**12.3.1 Abstraction**

Abstraction involves expressing a solution at different levels of detail. At the highest level, broad terms are used to state a solution. Lower levels involve more detailed descriptions. Procedural abstractions, such as the word "open" for a door, suppress specific details of a sequence of instructions. Data abstractions involve named collections of data, like attributes describing a "door" object.

**12.3.2 Architecture**

Software architecture refers to the overall structure of the software and its conceptual integrity. It involves the organization of program components, their interactions, and the structure of data used by these components. Architectural design provides a framework for detailed design activities, often using patterns that can be reused across similar systems.

**12.3.3 Patterns**

Design patterns are proven solutions to recurring design problems within a specific context. They describe design structures that help determine if a pattern is applicable, reusable, or serves as a guide for similar designs. Patterns facilitate solving problems efficiently by providing tested and reliable solutions.

**12.3.4 Separation of Concerns**

Separation of concerns involves breaking down a complex problem into manageable pieces. By isolating different aspects of the problem, it becomes easier to solve and optimize each part independently. This concept supports modularity, making it easier to manage and maintain the software.

**12.3.5 Modularity**

Modularity divides software into independent components or modules, each addressing a specific function. This approach makes the software more manageable, easier to develop, test, and maintain. It helps in reducing complexity and minimizing the cost of development and integration.

**12.3.6 Information Hiding**

Information hiding is about encapsulating the details within a module, making them inaccessible to other modules that do not need that information. This principle supports modularity by defining clear interfaces and reducing the chances of errors propagating through the system.

**12.3.7 Functional Independence**

Functional independence is achieved by developing modules with specific functions and minimal interaction with other modules. This reduces complexity, enhances maintainability, and limits the propagation of errors. Functional independence is assessed using cohesion (the strength of module's function) and coupling (the interdependence between modules).

**12.3.8 Refinement**

Stepwise refinement involves developing software by progressively elaborating on the details. Starting with a high-level abstraction, the design is detailed step by step until it reaches a level that can be implemented. Refinement complements abstraction, enabling a complete design model to evolve.

**12.3.9 Aspects**

Aspects represent crosscutting concerns that span multiple modules and cannot be easily compartmentalized. Identifying and managing aspects during the design phase ensures that these concerns are properly addressed and do not lead to scattered or tangled software components.

**12.3.10 Refactoring**

Refactoring is the process of restructuring existing code without changing its external behavior. This improves the internal structure of the software, making it easier to understand, maintain, and extend. Refactoring involves identifying and removing redundancy, optimizing algorithms, and improving data structures to enhance the overall quality of the software.

These design concepts provide the groundwork for developing robust, maintainable, and efficient software systems. By applying these principles, software engineers can create designs that stand the test of time and meet the evolving needs of users.

### 12.3.11 Object-Oriented Design Concepts

The object-oriented (OO) paradigm is widely utilized in contemporary software engineering due to its ability to model real-world scenarios and promote code reusability and maintainability. This approach encompasses several key concepts that provide a structured method to design and build software systems. For those unfamiliar with these concepts, Appendix 2 offers a detailed overview. Here, we summarize the main OO design concepts:

#### Classes and Objects

* **Classes**: A class is a blueprint for creating objects (specific instances of the class). It defines a set of attributes and methods that the objects created from the class will have.
* **Objects**: An object is an instance of a class. It encapsulates data and behavior (methods) that operate on the data. Each object has a state (represented by its attributes) and behavior (represented by its methods).

#### Inheritance

Inheritance is a mechanism by which one class (the subclass or derived class) can inherit attributes and methods from another class (the superclass or base class). This promotes code reuse and the creation of a hierarchical relationship between classes. There are different types of inheritance:

* **Single Inheritance**: A subclass inherits from one superclass.
* **Multiple Inheritance**: A subclass inherits from more than one superclass (note: some programming languages do not support multiple inheritance directly).
* **Multilevel Inheritance**: A chain of inheritance where a class is derived from another class, which is also derived from another class.
* **Hierarchical Inheritance**: Multiple classes inherit from a single superclass.

#### Polymorphism

Polymorphism allows methods to do different things based on the object it is acting upon, even if the method has the same name. It comes in two main forms:

* **Compile-time (Static) Polymorphism**: Achieved through method overloading and operator overloading.
* **Run-time (Dynamic) Polymorphism**: Achieved through method overriding, typically using inheritance and interfaces.

#### Encapsulation

Encapsulation is the concept of bundling the data (attributes) and the methods (functions) that operate on the data into a single unit, known as a class. It restricts direct access to some of the object's components, which can protect the object’s internal state from unintended or harmful interference. This is typically done using access specifiers:

* **Private**: Accessible only within the class.
* **Protected**: Accessible within the class and by derived class instances.
* **Public**: Accessible from any part of the program.

#### Abstraction

Abstraction is the concept of hiding the complex implementation details and showing only the necessary features of an object. It helps in reducing programming complexity and effort. In OO design, abstraction is achieved using abstract classes and interfaces.

#### Messages

Objects communicate with each other through messages. A message is a call to a method, where an object requests another object to execute one of its methods. The method may manipulate the data within the object, produce an output, or both.

#### Interface and Implementation Separation

OO design separates the interface from the implementation. The interface specifies what methods an object should have, while the implementation provides the actual code that executes the methods. This separation allows different objects to interact with each other via interfaces, without needing to know the details of each other’s implementations.

### Summary

Object-oriented design concepts provide a robust framework for software development, emphasizing modularity, reusability, and abstraction. By employing these principles, software engineers can create systems that are easier to manage, extend, and maintain. For a more in-depth exploration of these concepts, readers should refer to Appendix 2.

### 12.3.13 Dependency Inversion

In many older software architectures, the design often follows a hierarchical structure where "control" components depend on lower-level "worker" components to perform various tasks. This can lead to tight coupling between components, making the system more rigid and harder to maintain or extend. Consider a simple program with three components: a control module CCC, a keystroke reader module RRR, and a printer writer module WWW. The control module CCC coordinates the activities of the other two modules, RRR and WWW, to read keyboard strokes and print the results.

In this scenario, the design is highly coupled because CCC depends directly on RRR and WWW. To reduce this dependence, we can use abstractions to invoke the worker modules from the control module. In object-oriented software engineering, these abstractions are typically implemented as abstract classes. By introducing abstract classes R∗R^\*R∗ and W∗W^\*W∗, the control module CCC can interact with these abstractions instead of the concrete implementations of RRR and WWW.

These abstract classes serve as interfaces for the worker classes, allowing CCC to invoke methods without knowing the specific details of how they are implemented. For example, the abstract class R∗R^\*R∗ might define a method read(), which could be implemented by a keyboard reader class in one context or a sensor reader class in another. Similarly, W∗W^\*W∗ might define a method write(), which could be implemented by a printer writer class or another output device writer class.

This approach aligns with the Dependency Inversion Principle (DIP), which can be generalized as follows:

* **High-level modules (classes) should not depend directly on low-level modules. Both should depend on abstractions.**
* **Abstractions should not depend on details. Details should depend on abstractions.**

By adhering to the DIP, we achieve several benefits:

1. **Reduced Coupling**: The control module CCC no longer depends directly on the specific implementations of RRR and WWW. Instead, it depends on the abstractions R∗R^\*R∗ and W∗W^\*W∗, making the system more flexible and easier to modify.
2. **Improved Testability**: Since CCC interacts with abstractions, it becomes easier to substitute the real implementations of RRR and WWW with mock objects during testing. This isolation simplifies unit testing and enhances test coverage.
3. **Enhanced Maintainability**: Changes in the implementation of the worker modules RRR and WWW do not affect the control module CCC, as long as they adhere to the defined abstractions. This separation of concerns makes the system easier to maintain and extend.

