

Refactoring

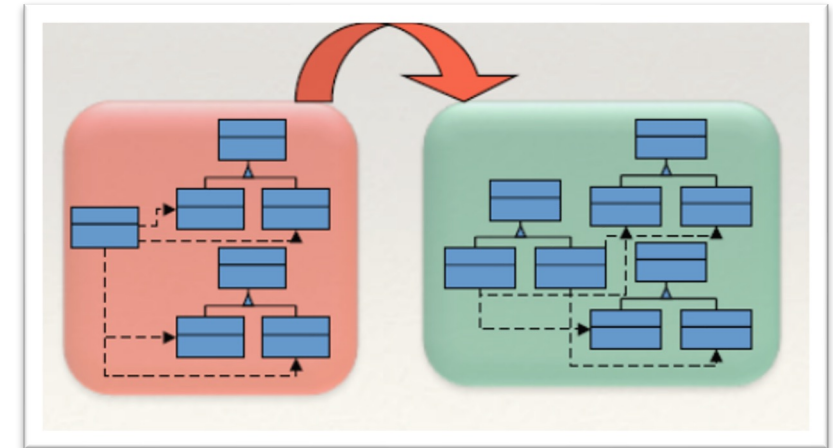
COMP2511, CSE, UNSW



UNSW
SYDNEY

Introduction to Refactoring

- ❖ Refactoring is the process of restructuring existing code **without changing** its **external behavior**.
- ❖ Aim is to:
 - **improve** internal structure/**design**, **readability**, and **maintainability**
 - help **detect bugs**.
 - increase **development speed**.
 - help conform to design principles and **eliminate** design/**code smells**.



When to Refactor

- ❖ Before adding **new features** if current structure is not suitable.
- ❖ While **fixing bugs**.
- ❖ During **code reviews**.


Code Smells

- ❖ **Code smells** are indicators of potential **design issues**.
- ❖ They hint at poor design but do **not** guarantee **defects**.
- ❖ **Refactoring** addresses code smells.

Common Code Smells:

Duplicated Code	Shotgun Surgery
Long Method	Feature Envy
Large Class	Lazy Classes
Long Parameter List	Data Classes
Divergent Change	

Refactoring Cycle

- 
- ❖ Step 1: Identify **code smell**.
 - ❖ Step 2: **Write tests** to confirm current behaviour.
 - ❖ Step 3: Apply small **refactoring** step.
 - ❖ Step 4: **Re-run tests**.
 - ❖ Step 5: **Repeat**.

Refactoring Technique: Extract Method

- ❖ Identify **logical chunks of code** and extract into separate methods.
- ❖ **Benefits:** improves readability, reduces duplication.

Before

```
void printOwing() {  
    printBanner();  
    // calculate outstanding  
    double outstanding = 0;  
    for (Order o : orders) {  
        outstanding += o.getAmount();  
    }  
    printDetails(outstanding);  
}
```

After

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

```
double calculateOutstanding() {  
    double result = 0;  
    for (Order o : orders) {  
        result += o.getAmount();  
    }  
    return result;  
}
```

Refactoring Technique: Move Method

- ❖ Move methods to the class whose data they use most.

```
class Customer {  
    double getDiscount(Product product) {  
        return product.getBasePrice() * 0.1;  
    }  
}
```

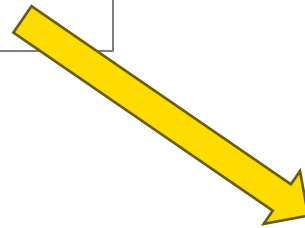
Move *getDiscount* to *Product* class.

```
class Product {  
    double getDiscount() {  
        return this.getBasePrice() * 0.1;  
    }  
}
```

Refactoring Technique: Replace Temp with Query

- ❖ Move expressions into methods instead of temporary variables.

```
double basePrice = quantity * itemPrice;  
if (basePrice > 1000)  
    return basePrice * 0.95;
```



```
if (basePrice() > 1000)  
    return basePrice() * 0.95;  
  
double basePrice() {  
    return quantity * itemPrice;  
}
```


Refactoring Technique: Replace Conditional with Polymorphism

- ❖ **Switch** or **if-else chains** based on type codes are hard to maintain and violate OOP principles.
 - Adding a new type **requires changes** to every switch statement.
 - Increases rigidity and breaks **Open/Closed Principle**.

Solution:

- Replace **switch** statements with inheritance.
- Define a superclass with an abstract method and implement this method in subclasses, each representing a case of the switch.

Refactoring Technique: Replace Conditional with Polymorphism

❖ Use **polymorphism** instead of conditionals.

```
class Movie {  
    int getPriceCode();  
}  
  
class Rental {  
    double getCharge() {  
        switch(movie.getPriceCode()) {  
            case REGULAR: return daysRented * 2;  
            case CHILDRENS: return daysRented * 1.5;  
        }  
    }  
}
```



```
abstract class Movie {  
    abstract double getCharge(int daysRented);  
}
```

```
class RegularMovie extends Movie {  
    double getCharge(int daysRented) {  
        return daysRented * 2;  
    }  
}
```

```
class ChildrensMovie extends Movie {  
    double getCharge(int daysRented) {  
        return daysRented * 1.5;  
    }  
}
```

Refactoring Using Composition

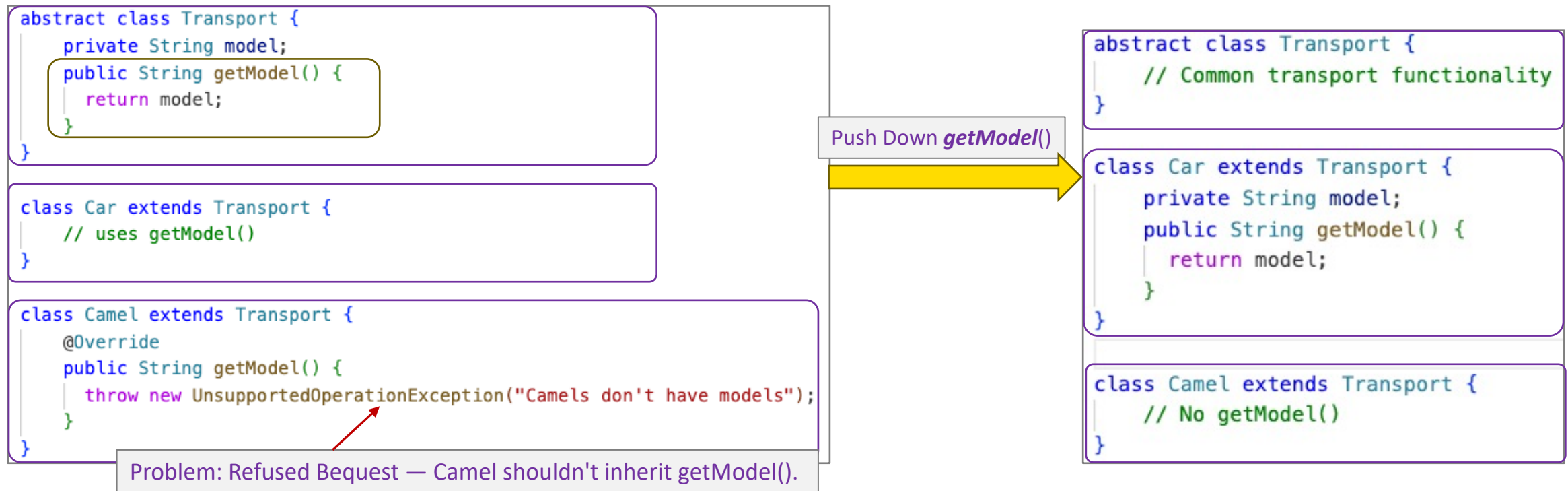
- ❖ Favor composition over inheritance.

Instead of extending *Logger* class,
use composition (has-a relation) and method forwarding.

```
class Application {  
    private Logger logger = new Logger();  
    void logInfo(String msg) {  
        logger.log(msg);  
    }  
}
```

Design Smell: Refused Bequest

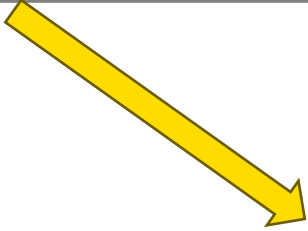
Refused Bequest: subclass inherits inappropriate behavior.



Smell: Long Parameter List

- ❖ To avoid long parameter lists, encapsulate related parameters into a data class and pass an instance of that class instead.

```
void createUser(String name, int age, String email, String phone)
```



```
class UserInfo {  
    String name;  
    int age;  
    String email;  
    String phone;  
}
```

```
void createUser(UserInfo user)
```

Smell: Large Method/Class

- ❖ **Large Method**: method with many lines doing multiple things.
- ❖ Refactor: use **Extract Method** to create new method(s)

- ❖ **Large Class**: Class with 20+ methods and many fields.
- ❖ Refactor: use **Extract Class** to separate concerns.

Smell: Similar Code Fragments

Case 1: Same code in multiple methods of the same class

- Use **Extract Method** and invoke it from each place.

Case 2: Same code in two subclasses of the same level

- Use **Extract Method** in both subclasses, Use **Pull Up Field** or **Pull Up Method** to unify code in the superclass.
- If inside constructors: use **Pull Up Constructor Body**.
- For similar but not identical code: use **Template Method**.
- If algorithms differ, use **Strategy Pattern**.

Case 3: Duplicate code in unrelated classes

- Use **Extract Superclass** to unify shared logic.

Smell: Feature Envy

- ❖ A method is more interested in **another class's data** than its own.

Symptoms

- The method invokes several methods on **another object** to calculate a value.
- Causes unnecessary coupling and **breaks encapsulation**.

Solution: Move the method to the class that owns the data (**Move Method**).

- If only part of the method accesses external data: use **Extract Method** followed by **Move Method**.
- If multiple external classes are involved: identify which one holds the majority of used data and move the method there.

Smell: Divergent Change

- ❖ A class is changed in many unrelated ways for different reasons.
- ❖ Violates Single Responsibility Principle.
- ❖ Increases risk of regression bugs due to unrelated modifications

Solution:

- Identify the reasons for change and separate them into cohesive classes.
- Use **Extract Class** to encapsulate each responsibility.

```
// Before
class DocumentManager {
    void print(Document doc) { ... }
    void save(Document doc) { ... }
    void exportToPDF(Document doc) { ... }
}

// After
class PrintService {
    void print(Document doc) { ... }
}

class PersistenceService {
    void save(Document doc) { ... }
}

class ExportService {
    void exportToPDF(Document doc) { ... }
}
```

Smell: Shotgun Surgery

- ❖ A small change requires updating many different classes.
- ❖ Makes code brittle and hard to maintain.

Solution:

- Consolidate related changes into a single class.
- Use **Move Method**, **Move Field**, or **Inline Class** to localize the change.

```
// Before: logic for logging exists in multiple classes
class Order {
    void logCreation() { Logger.log("Order created"); }
}
class Invoice {
    void logGeneration() { Logger.log("Invoice generated"); }
}
```



```
// After: Centralized logging
class LogService {
    static void logOrderCreation() { Logger.log("Order created"); }
    static void logInvoiceGeneration() { Logger.log("Invoice generated"); }
}
```

Divergent Change and Shotgun Surgery

- ❖ **Divergent Change** = One class changes for many unrelated reasons.
- ❖ **Shotgun Surgery** = One change spreads across many classes.
- ❖ Both can be addressed with refactoring to improve modularity and reduce fragility.

Useful Links

<https://refactoring.guru/refactoring/smells>

<https://www.refactoring.com/catalog/>

Demo

The Video Rental System

End