System Implementation, Configuration Management & Risk Management

Version Control:

Definition:
Version control is a system that manages the different versions of software configuration objects (like code, documents, etc.) during the software development process.

Four Major Capabilities of a Version Control System:

1. **Project Database:**
    - Stores all configuration objects (files, code, etc.) used in the software project.
2. **Version Management:**
    - Keeps track of all versions of each configuration object. Every time a change is made, a new version is created and saved.
3. **Make Facility:**
    - Helps in building (or constructing) a specific version of the software using the stored versions of objects.
4. **Issue Tracking:**
    - Tracks issues like bugs, enhancements, and changes. It's often integrated with version control to keep track of what changes are linked to specific issues or bugs.

Change Set:

- **Change Set:** A change set is a collection of all changes needed to create a specific version of the software.
- **Named Change Sets:** Each change set can be named for easy identification. You can refer to these names when you need to build or retrieve that version of the software.
- **Constructing a Version Using Change Sets:**
    - When building a version, you specify which change sets (by their names) should be included to get the correct version of the software.

Modeling Approach for Building New Versions:

1. **Template for Building Version:**
    - A predefined structure or blueprint for building a version of the software.
2. **Construction Rules:**
    - Rules that describe how to integrate changes and build the software correctly.
3. **Verification Rule:**

- After building a version, this rule ensures that the version is correct and works as expected (e.g., through tests or checks).

Change Control

- **Too much change control = problems**: If you have too many rules about changing things, it can slow everything down.
- **Uncontrolled change = chaos**: In big software projects, if changes are not controlled, things can get messy fast.
- **Change control combines human processes and tools**: You need both people (procedures) and software tools to manage changes in large projects.

Key Terms in Change Control

- **Engineering Change Order (ECO)**: This is a formal request to change something in the software. It's like an official order to make a change, and it gets tracked.

Elements of Change Management

1. **Access Control**:
    - Decides who can change what in the software.
    - Only authorized people (engineers) can access and modify certain parts of the software.
2. **Synchronization Control**:
    - Ensures that when two people are making changes at the same time, their work doesn't overwrite each other.
3. **Informal Change Control**:
    - In this case, developers can change things directly without asking for permission.
4. **Project-Level Change Control**:
    - Before making any change, the developer must get approval from the project manager.
5. **Formal Change Control**:
    - Used when the software is released to customers.
    - Changes must go through a formal process to make sure they don’t affect the existing product or cause new problems.

Component diagram

- Component diagram shows components, provided and required interfaces, ports, and relationships between them.
- It does not describe the functionality of the system but it describes the components used to make those functionalities.
- Component diagrams can also be described as a static implementation view of a system. Component diagram
- The component diagram's main purpose is to show the structural relationships between the components of a system

Figure 1: This simple component diagram shows the Order System's

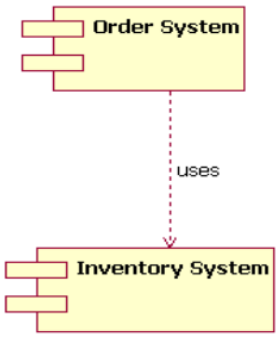


Figure 2: The different ways to draw a component's name compartment

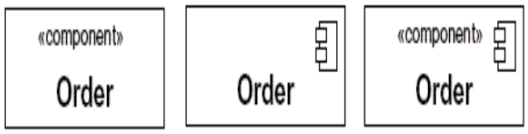


Figure 3: The additional compartment here shows the interfaces

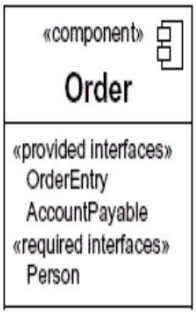
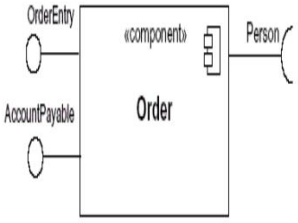


Figure 4: An alternative approach (compare with Figure 3) to showing a component's provided/required interfaces using interface symbols



The assembly connector bridges component's required interface (Component1) with the provided interface of another component (Component2); this allows one component to provide the services that another component requires

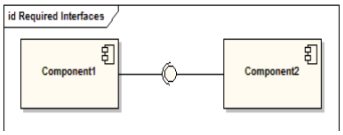
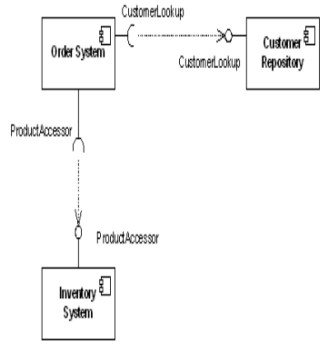
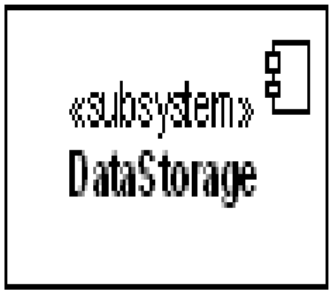


Figure 5: A component diagram that shows how the Order System component depends on other components

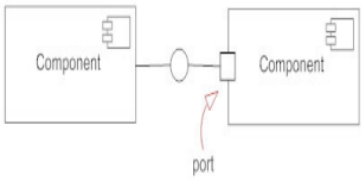


- Example of subsystem element

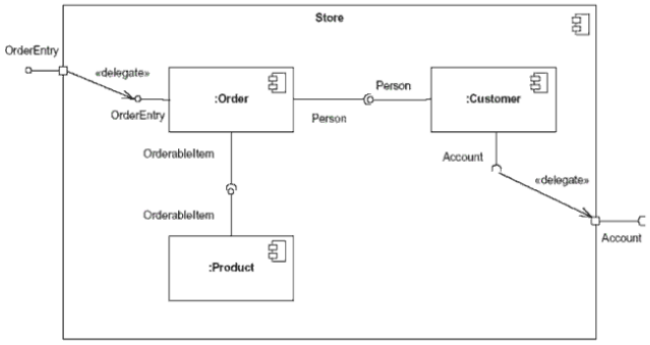


Port

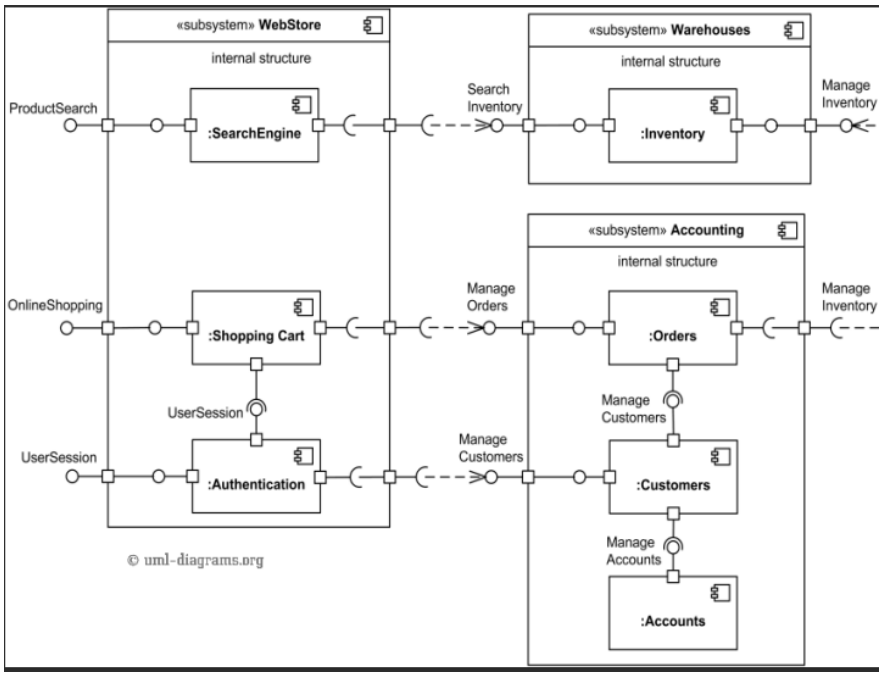
• Ports are represented using a square along the edge of the system or a component. A port is often used to help expose required and provided interfaces of a component



- This component's inner structure is composed of other components



Online shopping UML component diagram example with three related subsystems - WebStore, Warehouses, and Accounting



Deployment Diagram

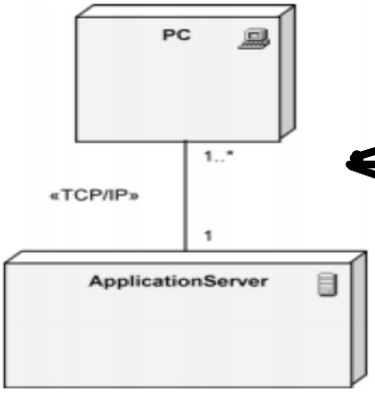
- A deployment diagram is used to show the allocation of artifacts to nodes in the physical design of a system.
- Deployment Diagrams are made up of a graph of nodes connected by communication associations to show the physical configuration of the software and hardware

Essential Elements of Deployment Diagram

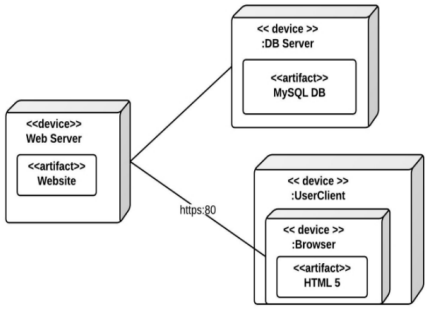
1. **Artifacts**
    - **Artifact** is a physical item that represents a piece of the software design.
    - It could be a **source file**, a **document**, or anything that's part of the code or design.
    - **Notation**: The artifact is shown as a **rectangle** with the label <<artifact>>.
2. **Nodes**
    - A **Node** represents a **hardware resource** that has memory and processing power. This is where the software (artifacts) runs.
    - **Notation**: The node is shown as a **3D cube** (like a box with depth).
    - **Important**: Nodes are **hardware** (computers, servers), not software.
3. **Connections**
    - **Connections** show how nodes communicate with each other using **messages** or **signals**.
    - **Notation**: The communication path between nodes is shown as a **solid line**.
    - By default, communication paths are **bidirectional** (both nodes can send and receive messages). If it's **unidirectional** (only one-way communication), an **arrow** is used.
    - You can add labels like <<http>> or <<TCP/IP>> to show the type of connection (e.g., web or network connection).
    - **Multiplicity** shows how many instances of a node are involved in the communication (e.g., one or many).

Summary

- **Artifacts** represent software components.
- **Nodes** represent hardware that runs the software.
- **Connections** represent how nodes communicate, and they can be labeled with the type of connection.



Notice that the box is a 3D box



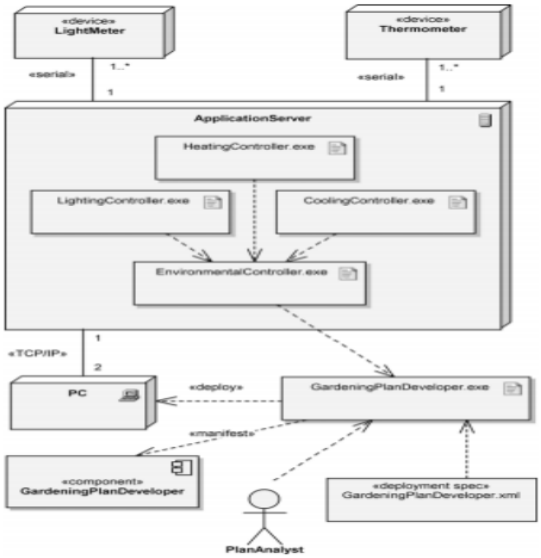


Figure 5–19 The Deployment Diagram for `EnvironmentalControlSystem`

Mapping Models to Code

- **Transformation**: This means converting a model (like a design or blueprint) into code while keeping all the important features, such as **functionality**, intact. The goal is to improve some aspect of the model, like **modularity** or **performance**, without losing its original purpose.

Transformation Activities:

1. **Optimization**
    - **Purpose**: Improve **performance** of the system by making it more efficient.
    - **Examples**:
        - **Reducing multiplicities**: Cutting down on unnecessary copies or instances.
        - **Adding redundant associations**: Creating extra links between components for faster access.
        - **Adding derived attributes**: Adding extra data or shortcuts to speed up access to objects.
2. **Realizing Associations**
    - **Purpose**: Turn the **associations** (connections between objects) in the model into real code constructs.
    - **Example**:
        - Using **references** or **collections of references** in the code to represent the relationships between objects.
3. **Mapping Contracts to Exceptions**
    - **Purpose**: Define what should happen when certain **rules** (contracts) are broken.
    - **Example**:
        - If a rule is violated (like invalid input), the system should **raise an exception** (an error message or stop the process).
4. **Mapping Class Models to a Storage Schema**
    - **Purpose**: Decide where and how the data should be stored.
    - **Example**:
        - Choose between different **storage strategies** (like a **database** or **flat files**) and then map the **class model** (structure) to a **storage schema** (like a database table).

Four Types of Transformations:

1. **Model Transformations**

- This involves changing an **object model** to another **object model**. The goal is to make the model better meet the requirements.
- **Changes** can include adding, removing, or renaming:
  - **Classes**, **operations**, **associations**, or **attributes**.

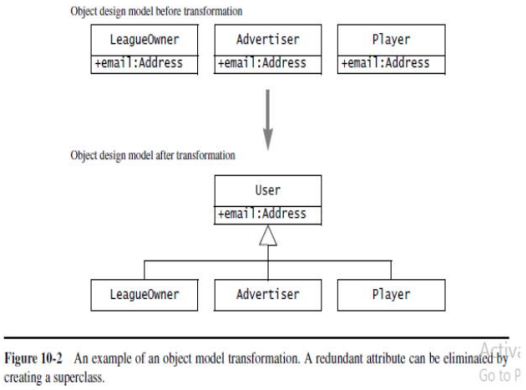


Figure 10-2 An example of an object model transformation. A redundant attribute can be eliminated by creating a superclass.

2. Refactoring

- **What is Refactoring?**
  - Refactoring is a process where the **source code** is changed or reorganized to improve its **readability** or **modifiability** without changing what the system actually **does** (its behavior).
  - It's like cleaning up the code to make it easier to understand or work with, but the software still behaves the same.
- **Goal of Refactoring**:
  - The aim is to **improve the design** of a working system, making the code cleaner, more efficient, and easier to maintain over time.
- **How Refactoring is Done**:
  - Refactoring happens in **small steps**, usually one change at a time. After each small change, the system is tested to make sure it still works correctly.
  - This process helps avoid big, risky changes and ensures the system stays stable during the improvement process.
- **Example of Refactoring in Action**:
  - The **object model transformation** (from the earlier "Mapping Models to Code") can be seen as a series of **refactorings** where the system is gradually improved through small, controlled changes.

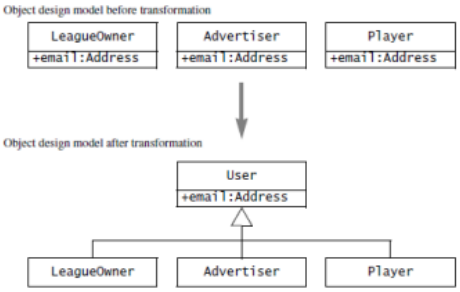


Figure 10-2 An example of an object model transformation. A redundant attribute can be eliminated by creating a superclass.

- The first one, Pull Up Field, moves the email field from the subclasses to the superclass User

Pull Up Field relocates the `email` field using the following steps (Figure 10-3):

1. Inspect `Player`, `LeagueOwner`, and `Advertiser` to ensure that the `email` field is equivalent. Rename equivalent fields to `email` if necessary.
2. Create public class `User`.
3. Set parent of `Player`, `LeagueOwner`, and `Advertiser` to `User`.
4. Add a protected field `email` to class `User`.
5. Remove fields `email` from `Player`, `LeagueOwner`, and `Advertiser`.
6. Compile and test.

Before refactoring

```
public class Player {
    private String email;
    //...
}
public class LeagueOwner {
    private String eMail;
    //...
}
public class Advertiser {
    private String email_address;
    //...
}
```

After refactoring

```
public class User {
    protected String email;
}
public class Player extends User {
    //...
}
public class LeagueOwner extends User
{
    //...
}
public class Advertiser extends User {
    //...
}
```

Figure 10-3 Applying the *Pull Up Field* refactoring.

- The second one, Pull Up Constructor Body, moves the initialization code from the subclasses to the superclass

Then, we apply the *Pull Up Constructor Body* refactoring to move the initialization code for `email` using the following steps (Figure 10-4):

1. Add the constructor `User(Address email)` to class `User`.
2. Assign the field `email` in the constructor with the value passed in the parameter.
3. Add the call `super(email)` to the `Player` class constructor.
4. Compile and test.
5. Repeat steps 1–4 for the classes `LeagueOwner` and `Advertiser`.

Before refactoring

```
public class User {
    private String email;
}

public class Player extends User {
    public Player(String email) {
        this.email = email;
        //...
    }
}
public class LeagueOwner extends User
{
    public LeagueOwner(String email) {
        this.email = email;
        //...
    }
}
public class Advertiser extends User {
    public Advertiser(String email) {
        this.email = email;
    //...
    }
}
```

After refactoring

```
public class User {
    public User(String email) {
        this.email = email;
    }
}
public class Player extends User {
    public Player(String email) {
        super(email);
    //...
    }
}
public class LeagueOwner extends User
{
    public LeagueOwner(String email) {
        super(email);
    //...
    }
}
public class Advertiser extends User {
    public Advertiser(String email) {
        super(email);
    //...
    }
}
```

• The third and final one, Pull Up Method, moves the methods manipulating the email field from the subclasses to the superclass.

1. Examine the methods of `Player` that use the `email` field. Note that `Player.notify()` uses `email` and that it does not use any fields or operations that are specific to `Player`.
2. Copy the `Player.notify()` method to the `User` class and recompile.
3. Remove the `Player.notify()` method.
4. Compile and test.
5. Repeat for `LeagueOwner` and `Advertiser`.

- Applying these three refactorings effectively transforms the ARENA source code in the same way the object model transformation

3. Forward Engineering

- Forward engineering is the process of taking elements from a **model** (such as a design or blueprint) and converting them into actual **source code** or **implementations**.
- For example, it could involve generating:
  - A **class declaration** in Java,
  - A **Java expression** (like a line of code),
  - A **database schema** (structure for storing data).

- **Purpose of Forward Engineering**:
  - The goal is to keep a **strong link** between the **design model** (the blueprint) and the **actual code**.

- This ensures that the code reflects the design properly and reduces the chance of **implementation errors** (mistakes made when coding).

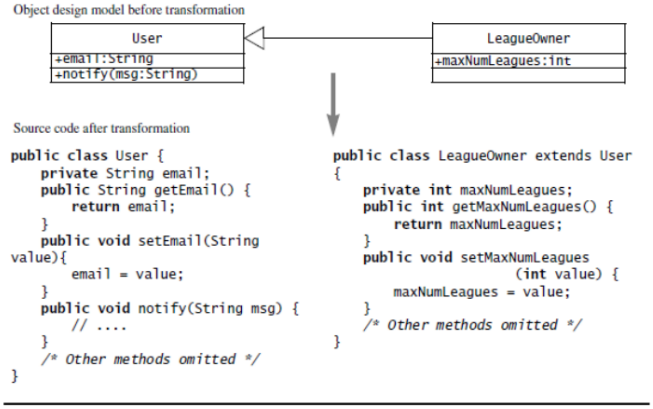


Figure 10-5 Realization of the `User` and `LeagueOwner` classes (UML class diagram and Java excerpts). In this transformation, the public visibility of `email` and `maxNumLeagues` denotes that the methods for getting and setting their values are public. The actual fields representing these attributes are private.

4. **Reverse Engineering**
    - Reverse engineering is the process of taking **existing source code** and converting it back into a **model** (like a design or blueprint).
    - This is the opposite of forward engineering, where you start with a model and generate code.

- **Purpose of Reverse Engineering**:
    - The goal is to **recreate the model** for a system that already exists. This can be done if:
        - The **original model was lost** or never created.
        - The model has become **out of sync** with the source code (i.e., the model no longer matches the code because it was never updated).
- **How It Works**:
    - Reverse engineering is like **backtracking** from the code to understand the original design, structure, or architecture of the system.
- **Inverse of Forward Engineering**:
        - If forward engineering takes a model and creates code, reverse engineering does the opposite: it takes code and recreates the model.

The Four Principles of Transformation:

1. **Each transformation must address a single criterion**
    - Each transformation should focus on improving **one aspect** of the design at a time (e.g., improving performance, reducing complexity, etc.).

- Don’t try to change too many things in one go.

2. **Each transformation must be local**
   - A transformation should affect **only a small part** of the system at a time (e.g., a few methods or classes).
   - This keeps changes manageable and easier to test.
3. **Each transformation must be applied in isolation**
   - Transformations should be made **independently** of other changes.
   - This means making sure that you apply a change before you move on to the next one, without other changes interfering.
4. **Each transformation must be followed by a validation step**
   - After making a transformation, you must **test and validate** it to ensure that the system still works as expected and no new errors were introduced.

Mapping Associations to Collections

1. Unidirectional one-to-one associations.

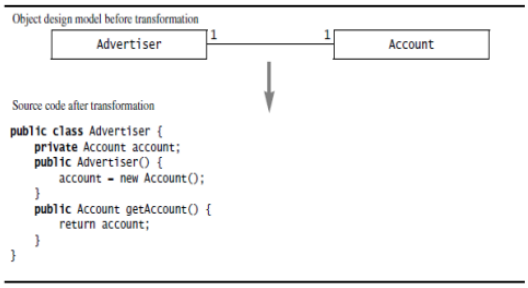


Figure 10-8 Realization of a unidirectional, one-to-one association (UML class diagram and Java).

2. Bidirectional one-to-one associations.

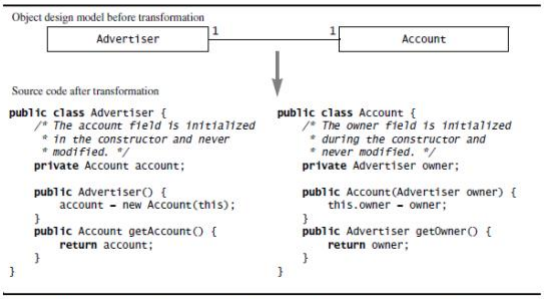


Figure 10-9 Realization of a bidirectional one-to-one association (UML class diagram and Java excerpts).

3. One-to-many associations.

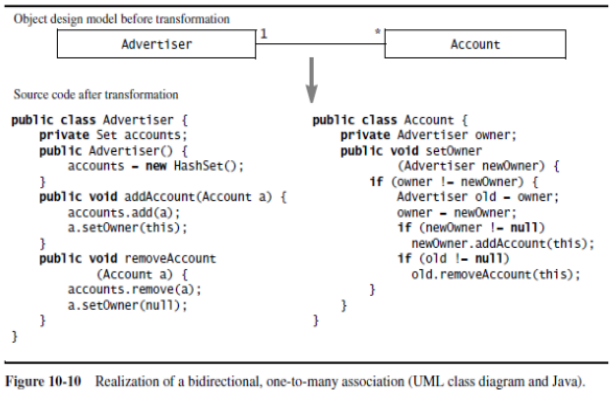


Figure 10-10 Realization of a bidirectional, one-to-many association (UML class diagram and Java).

4. Many-to-many associations.

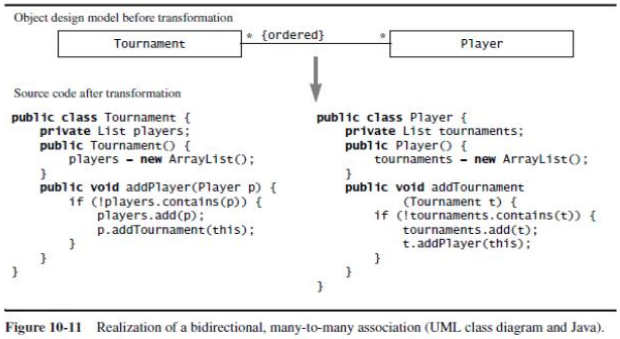


Figure 10-11 Realization of a bidirectional, many-to-many association (UML class diagram and Java).

Mapping Contracts to Exceptions

This process involves defining how a system should behave when certain **conditions (contracts)** are not met. Contracts in software often include **preconditions**, **postconditions**, and **invariants** that specify what must be true before, during, and after an operation. If these contracts are violated, exceptions are raised.

Steps in Mapping Contracts to Exceptions:

1. **Checking Preconditions**
    - **Preconditions** are conditions that must be true **before** a method is executed.
    - These checks should be done **at the start of the method**, before the actual work is performed.
    - If the precondition is false, an exception should be raised immediately.
    - **Each precondition** violation corresponds to a **specific exception**.
2. **Checking Postconditions**
    - **Postconditions** are conditions that must be true **after** a method has completed its work.
    - These checks should be done **at the end of the method**, after all work and state changes are finalized.
    - The postcondition is usually checked with a **Boolean expression** in an if statement. If the condition is violated, an exception is raised.
3. **Checking Invariants**
    - **Invariants** are conditions that should always be true during the operation of the system.
    - Invariants are often checked **along with postconditions**, after the method has finished running.
4. **Dealing with Inheritance**
    - When using **inheritance** (subclasses), the checks for **preconditions** and **postconditions** should be placed into **separate methods**.
    - These methods can then be **called from subclasses** to ensure consistency across the class hierarchy.
5. **Coding Effort**
    - Sometimes, the code needed to check **preconditions** and **postconditions** can be **longer and more complex** than the actual code that performs the main task of the method.
    - This increases the amount of code to maintain and can make the system harder to manage.

6. **Increased Opportunities for Defects**
    - Writing the checks for contracts also introduces the possibility of **errors** in the checking code.
    - This increases the **testing effort** because you need to test both the business logic and the validation logic.
7. **Performance Drawback**
    - Checking **all contracts systematically** can slow down the system, sometimes even by an order of magnitude (i.e., 10 times slower).
    - While ensuring correctness is important, it may negatively impact **performance goals** like **response time** and **throughput**.