE tools	Modern IDEs and static analysis tools can identify unused code efficiently

Payoff

- **Reduced code size** – less code to read, understand, and maintain
- **Simpler support** – no confusion about whether dead code should be maintained
- **Cleaner codebase** – easier to navigate and search

Key Insight

> Remember: **Git remembers everything** – you can always get deleted code back from version history. There is no reason to keep dead code "just in case."

Module E: Dispensable #6 – Speculative Generality

What Is It?

There is an **unused class, method, field, or parameter** that was created to support anticipated future features that **never got implemented**.

Signs and Symptoms

- Abstract classes with only one concrete implementation
- Methods with parameters that are never used by any caller
- Classes that exist "just in case we need them someday"
- Overly complex design patterns applied to simple problems

Reasons for the Problem

- "We'll need this someday" – but someday never comes
- Overly enthusiastic application of **design patterns** and **abstraction**
- Created during a design phase that envisioned features that were later **cancelled**
- Desire to create a "general-purpose" framework when a specific solution sufficed

Reference: RefactoringGuru – Speculative Generality[31]

Module E: Dispensable #6 – Speculative Generality (Java Example – Before)

```java
// SMELL: Speculative Generality -- over-engineered abstraction
public abstract class AbstractNotificationService {
    abstract void send(String message);
    abstract void schedule(String message, Date date);   // Never used
    abstract void retract(String messageId);             // Never used
    abstract void broadcast(String message, String group); // Never used
}

// Only ONE concrete implementation -- why is the parent abstract?
public class EmailNotificationService extends AbstractNotificationService {
    void send(String message) { /* actual implementation */ }
    void schedule(String message, Date date) { /* not implemented */ }
    void retract(String messageId) { /* not implemented */ }
```

[31] https://refactoring.guru/smells/speculative-generality

```
    void broadcast(String message, String group) { /* not implemented */ }
}

// Unused Shape methods
public abstract class Shape {
    abstract double area();
    abstract double perimeter();
    abstract void rotate(double degrees);       // Never called
    abstract void scale(double factor);         // Never called
    abstract Shape clone3D();                   // "We might need 3D someday"
}
```

Module E: Dispensable #6 – Speculative Generality (Java Example – After)

```
// REFACTORED: Collapse Hierarchy + Remove unused methods
// No abstract class needed -- only one implementation exists
public class EmailNotificationService {
    public void send(String message) {
        // actual email sending implementation
    }
    // schedule(), retract(), broadcast() -- DELETED (YAGNI)
}

// Shape with only the methods that are actually used
public abstract class Shape {
    abstract double area();
    abstract double perimeter();
    // rotate(), scale(), clone3D() -- DELETED
    // If we need them later, we can add them then
}

public class Circle extends Shape {
    private double radius;
    double area() { return Math.PI * radius * radius; }
    double perimeter() { return 2 * Math.PI * radius; }
}
```

Module E: Dispensable #6 – Speculative Generality (Treatment & Payoff)

Treatment (Refactoring Techniques)

Technique	When to Use
Collapse Hierarchy	Remove unnecessary abstract classes with only one subclass
Inline Class	Eliminate superfluous delegation between classes
Inline Method	Delete unused methods
Remove Parameter	Clean up methods with unneeded parameters
Direct deletion	Simply delete unused fields

Payoff

- **Slimmer code** – less to read, understand, and maintain
- **Cleaner architecture** – no phantom abstractions cluttering the design

- Follows the **YAGNI** principle (You Aren't Gonna Need It)

When to Ignore

- If you are working on a **framework**, it is reasonable to create functionality for users of the framework
- If a test uses an abstract class or interface for mocking purposes
- If the feature is in active planning and will be implemented in the next sprint

---

Module E: Takeaway

Key Points – Dispensables

Smell	Key Takeaway
Comments	Make code self-documenting; comments should explain "why," not "what"
Duplicate Code	Extract common code; every change should only need to be made once
Lazy Class	If a class does not earn its keep, inline it
Data Class	Move behavior into the class alongside its data
Dead Code	Delete it – Git remembers everything
Speculative Generality	YAGNI – do not build for imagined future requirements

Common Thread

Dispensables are about **removing unnecessary complexity**. Less code means fewer bugs, easier maintenance, and faster comprehension. The simplest code that works is usually the best code.

> "Perfection is achieved, not when there is nothing more to add, but when there is nothing left to take away." – Antoine de Saint-Exupery

---

Module F: Couplers

Smells Contributing to Excessive Coupling Between Classes

Module F: Couplers – Overview

All the smells in this group contribute to **excessive coupling** between classes or show what happens if coupling is replaced by **excessive delegation**.

The 4 Coupler Smells

#	Smell	Key Issue
1	**Feature Envy**	A method accesses data of another object more than its own
2	**Inappropriate Intimacy**	One class uses internal fields/methods of another class
3	**Message Chains**	`a.getB().getC().getD().doSomething()`
4	**Middle Man**	A class only delegates to another class

These smells represent different ways that excessive coupling manifests in code, either through direct data access or through chains of delegation that create brittle dependencies.

Reference: RefactoringGuru – Couplers[32]

Module F: Coupler #1 – Feature Envy

What Is It?

A method accesses the **data of another object** more than its own data. The method is "envious" of another class.

Signs and Symptoms

- A method that calls many getters of another object
- The method uses more fields/methods from another class than from its own
- The method seems like it belongs in another class

Reasons for the Problem

- This smell frequently emerges when fields get relocated to a **data class**
- If this is the case, you may want to move the operations on data to that class as well
- A method was placed in the **wrong class** during development
- It operates more on another class's data than its own

Reference: RefactoringGuru – Feature Envy[33]

Feature Envy - Before (Code Smell)

Feature Envy - After (Refactored)

Module F: Coupler #1 – Feature Envy (Java Example – Before)

```java
// SMELL: Feature Envy -- getCustomerReport() is envious of Customer's data
public class Order {
    private Customer customer;
    private List<Item> items;

    // This method accesses Customer's data far more than Order's data
    public String getCustomerReport() {
        String report = customer.getName() + "\n";
        report += customer.getAddress().getStreet() + "\n";
        report += customer.getAddress().getCity() + ", ";
        report += customer.getAddress().getState() + " ";
        report += customer.getAddress().getZip() + "\n";
        report += "Member since: " + customer.getMemberSince() + "\n";
        report += "Loyalty tier: " + customer.getLoyaltyTier() + "\n";
        report += "Total orders: " + customer.getOrderCount() + "\n";
        return report;
    }
}
```

[32] https://refactoring.guru/refactoring/smells/couplers
[33] https://refactoring.guru/smells/feature-envy

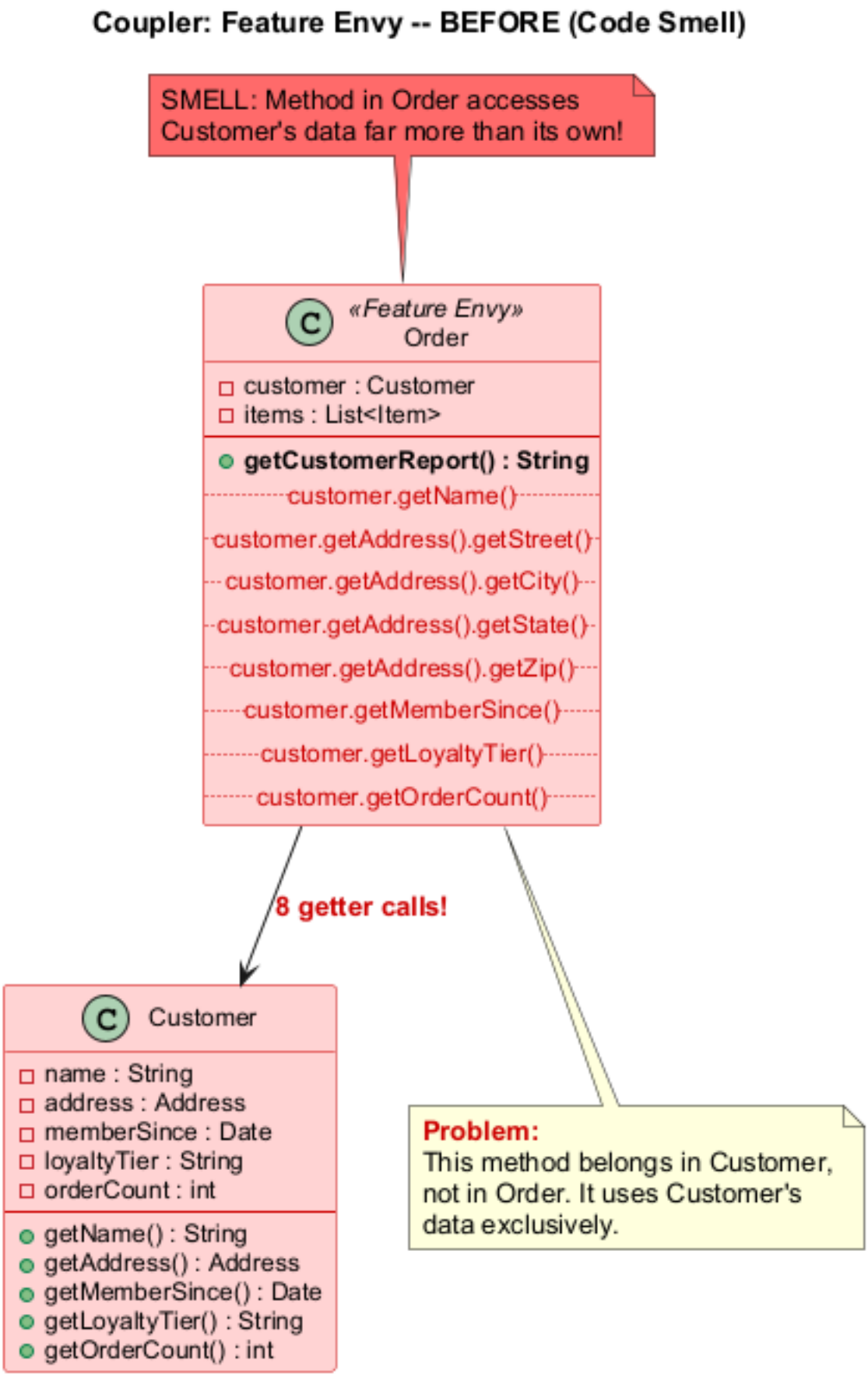


Figure 12: center

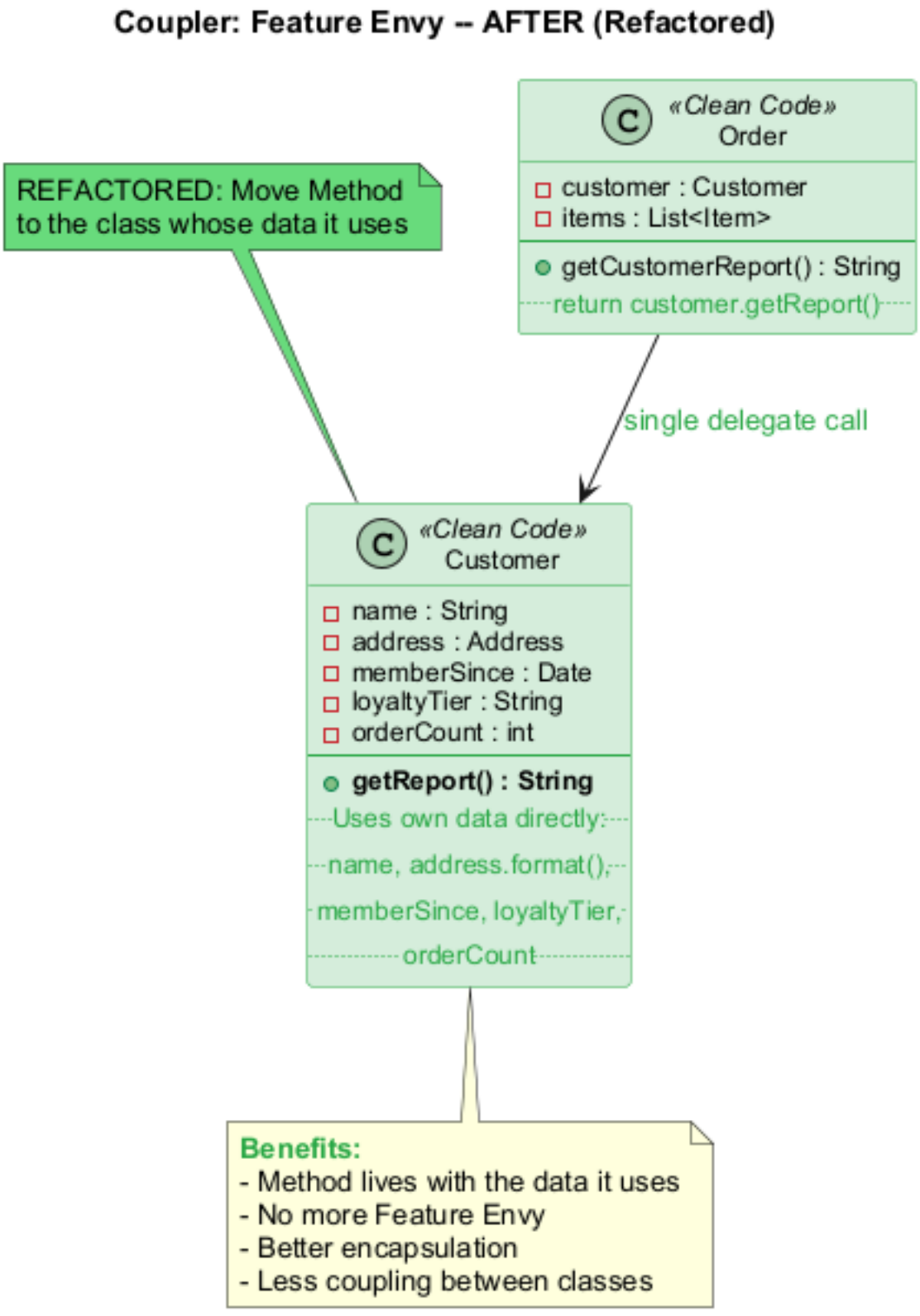


Figure 13: center

Module F: Coupler #1 – Feature Envy (Java Example – After)

```java
// REFACTORED: Move Method -- move the report generation to Customer
public class Customer {
    private String name;
    private Address address;
    private Date memberSince;
    private String loyaltyTier;
    private int orderCount;

    // Method now lives with the data it uses
    public String getReport() {
        String report = name + "\n";
        report += address.format() + "\n";
        report += "Member since: " + memberSince + "\n";
        report += "Loyalty tier: " + loyaltyTier + "\n";
        report += "Total orders: " + orderCount + "\n";
        return report;
    }
}

public class Order {
    private Customer customer;

    // Delegate to Customer -- no more Feature Envy
    public String getCustomerReport() {
        return customer.getReport();
    }
}
```

Module F: Coupler #1 – Feature Envy (Treatment & Payoff)

Treatment (Refactoring Techniques)

Technique	When to Use
Move Method	Move the method to the class whose data it uses most
Extract Method	If only a portion of the method envies another object, extract that part and move it

Payoff

- Less **code duplication** (if the envious logic was duplicated elsewhere)
- Better **code organization** – data and its behavior are in the same place

When to Ignore

- When Feature Envy is used **purposefully** in behavioral design patterns:
  - **Strategy Pattern**: Behavior is intentionally separated from the data class
  - **Visitor Pattern**: Operations are separated from the object structure
  - **Command Pattern**: Commands operate on data from other objects

  If a method clearly belongs to a design pattern that deliberately separates data from behavior, Feature Envy is acceptable.

Module F: Coupler #2 – Inappropriate Intimacy

What Is It?

One class uses the **internal fields and methods** of another class. Good classes should know as little about each other as possible – this makes classes easier to maintain and reuse.

Signs and Symptoms

- A class directly accesses another class's fields (especially non-private ones)
- A class depends on the internal implementation details of another class
- Bidirectional dependencies where two classes reference each other's internals
- Changing one class frequently breaks the other

Reasons for the Problem

- Classes that spend too much time together delving into each other's private parts
- Often results from **incremental development** where boundaries were not carefully maintained
- Tight coupling built up over time through convenience-driven coding

Reference: RefactoringGuru – Inappropriate Intimacy[34]

Module F: Coupler #2 – Inappropriate Intimacy (Java Example – Before)

```java
// SMELL: Inappropriate Intimacy -- Order accesses Customer's internals
public class Customer {
    int loyaltyPoints;              // Should be private!
    List<Order> orderHistory;       // Exposed to Order!
    String membershipTier;          // Directly accessed!
}

public class Order {
    private Customer customer;

    // Directly accessing and modifying Customer's internal fields!
    public void applyDiscount() {
        if (customer.loyaltyPoints > 1000) {  // Accessing internal field
            customer.loyaltyPoints -= 100;     // Modifying internal field!
            this.discount = 0.10;
        }
    }

    public void addToHistory() {
        customer.orderHistory.add(this);       // Directly modifying collection!
    }

    public boolean isVipOrder() {
        return customer.membershipTier.equals("GOLD")  // Internal details!
            && customer.orderHistory.size() > 50;
    }
}
```

[34] https://refactoring.guru/smells/inappropriate-intimacy

Module F: Coupler #2 – Inappropriate Intimacy (Java Example – After)

```java
// REFACTORED: Proper encapsulation -- Move Method + Hide fields
public class Customer {
    private int loyaltyPoints;
    private List<Order> orderHistory;
    private String membershipTier;

    public boolean canRedeemDiscount() {
        return loyaltyPoints > 1000;
    }

    public void redeemDiscount() {
        if (!canRedeemDiscount()) throw new IllegalStateException();
        loyaltyPoints -= 100;
    }

    public void addOrder(Order order) {
        orderHistory.add(order);
    }

    public boolean isVip() {
        return "GOLD".equals(membershipTier) && orderHistory.size() > 50;
    }
}

public class Order {
    private Customer customer;

    public void applyDiscount() {
        if (customer.canRedeemDiscount()) {
            customer.redeemDiscount();
            this.discount = 0.10;
        }
    }
}
```

Module F: Coupler #2 – Inappropriate Intimacy (Treatment & Payoff)

Treatment (Refactoring Techniques)

Technique	When to Use
Move Method / Move Field	Move parts of one class to the class they are most related to
Extract Class and **Hide Delegate**	Formalize the relationship by extracting shared functionality into a separate class
Change Bidirectional Association to Unidirectional	When two classes depend on each other mutually, convert to one-way dependency
Replace Delegation with Inheritance	When one class really is a specialized version of another

Payoff

- Enhanced **code organization** and structure
- Simplified **maintenance** and improved code reusability

- Better **encapsulation** – classes expose only what they need to
- Adheres to the **Law of Demeter** (principle of least knowledge)

  "Each unit should have only limited knowledge about other units: only units closely related to the current unit." – Law of Demeter

---

Module F: Coupler #3 – Message Chains

What Is It?

In code you see a series of calls resembling `a.getB().getC().getD()`. These chains mean that the client is **coupled to the navigation structure** of the class hierarchy.

Signs and Symptoms

- Long chains of method calls traversing an object graph
- The client needs to know the internal structure of multiple classes
- Any change to the intermediate relationships requires modifying the client

Reasons for the Problem

- Message chains are created when a client needs to navigate an **object graph** to find the information it needs
- The client requests an object, which requests another object, which requests yet another one
- This creates tight coupling to the entire navigation structure

Reference: RefactoringGuru – Message Chains[35]

Message Chains - Before (Code Smell)

Message Chains - After (Refactored)

Module F: Coupler #3 – Message Chains (Java Example – Before)

```java
// SMELL: Message Chains -- tight coupling to navigation structure
public class InvoiceGenerator {

    public String getCustomerCity(Order order) {
        // Chain: Order -> Customer -> Address -> City -> Name
        String cityName = order
            .getCustomer()
            .getAddress()
            .getCity()
            .getName();
        return cityName;
    }

    public double getManagerBalance(Company company) {
        // Chain: Company -> Department -> Manager -> Account -> Balance
        double balance = company
            .getDepartment("Sales")
```

[35] https://refactoring.guru/smells/message-chains

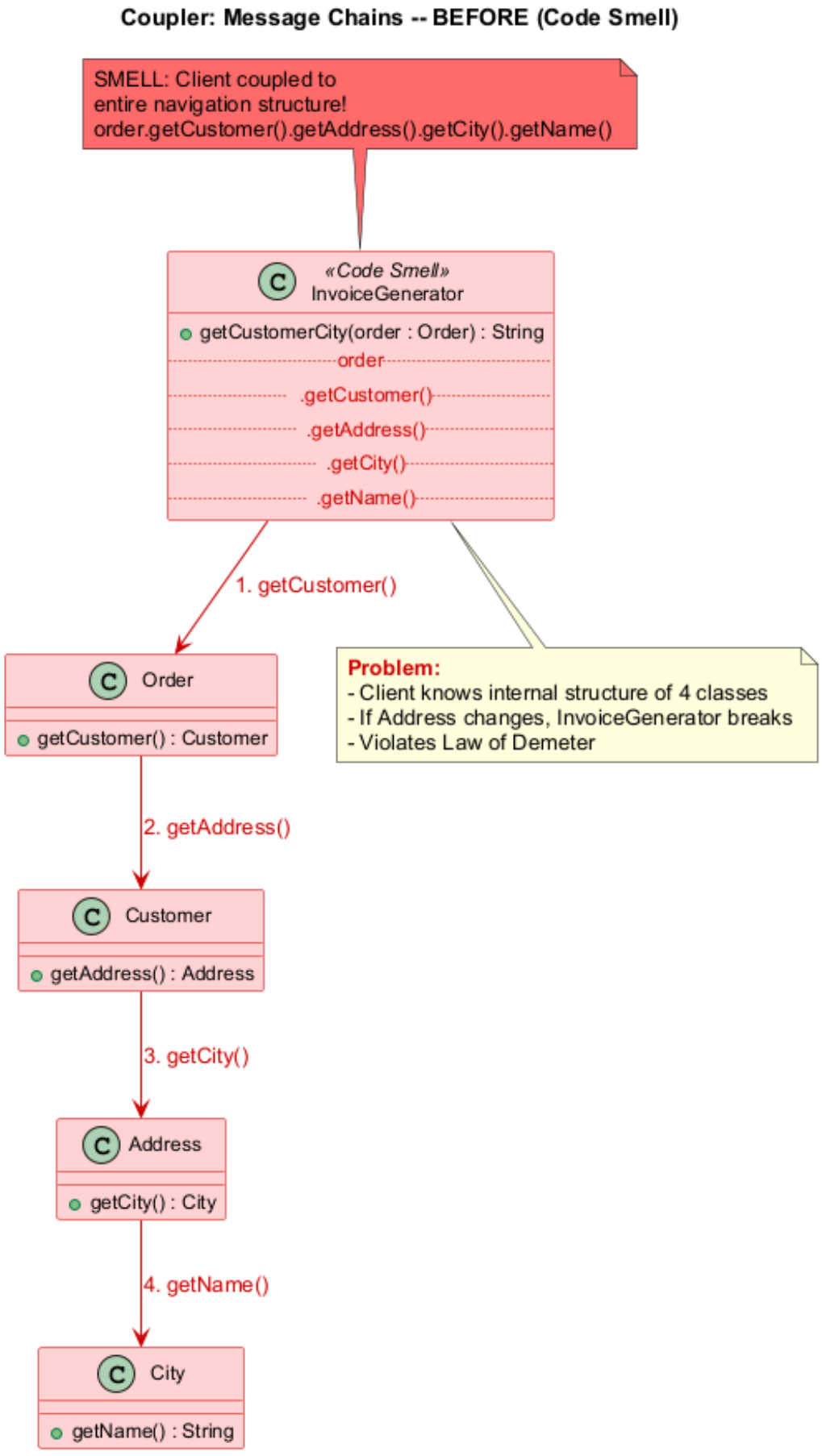


Figure 14: center

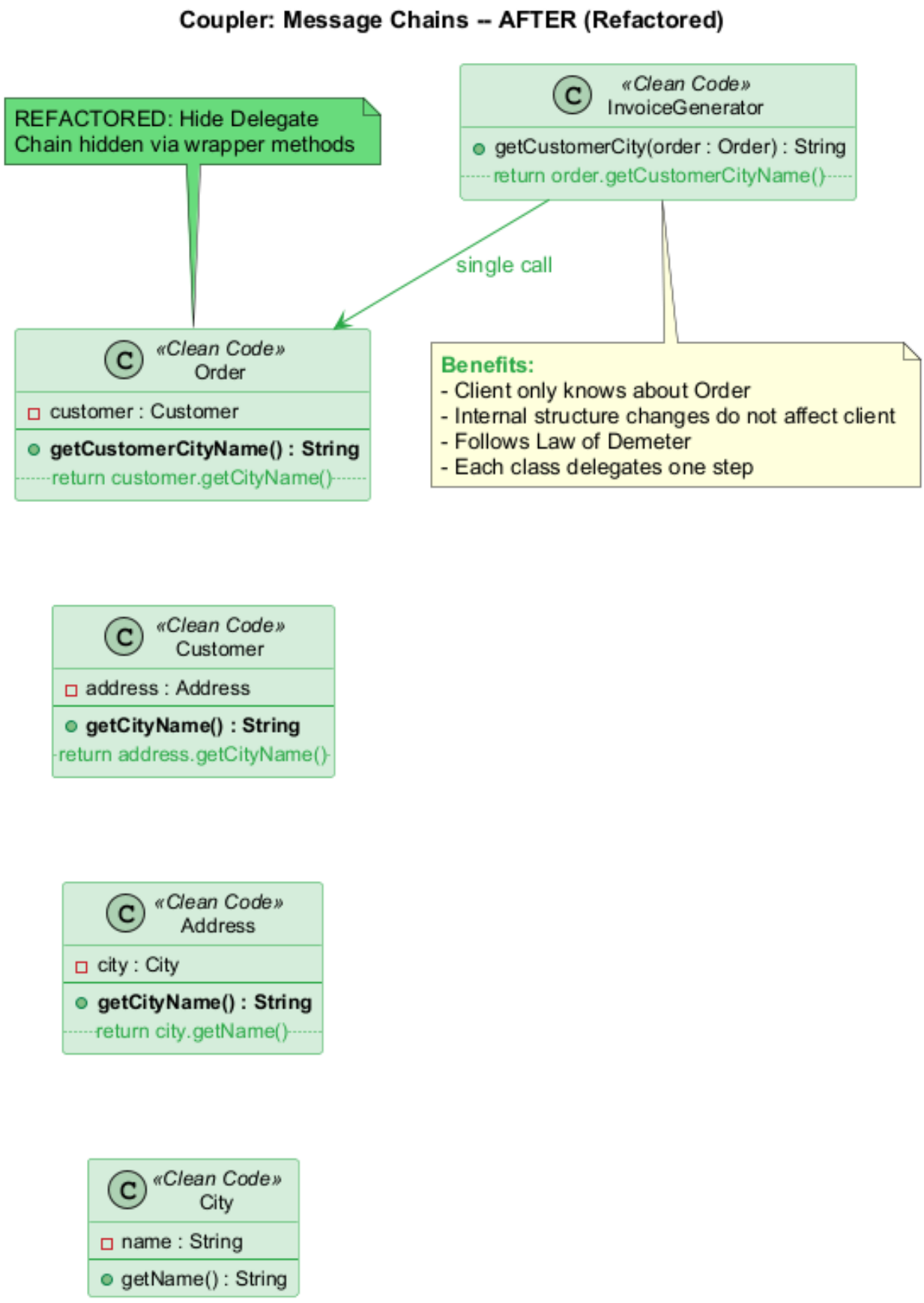


Figure 15: center

```
            .getManager()
            .getAccount()
            .getBalance();
        return balance;
    }
}
// If Address structure changes, InvoiceGenerator breaks!
// If Department-Manager relationship changes, InvoiceGenerator breaks!
```

Module F: Coupler #3 – Message Chains (Java Example – After)

```
// REFACTORED: Hide Delegate -- create wrapper methods to break the chain
public class Order {
    private Customer customer;

    // Hide delegate: Order provides customer city directly
    public String getCustomerCityName() {
        return customer.getCityName();
    }
}

public class Customer {
    private Address address;

    public String getCityName() {
        return address.getCityName();
    }
}

public class Address {
    private City city;

    public String getCityName() {
        return city.getName();
    }
}

// Client code -- clean, no chain!
public class InvoiceGenerator {
    public String getCustomerCity(Order order) {
        return order.getCustomerCityName();
    }
}
```

Module F: Coupler #3 – Message Chains (Treatment & Payoff)

Treatment (Refactoring Techniques)

Technique	When to Use
Hide Delegate	Create wrapper methods that encapsulate the chain at each level
Extract Method	Extract the chain into a well-named method

Move Method	Move the method that uses the chain to the class that has the data

Payoff

- Reduces **dependencies** between classes in a chain
- Reduces the amount of **bloated code** in the client
- The client is decoupled from the internal navigation structure
- Changes to intermediate classes do not ripple to clients

When to Ignore

- **Overly aggressive delegate hiding** can result in the **Middle Man** smell
- If the chain reflects a **stable, well-known structure** (e.g., `request.getSession().getAttribute()`), it may be acceptable
- Find the right balance: neither too many chains nor too many delegate methods

```
Message Chain  <------- Balance ------->  Middle Man
(too much coupling)                    (too much delegation)
```

Module F: Coupler #4 – Middle Man

What Is It?

If a class performs **only one action** – delegating work to another class – why does it exist at all?

Signs and Symptoms

- A class where the majority of methods simply forward calls to another class
- The class adds no logic, no validation, no transformation
- Clients could just as easily call the delegate directly

Reasons for the Problem

- **Overzealous elimination of Message Chains** – developers may have aggressively refactored to break up method chains, inadvertently creating unnecessary intermediary classes
- **Gradual erosion of responsibility** – a class's useful functionality gets progressively moved elsewhere, leaving behind an empty shell that merely delegates

Reference: RefactoringGuru – Middle Man[36]

Module F: Coupler #4 – Middle Man (Java Example – Before)

```java
// SMELL: Middle Man -- Department just delegates everything to Manager
public class Department {
    private Manager manager;

    public String getManagerName() {
        return manager.getName();       // Just delegates
    }

    public String getManagerEmail() {
        return manager.getEmail();      // Just delegates
    }
```

[36] https://refactoring.guru/smells/middle-man

```
    public void approveExpense(Expense expense) {
        manager.approveExpense(expense); // Just delegates
    }

    public Report getReport() {
        return manager.getReport();     // Just delegates
    }

    public void scheduleMeeting(Date date) {
        manager.scheduleMeeting(date);  // Just delegates
    }
    // Every single method just forwards to manager!
    // Department adds no value at all.
}
```

Module F: Coupler #4 – Middle Man (Java Example – After)

```
// REFACTORED: Remove Middle Man -- client accesses Manager directly
public class Department {
    private Manager manager;

    // Expose the manager so clients can interact directly
    public Manager getManager() {
        return manager;
    }

    // Keep methods that ADD value (not just delegate)
    public double getTotalBudget() {
        return manager.getTeamBudget() + operatingCosts;
    }
}

// Client code -- direct interaction
public class ExpenseProcessor {
    public void process(Department dept, Expense expense) {
        // Before: dept.approveExpense(expense);  // Middle Man
        // After: direct access to where the work actually happens
        dept.getManager().approveExpense(expense);
    }
}
```

Module F: Coupler #4 – Middle Man (Treatment & Payoff)

Treatment (Refactoring Techniques)

Technique	When to Use
Remove Middle Man	Have the client call the end object directly when most methods are pure delegation
Inline Method	For trivial delegation methods
Replace Delegation with Inheritance	If the middle man always delegates to the same class, consider making it a subclass

Payoff

- **Less bloated code** – no unnecessary intermediary classes
- **Clearer code** – you can see where work actually happens
- Simpler class hierarchy

When to Ignore

- A Middle Man may have been added to **avoid inter-class dependencies** intentionally
- Middle Man can be a deliberate part of the **Facade**, **Proxy**, or **Decorator** patterns
- If removing it would expose too many internals of the delegate class, keep it
- Do not remove Middle Man if doing so would create **Message Chains** – find the right balance

---

Module F: Takeaway

Key Points – Couplers

Smell	Key Takeaway
Feature Envy	Move the method to the class whose data it uses most
Inappropriate Intimacy	Enforce encapsulation; classes should know as little about each other as possible
Message Chains	Hide delegates to reduce client coupling to navigation structure
Middle Man	If a class only delegates, let clients call the delegate directly

The Coupling Spectrum

```
Tight Coupling <-------- Goal --------> Loose Coupling
(Feature Envy,          (Right balance)   (Middle Man,
 Inappropriate                             Excessive
 Intimacy)                                 Delegation)
```

> The goal is **just enough coupling** – not so much that changes ripple everywhere, and not so little that code becomes a maze of trivial delegations.

Summary: All 22 Code Smells at a Glance

Category	Smells	Core Problem
Bloaters	Long Method, Large Class, Primitive Obsession, Long Parameter List, Data Clumps	Code that has grown too large
OO Abusers	Switch Statements, Temporary Field, Refused Bequest, Alternative Classes with Different Interfaces	Incorrect application of OOP
Change Preventers	Divergent Change, Shotgun Surgery, Parallel Inheritance Hierarchies	Changes require disproportionate effort
Dispensables	Comments, Duplicate Code, Lazy Class, Data Class, Dead Code, Speculative Generality	Unnecessary complexity

Category	Smells	Core Problem
Couplers	Feature Envy, Inappropriate Intimacy, Message Chains, Middle Man	Excessive coupling or delegation

---

Summary: Key Refactoring Principles

Remember

- Code smells are **indicators**, not absolute rules – context matters
- Every smell has a "When to Ignore" case
- Refactoring is about making **small, safe changes** continuously
- The best time to refactor is **now** – technical debt only grows
- Never mix refactoring and feature development in the same commit
- All existing tests must pass after refactoring

The Refactoring Mindset

```
Detect Smell --> Choose Technique --> Apply Small Change --> Run Tests
     ^                                                          |
     |                                                          v
     +--------------------- Repeat if needed -------------------+
```

> "The key to refactoring is working in small steps. You make a small change, test, make another small change, test." – Martin Fowler

---

References

- Fowler, M. (2018). *Refactoring: Improving the Design of Existing Code* (2nd Edition). Addison-Wesley.
- Beck, K., & Fowler, M. (1999). *Bad Smells in Code.* In Refactoring: Improving the Design of Existing Code.
- Martin, R. C. (2008). *Clean Code: A Handbook of Agile Software Craftsmanship.* Prentice Hall.
- RefactoringGuru. *What is Refactoring?* https://refactoring.guru/refactoring/what-is-refactoring
- RefactoringGuru. *Clean Code.* https://refactoring.guru/refactoring/clean-code
- RefactoringGuru. *Technical Debt.* https://refactoring.guru/refactoring/technical-debt
- RefactoringGuru. *When to Refactor.* https://refactoring.guru/refactoring/when
- RefactoringGuru. *How to Refactor.* https://refactoring.guru/refactoring/how-to
- RefactoringGuru. *Code Smells.* https://refactoring.guru/refactoring/smells

---

References (Continued)

- RefactoringGuru. *Bloaters.* https://refactoring.guru/refactoring/smells/bloaters
- RefactoringGuru. *Object-Orientation Abusers.* https://refactoring.guru/refactoring/smells/oo-abusers
- RefactoringGuru. *Change Preventers.* https://refactoring.guru/refactoring/smells/change-preventers
- RefactoringGuru. *Dispensables.* https://refactoring.guru/refactoring/smells/dispensables
- RefactoringGuru. *Couplers.* https://refactoring.guru/refactoring/smells/couplers
- RefactoringGuru. *Long Method.* https://refactoring.guru/smells/long-method
- RefactoringGuru. *Large Class.* https://refactoring.guru/smells/large-class
- RefactoringGuru. *Primitive Obsession.* https://refactoring.guru/smells/primitive-obsession
- RefactoringGuru. *Long Parameter List.* https://refactoring.guru/smells/long-parameter-list
- RefactoringGuru. *Data Clumps.* https://refactoring.guru/smells/data-clumps

---

References (Continued)

- RefactoringGuru. *Switch Statements.* https://refactoring.guru/smells/switch-statements
- RefactoringGuru. *Temporary Field.* https://refactoring.guru/smells/temporary-field
- RefactoringGuru. *Refused Bequest.* https://refactoring.guru/smells/refused-bequest
- RefactoringGuru. *Alternative Classes with Different Interfaces.* https://refactoring.guru/smells/alternative-classes-with-different-interfaces
- RefactoringGuru. *Divergent Change.* https://refactoring.guru/smells/divergent-change
- RefactoringGuru. *Shotgun Surgery.* https://refactoring.guru/smells/shotgun-surgery
- RefactoringGuru. *Parallel Inheritance Hierarchies.* https://refactoring.guru/smells/parallel-inheritance-hierarchies
- RefactoringGuru. *Comments.* https://refactoring.guru/smells/comments
- RefactoringGuru. *Duplicate Code.* https://refactoring.guru/smells/duplicate-code
- RefactoringGuru. *Lazy Class.* https://refactoring.guru/smells/lazy-class

---

References (Continued)

- RefactoringGuru. *Data Class.* https://refactoring.guru/smells/data-class
- RefactoringGuru. *Dead Code.* https://refactoring.guru/smells/dead-code
- RefactoringGuru. *Speculative Generality.* https://refactoring.guru/smells/speculative-generality
- RefactoringGuru. *Feature Envy.* https://refactoring.guru/smells/feature-envy
- RefactoringGuru. *Inappropriate Intimacy.* https://refactoring.guru/smells/inappropriate-intimacy
- RefactoringGuru. *Message Chains.* https://refactoring.guru/smells/message-chains
- RefactoringGuru. *Middle Man.* https://refactoring.guru/smells/middle-man

---

$End - Of - Week - 12 - Module$