CEN206 Object-Oriented Programming

Week-13 (Refactoring Techniques)

Spring Semester, 2025-2026

Download DOC-PDF, DOC-DOCX, SLIDE

Week-13 Overview

Refactoring Techniques (All 66 Techniques)

Module	Topic	Techniques
A	Composing Methods	9 techniques
B	Moving Features between Objects	8 techniques
C	Organizing Data	15 techniques
D	Simplifying Conditional Expressions	8 techniques
E	Simplifying Method Calls	14 techniques
F	Dealing with Generalization	12 techniques

What is Refactoring?

- **Refactoring** is a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior.
- Each transformation (called a "refactoring") does little, but a sequence of transformations can produce a significant restructuring.
- The goal is to make code **easier to understand**, **cheaper to modify**, and **safer to extend**.

Why Refactor?

- **Improves software design** -- Without refactoring, the design of a program will decay over time.
- **Makes software easier to understand** -- Code that communicates its purpose clearly is easier to maintain.
- **Helps find bugs** -- Clarifying code structure makes bugs more visible.
- **Helps you program faster** -- Good design enables rapid development of new features.

When to Refactor?

- **Rule of Three**: The first time you do something, just do it. The second time, you wince at duplication but do it anyway. The third time, you refactor.
- **When adding a feature**: Refactor to make the new feature easier to add.
- **When fixing a bug**: Refactor to make the bug obvious.
- **During code review**: Refactor to improve readability and maintainability.

Module A: Composing Methods

9 Techniques for Better Method Structure

Module A Outline

Composing Methods

These refactorings deal with how methods are composed. Much of refactoring is about composing methods to properly package code. Almost all the time, the root problem is **methods that are too long**.

1. Extract Method
2. Inline Method
3. Extract Variable
4. Inline Temp
5. Replace Temp with Query
6. Split Temporary Variable
7. Remove Assignments to Parameters
8. Replace Method with Method Object
9. Substitute Algorithm

A1. Extract Method

Problem: You have a code fragment that can be grouped together.

Solution: Turn the fragment into a method whose name explains the purpose of the method.

Long methods are the root of all evil. The longer a method is, the harder it is to understand. Extract Method is one of the most common refactorings. If you feel the need to comment something inside a method, take the code and put it in a new method named after the intention, not the implementation.

A1. Extract Method -- Before

```java
public class Order {
    private String name;
    private List<LineItem> items;

    public void printOwing() {
        double outstanding = 0.0;

        // print banner
        System.out.println("**************************");
        System.out.println("***** Customer Owes ******");
        System.out.println("**************************");

        // calculate outstanding
        for (LineItem item : items) {
            outstanding += item.getAmount();
        }

        // print details
        System.out.println("name: " + name);
        System.out.println("amount: " + outstanding);
    }
}
```

A1. Extract Method -- After

```
public class Order {
    private String name;
    private List<LineItem> items;

    public void printOwing() {
        printBanner();
        double outstanding = calculateOutstanding();
        printDetails(outstanding);
    }

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

    private double calculateOutstanding() {
        double outstanding = 0.0;
        for (LineItem item : items) {
            outstanding += item.getAmount();
        }
        return outstanding;
    }

    private void printDetails(double outstanding) {
        System.out.println("name: " + name);
        System.out.println("amount: " + outstanding);
    }
}
```

A2. Inline Method

Problem: When a method body is just as clear as the method name, or the method body is trivially simple.

Solution: Put the method's body into the body of its callers and remove the method.

Use this refactoring when the delegation is needless. If the method body is as obvious as the method name, there is no value in the indirection. Also useful when you have a group of methods that seem badly factored -- inline them all into one big method and then re-extract.

A3. Extract Variable

Problem: You have an expression that is hard to understand.

Solution: Place the result of the expression or its parts in separate variables that are self-explanatory.

Complicated expressions are hard to understand. Variables with meaningful names can make the code self-documenting.

A3. Extract Variable -- Before

```java
public class Product {
    private int quantity;
    private double itemPrice;

    public double getPrice() {
        // price is base price - quantity discount + shipping
        return quantity * itemPrice
            - Math.max(0, quantity - 500) * itemPrice * 0.05
            + Math.min(quantity * itemPrice * 0.1, 100.0);
    }
}
```

A3. Extract Variable -- After

```java
public class Product {
    private int quantity;
    private double itemPrice;

    public double getPrice() {
        double basePrice = quantity * itemPrice;
        double quantityDiscount = Math.max(0, quantity - 500)
                                  * itemPrice * 0.05;
        double shipping = Math.min(basePrice * 0.1, 100.0);
        return basePrice - quantityDiscount + shipping;
    }
}
```

Each sub-expression now has a descriptive name that communicates its purpose. The code is self-documenting.

A4. Inline Temp

Problem: You have a temporary variable that is assigned to once with a simple expression, and the temp is getting in the way of other refactorings.

Solution: Replace all references to that temp with the expression.

```
// Before
double basePrice = order.basePrice();
return (basePrice > 1000);

// After
return (order.basePrice() > 1000);
```

Inline Temp is most often used as part of Replace Temp with Query, so it may not seem to have value on its own. But it is useful when the temp is blocking another refactoring.

A5. Replace Temp with Query

Problem: You are using a temporary variable to hold the result of an expression.

Solution: Extract the expression into a method. Replace all references to the temp with the method call. The new method can then be used in other methods.

```
// Before
double basePrice = quantity * itemPrice;
if (basePrice > 1000) { return basePrice * 0.95; }
else { return basePrice * 0.98; }

// After
if (basePrice() > 1000) { return basePrice() * 0.95; }
else { return basePrice() * 0.98; }

double basePrice() { return quantity * itemPrice; }
```

A6. Split Temporary Variable

Problem: You have a local variable that is assigned more than once, but it is not a loop variable or a collecting variable.

Solution: Make a separate temporary variable for each assignment. Each temp should be assigned only once.

When a temp is used for two different things, it can be very confusing to the reader. Use a separate variable for each purpose.

```
// Before
double temp = 2 * (height + width);
System.out.println(temp);
temp = height * width;
System.out.println(temp);

// After
double perimeter = 2 * (height + width);
System.out.println(perimeter);
double area = height * width;
System.out.println(area);
```

A7. Remove Assignments to Parameters

Problem: A value is assigned to a parameter inside the method's body.

Solution: Use a local variable instead of a parameter.

Assigning to parameters is confusing. It blurs the line between input values and local working variables. It can also prevent certain optimizations and confuse readers about the method's intent.

```
// Before
int discount(int inputVal, int quantity) {
    if (quantity > 50) inputVal -= 2;
    // ...
    return inputVal;
}

// After
int discount(int inputVal, int quantity) {
    int result = inputVal;
    if (quantity > 50) result -= 2;
    // ...
    return result;
}
```

A8. Replace Method with Method Object

Problem: You have a long method in which local variables are so intertwined that you cannot apply Extract Method.

Solution: Transform the method into a separate class so that the local variables become fields of the class. Then you can split the method into several methods within the same class.

A8. Replace Method with Method Object -- Before

```
public class Order {
    public double price() {
        double primaryBasePrice;
        double secondaryBasePrice;
        double tertiaryBasePrice;
        // long computation using all three variables...
        primaryBasePrice = /* ... */;
        secondaryBasePrice = /* ... */;
        tertiaryBasePrice = /* ... */;
        return primaryBasePrice + secondaryBasePrice + tertiaryBasePrice;
    }
}
```

A8. Replace Method with Method Object -- After

```
public class PriceCalculator {
    private Order order;
    private double primaryBasePrice;
    private double secondaryBasePrice;
    private double tertiaryBasePrice;

    public PriceCalculator(Order order) {
        this.order = order;
    }

    public double compute() {
        primaryBasePrice = /* ... */;
        secondaryBasePrice = /* ... */;
        tertiaryBasePrice = /* ... */;
        return primaryBasePrice + secondaryBasePrice + tertiaryBasePrice;
    }
}

// In Order class:
public double price() {
    return new PriceCalculator(this).compute();
}
```

Now local variables become fields, and you can freely extract sub-methods within `PriceCalculator`.

A9. Substitute Algorithm

Problem: You want to replace an existing algorithm with one that is clearer or more efficient.

Solution: Replace the body of the method with the new algorithm.

Sometimes you find a simpler way to do something. When that happens, replace the complicated algorithm with the simpler one.

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

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

Module A -- Takeaway

Composing Methods: Key Points

Technique	When to Use
Extract Method	Long methods, code needing comments
Inline Method	Trivial delegation, preparing for re-extraction
Extract Variable	Complex expressions
Inline Temp	Temp blocking other refactorings
Replace Temp with Query	Temps used in multiple places

Golden rule: Keep methods short, focused, and named for their intention.

Module B: Moving Features between Objects

8 Techniques for Better Responsibility Distribution

Module B Outline

Moving Features between Objects

One of the most fundamental decisions in OO design is where to put responsibilities. These refactorings help you **move functionality** between classes, **create new classes**, and **hide implementation details**.

1. Move Method
2. Move Field
3. Extract Class
4. Inline Class
5. Hide Delegate
6. Remove Middle Man
7. Introduce Foreign Method
8. Introduce Local Extension

B1. Move Method

Problem: A method is used more by another class than by its own class, or a method uses more features of another class than the class on which it is defined.

Solution: Create a new method in the class that uses the method the most, then either turn the old method into a delegation or remove it altogether.

B1. Move Method -- Before

```java
public class Account {
    private AccountType type;
    private int daysOverdrawn;

    public double overdraftCharge() {
        if (type.isPremium()) {
            double result = 10;
            if (daysOverdrawn > 7) {
                result += (daysOverdrawn - 7) * 0.85;
            }
            return result;
        } else {
            return daysOverdrawn * 1.75;
        }
    }

    public double bankCharge() {
        double result = 4.5;
        if (daysOverdrawn > 0) {
            result += overdraftCharge();
        }
        return result;
    }
}
```

B1. Move Method -- After

```java
public class AccountType {
    public double overdraftCharge(int daysOverdrawn) {
        if (isPremium()) {
            double result = 10;
            if (daysOverdrawn > 7) {
                result += (daysOverdrawn - 7) * 0.85;
            }
            return result;
        } else {
            return daysOverdrawn * 1.75;
        }
    }
}

public class Account {
    private AccountType type;
    private int daysOverdrawn;

    public double overdraftCharge() {
        return type.overdraftCharge(daysOverdrawn);
    }

    public double bankCharge() {
        double result = 4.5;
        if (daysOverdrawn > 0) {
            result += type.overdraftCharge(daysOverdrawn);
        }
        return result;
    }
}
```

B2. Move Field

Problem: A field is used more by another class than by the class in which it is defined.

Solution: Create a field in the target class and change all users of the old field to use the new one.

Moving state is often done in conjunction with Move Method. If a field is used primarily by methods that have been moved (or should be moved) to another class, move the field there too.

B3. Extract Class

Problem: When one class does the work of two, the result is awkwardness and complexity.

Solution: Create a new class and move the relevant fields and methods from the old class into the new class.

Classes grow over time. A class that does too much is hard to understand and hard to change. Split it!

B3. Extract Class -- Before

```java
public class Person {
    private String name;
    private String officeAreaCode;
    private String officeNumber;

    public String getName() { return name; }

    public String getTelephoneNumber() {
        return "(" + officeAreaCode + ") " + officeNumber;
    }

    public String getOfficeAreaCode() { return officeAreaCode; }
    public void setOfficeAreaCode(String arg) { officeAreaCode = arg; }

    public String getOfficeNumber() { return officeNumber; }
    public void setOfficeNumber(String arg) { officeNumber = arg; }
}
```

B3. Extract Class -- After

```java
public class TelephoneNumber {
    private String areaCode;
    private String number;

    public String getAreaCode() { return areaCode; }
    public void setAreaCode(String arg) { areaCode = arg; }
    public String getNumber() { return number; }
    public void setNumber(String arg) { number = arg; }

    public String getTelephoneNumber() {
        return "(" + areaCode + ") " + number;
    }
}

public class Person {
    private String name;
    private TelephoneNumber officeTelephone = new TelephoneNumber();

    public String getName() { return name; }
    public String getTelephoneNumber() {
        return officeTelephone.getTelephoneNumber();
    }
    public TelephoneNumber getOfficeTelephone() {
        return officeTelephone;
    }
}
```

B4. Inline Class

Problem: A class does almost nothing, has no real responsibility, and no additional responsibilities are planned for it.

Solution: Move all features from the class into another one and delete it.

Inline Class is the reverse of Extract Class. Use it when a class is no longer pulling its weight and should be folded into another class.

B5. Hide Delegate

Problem: A client is calling a delegate class of an object directly.

Solution: Create methods on the server to hide the delegate.

Encapsulation is one of the key principles of OOP. When a client needs to call methods on a delegate object obtained from another object, the client needs to know about the delegate. If the delegate changes, the client may need to change too.

B5. Hide Delegate -- Before

```java
// Client code
public class Client {
    public void doSomething() {
        Person manager = john.getDepartment().getManager();
        // Client knows about Department -- too much coupling!
    }
}

public class Person {
    private Department department;
    public Department getDepartment() { return department; }
}

public class Department {
    private Person manager;
    public Person getManager() { return manager; }
}
```

B5. Hide Delegate -- After

```java
// Client code -- no longer knows about Department
public class Client {
    public void doSomething() {
        Person manager = john.getManager();
    }
}

public class Person {
    private Department department;

    // Delegate method hides the Department class
    public Person getManager() {
        return department.getManager();
    }
}

public class Department {
    private Person manager;
    public Person getManager() { return manager; }
}
```

Now the client does not depend on `Department`. If `Department` changes its interface, only `Person` needs to change.

B6. Remove Middle Man

Problem: A class has too many methods that simply delegate to another class.

Solution: Delete the delegation methods and let the client call the delegate directly.

Remove Middle Man is the inverse of Hide Delegate. If a class has become a "middle man" that adds no value, it is better to let clients call the delegate directly rather than maintain dozens of pass-through methods.

B7. Introduce Foreign Method

Problem: A utility class does not contain the method you need, and you cannot modify the class.

Solution: Create the method in the client class and pass an instance of the utility class as the first argument.

```
// Before -- awkward workaround
Date newStart = new Date(previousEnd.getYear(),
    previousEnd.getMonth(), previousEnd.getDate() + 1);

// After -- introduce foreign method
Date newStart = nextDay(previousEnd);

private static Date nextDay(Date date) {
    return new Date(date.getYear(),
        date.getMonth(), date.getDate() + 1);
}
```

B8. Introduce Local Extension

Problem: A utility class does not contain several methods that you need, and you cannot modify the class.

Solution: Create a new class that contains these extra methods. Make this extension class either a subclass or a wrapper of the original.

Unlike Introduce Foreign Method (which is for one or two methods), Introduce Local Extension is for when you need to add many methods to a class you cannot modify. Create a subclass or wrapper class that provides the additional functionality.

Module B -- Takeaway

Moving Features between Objects: Key Points

Technique	When to Use
Move Method	Method uses more features of another class
Move Field	Field used more by another class
Extract Class	Class doing the work of two
Inline Class	Class has no real responsibility
Hide Delegate	Client calling chain of objects

Golden rule: Each class should have one clear responsibility with properly assigned features.

Module C: Organizing Data

15 Techniques for Better Data Handling

Module C Outline

Organizing Data

These refactorings deal with data handling, replacing primitives with rich class functionality, and untangling class associations.

1. Self Encapsulate Field
2. Replace Data Value with Object
3. Change Value to Reference
4. Change Reference to Value
5. Replace Array with Object
6. Duplicate Observed Data
7. Change Unidirectional Association to Bidirectional
8. Change Bidirectional Association to Unidirectional

Module C Outline (continued)

C1. Self Encapsulate Field

Problem: You are accessing a field directly, but the coupling to the field is becoming awkward.

Solution: Create getting and setting methods for the field and use only those to access the field, even within the class itself.

```java
// Before
private int low, high;
boolean includes(int arg) {
    return arg >= low && arg <= high;
}

// After
private int low, high;
boolean includes(int arg) {
    return arg >= getLow() && arg <= getHigh();
}
int getLow() { return low; }
int getHigh() { return high; }
```

This allows subclasses to override how the field is accessed.

C2. Replace Data Value with Object

Problem: A data item that needs additional data or behavior (e.g., a phone number string that needs formatting, validation, or area code extraction).

Solution: Turn the data item into an object.

In early stages of development, simple data items represented as Strings or numbers are fine. As the system grows, these simple items often need formatting, validation, or associated behavior. At that point, wrap them in an object.

C3. Change Value to Reference

Problem: You have many identical instances of a class that you need to replace with a single object.

Solution: Convert the equal objects into a single reference object.

When you have many copies of the same object (e.g., multiple `Customer` objects for the same real-world customer), changes to one copy are not reflected in the others. Convert to a reference object so there is only one instance per real-world entity.

C4. Change Reference to Value

Problem: You have a reference object that is too small and infrequently changed to justify the complexity of managing its lifecycle.

Solution: Make it a value object.

Reference objects require lifecycle management (who creates them, how to access them). If the object is small and immutable, treat it as a value object instead -- simpler and easier to reason about.

C5. Replace Array with Object

Problem: You have an array in which certain elements mean different things.

Solution: Replace the array with an object that has a field for each element.

```
// Before
String[] row = new String[3];
row[0] = "Liverpool";
row[1] = "15";    // wins

// After
Performance row = new Performance();
row.setName("Liverpool");
row.setWins(15);
```

Arrays should be used only for collections of similar objects. When each element has a different meaning, use an object.

C6. Duplicate Observed Data

Problem: Domain data is stored in GUI-only classes but needs to be accessed by domain objects.

Solution: Copy the data to a domain object and set up an observer to synchronize the two.

This separates the presentation layer from the business logic. In MVC-style architectures, domain data should live in domain objects, not in GUI components. Use the Observer pattern to keep the two in sync.

C7. Change Unidirectional Association to Bidirectional

Problem: You have two classes that need to use each other's features, but only a one-way link exists.

Solution: Add back pointers, and change modifiers to update both sets.

When class A references class B, but class B also needs to know about class A, add a back-pointer from B to A and manage the link from both sides.

C8. Change Bidirectional Association to Unidirectional

Problem: You have a bidirectional association, but one class no longer needs features from the other.

Solution: Drop the unneeded end of the association.

Bidirectional associations add complexity (both classes need to be kept in sync, create interdependency). If one direction is no longer used, remove it.

C9. Replace Magic Number with Symbolic Constant

Problem: Your code uses a number that has a certain meaning to it.

Solution: Create a constant, give it a human-readable name, and replace the number with it.

```
// Before
double potentialEnergy(double mass, double height) {
    return mass * 9.81 * height;
}

// After
static final double GRAVITATIONAL_CONSTANT = 9.81;

double potentialEnergy(double mass, double height) {
    return mass * GRAVITATIONAL_CONSTANT * height;
}
```

Magic numbers are one of the oldest problems in programming. They are hard to understand and hard to change (especially if the same number appears in multiple places with different meanings).

C10. Encapsulate Field

Problem: There is a public field.

Solution: Make the field private and provide access methods.

```java
// Before
public class Person {
    public String name;
}

// After
public class Person {
    private String name;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

One of the key tenets of OOP is **encapsulation** -- hiding data. Public fields expose implementation details and prevent you from adding validation or side effects later.

C11. Encapsulate Collection

Problem: A method returns a collection (List, Set, Map) directly.

Solution: Return a read-only view of the collection and provide add/remove methods.

```
// Before
public List<Course> getCourses() { return courses; }
public void setCourses(List<Course> courses) {
    this.courses = courses;
}

// After
public List<Course> getCourses() {
    return Collections.unmodifiableList(courses);
}
public void addCourse(Course c) { courses.add(c); }
public void removeCourse(Course c) { courses.remove(c); }
```

Returning a raw collection lets clients modify it without the owning object knowing.

C12. Replace Type Code with Class

Problem: A class has a numeric type code that does not affect behavior.

Solution: Replace the number with a new class (or enum).

```
// Before
public static final int O = 0;
public static final int A = 1;
public static final int B = 2;
public static final int AB = 3;

// After
public enum BloodGroup {
    O, A, B, AB;
}
```

Using raw ints for type codes provides no compile-time type safety. Using a class or enum gives type safety and prevents invalid values.

C13. Replace Type Code with Subclasses

Problem: You have an immutable type code that affects the behavior of a class.

Solution: Replace the type code with subclasses.

```java
// Before
public class Employee {
    static final int ENGINEER = 0;
    static final int SALESMAN = 1;
    static final int MANAGER = 2;
    int type;
}

// After
public abstract class Employee { }
public class Engineer extends Employee { }
public class Salesman extends Employee { }
public class Manager extends Employee { }
```

When the type code affects behavior (different types have different implementations), subclasses let you use **polymorphism** instead of conditionals.

C14. Replace Type Code with State/Strategy

Problem: You have a type code that affects behavior, but you cannot use subclassing (e.g., the type changes at runtime).

Solution: Replace the type code with a state object (State or Strategy pattern).

When subclassing is not possible (because the type of an object changes during its lifetime), use the State or Strategy pattern instead. Create a type hierarchy for the "type" and delegate behavior to the type object.

C15. Replace Subclass with Fields

Problem: You have subclasses that differ only in the values of certain constant fields or methods that return constants.

Solution: Replace the subclasses with fields in the superclass and eliminate the subclasses.

```
// Before
abstract class Person {
    abstract boolean isMale();
}
class Male extends Person {
    boolean isMale() { return true; }
}
class Female extends Person {
    boolean isMale() { return false; }
}

// After
class Person {
    private boolean isMale;
    Person(boolean isMale) { this.isMale = isMale; }
    boolean isMale() { return isMale; }
}
```

Module C -- Takeaway

Organizing Data: Key Points

Technique	When to Use
Self Encapsulate Field	Need flexibility in field access
Replace Data Value with Object	Data item needs behavior
Change Value to Reference / Reference to Value	Control object identity semantics
Replace Array with Object	Array elements have different meanings
Duplicate Observed Data	Separate domain data from GUI

Module D: Simplifying Conditional Expressions

8 Techniques for Cleaner Logic

Module D Outline

Simplifying Conditional Expressions

Conditional logic has a way of getting tricky, so here are a number of refactorings to **simplify** it. The core refactoring is Decompose Conditional, which entails breaking a conditional into pieces. The logical opposite is Consolidate Conditional Expression.

1. Decompose Conditional
2. Consolidate Conditional Expression
3. Consolidate Duplicate Conditional Fragments
4. Remove Control Flag
5. Replace Nested Conditional with Guard Clauses
6. Replace Conditional with Polymorphism
7. Introduce Null Object
8. Introduce Assertion

D1. Decompose Conditional

Problem: You have a complicated conditional (if-then-else) chain.

Solution: Extract methods from the condition, then-part, and else-part.

Complex conditionals make code hard to read. Extract the condition and each branch into clearly named methods.

D1. Decompose Conditional -- Before

```java
public class PriceCalculator {
    private Date summerStart, summerEnd;
    private double winterRate, winterServiceCharge;
    private double summerRate;

    public double getCharge(int quantity, Date date) {
        double charge;
        if (date.before(summerStart) || date.after(summerEnd)) {
            charge = quantity * winterRate + winterServiceCharge;
        } else {
            charge = quantity * summerRate;
        }
        return charge;
    }
}
```

D1. Decompose Conditional -- After

```java
public class PriceCalculator {
    private Date summerStart, summerEnd;
    private double winterRate, winterServiceCharge;
    private double summerRate;

    public double getCharge(int quantity, Date date) {
        if (isSummer(date)) {
            return summerCharge(quantity);
        } else {
            return winterCharge(quantity);
        }
    }

    private boolean isSummer(Date date) {
        return !date.before(summerStart) && !date.after(summerEnd);
    }

    private double summerCharge(int quantity) {
        return quantity * summerRate;
    }

    private double winterCharge(int quantity) {
        return quantity * winterRate + winterServiceCharge;
    }
}
```

D2. Consolidate Conditional Expression

Problem: You have a sequence of conditional tests that all have the same result.

Solution: Combine them into a single conditional expression and extract it into a method.

```
// Before
double disabilityAmount() {
    if (seniority < 2) return 0;
    if (monthsDisabled > 12) return 0;
    if (isPartTime) return 0;
    // compute the disability amount ...
}

// After
double disabilityAmount() {
    if (isNotEligibleForDisability()) return 0;
    // compute the disability amount ...
}

boolean isNotEligibleForDisability() {
    return seniority < 2
        || monthsDisabled > 12
        || isPartTime;
}
```

D3. Consolidate Duplicate Conditional Fragments

Problem: The same fragment of code is in all branches of a conditional expression.

Solution: Move it outside of the expression.

```
// Before
if (isSpecialDeal()) {
    total = price * 0.95;
    send();
} else {
    total = price * 0.98;
    send();
}

// After
if (isSpecialDeal()) {
    total = price * 0.95;
} else {
    total = price * 0.98;
}
send();
```

D4. Remove Control Flag

Problem: You have a boolean variable that acts as a control flag for a series of conditional expressions.

Solution: Use `break` or `return` instead of the control flag.

```java
// Before
boolean found = false;
for (Person p : people) {
    if (!found) {
        if (p.getName().equals("Don")) {
            sendAlert();
            found = true;
        }
    }
}

// After
for (Person p : people) {
    if (p.getName().equals("Don")) {
        sendAlert();
        break;
    }
}
```

D5. Replace Nested Conditional with Guard Clauses

Problem: You have a group of nested conditionals that make it hard to determine the normal flow of code execution.

Solution: Use guard clauses for all special cases. Guard clauses either return or throw an exception.

D5. Replace Nested Conditional with Guard Clauses -- Before

```java
public double getPayAmount() {
    double result;
    if (isDead) {
        result = deadAmount();
    } else {
        if (isSeparated) {
            result = separatedAmount();
        } else {
            if (isRetired) {
                result = retiredAmount();
            } else {
                result = normalPayAmount();
            }
        }
    }
    return result;
}
```

D5. Replace Nested Conditional with Guard Clauses -- After

```java
public double getPayAmount() {
    if (isDead) return deadAmount();
    if (isSeparated) return separatedAmount();
    if (isRetired) return retiredAmount();
    return normalPayAmount();
}
```

Guard clauses say "this is rare; if it happens, do something and get out." The key insight is that one branch is the normal behavior and the other is an unusual condition. If both are part of normal behavior, use if/else. If the condition is unusual, use a guard clause.

D6. Replace Conditional with Polymorphism

Problem: You have a conditional that chooses different behavior depending on the type of an object.

Solution: Move each leg of the conditional to an overriding method in a subclass. Make the original method abstract.

D6. Replace Conditional with Polymorphism -- Before

```java
public class Bird {
    enum BirdType { EUROPEAN, AFRICAN, NORWEGIAN_BLUE }
    BirdType type;
    int numberOfCoconuts;
    double voltage;
    boolean isNailed;

    public double getSpeed() {
        switch (type) {
            case EUROPEAN:
                return getBaseSpeed();
            case AFRICAN:
                return getBaseSpeed() - getLoadFactor() * numberOfCoconuts;
            case NORWEGIAN_BLUE:
                return isNailed ? 0 : getBaseSpeed(voltage);
            default:
                throw new RuntimeException("Unknown bird type");
        }
    }
}
```

D6. Replace Conditional with Polymorphism -- After

```java
public abstract class Bird {
    public abstract double getSpeed();
    protected double getBaseSpeed() { /* ... */ }
}

public class European extends Bird {
    public double getSpeed() {
        return getBaseSpeed();
    }
}

public class African extends Bird {
    private int numberOfCoconuts;
    public double getSpeed() {
        return getBaseSpeed() - getLoadFactor() * numberOfCoconuts;
    }
}

public class NorwegianBlue extends Bird {
    private double voltage;
    private boolean isNailed;
    public double getSpeed() {
        return isNailed ? 0 : getBaseSpeed(voltage);
    }
}
```

D7. Introduce Null Object

Problem: You have repeated checks for `null` throughout your code.

Solution: Instead of returning `null`, return a special-case object that implements the expected interface with default ("do nothing") behavior.

```java
// Before
if (customer == null) {
    plan = BillingPlan.basic();
} else {
    plan = customer.getPlan();
}

// After -- NullCustomer always returns the basic plan
plan = customer.getPlan();

public class NullCustomer extends Customer {
    public BillingPlan getPlan() {
        return BillingPlan.basic();
    }
}
```

D8. Introduce Assertion

Problem: A section of code assumes something about the state of the program.

Solution: Make the assumption explicit with an assertion.

```
// Before
double getExpenseLimit() {
    // should have either expense limit or a primary project
    return (expenseLimit != NULL_EXPENSE)
        ? expenseLimit
        : primaryProject.getMemberExpenseLimit();
}

// After
double getExpenseLimit() {
    assert (expenseLimit != NULL_EXPENSE || primaryProject != null)
        : "Must have expense limit or primary project";
    return (expenseLimit != NULL_EXPENSE)
        ? expenseLimit
        : primaryProject.getMemberExpenseLimit();
}
```

Assertions document assumptions and fail fast when assumptions are violated.

Module D -- Takeaway

Simplifying Conditional Expressions: Key Points

Technique	When to Use
Decompose Conditional	Complex if/else chains
Consolidate Conditional	Multiple conditions, same outcome
Consolidate Duplicate Fragments	Same code in all branches
Remove Control Flag	Boolean flags controlling flow
Guard Clauses	Deeply nested conditionals

Golden rule: Conditional logic should read like a clear, linear story, not a maze.

Module E: Simplifying Method Calls

14 Techniques for Better Interfaces

Module E Outline

Simplifying Method Calls

These refactorings make **method interfaces** simpler and easier to understand. A clean interface is the key to creating easy-to-use classes.

1. Rename Method
2. Add Parameter
3. Remove Parameter
4. Separate Query from Modifier
5. Parameterize Method
6. Replace Parameter with Explicit Methods
7. Preserve Whole Object

Module E Outline (continued)

8. Replace Parameter with Method Call
9. Introduce Parameter Object
10. Remove Setting Method
11. Hide Method
12. Replace Constructor with Factory Method
13. Replace Error Code with Exception
14. Replace Exception with Test

E1. Rename Method

Problem: The name of a method does not reveal its purpose.

Solution: Rename the method.

```java
// Before
public class Customer {
    public String getsnm() {
        return surname;
    }
}

// After
public class Customer {
    public String getSurname() {
        return surname;
    }
}
```

Code is read far more often than it is written. A good method name is the single best tool for communicating intent. If you have to look at the body of a method to understand what it does, the name should be improved.

E2. Add Parameter

Problem: A method does not have enough data to perform its actions.

Solution: Create a new parameter to pass in this data.

This is a common refactoring, but be careful -- too many parameters is a code smell in itself. Before adding a parameter, consider: Can the method obtain this data some other way? Should the data be part of the object's state?

E3. Remove Parameter

Problem: A parameter is no longer used by the method body.

Solution: Remove the unused parameter.

Every parameter represents work the caller has to do. If the parameter is not used, it misleads the reader and creates unnecessary coupling. Remove it.

E4. Separate Query from Modifier

Problem: You have a method that returns a value but also changes the state of an object.

Solution: Create two methods, one for the query and one for the modification.

```
// Before
int getTotalAndSendBill() {
    int total = calculateTotal();
    sendBill(total);
    return total;
}

// After
int getTotal() {
    return calculateTotal();
}
void sendBill() {
    sendBill(calculateTotal());
}
```

It is a good idea to clearly signal the difference between methods with side effects and those without. A query method (no side effects) can be called freely without worry.

E5. Parameterize Method

Problem: Several methods do similar things but with different values contained in the method body.

Solution: Create one method that uses a parameter for the different values.

```
// Before
void fivePercentRaise() { salary *= 1.05; }
void tenPercentRaise() { salary *= 1.10; }

// After
void raise(double factor) { salary *= (1 + factor); }
```

E6. Replace Parameter with Explicit Methods

Problem: A method is split into parts, each of which is run depending on the value of a parameter.

Solution: Create a separate method for each value of the parameter.

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

// After
void setHeight(int value) { height = value; }
void setWidth(int value) { width = value; }
```

This is the inverse of Parameterize Method. Use it when discrete parameter values lead to very different behavior.

E7. Preserve Whole Object

Problem: You get several values from an object, then pass them as parameters to a method.

Solution: Send the whole object instead.

```
// Before
int low = daysTempRange.getLow();
int high = daysTempRange.getHigh();
boolean withinPlan = plan.withinRange(low, high);

// After
boolean withinPlan = plan.withinRange(daysTempRange);
```

Passing the whole object reduces the parameter list and makes the code more adaptable if the called method needs more data from the object in the future.

E8. Replace Parameter with Method Call

Problem: Calling a query method and passing its result as a parameter of another method, while the second method could call the query directly.

Solution: Remove the parameter and let the method invoke the query itself.

```
// Before
int basePrice = quantity * itemPrice;
double seasonDiscount = getSeasonalDiscount();
double finalPrice = discountedPrice(basePrice, seasonDiscount);

// After
double finalPrice = discountedPrice(basePrice);

double discountedPrice(int basePrice) {
    return basePrice - getSeasonalDiscount();
}
```

E9. Introduce Parameter Object

Problem: Several parameters always go together.

Solution: Replace them with an object.

```java
// Before
public class Account {
    double getFlowBetween(Date start, Date end) {
        double result = 0;
        for (Entry e : entries) {
            if (e.getDate().compareTo(start) >= 0
                && e.getDate().compareTo(end) <= 0) {
                result += e.getValue();
            }
        }
        return result;
    }
}

// After
public class DateRange {
    private final Date start;
    private final Date end;
    public DateRange(Date start, Date end) { this.start = start; this.end = end; }
    public boolean includes(Date date) {
        return date.compareTo(start) >= 0 && date.compareTo(end) <= 0;
    }
}

public class Account {
    double getFlowBetween(DateRange range) {
        double result = 0;
        for (Entry e : entries) {
            if (range.includes(e.getDate())) {
                result += e.getValue();
            }
        }
        return result;
    }
}
```

E10. Remove Setting Method

Problem: A field should be set at creation time and never altered.

Solution: Remove any setter for the field. Make the field final if possible.

```
// Before
class Employee {
    private String id;
    public void setId(String id) { this.id = id; }
}

// After
class Employee {
    private final String id;
    public Employee(String id) { this.id = id; }
}
```

If you do not want a field to change after the object is constructed, do not provide a way to change it. This communicates intent and prevents bugs.

E11. Hide Method

Problem: A method is not used by any other class, or is only used inside its own class hierarchy.

Solution: Make the method private (or protected).

Every public method is part of the class's contract. The more public methods, the harder it is to understand and maintain the class. If a method is only used internally, make it private.

E12: Replace Constructor with Factory Method

Problem: You have a complex constructor that does more than just setting fields, or you want to return different subclasses depending on parameters.

Solution: Replace the constructor with a factory method.

```
// Before
class Employee {
    Employee(int type) {
        this.type = type;
    }
}
Employee eng = new Employee(Employee.ENGINEER);
Employee mgr = new Employee(Employee.MANAGER);

// After
class Employee {
    static Employee create(int type) {
        switch (type) {
            case ENGINEER: return new Engineer();
            case MANAGER:  return new Manager();
            default: throw new IllegalArgumentException();
        }
    }
}
Employee eng = Employee.create(Employee.ENGINEER);
Employee mgr = Employee.create(Employee.MANAGER);
```

Factory methods can return subclass instances, have meaningful names, and cache objects.

E13. Replace Error Code with Exception

Problem: A method returns a special error code to indicate an error.

Solution: Throw an exception instead.

```java
// Before
int withdraw(int amount) {
    if (amount > balance) return -1;
    balance -= amount;
    return 0;
}

// After
void withdraw(int amount) throws BalanceException {
    if (amount > balance)
        throw new BalanceException("Insufficient funds");
    balance -= amount;
}
```

Error codes clutter the normal flow with error-handling checks. Exceptions separate the normal path from the exceptional one.

E14. Replace Exception with Test

Problem: You throw an exception in a place where a simple test would do.

Solution: Replace the exception with a conditional test.

```
// Before
double getValueForPeriod(int periodNumber) {
    try {
        return values[periodNumber];
    } catch (ArrayIndexOutOfBoundsException e) {
        return 0;
    }
}

// After
double getValueForPeriod(int periodNumber) {
    if (periodNumber >= values.length) return 0;
    return values[periodNumber];
}
```

Exceptions should be used for unexpected conditions, not for regular flow control. A simple `if` check is far more readable and performant than a `try/catch`.

Module E -- Takeaway

Simplifying Method Calls: Key Points

Technique	When to Use
Rename Method	Name does not reveal purpose
Add / Remove Parameter	Parameter needed or no longer used
Separate Query from Modifier	Method both returns value and has side effects
Parameterize Method	Similar methods differ only in values
Replace Parameter with Explicit Methods	Parameter controls branching

Module F: Dealing with Generalization

12 Techniques for Better Inheritance Hierarchies

Dealing with Generalization

These refactorings deal with generalization -- primarily moving methods and fields along the inheritance hierarchy, creating new classes, and replacing inheritance with delegation or vice versa.

1. Pull Up Field
2. Pull Up Method
3. Pull Up Constructor Body
4. Push Down Method
5. Push Down Field
6. Extract Subclass
7. Extract Superclass
8. Extract Interface
9. Collapse Hierarchy
10. Form Template Method

F1. Pull Up Field

Problem: Two subclasses have the same field.

Solution: Move the field to the superclass.

When subclasses develop independently, they often end up with duplicate fields. Moving the field to the superclass eliminates duplication and makes the relationship between superclass and subclass clearer.

```java
// Before
class Salesman extends Employee { private String name; }
class Engineer extends Employee { private String name; }

// After
class Employee { protected String name; }
class Salesman extends Employee { }
class Engineer extends Employee { }
```

F2. Pull Up Method

Problem: Your subclasses have methods that perform similar work.

Solution: Make the methods identical, then move them to the relevant superclass.

F2. Pull Up Method -- Before

```java
public class Employee {
    // no common method
}

public class Salesman extends Employee {
    public String getName() {
        return name;
    }
    public double getAnnualCost() {
        return salary + bonus;
    }
}

public class Engineer extends Employee {
    public String getName() {
        return name;
    }
    public double getAnnualCost() {
        return salary;
    }
}
```

F2. Pull Up Method -- After

```java
public class Employee {
    protected String name;

    public String getName() {
        return name;
    }

    // getAnnualCost stays in subclasses -- it differs
}

public class Salesman extends Employee {
    public double getAnnualCost() {
        return salary + bonus;
    }
}

public class Engineer extends Employee {
    public double getAnnualCost() {
        return salary;
    }
}
```

The identical `getName()` is pulled up. The differing `getAnnualCost()` stays in each subclass.

F5: Pull Up Constructor Body

Problem: Your subclasses have constructors with mostly identical code.

Solution: Create a superclass constructor and move the common code there. Call the superclass constructor from subclass constructors.

```java
// Before
class Manager extends Employee {
    Manager(String name, String id, int grade) {
        this.name = name;
        this.id = id;
        this.grade = grade;
    }
}

// After
class Employee {
    Employee(String name, String id) {
        this.name = name;
        this.id = id;
    }
}
class Manager extends Employee {
    Manager(String name, String id, int grade) {
        super(name, id);
        this.grade = grade;
    }
}
```

F4. Push Down Method

Problem: A method in a superclass is relevant to only one (or a few) of its subclasses.

Solution: Move the method to the subclass(es) that need it.

```java
// Before
class Employee {
    double getQuota() { /* ... */ }  // only for Salesman
}

// After
class Salesman extends Employee {
    double getQuota() { /* ... */ }
}
```

Push Down Method is the inverse of Pull Up Method. Use it when behavior in a superclass is needed by only one subclass.

F5. Push Down Field

Problem: A field is used by only one (or a few) subclasses.

Solution: Move the field to those subclasses.

```
// Before
class Employee {
    protected double quota;  // only used by Salesman
}

// After
class Salesman extends Employee {
    protected double quota;
}
```

Push Down Field is the inverse of Pull Up Field.

F6. Extract Subclass

Problem: A class has features that are used only in some instances.

Solution: Create a subclass for that subset of features.

The main sign: a class has behavior that is used only for some instances. Sometimes this is indicated by a type code. Create a subclass to house the specialized behavior and move the relevant fields and methods down.

F7. Extract Superclass

Problem: You have two classes with similar features.

Solution: Create a superclass and move the common features to it.

If two classes do similar things, you can move the similarities into a common superclass. This is a common alternative to Extract Class.

F8. Extract Interface

Problem: Several clients use the same subset of a class's interface, or two classes have part of their interfaces in common.

Solution: Extract the common subset into an interface.

F8. Extract Interface -- Before

```
public class Employee {
    public double getRate() { return rate; }
    public boolean hasSpecialSkill() { return hasSkill; }
    public String getName() { return name; }
    public String getDepartment() { return department; }
}

// Client only uses getRate() and hasSpecialSkill() for billing
double charge(Employee emp, int days) {
    int base = emp.getRate() * days;
    // ...
}
```

F8. Extract Interface -- After

```java
public interface Billable {
    double getRate();
    boolean hasSpecialSkill();
}

public class Employee implements Billable {
    public double getRate() { return rate; }
    public boolean hasSpecialSkill() { return hasSkill; }
    public String getName() { return name; }
    public String getDepartment() { return department; }
}

// Client depends only on the interface
double charge(Billable emp, int days) {
    double base = emp.getRate() * days;
    // ...
}
```

Now any class implementing `Billable` can be used for billing, not just `Employee`.

F9. Collapse Hierarchy

Problem: A superclass and subclass are not very different.

Solution: Merge them together.

If a subclass has grown to be almost the same as its parent, the hierarchy is not adding value. Merge the subclass into the parent (or vice versa).

F10. Form Template Method

Problem: You have two methods in subclasses that perform similar steps in the same order, yet the steps are different.

Solution: Get the steps into methods with the same signature, so that the original methods become the same. Then pull them up into the superclass using the Template Method pattern.

The Template Method pattern defines the skeleton of an algorithm in a superclass and lets subclasses override specific steps. If subclasses have similar overall structure but differ in details, Form Template Method can unify the structure.

F11. Replace Inheritance with Delegation

Problem: You have a subclass that uses only a portion of the methods of its superclass (or it is not possible to pass superclass data to the subclass).

Solution: Create a field for the superclass, adjust methods to delegate to the superclass, and remove the inheritance.

F11. Replace Inheritance with Delegation -- Before

```java
public class Stack<E> extends Vector<E> {
    // Stack IS-A Vector, but a stack should not expose
    // random access methods like get(index), set(index, e)
    public E push(E item) {
        addElement(item);
        return item;
    }
    public E pop() {
        E obj = peek();
        removeElementAt(size() - 1);
        return obj;
    }
    public E peek() {
        return elementAt(size() - 1);
    }
}
```

F11. Replace Inheritance with Delegation -- After

```java
public class Stack<E> {
    private Vector<E> storage = new Vector<>();

    public E push(E item) {
        storage.addElement(item);
        return item;
    }

    public E pop() {
        E obj = peek();
        storage.removeElementAt(storage.size() - 1);
        return obj;
    }

    public E peek() {
        return storage.elementAt(storage.size() - 1);
    }

    public int size() {
        return storage.size();
    }
}
```

Now `Stack` only exposes stack operations. Clients cannot use `Vector` methods like `get(index)`.

F12. Replace Delegation with Inheritance

Problem: You use delegation and are constantly writing many simple delegating methods.

Solution: Make the delegating class a subclass of the delegate.

This is the inverse of Replace Inheritance with Delegation. If you are writing many pass-through methods and the class genuinely IS-A type of the delegate, inheritance may be simpler.

Warning: Only use this when the class genuinely satisfies the IS-A relationship. Do not use inheritance just to avoid writing delegation methods.

Module F -- Takeaway

Dealing with Generalization: Key Points

Technique	When to Use
Pull Up Field / Method	Common features in subclasses
Pull Up Constructor Body	Duplicate constructor code
Push Down Method / Field	Feature only relevant to some subclasses
Extract Subclass	Features used only in some instances
Extract Superclass	Two classes with similar features
Extract Interface	Clients use subset of a class's interface

Module G: Code Smells to Refactoring Cross-Reference

Mapping Problems to Solutions

Code Smells Overview

A **code smell** is a surface indication that usually corresponds to a deeper problem in the system. Code smells are not bugs -- the code works -- but they indicate weaknesses in design that may slow development or increase the risk of bugs.

Five Categories of Code Smells

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

Code Smell to Refactoring: Bloaters

Code Smell	Description	Recommended Refactorings
Long Method	A method has too many lines of code (generally, any method longer than 10-15 lines)	Extract Method, Replace Temp with Query, Introduce Parameter Object, Preserve Whole Object, Replace Method with Method Object, Decompose Conditional
Large Class	A class contains too many fields, methods, or lines of code	Extract Class, Extract Subclass, Extract Interface, Duplicate Observed Data
Primitive Obsession	Using primitives instead of small objects (money, ranges, phone numbers), using constants for type codes	Replace Data Value with Object, Replace Type Code with Class, Replace Type Code with Subclasses, Replace Type Code with State/Strategy, Replace Array with Object, Introduce Parameter Object
Long Parameter List	A method has more than 3-4 parameters	Replace Parameter with Method Call, Preserve Whole Object, Introduce Parameter Object
Data Clumps	Groups of data that appear together in fields and parameter lists	Extract Class, Introduce Parameter Object, Preserve Whole Object

Code Smell to Refactoring: Object-Orientation Abusers

Code Smell	Description	Recommended Refactorings
Switch Statements	Complex `switch` or `if/else` chains, often on type codes	Extract Method, Move Method, Replace Type Code with Subclasses, Replace Type Code with State/Strategy, Replace Conditional with Polymorphism, Replace Parameter with Explicit Methods, Introduce Null Object
Temporary Field	An object field that is only set/used in certain circumstances	Extract Class, Introduce Null Object
Refused Bequest	A subclass uses only a few of the methods and properties inherited from its parent	Replace Inheritance with Delegation, Extract Superclass
Alternative Classes with Different Interfaces	Two classes perform identical functions but have different method names	Rename Method, Move Method, Extract Superclass

Code Smell to Refactoring: Change Preventers

Code Smell	Description	Recommended Refactorings
Divergent Change	You have to change many unrelated methods when you make changes to a class (the class has too many reasons to change)	Extract Class
Shotgun Surgery	Making any modification requires you to make many small changes to many different classes	Move Method, Move Field, Inline Class
Parallel Inheritance Hierarchies	Whenever you make a subclass of one class, you also have to make a subclass of another	Move Method, Move Field

Code Smell to Refactoring: Dispensables

Code Smell	Description	Recommended Refactorings
Comments	A method is filled with explanatory comments (the code itself should be self-explanatory)	Extract Method, Extract Variable, Rename Method, Introduce Assertion
Duplicate Code	Two code fragments look almost identical	Extract Method, Pull Up Method, Form Template Method, Extract Class
Lazy Class	A class does not do enough to earn your attention / justify its existence	Inline Class, Collapse Hierarchy
Data Class	A class that contains only fields, getters, and setters with no additional behavior	Encapsulate Field, Encapsulate Collection, Move Method, Extract Method, Remove Setting Method, Hide Method
Dead Code	A variable, parameter, field, method, or class is no longer used anywhere	Delete the unused code (Inline Class, Collapse Hierarchy, Remove Parameter)
Speculative Generality	There is an unused class, method, field, or parameter created "just in case" for future use	Collapse Hierarchy, Inline Class, Inline Method, Remove Parameter

Code Smell to Refactoring: Couplers

Code Smell	Description	Recommended Refactorings
Feature Envy	A method accesses the data of another object more than its own data	Move Method, Extract Method
Inappropriate Intimacy	One class uses the internal fields and methods of another class	Move Method, Move Field, Extract Class, Hide Delegate, Change Bidirectional Association to Unidirectional, Replace Inheritance with Delegation
Message Chains	You see a series of calls like `a.getB().getC().getD()`	Hide Delegate, Extract Method, Move Method
Middle Man	A class does nothing except delegate to another class	Remove Middle Man, Inline Method, Replace Delegation with Inheritance
Incomplete Library Class	A library class does not have a method you need	Introduce Foreign Method, Introduce Local Extension

Complete Refactoring Techniques Summary (1/3)

Composing Methods (9)

#	Technique	Purpose
1	Extract Method	Break long methods into smaller, named pieces
2	Inline Method	Remove unnecessary indirection
3	Extract Variable	Make complex expressions readable
4	Inline Temp	Remove blocking temporary variables
5	Replace Temp with Query	Turn temps into reusable query methods
6	Split Temporary Variable	One purpose per variable
7	Remove Assignments to Parameters	Preserve parameter semantics
8	Replace Method with Method Object	Handle complex local variable dependencies
9	Substitute Algorithm	Replace with clearer/faster algorithm

Complete Refactoring Techniques Summary (2/3)

Moving Features (8) + Organizing Data (15)

#	Technique	Purpose
10	Move Method	Place method with the data it uses
11	Move Field	Place field with the methods that use it
12	Extract Class	Split overloaded classes
13	Inline Class	Merge underpowered classes
14	Hide Delegate	Reduce coupling
15	Remove Middle Man	Eliminate excessive delegation
16	Introduce Foreign Method	Add method to unmodifiable class (one-off)
17	Introduce Local Extension	Add methods to unmodifiable class (many)
18-32	Organizing Data (15 techniques)	Encapsulate fields, replace type codes, manage associations

Complete Refactoring Techniques Summary (3/3)

Simplifying Conditionals (8) + Method Calls (14) + Generalization (12)

#	Technique	Purpose
33-40	Simplifying Conditionals (8)	Decompose, consolidate, guard clauses, polymorphism
41-54	Simplifying Method Calls (14)	Rename, add/remove params, factory methods, exceptions
55-66	Dealing with Generalization (12)	Pull up/push down, extract subclass/superclass/interface

Total: 66 Refactoring Techniques across 6 categories

Module G -- Takeaway

Code Smells to Refactoring: Key Points

- Every **code smell** has a set of recommended refactoring techniques to address it
- The same refactoring technique may address **multiple code smells**
- Refactoring is a **continuous** process -- not a one-time event
- The goal is **incremental improvement** -- each refactoring is small and safe
- **Automated tests** are essential to ensure refactoring does not change behavior
- Modern IDEs (IntelliJ IDEA, Eclipse, VS Code) have built-in support for many refactoring techniques

Refactoring Best Practices

1. **Always have tests** before refactoring -- refactoring without tests is risky
2. **Small steps** -- each refactoring should be a small, verifiable change
3. **Commit frequently** -- after each successful refactoring step
4. **Do not add functionality** while refactoring -- separate concerns
5. **Follow your nose** -- if code smells, investigate and refactor
6. **Use IDE support** -- automated refactoring tools reduce manual errors
7. **Review the result** -- ensure the refactored code is actually better

References

- Fowler, M. (1999). *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional.
- Fowler, M. (2018). *Refactoring: Improving the Design of Existing Code* (2nd Edition). Addison-Wesley Professional.
- RefactoringGuru. *Refactoring Techniques*. https://refactoring.guru/refactoring/techniques
- RefactoringGuru. *Composing Methods*. https://refactoring.guru/refactoring/techniques/composing-methods
- RefactoringGuru. *Moving Features between Objects*. https://refactoring.guru/refactoring/techniques/moving-features-between-objects
- RefactoringGuru. *Organizing Data*. https://refactoring.guru/refactoring/techniques/organizing-data

References (continued)

- RefactoringGuru. *Simplifying Conditional Expressions*. https://refactoring.guru/refactoring/techniques/simplifying-conditional-expressions
- RefactoringGuru. *Simplifying Method Calls*. https://refactoring.guru/refactoring/techniques/simplifying-method-calls
- RefactoringGuru. *Dealing with Generalization*. https://refactoring.guru/refactoring/techniques/dealing-with-generalization
- RefactoringGuru. *Code Smells*. https://refactoring.guru/refactoring/smells
- Kerievsky, J. (2004). *Refactoring to Patterns*. Addison-Wesley Professional.
- Martin, R. C. (2008). *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall.

$$End - Of - Week - 13 - Module$$