Chapter 3: Clean code & SOLID

📌 What is Clean Code?

Clean Code refers to code that is easy to read, understand, and maintain. It doesn't just work — it communicates clearly what it does and why. Clean code helps teams collaborate more efficiently and reduces bugs in the long run.

"Clean code always looks like it was written by someone who cares." — *Robert C. Martin (Uncle Bob)*

🧠 Why is Clean Code Important?

- Easier for others (and your future self) to understand.
- Faster to debug and enhance.
- Encourages best practices in software design.
- Reduces technical debt.

🧼 Clean Code – Use Intention-Revealing Names

1. Use Intention-Revealing Names

Names should clearly express the purpose of a variable, function, or

class.

Bad Example:

```
public List<int[]> getThem() {
   List<int[]> list1 = new ArrayList<>();
   for (int[] x : theList) {
      if (x[0] == 4)
         list1.add(x);
   }
   return list1;
}
```

Good Example:

```
public List<int[]> getFlaggedCells() {
   List<int[]> flaggedCells = new ArrayList<>();
   for (int[] cell : gameBoard) {
      if (cell[STATUS_VALUE] == FLAGGED)
         flaggedCells.add(cell);
   }
   return flaggedCells;
}
```

Best Example (Using Cell class):

```
public List<Cell> getFlaggedCells() {
   List<Cell> flaggedCells = new ArrayList<>();
   for (Cell cell : gameBoard) {
```

```
        if (cell.isFlagged())
            flaggedCells.add(cell);
    }
    return flaggedCells;
}
```

---

2. Avoid Disinformation

Do not use misleading or confusing names.

Bad Example:

```
int O = 0, l = 1;
if (O == l) {
    System.out.println("Hard to read");
}
```

Good Example:

```
int zero = 0, one = 1;
if (zero == one) {
    System.out.println("Clear to read");
}
```

---

3. Use Meaningful Distinctions

Avoid names that differ only slightly or without meaning.

Bad Example:

```
public static void copyChars(char[] a1, char[] a2) {
   for (int i = 0; i < a1.length; i++) {
      a2[i] = a1[i];
   }
}
```

Good Example:

```
public static void copyChars(char[] source, char[] destination) {
   for (int i = 0; i < source.length; i++) {
      destination[i] = source[i];
   }
}
```

———

4. Eliminate Noise

Avoid unnecessary or redundant words in names.

Bad:

```
String customerNameString = "John";
```

Good:
String customerName = "John";

———

5. Use Searchable Names

Avoid magic numbers or meaningless constants.

Bad:
for (int i = 0; i < 7; i++) { ... }

Good:
final int MAX_CLASSES = 7;
for (int i = 0; i < MAX_CLASSES; i++) { ... }

———

6. Class & Method Naming

Use nouns for class names and verbs for method names.

Class Example:
public class InvoiceManager { ... }

Method Example:
public void calculateTax() { ... }

———

7. Avoid Encodings

Do not include type information in variable names (e.g., Hungarian Notation).

Bad:
int iAge;

Good:
int age;

———

8. Interfaces and Implementations

Do not prefix interface names with “I”.

Preferred:
interface ShapeFactory { ... }
class DefaultShapeFactory implements ShapeFactory { ... }

---

9. Use Pronounceable Names

Choose names that can be easily pronounced and discussed.

Bad:
```
class DtaRcrd102 { ... }
```

Good:
```
class CustomerRecord { ... }
```

---

10. Avoid Mental Mapping

Use descriptive names instead of cryptic abbreviations.

Bad:
```
String r = "https://api.example.com";
String p = "id=1";
```

Good:
```
String apiUrl = "https://api.example.com";
String queryParams = "id=1";
```

---

11. Use Solution & Problem Domain Names

Use names from the business or technical domain.

Example (Solution Domain):
public class InvoiceGenerator { ... }

Example (Problem Domain):
public class Customer { ... }

---

12. Don’t Be Cute

Avoid slang, jokes, or non-standard terms in code.

Bad:
killAllZombies();

Good:
terminateProcesses();

---

13. Pick One Word Per Concept

Be consistent with terminology across your codebase.

Bad:
getUser(); fetchUser(); retrieveUser();

Good:
getUser();

———

14. Add Meaningful Context

Add context to make names clear and specific.

Bad:
int state;

Good:
String addressState;
String customerFirstName;

📘 Clean Code - Functions

1. Keep Functions Small

- Functions should be small.
- If a function is more than 10 lines, start questioning its design.
- Smaller functions are easier to test and understand.
- Your function should be readable at a glance.

2. Bad Example – Long Method

This method is doing too much:

```
public static void addDataToFile(String path, Person person) {
   FileInputStream readData = new FileInputStream(path);
   ObjectInputStream readStream = new ObjectInputStream(readData);
   ArrayList people = (ArrayList<Person>) readStream.readObject();
   readStream.close();

   people.add(person);

   FileOutputStream writeData = new FileOutputStream(path);
   ObjectOutputStream writeStream = new
ObjectOutputStream(writeData);
   writeStream.writeObject(people);
   writeStream.flush();
   writeStream.close();

   System.out.println(people.toString());
```

```
}
```

3. Refactor into Small Functions

Step 1: Extract helper methods:

```
private static ArrayList readDataFromFile(String path) {
    FileInputStream readData = new FileInputStream(path);
    ObjectInputStream readStream = new ObjectInputStream(readData);
    ArrayList people = (ArrayList<Person>) readStream.readObject();
    readStream.close();
    return people;
}

private static void writeDataToFile(String path, ArrayList people) {
    FileOutputStream writeData = new FileOutputStream(path);
    ObjectOutputStream writeStream = new
ObjectOutputStream(writeData);
    writeStream.writeObject(people);
    writeStream.flush();
    writeStream.close();
}

private static void printPeople(ArrayList people) {
    System.out.println(people.toString());
}
```

Step 2: Simplify the main function:

```
public static void addDataToFile(String path, Person person) {
   ArrayList people = readDataFromFile(path);
   people.add(person);
   writeDataToFile(path, people);
   printPeople(people);
}
```

4. Do One Thing

- A function should do only one thing.
- If you can extract another function from it, it’s doing more than one thing.

5. Better Separation of Responsibilities

Refactor the add logic into a separate function:

```
public static ArrayList addPersonToList(Person person, ArrayList
people) {
   people.add(person);
   return people;
}
```

Then refactor the save function:

```
public static void savePeopleToFile(String path, ArrayList people) {
    writeDataToFile(path, people);
}
```

And the main:

```
public static void main(String[] args) {
    Person p = new Person("Ahmed", "Hesham", 1990);
    String path = "peopledata.ser";

    ArrayList people = readDataFromFile(path);
    addPersonToList(p, people);
    savePeopleToFile(path, people);

    ArrayList afterAdd = readDataFromFile(path);
    printPeople(afterAdd);
}
```

6. One Level of Abstraction per Function

Bad Example (mixing abstraction levels):

```
private static void printNumbersOneToThirtyFive() {
    printNumbersOneToTen(); // high-level

    // low-level starts
```

```
    for (int i = 11; i < 21; i++) {
        System.out.println(i);
    }
    System.out.println(21);
    if (22 % 11 == 0) {
        System.out.println(22);
        for (int i = 23; i < 31; i++) {
            System.out.println(i);
            if (i == 30) {
                for (int j = 31; j < 36; j++) {
                    System.out.println(j);
                }
            }
        }
    }
}
```

Better (same level of abstraction):

```
private static void printNumbersOneToThirtyFive() {
    printNumbersOneToTen();
    printNumbersElevenToTwenty();
    printNumberTwentyOne();
    printNumbersTwentyTwoToThirtyFive();
}
```

7. Minimize Function Arguments

- Prefer 0-1 argument functions.
- Avoid functions with more than 2 arguments.
- Too many arguments → hard to understand & test.

Example:

```
// Too many parameters
public void sendRequest(String ip, int port, String user, String password, String body);

// Better: wrap them in a class
public void sendRequest(Request request);
```

8. Avoid Flag Arguments

```
// Bad
public void printAddress(boolean full) {
    if (full)
        printLongAddress();
    else
        printShortAddress();
}

// Better
public void printShortAddress() { ... }
public void printLongAddress() { ... }
```

9. Small if/else Blocks
- Keep blocks in if, else, while as short as possible.
- Ideally just one line, usually calling a clearly named function.

Example:

```
if (user.isActive()) {
   sendWelcomeEmail(user);
}
```

10. Replace Conditionals with Polymorphism

Bad (using switch):

```
public class Vehicle {
   private Type type;
   enum Type { BICYCLE, BIKE, CAR, BUS }
   enum ParkingSpotSize { SMALL, MEDIUM, LARGE, XL }

   public ParkingSpotSize spaceRequiredForParking() {
      switch (type) {
         case BICYCLE: return ParkingSpotSize.SMALL;
         case BIKE: return ParkingSpotSize.MEDIUM;
         case CAR: return ParkingSpotSize.LARGE;
         case BUS: return ParkingSpotSize.XL;
```

```
        }
    }

    public BigDecimal parkingCharges() {
        switch (type) {
            case BICYCLE: return BigDecimal.valueOf(10);
            case BIKE: return BigDecimal.valueOf(20);
            case CAR: return BigDecimal.valueOf(50);
            case BUS: return BigDecimal.valueOf(100);
        }
    }
}
```

Better: Use Polymorphism

Define an interface:

```
public interface Vehicle {
    ParkingSpotSize spaceRequiredForParking();
    BigDecimal parkingCharges();

    enum ParkingSpotSize { SMALL, MEDIUM, LARGE, XL }
}
```

Implement per class:

```
public class Car implements Vehicle {
    public ParkingSpotSize spaceRequiredForParking() {
        return ParkingSpotSize.LARGE;
    }

    public BigDecimal parkingCharges() {
        return BigDecimal.valueOf(50);
    }
}
```

📘 Clean Code - Error Handling

1. Use Exceptions Rather Than Return Codes

- Avoid returning special values like `null`, `-1`, or `false` to indicate an error.
- Throw exceptions instead to make error handling explicit and avoid hidden bugs.

❌ Bad Example (using return code):

```
public int getUserAge(String username) {
    if (!userExists(username)) {
        return -1;
    }
    return findUser(username).getAge();
```

```
}
```

✅ **Good Example (using exception):**

```
public int getUserAge(String username) {
    if (!userExists(username)) {
        throw new UserNotFoundException("User not found: " +
username);
    }
    return findUser(username).getAge();
}
```

---

2. Provide Meaningful Exception Messages

- Exception messages should help identify the issue quickly.
- Include important details like input values or operation names.

✅ **Example:**

```
throw new FileNotFoundException("File not found: " + filePath);
```

---

3. Don’t Use Exceptions for Control Flow

- Don’t use exceptions to handle expected situations.
- Use regular control structures like `if-else` instead.

❌ **Bad Example:**

```
try {
   int age = Integer.parseInt(input);
} catch (Exception e) {
   age = 0;
}
```

✅ **Good Example:**

```
if (input.matches("\\d+")) {
   int age = Integer.parseInt(input);
} else {
   age = 0;
}
```

---

4. Use Custom Exceptions When Appropriate

- Avoid using generic exceptions like `Exception` or `RuntimeException` for everything.
- Create custom exceptions that describe the specific error.

✅ **Example:**

```
public class InvalidUserInputException extends RuntimeException {
   public InvalidUserInputException(String message) {
      super(message);
   }
```

}

---

5. Define Exception Hierarchies
- Organize related exceptions under a common parent class for cleaner handling.

✅ Example:
```
public class ApplicationException extends Exception { }

public class DatabaseException extends ApplicationException { }

public class NetworkException extends ApplicationException { }
```

---

6. Clean Up Resources in finally or Use try-with-resources
- Ensure that files, streams, or database connections are always closed, even when an error occurs.

✅ Example using try-with-resources:
```
try (FileInputStream fis = new FileInputStream("data.txt")) {
    // read data
} catch (IOException e) {
    // handle error
```

```
}
```

---

7. Don’t Swallow Exceptions

- Catching an exception and doing nothing is a bad practice.
- At the very least, log it or rethrow it.

❌ **Bad Example:**

```
try {
    doSomething();
} catch (Exception e) {
    // nothing here!
}
```

✅ **Good Example:**

```
try {
    doSomething();
} catch (Exception e) {
    System.err.println("Error occurred: " + e.getMessage());
    e.printStackTrace();
}
```

---

8. Wrap Low-Level Exceptions

- When catching low-level exceptions, wrap them in higher-level custom exceptions with meaningful context.

✅ **Example:**

```
try {
    connectToDatabase();
} catch (SQLException e) {
    throw new DatabaseException("Failed to connect to database", e);
}
```

📘 Entity-Attribute-Value (EAV) Model

🔷 What is EAV?

The EAV model is a data modeling technique used to represent entities with a variable number of attributes, especially when the attributes differ significantly across records and are not known in advance.

Instead of having a table with many columns (one for each attribute), we store data in a more flexible structure:

- **Entity**: the item or object (e.g., a product or patient).
- **Attribute**: a property or characteristic (e.g., color, weight, temperature).

- **Value**: the value for that attribute.

This model is also called **Open Schema** or **Vertical Schema**.

🔷 Why Use EAV?

✅ When the number of attributes is very large and sparse (most columns would be null).
✅ When attributes are dynamic or user-defined.
✅ Useful in medical records, product catalogs, sensor data, etc.

🔷 EAV Table Structure (Generic Schema)

entity_id	attribute	value
1	name	"Laptop"
1	price	"1000"
1	weight	"1.5kg"
2	name	"Smartphone"
2	screen_size	"6.1 inch"

So instead of having a Product table like:

id	name	price	weight	screen_size
1	Laptop	1000	1.5kg	NULL
2	Smartphone	NULL	NULL	6.1 inch

We store data in rows instead of columns using EAV.

🔷 **Java Example for EAV Representation**

```
// Entity class
public class Entity {
    private int id;
    private List<AttributeValue> attributes;

    // constructor, getters, setters
}

// AttributeValue class
public class AttributeValue {
    private String attribute;
    private String value;

    // constructor, getters, setters
```

```
}

// Usage Example
Entity product = new Entity();
product.setId(1);
product.setAttributes(Arrays.asList(
    new AttributeValue("name", "Laptop"),
    new AttributeValue("price", "1000"),
    new AttributeValue("weight", "1.5kg")
));
```

🔷 **Disadvantages of EAV**

❌ **Querying becomes more complex (especially for filtering or joining).**
❌ **Data types are not enforced (all values usually stored as strings).**
❌ **Validation and constraints are harder to implement.**
❌ **Difficult to use indexes effectively.**

📘 **SOLID Principles**

The SOLID principles are a set of five design principles intended to make software designs more understandable, flexible, and maintainable. Each letter stands for a principle:

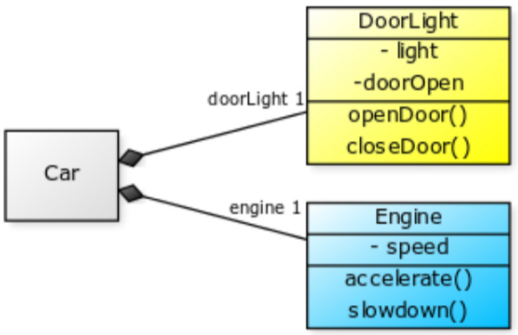
1️⃣ **S - Single Responsibility Principle (SRP)**
2️⃣ **O - Open/Closed Principle (OCP)**
3️⃣ **L - Liskov Substitution Principle (LSP)**
4️⃣ **I - Interface Segregation Principle (ISP)**
5️⃣ **D - Dependency Inversion Principle (DIP)**

1️⃣ Single Responsibility Principle (SRP)

"A class should have one, and only one, reason to change."

🔶 Problem:

A class doing too many things (e.g., handling business logic and saving to a file).



🔶 **Java Example:**

```
// Bad Example
public class Report {
    public void generateReport() {
        // logic to generate report
    }

    public void saveToFile(String filename) {
        // logic to save report to file
    }
}
```

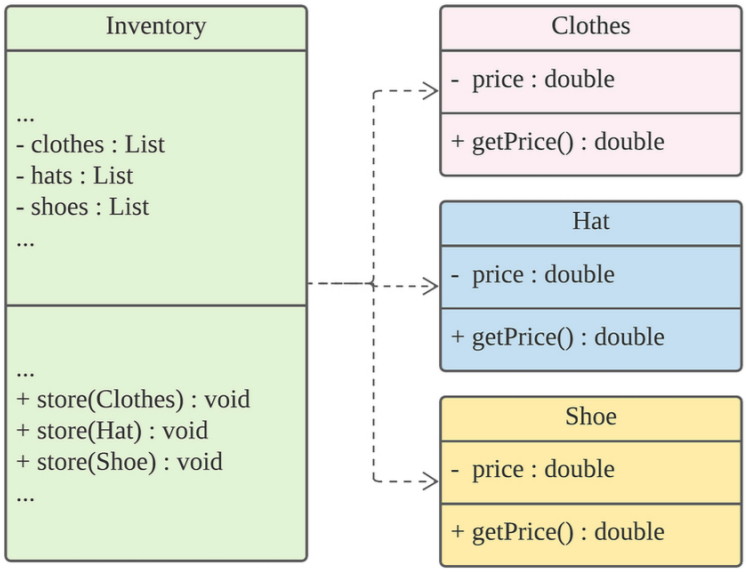
```
// Good Example - Separate Responsibilities
public class Report {
    public void generateReport() {
        // logic to generate report
    }
}

public class ReportSaver {
    public void saveToFile(Report report, String filename) {
        // logic to save report to file
    }
}
```

2️⃣ Open/Closed Principle (OCP)

"Software entities should be open for extension, but closed for modification."

🔶 **Problem:**

Modifying existing code every time a new behavior is added.



🔶 **Java Example:**

```
// Bad Example
public class DiscountCalculator {
    public double calculateDiscount(String customerType) {
        if (customerType.equals("Regular"))
            return 0.1;
        else if (customerType.equals("Premium"))
            return 0.2;
        return 0.0;
    }
}

// Good Example - Use Polymorphism
public interface DiscountStrategy {
```

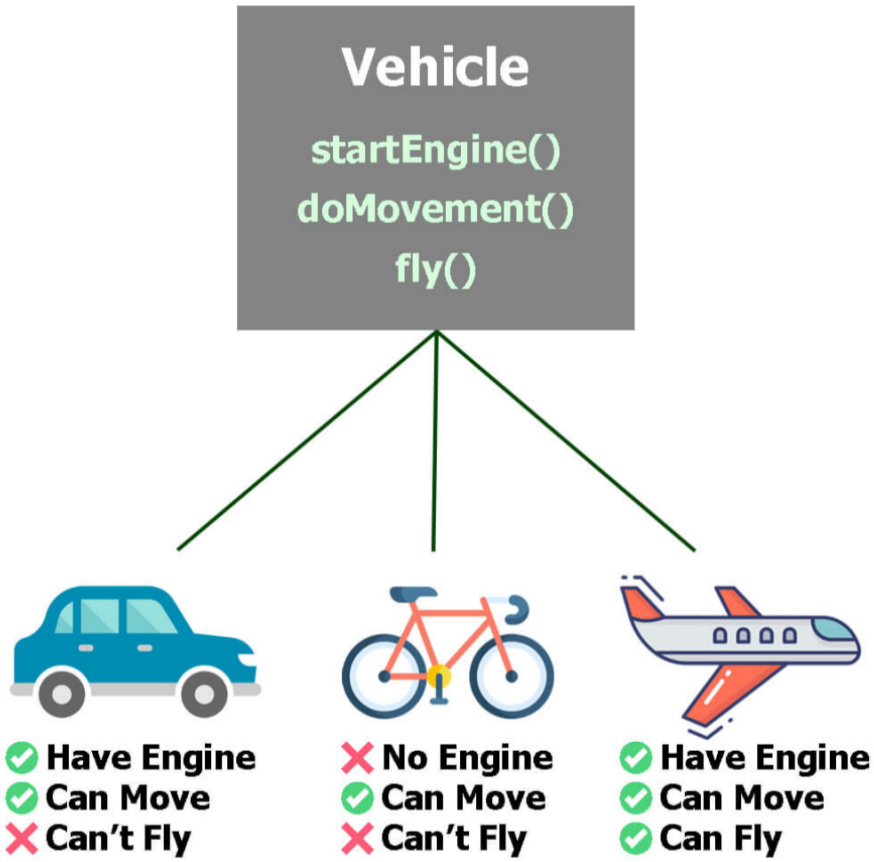
```
    double getDiscount();
}

public class RegularDiscount implements DiscountStrategy {
    public double getDiscount() { return 0.1; }
}

public class PremiumDiscount implements DiscountStrategy {
    public double getDiscount() { return 0.2; }
}

public class DiscountCalculator {
    public double calculateDiscount(DiscountStrategy strategy) {
        return strategy.getDiscount();
    }
}
```

3️⃣ Liskov Substitution Principle (LSP)

"Subtypes must be substitutable for their base types."

🔶 Problem:

Subclass modifies behavior in a way that breaks the parent class expectations.



🔶 **Java Example:**

```
// Bad Example
public class Bird {
    public void fly() {
        System.out.println("Flying");
    }
}

public class Ostrich extends Bird {
```

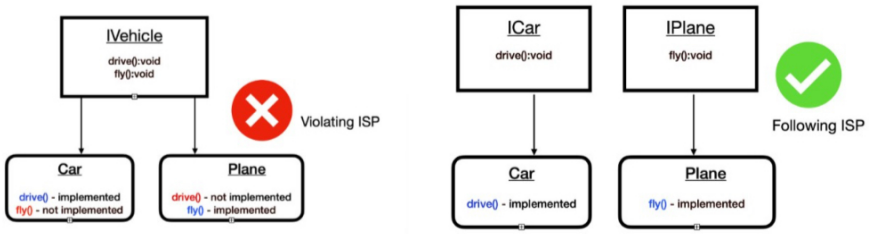
```
    public void fly() {
        throw new UnsupportedOperationException("Ostrich can't fly!");
    }
}

// Good Example - Redesign Hierarchy
public interface Bird {}
public interface FlyingBird extends Bird {
    void fly();
}

public class Sparrow implements FlyingBird {
    public void fly() {
        System.out.println("Sparrow flying");
    }
}

public class Ostrich implements Bird {
    // No fly method needed
}
```

4️⃣ Interface Segregation Principle (ISP)

"Clients should not be forced to depend on interfaces they do not use."

🔶 Problem:

Fat interfaces force implementing unnecessary methods.



🔶 Java Example:

```
// Bad Example
public interface Machine {
    void print();
    void scan();
    void fax();
}

public class OldPrinter implements Machine {
    public void print() { }
    public void scan() { throw new UnsupportedOperationException(); }
    public void fax() { throw new UnsupportedOperationException(); }
}

// Good Example - Split interfaces
```

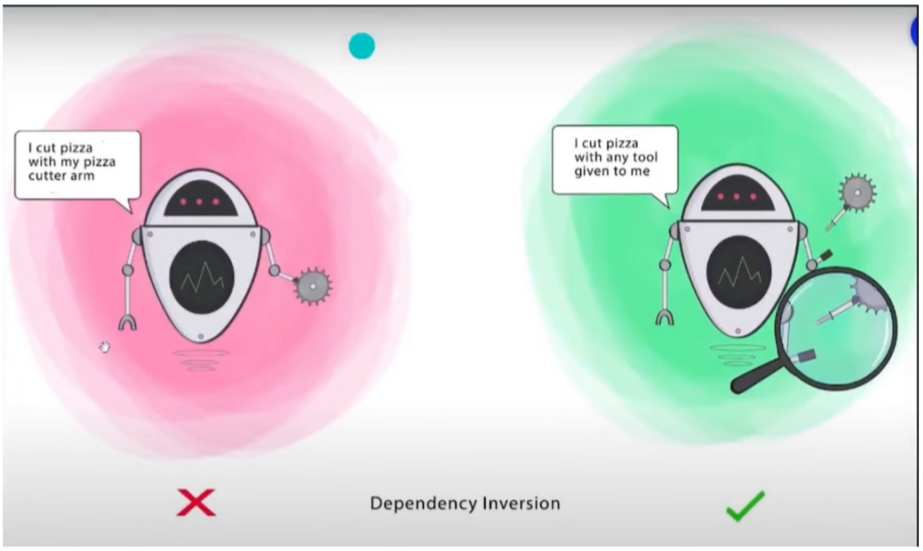
```
public interface Printer {
    void print();
}

public interface Scanner {
    void scan();
}

public class SimplePrinter implements Printer {
    public void print() {
        System.out.println("Printing...");
    }
}
```

5️⃣ Dependency Inversion Principle (DIP)

"High-level modules should not depend on low-level modules. Both should depend on abstractions."

🔶 **Problem:**

Tight coupling between high-level and low-level classes.



🔶 **Java Example:**

```
// Bad Example
public class MySQLDatabase {
    public void connect() { }
}

public class DataManager {
    private MySQLDatabase db = new MySQLDatabase();

    public void saveData() {
        db.connect();
        // save data
    }
}
```

```
// Good Example - Depend on abstraction
public interface Database {
    void connect();
}

public class MySQLDatabase implements Database {
    public void connect() { }
}

public class DataManager {
    private Database db;

    public DataManager(Database db) {
        this.db = db;
    }

    public void saveData() {
        db.connect();
    }
}
```
