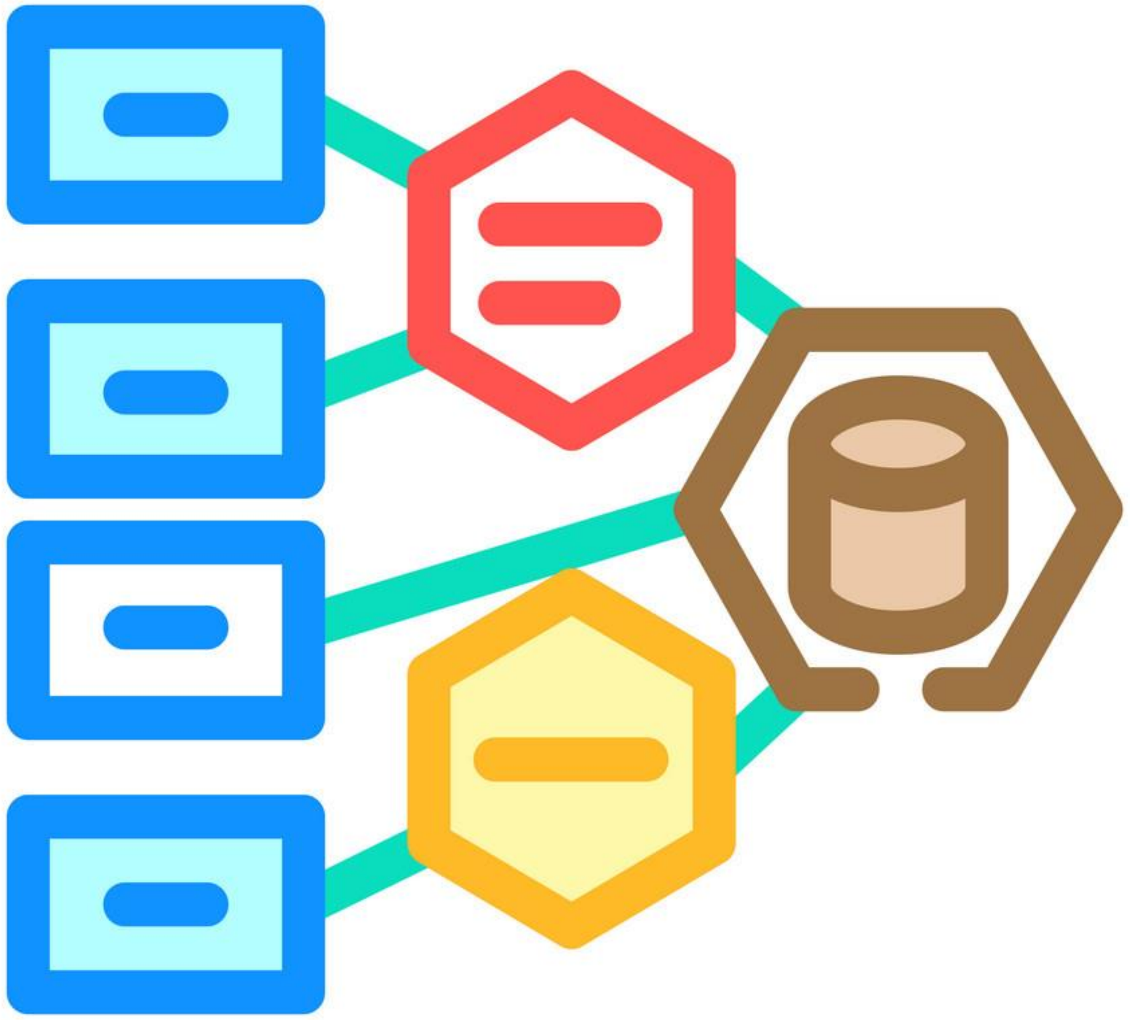
Software Design and architecture

Fall 2024

Dr. Natalia Chaudhry



Outline

- Fundamental software design concepts

Readings

Chapter 12, Design concepts, Software Engineering: A Practitioner’s Approach, Roger S. Pressman, Bruce R. Maxim, 8th Ed, McGraw-Hill Education, 2015.

- Design creates a representation or model of the software, but unlike the requirements model (that focuses on describing required data, function, and behavior), the design model provides detail about software architecture, data structures, interfaces, and components that are necessary to implement the system.

What is the work product?

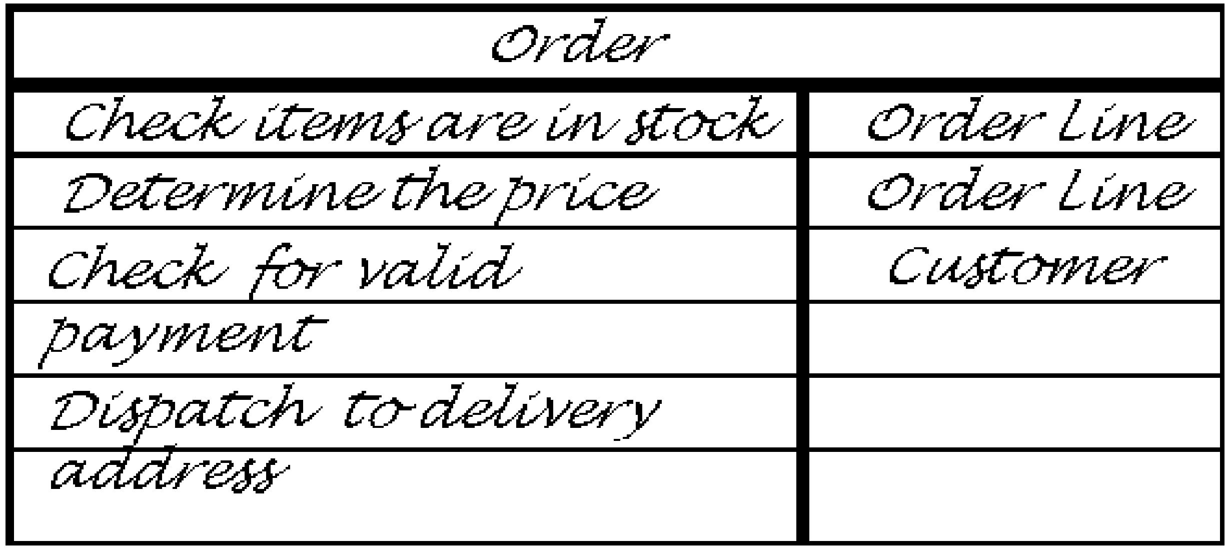
- A design model that encompasses architectural, interface, component-level, and deployment representations is the primary work product that is produced during software design

- The requirements model, manifested by scenario-based, class-based, and behavioral elements, feed the design task.
- Design produces a data/class design, an architectural design, an interface design, and a component design.

CLASS -RESPONSIBILITY-COLLABORATOR MODELING

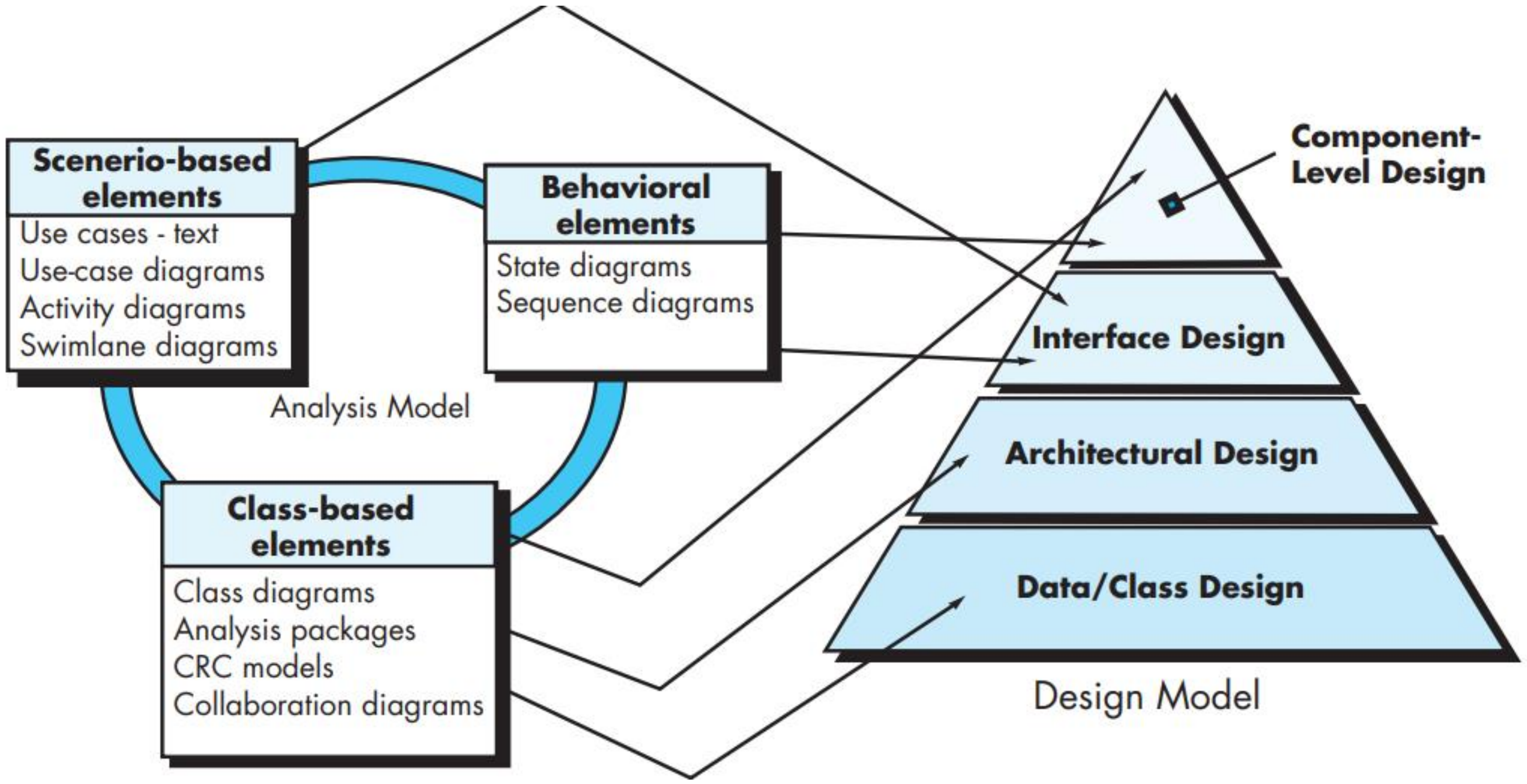
- A CRC model is really a collection of standard index cards that represent classes. The cards are divided into three sections.
- Along the top of the card you write the name of the class.
- In the body of the card you list the class responsibilities on the left and the collaborators on the right.

- Responsibilities are the attributes and operations that are relevant for the class. Stated simply, a responsibility is "anything the class knows or does"
- Collaborators are those classes that are required to provide a class with the information needed to complete a responsibility.
- Classes fulfill their responsibilities in one of two ways: (1) A class can use its own operations to manipulate its own attributes, thereby fulfilling a particular responsibility, or (2) a class can collaborate with other classes.



Order	
Check items are in stock	Order Line
Determine the price	Order Line
Check for valid payment	Customer
Dispatch to delivery address	

Translating the requirements model into the design model



- **Scenario-based elements (use cases)** → Sequence diagrams, object interactions.
- **Behavioral elements (state machines, activity diagrams)** → State management in classes, workflows.
- **Class-based elements (class diagrams)** → Detailed class structures, relationships, and methods, applying patterns.

Example: Translation Process (Requirements to Design)

Scenario-Based Element (Use Case):

Requirements:

- **Use Case**: "A customer can place an order."

Design:

- Design a sequence diagram:
  - Objects: Customer, ShoppingCart, Order, PaymentGateway.
  - Sequence:
    1. Customer adds items to ShoppingCart.
    2. ShoppingCart creates an Order.
    3. Order is processed through the PaymentGateway.
    4. Confirmation is sent to Customer.

Behavioral Element (State Machine):

Requirements:

•**State Machine**: "An order moves through the states: 'Created', 'Processed', 'Shipped', 'Delivered'."

Design:

•Refine into an OrderManager class that manages state transitions.

- •Methods: processOrder(), shipOrder(), deliverOrder().
- •Each state becomes a method call, with transitions handled by business rules coded into the OrderManager.

Class-Based Element (Class Diagram):

Requirements:

- **Class Diagram**: Define the structure with classes like Order, Customer, Payment.

Design:

- Create detailed class diagrams:
    - Order: Attributes (orderId, orderDate, status), Methods (addItem(), processOrder()).
    - Customer: Attributes (customerId, name, address), Methods (placeOrder(), viewOrderStatus()).
    - Apply appropriate design patterns (e.g., use **Observer pattern** to notify customers when an order state changes).

Design concepts

Abstraction

- When you consider a modular solution to any problem, many levels of abstraction can be posed.
- At the highest level of abstraction, a solution is stated in broad terms using the language of the problem environment.
- At lower levels of abstraction, a more detailed description of the solution is provided.
- Finally, at the lowest level of abstraction, the solution is stated in a manner that can be directly implemented.
- **Procedural** and **data abstractions**

Procedural Abstraction:

- **Focus:** The behavior or logic of a program (the "how").
- **Definition:** Procedural abstraction involves defining a sequence of steps (or procedures) to perform a task without **exposing the internal workings of those steps to the user**. The details of how the procedure is implemented are hidden behind a simple interface, which might be a function or method.
- **Example:** A sorting function like sort(array) hides the specific algorithm (bubble sort, quick sort, etc.) from the user, who only needs to know that it sorts the array.
- **Goal:** To simplify complex operations by allowing the user to think in terms of "what needs to be done" instead of "how it is done."

Data Abstraction:

- **Focus:** Hides *how* data is stored or structured.
- **Definition:** Data abstraction involves defining a data structure or type without exposing the underlying representation. The focus is on the operations that can be performed on the data, rather than on how the data is stored or organized.
- **Example:** A class Stack that has methods push() and pop() to add and remove elements, without revealing how the stack is implemented internally (e.g., as an array or linked list).
- **Goal:** To manage complexity by **hiding the internal details of data representation,** allowing the user to interact with data in a consistent and predictable way.

Architecture

- Architecture is the structure or organization of program components (modules), the manner in which these components interact, and structure of data that are used by the components.
- Different *architectural design models*
- **Structural, framework, dynamic, process** and **functional model**

Structural Model

- The structural model focuses on how the system is composed, including the organization of system components, their relationships, and how they interact.
- It typically involves elements like classes, objects, modules, and their interconnections (e.g., associations, generalizations).
- Class diagrams, object diagrams, and component diagrams in UML (Unified Modeling Language).
- A class diagram that shows the classes in a software system and the relationships (like inheritance and association) between them.

Framework Model.

•It outlines the overall reusable **architecture template** or set of rules for building a system.

•It defines high-level guidelines for organizing the system and can be reused across different projects.

•Framework models often refer to design patterns or pre-existing architectural frameworks like MVC (Model-View-Controller)

•**Example:** The MVC framework used in web development where:

- **Model**: Handles the data and business logic.
- **View**: Renders the user interface.
- **Controller**: Manages communication between the model and the view.

Dynamic Model

•The dynamic model captures the **time-dependent** behavior of the system.

•It focuses on how the system's state changes in response to internal and external events.

•Dynamic models often represent the **flow of control** between components or how the system reacts to user inputs, signals, or changes in data.

•Sequence diagrams, state diagrams, activity diagrams in UML.

•**Example:** A sequence diagram showing interactions between a user and a system during the login process

Process Model

•A process model describes how various processes or tasks are coordinated and executed, focusing on the **sequence of activities** and how tasks or threads in a system execute.

•It may include concurrency, task distribution, or synchronization between processes.

•**Example:** An activity diagram in UML that shows the flow of tasks during an online order process

Functional Model

•The functional model emphasizes the **functions** or operations that the system performs, along with how data flows between these functions.

•It captures how the system transforms inputs into outputs, representing the business logic or functionality.

•Data Flow Diagrams (DFD), Use Case Diagrams in UML.

•**Example:** A Data Flow Diagram (DFD) showing how data flows through a process like online shopping

•**Structural Model:** Focuses on the **static structure** of the system (components and relationships).

•**Framework Model:** Provides a **template or reusable architecture** for organizing the system.

•**Dynamic Model:** Captures the system’s **behavior and state changes** over time.

•**Process Model:** Emphasizes the **flow of activities** and processes within the system.

•**Functional Model:** Focuses on the **functions and transformations** the system performs on data.

A number of different architectural description languages (ADLs) have been developed to represent these models

Patterns

An architectural pattern provides a blueprint for **how to organize and structure an entire system**. It deals with the **overall organization** of the system and the relationships between the high-level components or subsystems. Architectural patterns guide the overall framework or structure of the software.

A design pattern provides a reusable solution for solving **common problems** in the **implementation** phase of development. It deals with the design of classes, objects, or small groups of components, focusing on solving specific design issues such as object creation, structural relationships, or object behavior.

Separation of Concerns

- Separation of concerns is a design concept that suggests that any complex problem can be more easily handled if it is subdivided into pieces that can each be solved and/or optimized independently.
- A concern refers to a specific aspect of the system, such as a feature, functionality, or responsibility.
- By separating concerns into smaller, and therefore more manageable pieces, a problem takes less effort and time to solve.
- **Why useful?**

The goal of SoC is to ensure that each part of the system deals with **only one responsibility** or **aspect** of functionality, making it easier to:

- **Develop**: Components are smaller and more focused, leading to faster development.
- **Test**: Isolated parts are easier to test independently.
- **Maintain**: Changes in one part of the system (e.g., business logic) don't affect other parts (e.g., user interface).
- **Reuse**: Modules can be reused across different parts of a system or even in different systems.

Example of Separation of Concerns:
web application

1. **User Interface (UI)**: How the application is displayed to the user.
2. **Business Logic**: The core functionality and operations (e.g., processing orders, handling transactions).
3. **Data Management**: Interacting with the database to store and retrieve data.

Modularity

Modularity refers to the practice of breaking down a system into **independent, interchangeable modules**.

Each module encapsulates a specific functionality or piece of the system, and modules are designed to interact with each other through well-defined interfaces.

Separation of Concerns often leads to **Modularity**. **SoC** informs how a system is organized **conceptually**, while **modularity** determines how those concepts are physically implemented in a **modular structure**.

Information Hiding

It ensures that only necessary details (or an interface) are exposed, while the implementation details remain hidden and protected from outside interference.

Key Concepts:

•**Encapsulation**: Information hiding is often achieved through **encapsulation**, where the internal state and behavior of an object are hidden and accessed only through well-defined interfaces (such as public methods).

•**Abstraction**: By hiding the unnecessary details, developers focus on the **abstract behavior** rather than the implementation specifics.

•**Interfaces**: Exposing only the necessary functions and hiding the implementation details ensures that other parts of the system interact only with the **interface** and not the internal data.

Functional Independence

- Functional independence is achieved by developing modules with “ singleminded” function
- Independence is assessed using two qualitative criteria: **cohesion** and **coupling.**
- **Cohesion** is an indication of the relative functional strength of a module. A cohesive module performs a single task, requiring little interaction with other components in other parts of a program **<<High..ideal!>>**
- **Coupling** is an indication of the relative interdependence among modules. Simple connectivity among modules results in software that is easier to understand and less prone to a “ripple effect”, caused when errors occur at one location and propagate throughout a system. **<<Low..ideal!>>**

Refinement

- Refinement is actually a ***process of elaboration***.

- You begin with a statement of function (or description of information) that is defined at a high level of abstraction. That is, the statement describes function or information conceptually but provides no indication of the internal workings of the function or the internal structure of the information.
- You then elaborate on the original statement, providing more and more detail as each successive refinement (elaboration) occurs.

- Abstraction and refi nement are complementary concepts

Aspects

A **cross-cutting concern** refers to a functionality or behavior that affects multiple parts of a system but is not confined to a single module or layer.

These concerns often impact various components of an application and are usually implemented independently from the core business logic.

Cross-cutting concerns can lead to **code duplication** and **tangled code** if not managed properly, making maintenance and updates more challenging.

Logging is a common cross-cutting concern because many different parts of an application (e.g., services, controllers, modules) may need to log information (errors, user actions, performance metrics).

Example of Cross-Cutting Concerns

Logging

In most applications, logging is used to record events, errors, and system behavior for debugging, auditing, or performance tracking. Logging affects many parts of the system—each function or module may need to log information, but logging itself is not central to the business logic.

Security

Security is a concern that cuts across various components. Every part of an application needs to ensure that only authorized users can access certain features, but security checks are not part of the core functionality of those features.

Example: In an e-commerce system, user authentication and authorization are required for viewing profiles, making purchases, or accessing order histories. The actual operations (viewing profiles or making purchases) are core to the business, but the security aspect (checking if the user is logged in or has the right permissions) is a cross-cutting concern.

Transaction Management

•In systems where database operations need to be atomic (all operations succeed or none do), transaction management becomes a cross-cutting concern. It affects any module that interacts with the database but is not a part of the core logic of these operations.

•**Example**: In a banking system, transferring funds between accounts requires both accounts to be updated in a single transaction. If any part of the operation fails, the whole transaction must be rolled back. However, transaction management is separate from the actual fund transfer logic.

- Cross-cutting concerns are often handled using **aspect-oriented programming (AOP)** techniques.
- AOP allows developers to separate these concerns from the core business logic by defining **aspects** that can be applied to multiple points in the program, often without modifying the actual code.

- **In Java**: AOP can be implemented using frameworks like **AspectJ** or **Spring AOP**, where aspects (like logging or security) are applied to certain points in the codebase.

Aspect is a concrete implementation of a cross-cutting concern, usually encapsulated in a module, defining how the concern is handled throughout the system.

Refactoring

- Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code [design] yet improves its internal structure.
- When software is refactored, the existing design is examined for redundancy, unused design elements, inefficient or unnecessary algorithms, poorly constructed or inappropriate data structures, or any other design failure that can be corrected to yield a better design.

- For example, a first design iteration might yield a component that exhibits low cohesion. After careful consideration, you may decide that the component should be refactored into three separate components, each exhibiting high cohesion. *The result will be software that is easier to integrate, easier to test, and easier to maintain.*

Refactoring process using Eclipse with a Java

"Extract Method" refactoring

Step 1: Open Eclipse and Create a Java Project
Step 2: Write Sample Java Code

```java
public class Calculator {
    public int add(int a, int b) {
        int result = a + b;
        return result;
    }
    public int multiply(int a, int b) {
        int result = a * b;
        return result;
    }
    public int calculate(int x, int y) {
        // Some complex calculations here
        int intermediateResult = add(x, y);
        int finalResult = multiply(intermediateResult, 2);
        return finalResult;
    }
    public static void main(String[] args) {
        Calculator calculator = new Calculator();
        int result = calculator.calculate(5, 3);
        System.out.println("Result: " + result);
    }
}
```

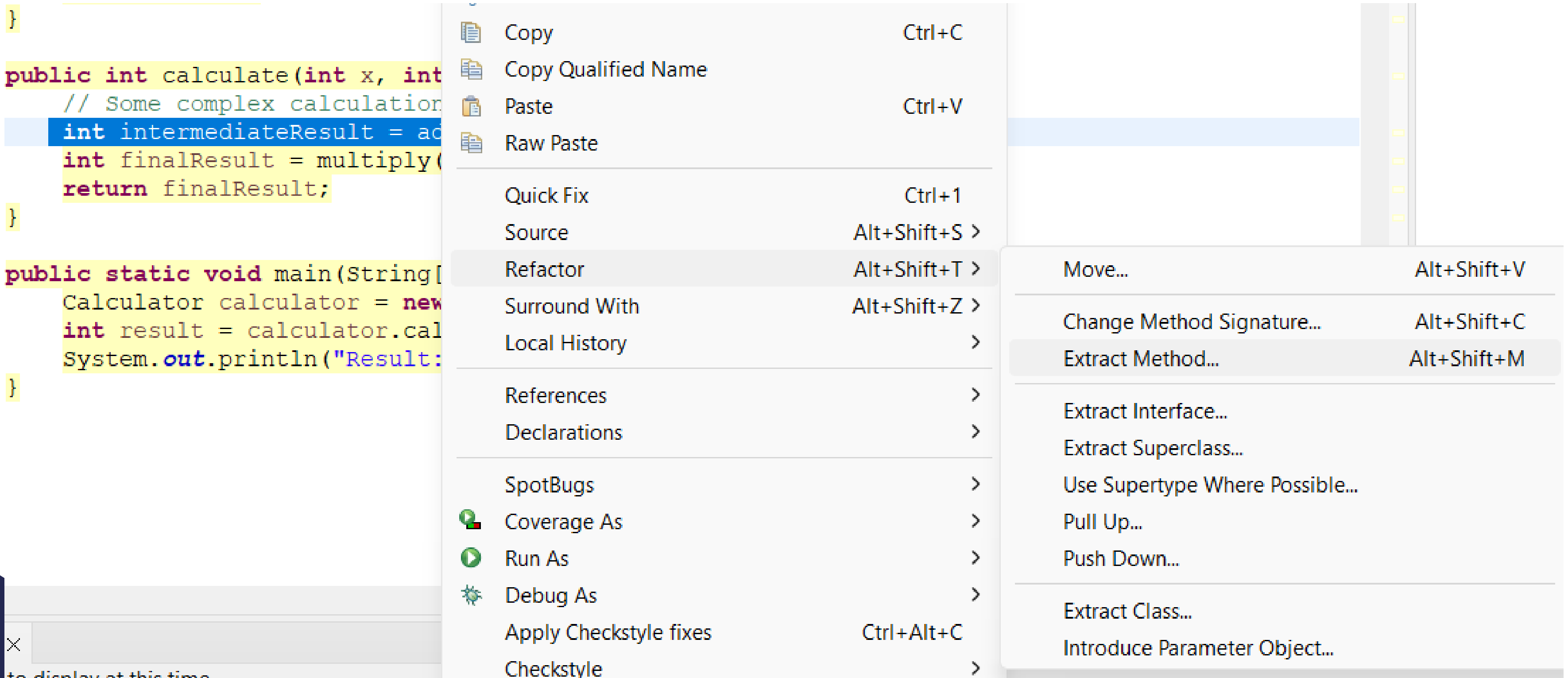
Now, let's say you want to refactor the calculate method by extracting a new method to calculate the intermediate result. Here's how you can do it using Eclipse:

1. Open Eclipse and load your Java project.
2. Locate the calculate method in your code, which contains the complex calculations.
3. Place the cursor inside the calculate method, specifically where the intermediate calculation is done:

int intermediateResult = add(x, y);

Right-click on this line of code and select "Refactor" from the context menu.

In the submenu, choose "Extract Method."

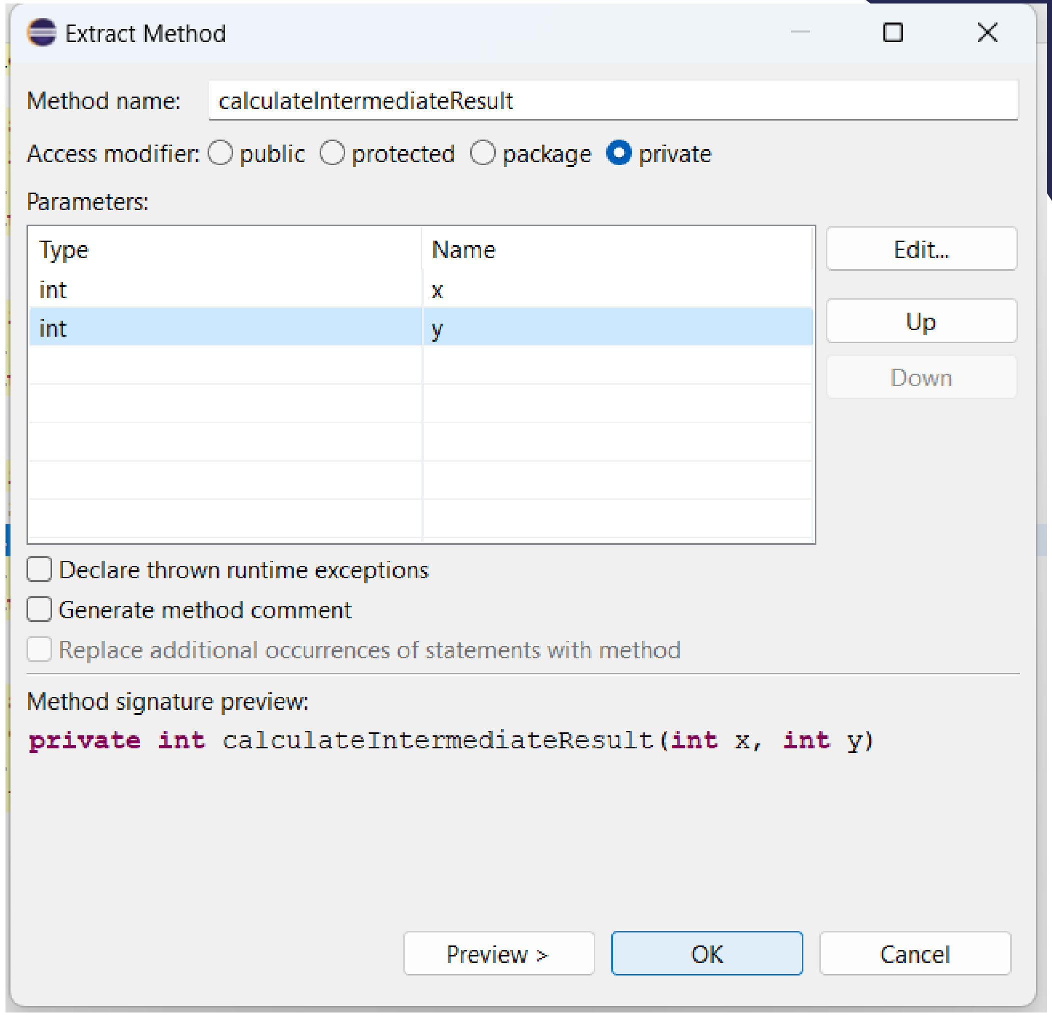


```
}

public int calculate(int x, int
    // Some complex calculation
    int intermediateResult = ad
    int finalResult = multiply(
    return finalResult;
}

public static void main(String[
    Calculator calculator = new
    int result = calculator.cal
    System.out.println("Result:
}
```

A dialog box will appear. Enter a name for the new method, let's call it calculateIntermediateResult.

You can choose to have Eclipse create parameters for the extracted method by selecting the appropriate variables, in this case, x and y. Make sure the checkbox for "Save as a method parameter" is checked.

Click the "OK" button to create the new method.



Extract Method

Method name: calculateIntermediateResult

Access modifier: public protected package [✓ private]

Parameters:

Type	Name
int	x
int	y

- [ ] Declare thrown runtime exceptions
- [ ] Generate method comment
- [ ] Replace additional occurrences of statements with method

Method signature preview:

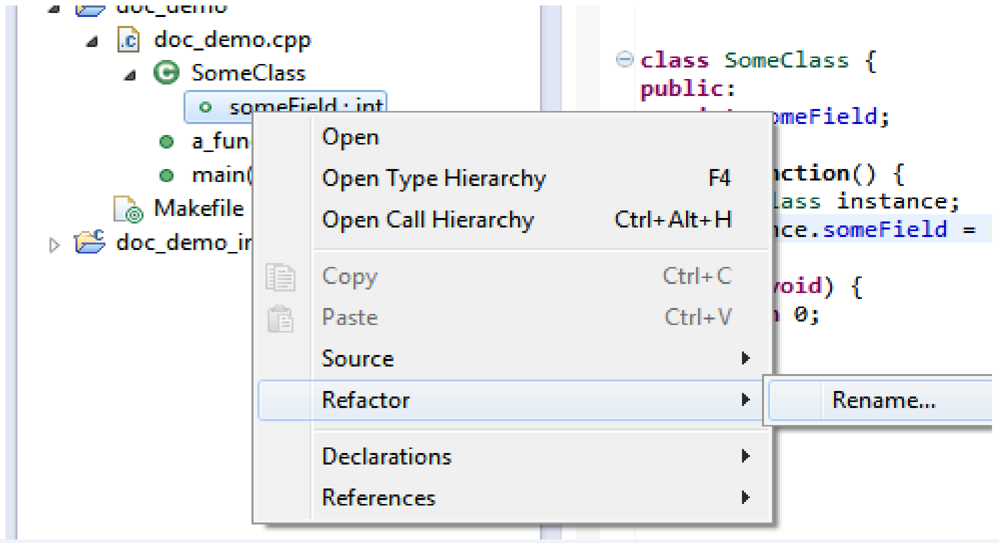
```
private int calculateIntermediateResult(int x, int y)
```

```
package demo1;

public class Calculator {
public int add(int a, int b) {
int result = a + b;
return result;
}
public int multiply(int a, int b) {
int result = a * b;
return result;
}
public int calculate(int x, int y) {
// Some complex calculations here
int intermediateResult = calculateIntermediateResult(x, y);
int finalResult = multiply(intermediateResult, 2);
return finalResult;
}

private int calculateIntermediateResult(int x, int y) {
int intermediateResult = add(x, y);
return intermediateResult;
}
public static void main(String[] args) {
Calculator calculator = new Calculator();
int result = calculator.calculate(5, 3);
System.out.println("Result: " + result);
}
}
```

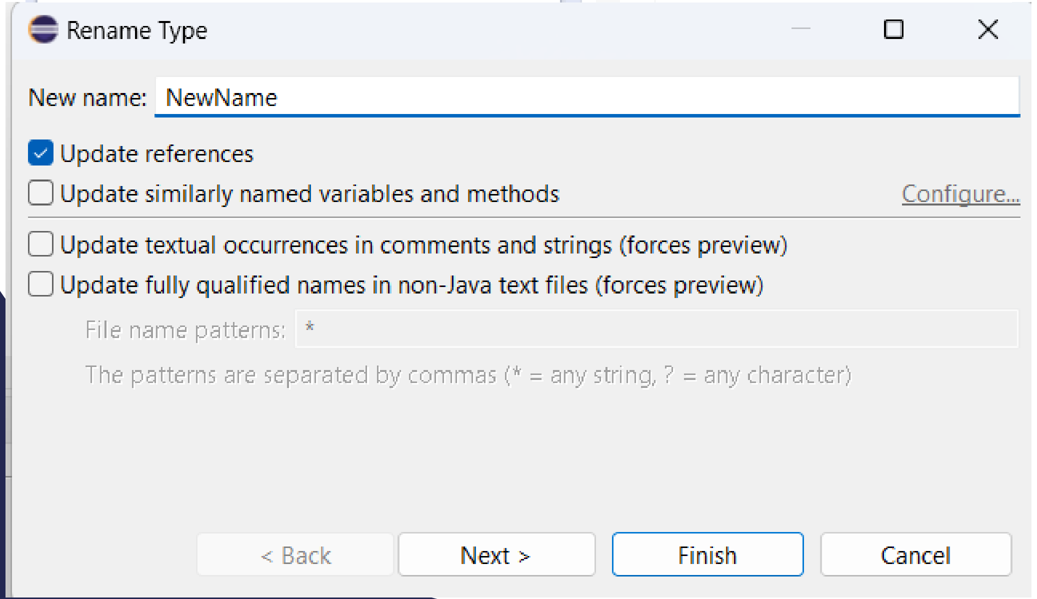
You've successfully extracted a new method, calculateIntermediateResult, which encapsulates the intermediate calculation.

Rename

Rename: If you want to change the name of the type, select "Rename." You can then provide a new name for the type, and Eclipse will automatically update all references to the type within your code.



```
class SomeClass {
public:
          omeField;

        nction() {
        lass instance;
        nce.someField =

        void) {
         0;
```



New name: NewName

- [x] Update references
- [ ] Update similarly named variables and methods
- [ ] Update textual occurrences in comments and strings (forces preview)
- [ ] Update fully qualified names in non-Java text files (forces preview)

File name patterns: *

The patterns are separated by commas (* = any string, ? = any character)

```
public class MyClass {
    private int varOne;
    private String varTwo;

    public MyClass(int a, String b) {
        this.varOne = a;
        this.varTwo = b;
    }

    public void printValues() {
        System.out.println("varOne: " + varOne);
        System.out.println("varTwo: " + varTwo);
    }

    public int calculateSum(int x, int y) {
        return x + y;
    }
}
```

Move

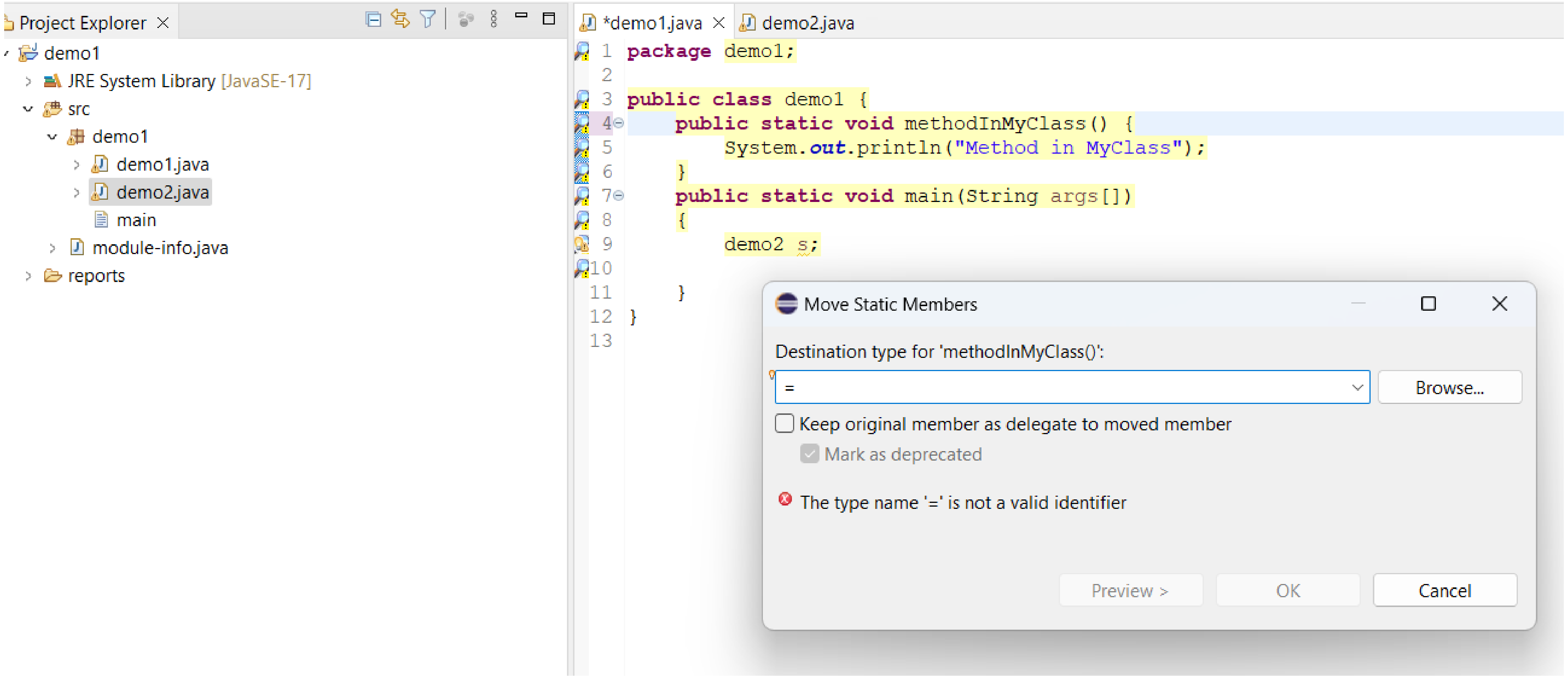
Move: If you want to move the type to a different package or source folder, select "Move." Eclipse will guide you through the process of selecting the destination.

demo1.java

```java
package demo1;


public class demo1 {
public static void methodInMyClass() {
System.out.println("Method in MyClass");
}
public static void main(String args[])
{
demo2 s;
}
}
```

demo2.java

```java
package demo1;


public class demo2 {
public void methodInAnotherClass() {
System.out.println("Method in AnotherClass");
}


}
```


Project Explorer

- demo1
  - JRE System Library [JavaSE-17]
  - src
    - demo1
      - demo1.java
      - demo2.java
      - main
    - module-info.java
  - reports

*demo1.java demo2.java

```java
package demo1;

public class demo1 {
    public static void methodInMyClass() {
        System.out.println("Method in MyClass");
    }
    public static void main(String args[])
    {
        demo2 s;

    }
}
```

Move Static Members

Destination type for 'methodInMyClass()':

=

Browse...

Keep original member as delegate to moved member

Mark as deprecated

The type name '=' is not a valid identifier

Preview > OK Cancel

demo2.java

```
package demo1;


public class demo2 {
public void methodInAnotherClass() {
System.out.println("Method in
AnotherClass");
}


public static void methodInMyClass() {
System.out.println("Method in MyClass");
}


}
```

Extract method

Extract method

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

```
void printOwing() {
  printBanner();

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

Solution
Move this code to a separate new method (or function) and replace the old code with a call to the method.

```
void printOwing() {
  printBanner();
  printDetails(getOutstanding());
}

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

Extract variable

The main reason for extracting variables is to make a complex expression more understandable, by dividing it into its intermediate parts.

How to Refactor

1. Insert a new line before the relevant expression and declare a new variable there. Assign part of the complex expression to this variable.
2. Replace that part of the expression with the new variable.
3. Repeat the process for all complex parts of the expression.

Extract variable

Problem
You have an expression that’s hard to understand.

```
void renderBanner() {
  if ((platform.toUpperCase().indexOf("MAC") > -1)
&&
       (browser.toUpperCase().indexOf("IE") > -1) &&
        wasInitialized() && resize > 0 )
  {
    // do something
  }
}
```

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

```
void renderBanner() {
  final boolean isMacOs =
platform.toUpperCase().indexOf("MAC") > -1;
  final boolean isIE = browser.toUpperCase().indexOf("IE") >
-1;
  final boolean wasResized = resize > 0;

  if (isMacOs && isIE && wasInitialized() && wasResized) {
    // do something
  }
}
```

Inline method

How to Refactor

1.Make sure that the method isn’t redefined in subclasses. If the method is redefined, refrain from this technique.
2.Find all calls to the method. Replace these calls with the content of the method.
3.Delete the method.

Inline methods

Problem: When a method body is more obvious than the method itself, use this technique.
Solution: Replace calls to the method with the method's content and delete the method itself.

Problem

When a method body is more obvious than the method itself, use this technique.

```
class PizzaDelivery {
  // ...
  int getRating() {
    return moreThanFiveLateDeliveries() ? 2 : 1;
  }
  boolean moreThanFiveLateDeliveries() {
    return numberOfLateDeliveries > 5;
  }
}
```

Solution

Replace calls to the method with the method's content and delete the method itself.

```
class PizzaDelivery {
  // ...
  int getRating() {
    return numberOfLateDeliveries > 5 ? 2 : 1;
  }
}
```

Inline Temp

Inline Temp

Problem
You have a temporary variable that's assigned the result of a simple expression and nothing more.

```
boolean hasDiscount(Order order) {
  double basePrice = order.basePrice();
  return basePrice > 1000;
}
```

Solution
Replace the references to the variable with the expression itself.

```
boolean hasDiscount(Order order) {
  return order.basePrice() > 1000;
}
```

Pull Up and Push Down

Pull Up and Push Down: These refactorings are used for moving methods and fields between superclasses and subclasses. "Pull Up" moves members from a subclass to a superclass, while "Push Down" moves members from a superclass to a subclass.

```
class Animal {
    // Common properties and methods for all animals.
}

class Dog extends Animal {
    public void bark() {
        System.out.println("Woof!");
    }
}

class Cat extends Animal {
    public void meow() {
        System.out.println("Meow!");
    }
}
```

pull up the meow method from Cat to the superclass Animal.

Extract interface

Extract Interface: If you want to create an interface from a class, select "Extract Interface." This can be useful for defining a contract that multiple classes need to implement.

- Extracting an interface is a common refactoring technique in Java used to define a new interface based on existing class methods.
- This allows you to abstract the behavior of one or more classes into a shared interface, promoting code flexibility and maintainability.

```
public class MyServiceImpl {
    public void methodOne() {
        // Implementation of methodOne
    }

    public void methodTwo() {
        // Implementation of methodTwo
    }
}
```

extract an interface from MyServiceImpl that includes methodOne and methodTwo.

```
public interface MyService {
    void methodOne();

    void methodTwo();
}

public class MyServiceImpl implements MyService {
    public void methodOne() {
        // Implementation of methodOne
    }

    public void methodTwo() {
        // Implementation of methodTwo
    }
}
```

Extract Superclass

Extract Superclass: If you want to create a common superclass for several classes, select "Extract Superclass." This can help in reducing duplicated code.

```
public class Dog {
    private String name;
    private int age;

    public Dog(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public void bark() {
        System.out.println("Woof!");
    }

    public void fetch() {
        System.out.println("Fetching the ball.");
    }
}
```

```
public class Cat {
    private String name;
    private int age;

    public Cat(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public void meow() {
        System.out.println("Meow!");
    }

    public void sleep() {
        System.out.println("Sleeping...");
    }
}
```

Select one of the classes from which you want to extract a superclass (e.g., Dog).

Step 4: Right-click on the selected class and choose "Refactor" from the context menu, then select "Extract Superclass."

Step 5: A dialog or window will appear, allowing you to configure the extraction process. You can:

Specify the name of the new superclass (e.g., Animal).
Choose the attributes and methods you want to include in the superclass (select name, age, bark, and meow).
Step 6: Click the "OK" or "Finish" button to confirm the extraction.

Step 7: After confirming, your code will be updated as follows:

```java
public class Animal {
    private String name;
    private int age;

    public Animal(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public void bark() {
        System.out.println("Woof!");
    }

    public void meow() {
        System.out.println("Meow!");
    }
}

public class Dog extends Animal {
    public Dog(String name, int age) {
        super(name, age);
    }

    // Dog-specific methods
}
```

```java
public class Cat extends Animal {
    public Cat(String name, int age) {
        super(name, age);
    }

    // Cat-specific methods
}
```

Delete

Delete: If you want to remove a type from your project, select "Delete." Be careful when using this option, as it will permanently delete the type and all its references.

Replace Temp with Query

Problem

You place the result of an expression in a local variable for later use in your code.

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

Solution

Move the entire expression to a separate method and return the result from it. Query the method instead of using a variable. Incorporate the new method in other methods, if necessary.

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

Bad Code Example (Before Refactoring):

```
import java.util.ArrayList;
import java.util.List;

public class Library {
    public List<String> books = new ArrayList<>();
    public List<String> borrowedBooks = new
ArrayList<>();
    public List<String> users = new ArrayList<>();
    public List<String> borrowedBy = new ArrayList<>();

    public void addBook(String book) {
        books.add(book);
    }

    public void addUser(String user) {
        users.add(user);
    }
```

```
public void borrowBook(String book, String user) {
        if (books.contains(book) &&
users.contains(user)) {
            borrowedBooks.add(book);
            borrowedBy.add(user);
            books.remove(book);
        }
    }

    public void returnBook(String book, String user) {
        if (borrowedBooks.contains(book) &&
borrowedBy.contains(user)) {
            books.add(book);
            borrowedBooks.remove(book);
            borrowedBy.remove(user);
        }
    }
```

```
public void printBorrowedBooks() {
        for (int i = 0; i < borrowedBooks.size(); i++) {
            System.out.println(borrowedBooks.get(i) + "
borrowed by " + borrowedBy.get(i));
        }
    }

    public double calculateOverdueFine(int daysOverdue) {
        double fine = 0.0;
        if (daysOverdue > 0) {
            fine = daysOverdue * 0.50; // fixed rate per day
        }
        return fine;
    }
}
```

- Create separate classes for Book, User, and Loan to improve encapsulation and separation of concerns.
- Encapsulate the books, users, and borrowedBooks lists. Use getter and setter methods where necessary.

In Eclipse:

- Extract Class for separating responsibilities into Book, User, and Loan classes.
- Encapsulate Fields to hide internal data (Right-click > Refactor > Encapsulate Field).
- Extract Interface for creating interfaces for fine calculation strategies and user management.

```java
// Book class to encapsulate book details
public class Book {
   private String title;
   private String author;
    public Book(String title, String author) {
      this.title = title;
      this.author = author;
   }
    public String getTitle() {
      return title;
   }
    public String getAuthor() {
      return author;
   }
}
```

```java
// User class to encapsulate user details
public class User {
   private String name;
   private boolean isPremium;

   public User(String name, boolean isPremium) {
      this.name = name;
      this.isPremium = isPremium;
   }
    public String getName() {
      return name;
   }
    public boolean isPremium() {
      return isPremium;
   }
}
```

```
// Loan class to represent a borrowed book
public class Loan {
  private Book book;
  private User user;
  private int daysOverdue;
  public Loan(Book book, User user) {
    this.book = book;
    this.user = user;
    this.daysOverdue = 0;
  }
  public void returnBook() {
    // Return logic here
  }
  public void setDaysOverdue(int days) {
    this.daysOverdue = days;
  }
  public int getDaysOverdue() {
    return daysOverdue;
  }
  public Book getBook() {
    return book;
  }

  public User getUser() {
    return user;
  }
}
```

Object-Oriented Design Concepts

OO design concepts such as classes and objects, inheritance, messages, and polymorphism

Design Classes

- Design classes that refine the analysis classes by providing design detail that will enable the classes to be implemented, and implement a software infrastructure that supports the business solution.
- Various types: User interface classes, Business domain classes, Process classes, Persistent classes, system classes

User Interface (UI) Classes:

These classes represent the components that the end-user interacts with. They manage user inputs, display data, and handle interactions such as button clicks, form submissions, and displaying results.

•**Example**:

- **LoginForm**: A class that represents the user login form where users input their credentials.
- **DashboardUI**: A class that manages the graphical user interface for a dashboard

Business Domain Classes:

These classes encapsulate the core business logic of the system. They represent the entities and operations within the business domain, including the rules, policies, and computations related to the domain.

- **Example**:
  - **Customer**: A class that manages customer details, such as name, contact info, and order history.
  - **Order**: A class responsible for processing customer orders, including validations and applying business rules like discounts or promotions.

Process Classes:

These classes represent the workflows or processes that the system must perform. They coordinate and manage complex activities that may involve multiple domain entities and other system components.

•**Example**:

- **OrderProcessor**: A class that handles the steps involved in processing a customer's order, such as validating the order, checking inventory, and calculating total cost.
- **PaymentProcess**: A class responsible for managing the entire payment process, from payment validation to confirmation.

Persistent Classes:

These classes are responsible for managing the **persistence** (storage) of business domain objects. They interact with databases or file systems to store, retrieve, update, and delete data.

•**Example**:

- **CustomerRepository**: A class responsible for saving and retrieving customer data from the database.
- **OrderDAO**: A Data Access Object (DAO) class responsible for persisting and retrieving order data from a database.

System Classes:

These classes represent the system-level operations and utilities that are necessary for the functioning of the application but are not directly related to the business logic. This could include utility classes, communication with external systems, or hardware interaction.

•**Example**:

- **FileManager**: A class that handles reading from or writing to files in the file system.
- **EmailService**: A system class that handles sending emails to users for notifications

Dependency Inversion

High-Level Modules
These are more **abstract** and typically deal with the overall functionality, rules, or business logic of a system. They are closer to the user or business domain and are more concerned with **what** needs to be done rather than **how** it's done.

In an e-commerce application, a **high-level module** could be:
•A **shopping cart service** that handles adding and removing items, calculating total costs, and managing checkout flows.
•It would use abstract interfaces to interact with the payment gateway, inventory system, or user authentication system, without caring about their specific implementation.
.

Low-level modules: These modules are more **concrete** and handle the **specific details** of system operations, such as interacting with hardware, databases, or third-party services. They focus on **how** things are done and implement the specifics required by high-level modules.

Example:

In the same e-commerce application, a **low-level module** could be:

- A **database module** that handles operations like retrieving product information or updating the user's order history.
- This module would contain specific SQL queries or ORM (Object-Relational Mapping) logic, which implements the abstract database interactions defined by the high-level module.

Abstraction: An abstraction in this context is usually an interface or an abstract class that defines the expected behavior without dictating how that behavior is implemented.

Dependency Inversion Principle (DIP)

- The **Dependency Inversion Principle (DIP)** is one of the five SOLID principles of object-oriented design. It suggests that **high-level modules** (which contain the core business logic) should not depend on **low-level modules** (such as utility classes or data access logic). Instead, both should depend on **abstractions**, such as interfaces or abstract classes.
- In essence, the goal of the Dependency Inversion Principle is to **reduce coupling** between the high-level and low-level components by introducing an abstraction layer, making the codebase more modular, flexible, and maintainable.

Example without DIP (Tight Coupling)
Let's say you have an application where an OrderService directly depends on a MySQLDatabase class to save data.

```
class MySQLDatabase:
    def save_order(self, order):
        # Logic to save order to MySQL database
        print("Order saved to MySQL")

class OrderService:
    def __init__(self):
        self.database = MySQLDatabase()

    def place_order(self, order):
        self.database.save_order(order)
```

The OrderService class is directly dependent on the MySQLDatabase class.If you wanted to change the database (e.g., switch from MySQL to MongoDB), you'd have to modify the OrderService class, which violates open/closed principle and creates a strong coupling.

```
class MySQLDatabase:
    def save_order(self, order):
        # Logic to save order to MySQL database
        print("Order saved to MySQL")

class OrderService:
    def __init__(self):
        self.database = MySQLDatabase()

    def place_order(self, order):
        self.database.save_order(order)
```

Example with DIP (Loose Coupling)
Now, let's apply DIP by introducing an abstraction (an interface) for the database interaction.

```
# Abstraction
class Database:
    def save_order(self, order):
        pass

# Low-level module (depends on the abstraction)
class MySQLDatabase(Database):
    def save_order(self, order):
        print("Order saved to MySQL")
```

```
# Low-level module (another possible implementation)
class MongoDB(Database):
    def save_order(self, order):
        print("Order saved to MongoDB")

# High-level module (depends on the abstraction)
class OrderService:
    def __init__(self, database: Database):
        self.database = database
    def place_order(self, order):
        self.database.save_order(order)
```

- **OrderService** now depends on the Database abstraction, not on the low-level implementation (MySQLDatabase or MongoDB).
- You can easily switch between MySQL and MongoDB without modifying the OrderService.

- This way, the OrderService is flexible, extensible, and loosely coupled with the database layer.

- The **Dependency Inversion Principle** (DIP) aims to reduce tight coupling between high-level and low-level modules by making both depend on abstractions (interfaces or abstract classes).
- This results in more flexible, maintainable, and testable code, and it allows for easier modifications without affecting other parts of the system.

SOLID Principles

The **SOLID principles** are a set of five design principles in object-oriented programming that help developers build maintainable, scalable, and flexible software.

SOLID stands for:
S - Single Responsibility Principle (SRP)
O - Open/Closed Principle (OCP)
L - Liskov Substitution Principle (LSP)
I - Interface Segregation Principle (ISP)
D - Dependency Inversion Principle (DIP)

Single Responsibility Principle (SRP)

Each class should do one thing well. If a class has multiple responsibilities (e.g., handling user authentication and data storage), it becomes harder to maintain because changes in one responsibility may affect the others.

- **Example**: A class that handles user authentication should not also manage logging errors. Instead, there should be separate classes: one for authentication and one for logging.

Open/Closed Principle (OCP)
Software entities (classes, modules, functions) should be **open for extension** but **closed for modification**.

You should be able to add new functionality to a system (by extending it) without changing the existing code. This reduces the risk of introducing bugs when modifying working code.

Example: If a system needs to support new file formats, you should add new classes to handle these formats, rather than modifying existing file handling code. This can be achieved using **polymorphism** or **interfaces**.

Liskov Substitution Principle (LSP)

Subtypes must be substitutable for their base types. In other words, objects of a derived class should be able to replace objects of the base class without altering the correctness of the program.

If a class B is a subclass of class A, you should be able to use an object of B wherever an object of A is expected, without the program breaking or behaving unexpectedly.

If you have a class Bird and a subclass Penguin, the Penguin should be able to replace Bird in any place where Bird is expected. If Penguin cannot fly, while Bird can, then Penguin breaks the LSP, and there may need to be a redesign (e.g., introducing an abstract class for flight behavior).

Interface Segregation Principle (ISP)

Clients should not be forced to depend on interfaces they do not use. Essentially, large interfaces should be broken down into smaller, more specific ones.

It’s better to have many small, specialized interfaces rather than one large interface that forces classes to implement methods they don’t need.

Instead of having one large UserOperations interface that includes methods like login(), register(), deleteAccount(), and updateProfile(), you should break it down into smaller interfaces such as Authentication, AccountManagement, etc. A class should only implement the interfaces it needs.

Dependency Inversion Principle (DIP)

High-level modules should not depend on **low-level modules**. Both should depend on **abstractions** (e.g., interfaces). Also, **abstractions should not depend on details**; details should depend on abstractions.

Instead of having high-level components (like business logic) directly depend on low-level components (like database access), both should rely on abstractions, which can be interfaces or abstract classes. This allows for better flexibility and easier maintenance.

Instead of a business logic class depending on a concrete MySQLDatabase class, it should depend on a DatabaseInterface. This way, you can swap out MySQLDatabase with PostgreSQLDatabase or MongoDBDatabase without modifying the business logic.

Benefits of SOLID Principles:

1. **Maintainability**: Adhering to SOLID principles ensures that the code is easier to understand, maintain, and modify over time.
2. **Scalability**: By separating responsibilities and allowing for extension without modification, systems are more adaptable to changes and new requirements.
3. **Testability**: Code that follows SOLID principles is often easier to test because components are decoupled and follow predictable behaviors.
4. **Loose Coupling**: SOLID principles help in reducing dependencies between components, making the system more modular and reducing the ripple effect of changes.

Design for test

- Designing for testability is an important aspect of software architecture and development.
- Testable design ensures that the system's components can be easily tested in isolation and that issues are detected early.
- This approach leads to more reliable, maintainable, and scalable software.
- Testable design often incorporates several principles and practices that make unit testing, integration testing, and system testing easier.

That’s it