CSC 540 2.0
Software Engineering

M. K. A. Ariyaratne

Department of Computer Science
Faculty of Applied Sciences
University of Sri Jayewardenepura
Sri Lanka

UNIT 5
Software Design Concepts

Design - Introduction

- Design can be a plan or a drawing.
- Design allows you to model the system that is to be built.

> The beginning of wisdom for a software engineer is to recognize the difference between getting a program to work, and getting it right.
>
> Fundamental software design concepts provide the necessary framework for **"Getting It Right"**.

Introduction to Software Design

Software Design is the process of converting requirements into a blueprint for building the software. It bridges the gap between requirements analysis and implementation.

- Provides a roadmap for implementation.
- Emphasizes quality, modularity, and maintainability.
- Ensures software meets both functional and non-functional requirements.

Design - Introduction

- The goal of design is to produce a model or representation that exhibits firmness, commodity and delight.

    1. Firmness: A program should not have any bugs that inhibit its function.

    2. Commodity: A program should be suitable for the purposes for which it was intended.

    3. Delight: The experience of using the program should be pleasurable one.

Design Concepts

Design Concepts - Introduction

- Design concepts in software engineering are foundational principles, guidelines, and best practices used to create effective, efficient, and maintainable software systems.
- These concepts are universal and applicable across various methodologies, tools, and languages.
- They help ensure that the software design is robust, modular, scalable, and easy to understand or modify.

Design concepts overview

1. Abstraction
2. Architecture
3. Patterns
4. Separation of Concerns (SoC)
5. Modularity
6. Information Hiding
7. Functional Independence
   - 7.1 Cohesion
   - 7.2 Coupling
8. Refinement
9. Refactoring

1. Abstraction

"Abstraction permits one to concentrate on a problem at some level of generalization without regard to irrelevant low level details.."

Abstraction

- The word "Abstract" means conceptual or existing in thought or as an idea but not having a physical or concrete existence.
- During requirements definition and design, abstraction permits separation of the conceptual aspects of a system from the (yet to be specified) implementation details.
- In software Engineering, the concept abstraction has a broad perspective.
- Abstraction can be used in many phases in SDLC and in many levels.

Abstraction (2)

- Use of highest Level abstraction of a solution: A solution is stated in broad terms using the language of the problem environment.

  Eg. Loyalty management system: We want to reward loyal (best buyers) customers using a good method.

- At lower levels of abstraction, a more detailed description of the solution is provided.

- Problem-oriented terminology is coupled with implementation-oriented terminology in an effort to state a solution.

  Eg. How you are going to reward : Give points according to their bill value. (If bill value is greater than 100, less than 500, give one point, greater than 500, less than 1000, give 2 points, greater than 1000, give 5 points)

Abstraction (3)

- Finally, at the lowest level of abstraction, the solution is stated in a manner that can be directly implemented (detailed).

  e.g. Points are added to a loyalty card. Card can be read using a card reader..

- Presumably, the abstract definition of a component is much simpler than the component itself.

Abstraction (4)

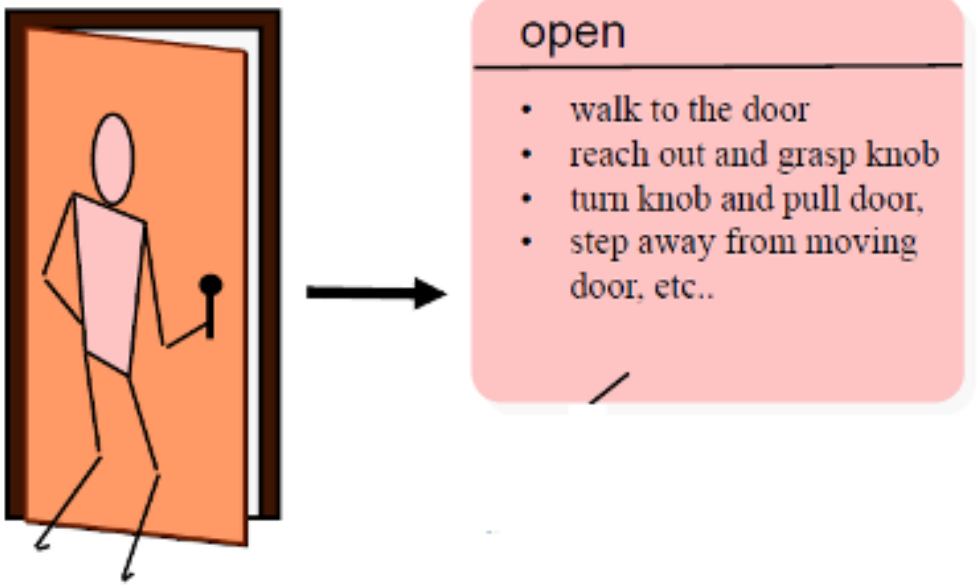
- Different levels of abstraction is used in Software Engineering, as well as different types of abstraction; for an example Procedural abstraction and Data abstraction.
- A procedural abstraction refers to a sequence of instructions that have a specific and limited function.
- The **name of a procedural abstraction** implies these functions, **but specific details are suppressed**.

Procedural Abstraction

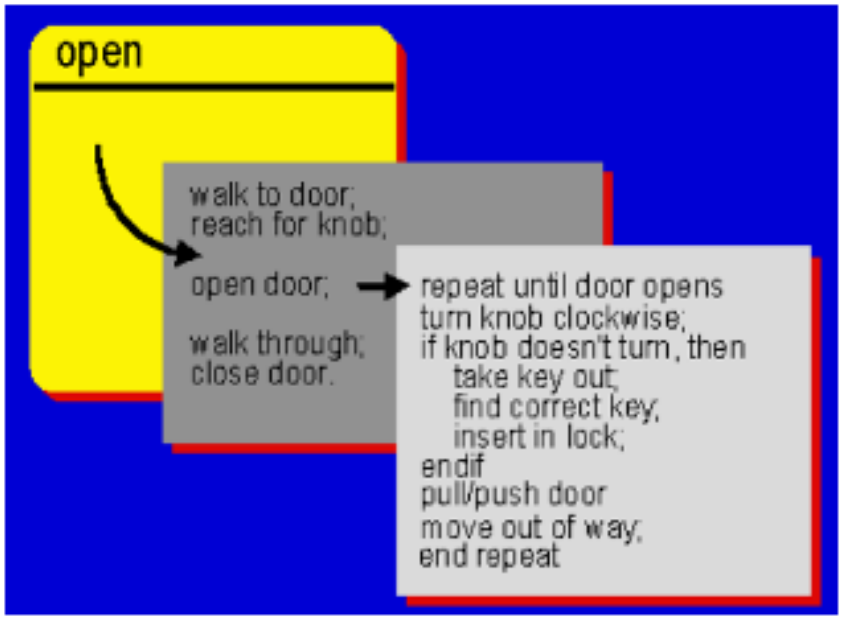
- An example of a procedural abstraction would be the word **open** for a door.
- **Open** implies a long sequence of procedural steps.

  e.g.,

  1. **walk to the door**
  2. **reach out and grasp knob**
  3. **turn knob and pull door**
  4. **step away from moving door, etc.**



Procedural Abstraction (3)



1. **High-Level:** What the system does (e.g., "Open a door").
2. **Mid-Level:** Logical steps to achieve functionality (e.g., "Turn the knob and push").
3. **Low-Level:** Specific instructions for implementation (e.g., "Send signal to motor").

Data Abstraction

- A data abstraction is a named collection of data that describes a data object.
- For an example In the context of the procedural abstraction **Open**, we can define a data abstraction called **door**.
- Like any data object, the data abstraction for door would encompass a set of attributes that describe the door (e.g., door type, swing direction, opening mechanism, weight, dimensions).
- It follows that the procedural abstraction open would make use of information contained in the attributes of the data abstraction door .

Architecture

The structure or organization of program components (modules), the manner in which these components interact and the structure of data that are used by the components.

Architecture: The blueprint that defines the structure and interaction of software components.

What is Software Architecture?

- Software Architecture defines the high-level structure of a software system and the guidelines for its design and evolution.
- It provides a blueprint for system structure, addressing:
  - Components
  - Relationships among components
  - Principles guiding the design
- The architecture ensures scalability, maintainability, and reliability of the software.

Key Architectural Concepts

- **Architectural Styles:**
  - Layered Architecture
  - Client-Server Architecture
  - Event-Driven Architecture
  - Microservices Architecture
  - Service-Oriented Architecture (SOA)
- **Design Patterns:**
  - Singleton, Factory, Observer, Adapter
  - Architectural patterns like MVC (Model-View-Controller) and MVVM (Model-View-ViewModel)
- **System Quality Attributes:**
  - Scalability, Security, Maintainability, Performance, Reliability

Layered Architecture

- Divides the system into layers, each with specific responsibilities.
- Common layers:
  - Presentation Layer (UI)
  - Business Logic Layer
  - Data Access Layer
- Ensures separation of concerns and improves maintainability.

Client-Server Architecture

- A distributed system structure with clients requesting services and servers providing them.
- Example: Web applications (browser as the client, web server as the server).
- Advantages:
  - Centralized control
  - Scalability

Microservices Architecture

- A system composed of small, independent services that communicate via APIs.
- Benefits:
  - Flexibility in development and deployment
  - Scalability and fault isolation
- Examples: E-commerce platforms, cloud-native applications.

Key Quality Attributes in Architecture

- **Scalability:** Ability to handle increasing workload.
- **Performance:** Ensuring the system responds quickly to requests.
- **Reliability:** Maintaining functionality under different conditions.
- **Security:** Protecting the system from threats.
- **Maintainability:** Ease of making updates or fixing issues.

Design Patterns in Architecture

- **Singleton Pattern:** Ensures only one instance of a class exists.
- **Factory Pattern:** Creates objects without specifying their exact class.
- **Observer Pattern:** Notifies dependent objects of state changes.
- **Adapter Pattern:** Bridges interfaces to make them compatible.

Summary

- Software architecture defines the structure and design principles of a system.
- Key concepts include architectural styles, design patterns, and quality attributes.
- The right architecture ensures a system is scalable, maintainable, and reliable.

Design Patterns

In software engineering, a software design pattern is a general, reusable solution to a commonly occurring problem within a given context in software design.

Design Patterns

- Design patterns represent the best practices used by experienced object-oriented software developers.
- In 1994, four authors Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides published a book titled Design Patterns - Elements of Reusable Object-Oriented Software which initiated the concept of Design Pattern in Software development.
- A pattern conveys the essence of a proven solution to a recurring problem within a certain context amidst competing concerns
- A design pattern isn't a finished design that can be transformed directly into code.

Design Patterns (2)

- It is a description or template for how to solve a problem that can be used in many different situations.
- Design patterns can speed up the development process by providing tested, proven development paradigms.
- You have a problem, you select the most suitable pattern for that, then you apply the pattern according to your problem.

Patterns in programming are like recipes in cooking. They are not ready dishes, but instructions for slicing and dicing products, cooking them, serving them and so forth.

Design Patterns - Why should we use them?

- Design patterns are, by principle, well-thought out solutions to programming problems.
- Many programmers have encountered these problems before, and have used these 'solutions' to remedy them.
- If you encounter these problems, why recreate a solution when you can use an already proven answer?
  - Examples:
    - **Singleton:** Ensures a class has only one instance.
    - **Observer:** Notifies dependent objects of changes.
    - **Factory:** Creates objects without specifying their exact type.

Separation of Concerns

In computer science, separation of concerns (SoC) is a design principle for separating a computer program into distinct sections, such that each section addresses a separate concern

What is Separation of Concerns?

- Separation of Concerns (SoC) is a software design principle that promotes dividing a system into distinct parts, each addressing a separate concern.
- A **concern** refers to a specific functionality or behavior of a software system (e.g., user interface, data processing, or security).
- Benefits of SoC:
  - Simplifies system design.
  - Enhances maintainability and readability.
  - Enables parallel development by different teams.
  - Improves scalability and testability.

Key Concepts of SoC

- **Modularity:**
  - Divides the software system into self-contained modules.
  - Example: A module for user authentication, another for database access.
- **Encapsulation:**
  - Hides the internal workings of a module.
  - Allows changes within a module without impacting others.
- **Abstraction:**
  - Focuses on essential details, ignoring unnecessary complexities.
  - Example: Abstract interfaces hide implementation details.
- **Responsibility Assignment:**
  - Assigns distinct responsibilities to different parts of the system.
  - Example: MVC pattern assigns responsibilities to Model, View, and Controller layers.

Implementation Techniques for SoC

- **Layered Architecture:**
  - Divides the system into layers, such as Presentation, Business Logic, and Data Access.
  - Each layer addresses a specific concern and communicates only with adjacent layers.
- **Aspect-Oriented Programming (AOP):**
  - Separates cross-cutting concerns like logging, authentication, and error handling.
  - Uses aspects to encapsulate these concerns.
- **Design Patterns:**
  - Patterns like MVC (Model-View-Controller) inherently support SoC.
  - Example: MVC separates concerns of data handling (Model), user interface (View), and application logic (Controller).

Benefits of SoC

- **Maintainability:**
  - Changes in one part of the system do not affect other parts.
  - Easier to identify and fix bugs.
- **Reusability:**
  - Modules can be reused in different projects or parts of the system.
- **Parallel Development:**
  - Teams can work on different concerns simultaneously without interference.
- **Scalability:**
  - Makes it easier to scale individual components.
- **Testability:**
  - Concerns can be tested independently, simplifying debugging and verification.

Examples of SoC in Practice

- **Web Applications:**
  - Frontend (UI) is separated from Backend (logic and database).
- **Microservices Architecture:**
  - Each service handles a specific business function.
- **Aspect-Oriented Logging:**
  - Logging functionality is encapsulated in a separate module, leaving core business logic unaffected.
- **MVC Frameworks:**
  - Frameworks like Django and Spring MVC demonstrate SoC by segregating data, logic, and presentation layers.

Summary

- Separation of Concerns is a foundational principle in software design that improves system modularity, maintainability, and scalability.
- It is implemented through techniques like layered architecture, AOP, and design patterns.
- Practical examples include web applications, microservices, and MVC frameworks.

Modularity

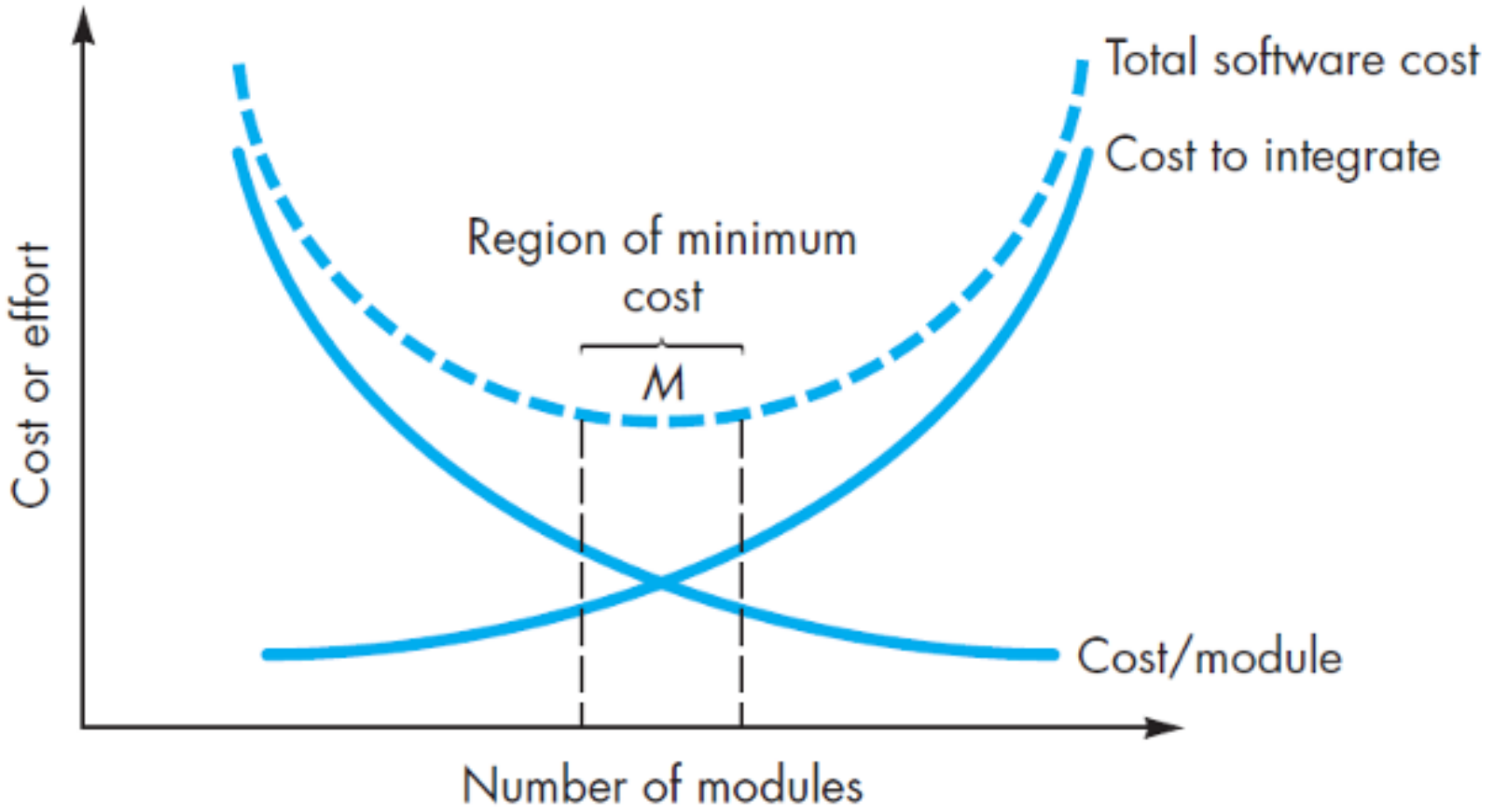
Modularity is the most common manifestation / demonstration of separation of concerns

Modularity

- In software engineering, modularity refers to the extent to which a software/Web application may be divided into smaller modules.
- Software is divided into separately named and addressable components, sometimes called modules, that are integrated to satisfy problem requirements.

Modularity

How, Effort / Cost to build and integrate change over number of modules is shown in the following graph.



Why Modularity Demonstrates Separation of Concerns

Definition of Modularity:

- Modularity refers to dividing a software system into smaller, self-contained, and independent parts or modules.
- Each module encapsulates a specific concern or responsibility.

Alignment with Separation of Concerns:

- SoC promotes isolating different concerns (e.g., UI, business logic, data access) into distinct parts.
- Modularity is the practical implementation of SoC, as it organizes concerns into separate modules.

Benefits of Modularity in SoC

- **Improved Maintainability:** Changes in one module do not impact others.
- **Enhanced Testability:** Modules can be tested independently.
- **Parallel Development:** Teams can work on different modules concurrently.
- **Reusability:** Modules can often be reused in other projects.

Examples of Modularity Demonstrating SoC

- **Layered Architecture:**
  - Layers like Presentation, Business Logic, and Data Access are modular components, each addressing a specific concern.
- **Microservices:**
  - Each microservice handles a single responsibility, encapsulating a concern.
- **Object-Oriented Design:**
  - Classes and objects encapsulate concerns, demonstrating modularity.

Information Hiding

Information Hiding is about handling accessibility

What is Information Hiding?

- **Information Hiding** is a software design principle that involves hiding implementation details of a module or component from other parts of the system.
- Focuses on exposing only necessary functionality while keeping internal details private.
- Promotes encapsulation and reduces system complexity by minimizing interdependencies.
- First introduced by David L. Parnas in the 1970s.

Key Concepts of Information Hiding

- **Encapsulation:**
  - Groups data and methods that operate on the data into a single unit.
  - Example: A class in Object-Oriented Programming.
- **Abstraction:**
  - Provides only essential details while hiding implementation specifics.
  - Example: Abstract interfaces in programming.
- **Access Control:**
  - Uses access modifiers (e.g., public, private, protected) to limit access to class members.
  - Ensures that internal details are hidden from external components.
- **Minimized Coupling:**
  - Reduces dependencies between modules by hiding internal workings.
  - Enhances modularity and independence.

Benefits of Information Hiding

- **Improved Maintainability:**
  - Internal changes to a module do not affect other parts of the system.
  - Easier to locate and fix issues within a module.
- **Enhanced Security:**
  - Restricts access to sensitive data or critical functionality.
- **Simplified Interface:**
  - Provides a clean, easy-to-use interface for interacting with the module.
- **Supports Modularity:**
  - Promotes separation of concerns by isolating functionality.
- **Facilitates Parallel Development:**
  - Teams can work on separate modules without worrying about internal dependencies.

Techniques for Information Hiding

- **Access Modifiers:**
  - Use public, private, and protected keywords to control access.
- **Encapsulation:**
  - Use classes and objects to bundle data and methods.
- **Abstract Interfaces:**
  - Define interfaces to expose only the necessary methods.
- **Use of Modules or Packages:**
  - Organize code into logical units to encapsulate functionality.

Examples of Information Hiding

- **Object-Oriented Programming (OOP):**
  - Private fields in a class with public getter and setter methods.
- **APIs:**
  - APIs expose only necessary endpoints, hiding backend logic.
- **Database Systems:**
  - Hides internal schema and exposes only views or stored procedures.
- **Encryption Libraries:**
  - Provides encryption and decryption methods without exposing underlying algorithms.

Summary

- Information Hiding is a foundational principle for managing complexity in software design.
- It promotes encapsulation, abstraction, and modularity.
- Benefits include improved maintainability, security, and scalability.
- Techniques like access modifiers, encapsulation, and abstract interfaces help achieve effective information hiding.

Functional Independence

Functional independence occurs where modules address a specific and constrained range of functionality

Functional Independence

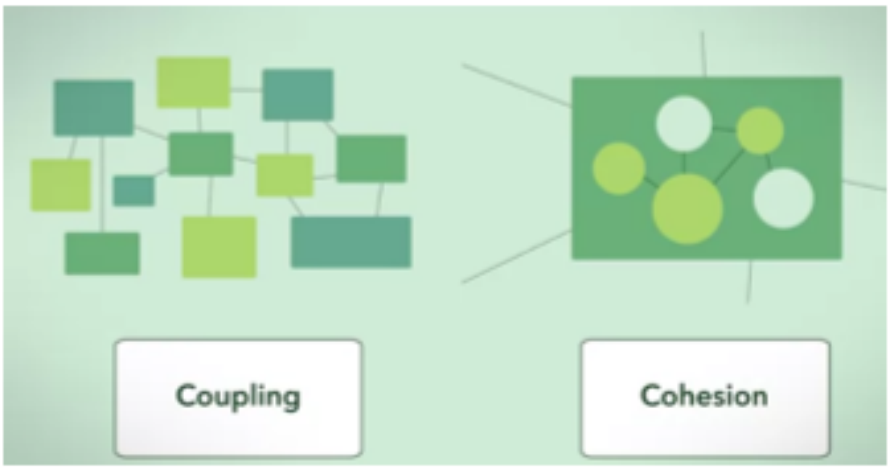
- Functional independence is defined as a direct improvement or enhancement to the concepts; modularity, abstraction and information hiding.
- It can be achieved by developing the modules (or components) for **their single, isolated respective functions** and then **combine these modules through interacting approaches to achieve a whole sole objective**.
- When a module has its single function to perform then it will be easy to achieve its objective.
- So functional independence in this respect is associated with the **effective modularity**.

Functional Independence (2)

- Functional independence is a key of good design and design is the key to software quality.
- To recognize and measure / evaluate the degree of module independence in a design, two qualitative criteria are defined.
- They are
  1. **Coupling**
  2. **Cohesion**
- A module having **high cohesion** and **low coupling** is said to be functionally independent of other modules.

Coupling and Cohesion

- When you are programming, keeping modules simple is critical.
- If the design complexity exceeds what developers can mentally handle, bugs will occur more often.
- So there must be a way to evaluate the design complexity.
- Coupling and cohesion are to evaluate this complexity.



Coupling

- Coupling focuses on complexity between a module and other modules.
- When you design a system, you combine various modules together.
- Think about a bad design like puzzle pieces where your modules are the pieces.

- You can only connect a puzzle piece to a specific puzzle piece.

Coupling (2)

- On the other hand think about a well design system like Lego blocks.

- You can connect any two blocks without much trouble.
- All Lego blocks are compatible with one another.

Coupling (3)

- When designing your system you want to make it like Lego.
- That way you can easily connect and reuse modules together.
- Coupling for a module captures the complexity of connecting the module to the other modules.
- **If your module highly depend on other modules, you would say that the module is tightly coupled to others (Like puzzle pieces)**
- On the other hand **if your module find its easy to connect with other modules, this module is loosely coupled (Like Lego blocks)**.

Coupling (4)

- You want coupling to your module to be loose or low, not tight.
- When evaluating the coupling of a module, you need to consider.
  1. Degree.
  2. Ease.
  3. Flexibility.

Coupling (5)

Degree

Degree is the number of connections between the module and others. With coupling, you want to keep the degree small. For instance, if the module needed to connect to other modules through a few parameters or narrow interfaces, then the degree would be small, and coupling would be loose.

Ease

Ease is how obvious are the connections between the module and others. With coupling, you want the connections to be easy to make without needing to understand the implementations of the other modules.

Flexibility

Flexibility is how interchangeable the other modules are for this module. With coupling, you want the other modules easily replaceable for something better in the future.

Cohesion

- You also need to consider **complexity within the module**.
- Cohesion represents the clarity of the responsibilities of a module.
- If your module performs a one task and nothing else, or has a clear purpose, **your module has high cohesion**.
- If your module tries to encapsulate more than one purpose, or an unclear purpose, your module has low cohesion.
- All you need is **high cohesion**. If you module has several tasks, then its better to split the module.

Coupling and Cohesion example

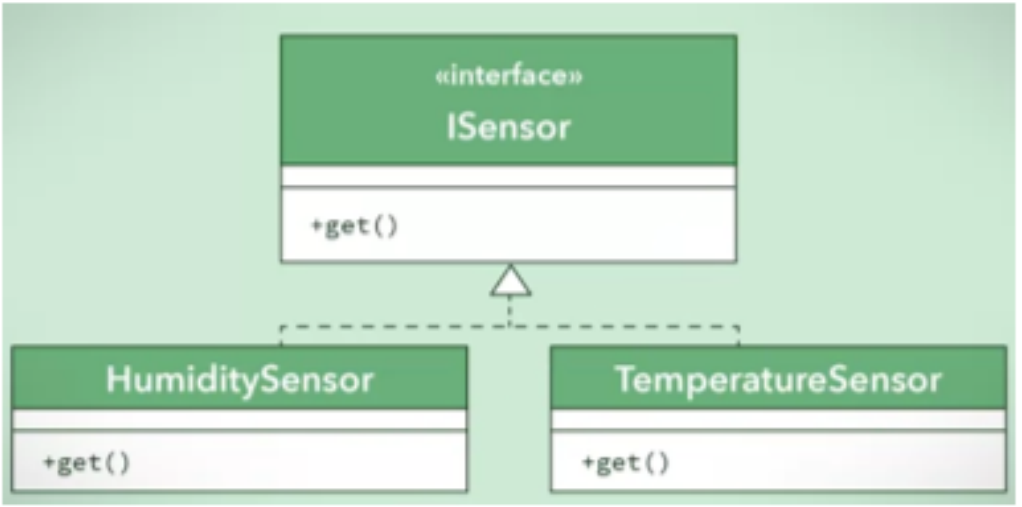

```
public void get (int controlFlag) {
     switch (controlFlag) {
          case 0:
               return this.humidity;
               break;
          case 1:
               return this.temperature;
               break;
          default:
               throw new UnknownControlFlagException();
     }
}
```

Coupling and Cohesion example (2)

- Since the **'Sensor'** class does not have a clear single purpose, it suffers from **Low cohesion**.
- It is unclear what the **'controlFlag'** means. You need to read the inside of the method to know what values to give it.
- Method is unclear and lacks **ease**.
- This lack of ease makes the 'get' method harder to use, and makes any caller of this method **tightly coupled** to it.

Coupling and Cohesion example (3)

An appropriate design



- Now you have two separate classes each having clear purposes, So **High cohesion** is there.
- get methods are not hiding any information and easily think that humidity method's 'get' method will return humidity and so the 'get' in temperature method.

Coupling and Cohesion example (3)

- That makes other module which use the either be **loosely coupled**.
- In general, there is a balance to be made with high cohesion and low coupling.
- If modules are simplified to have high cohesion, then they may depend more on other modules, thus increasing coupling.
- If you simplified the connection between modules to achieve low coupling modules may need to take more responsibility and may have low cohesion.
- You need to have a good design which is balanced but have **low coupling** and **high cohesion**.

Refinement

The Process of Elaboration

Refinement

- In software design, **refinement is the verifiable transformation of an abstract (high-level) specification into a concrete (low-level) executable program.**
- Stepwise refinement allows this process to be done in stages.
- A hierarchy is developed by decomposing a macroscopic statement of function (a procedural abstraction) in a stepwise fashion until programming language statements are reached.
- Refinement is actually a **process of elaboration**.

Refinement (2)

- You begin with a statement of function that is defined at a high level of abstraction by providing no indication of the internal workings of the function .
- You then elaborate on the original statement, providing more and more detail as each successive refinement (elaboration) occurs.

Refinement (3)

Abstraction

A description of something that omits some details that are not relevant to the purpose of the abstraction; the converse of refinement.

Refinement

A detailed description that conforms to another (its abstraction). Everything said about the abstraction holds, perhaps in a somewhat different form, in the refinement. Also called realization.

Refactoring

Revision or cleaning up of source code

What is Refactoring?

- **Refactoring** is the process of restructuring existing code without changing its external behavior.
- It improves the code's internal structure, making it easier to read, maintain, and extend.
- Refactoring focuses on improving code quality while preserving functionality.
- Coined by Martin Fowler and popularized in his book *Refactoring: Improving the Design of Existing Code*.

Why Refactor?

- **Improves Code Readability:** Makes the code easier for developers to understand.
- **Enhances Maintainability:** Simplifies debugging and updating the code.
- **Reduces Technical Debt:** Addresses shortcuts and quick fixes in the codebase.
- **Supports Scalability:** Prepares the code for future growth and feature additions.
- **Encourages Best Practices:** Promotes adherence to coding standards and principles.

Key Refactoring Techniques

- **Rename Variables and Methods:**
  - Use descriptive names that clearly indicate their purpose.
- **Extract Method:**
  - Split large methods into smaller, reusable methods.
- **Inline Method:**
  - Replace a method call with the method's body if the method is trivial.
- **Remove Dead Code:**
  - Eliminate unused or unnecessary code.
- **Simplify Conditional Expressions:**
  - Replace complex conditionals with simpler constructs.

More Refactoring Techniques

- **Introduce Parameter Object:**
  - Combine multiple parameters into a single object.
- **Encapsulate Field:**
  - Make fields private and provide getters and setters.
- **Replace Magic Numbers:**
  - Replace hardcoded values with named constants.
- **Decompose Switch Statements:**
  - Replace long switch statements with polymorphism or strategy patterns.
- **Introduce Null Object:**
  - Use a null object to avoid null checks.

Benefits of Refactoring

- **Improved Code Quality:** Reduces complexity and enhances clarity.
- **Better Collaboration:** Easier for teams to understand and modify code.
- **Reduced Bugs:** Cleaner code is less prone to errors.
- **Faster Development:** Simplifies future additions and changes.
- **Compliance with Standards:** Aligns code with industry best practices.

When to Refactor?

- **Before Adding New Features:** Ensure the codebase is clean and scalable.
- **During Code Reviews:** Address feedback related to structure and readability.
- **When Fixing Bugs:** Refactor related code to prevent future issues.
- **When Learning the Code:** Improve the structure as you understand it better.
- **When Tests Are Available:** Ensure functionality is preserved with automated tests.

Summary

- Refactoring improves code structure and maintainability without changing functionality.
- Key techniques include renaming, extracting methods, and removing dead code.
- Benefits include better readability, reduced bugs, and faster development.
- Refactoring should be an ongoing process to maintain a clean and efficient codebase.

Design Concepts - Summary

You can brief the idea of design concepts as follows.

1. **Abstraction:** It's the bird's eye view; Macroscopic
2. **Refinement:** Opposite of abstraction; Elaborating
3. **Architecture:** The structure
4. **Design Patters:** Proven solutions to reoccurring problems
5. **Separation of concerns (SoC):** Separate the system in to sections
6. **Modularity:** Demonstration of SoC

Design Concepts - Summary (2)

7. **Information hiding:** Handling accessibility. (information contained within a module is inaccessible to other modules that have no need for such information.)

8. **Functional Independence:** A Module's dependency

   8.1 **Coupling:** A measure of Complexity between a module and other modules (Prefer Low coupling)

   8.2 **Cohesion:** A measure of Complexity within a module (Prefer high cohesion)

9. **Refactoring:** Improving internal structure without changing the external structure / behavior

Design Concepts - Summary (3)

- Software design commence as the first iteration of requirements engineering comes to a conclusion.
- The intent of software design is to apply a set of principles, concepts and practice that lead to the development of high quality system or product.
- Design concepts describe attributes of computer software that should be present regardless of the software engineering process chosen, the design methods that are applied or the programming languages that are used.

Design Concepts - Summary (4)

Design concepts emphasize

- The need for abstraction as a mechanism or creating reusable software components.
- The importance of architecture as a way of better understand the over all structure of the system.
- The benefits of pattern based engineering as a technique for designing software with proven capabilities.
- The value of separation concern and effective modularity as a way to make software more understandable, more testable, and more maintainable.
- The consequences of information hiding as a mechanism for reducing the propagation of side effects when errors do occur.
- The impact of functional independence as a criterion for building effective modules.

1. Which of the following is/are true? In each case justify your answer.

   (a) At lower levels of abstraction, a more detailed description of the software solution can be provided.

   (b) Design patterns can slow down the software development process.

   (c) When the number of modules in the software increase, the integration cost gets lower.

(03x3 Marks)

7. Cohesion is a qualitative indication of the degree to which a module

   (a) can be written more compactly

   (b) focuses on just one thing

   (c) is able to complete its function in a timely manner

   (d) is connected to other modules and the outside world

8. Coupling is a qualitative indication of the degree to which a module

   (a) can be written more compactly

   (b) focuses on just one thing

   (c) is able to complete its function in a timely manner

   (d) is connected to other modules and the outside world

9. When it is about functional independence, we need modules to have
   (a) high cohesion and high coupling
   (b) Low cohesion and high coupling
   (c) high cohesion and low coupling
   (d) Low cohesion and low coupling

10. Which of the property of software modularity is incorrect with respect to benefits software modularity?
   (a) Modules are robust
   (b) Module can use other modules
   (c) Modules Can be separately compiled and stored in a library
   (d) Modules are mostly dependent