DESIGN AND IMPLEMENTATION

Prepared By: Natabar Khatri

New Summit College

Unit 7

INTRODUCTION TO SOFTWARE DESIGN AND IMPLEMENTATION

- Software design and implementation are the core, executable phases of the software engineering lifecycle where a conceptual idea is transformed into a tangible software system. They are deeply interconnected and iterative processes
- Software design involves a systematically structured approach to transform the requirements into a design model and software implementation involves a systematically structured approach to transform the design model into a working product.
- The relationship between software design and implementation, and the formality with which they are approached, varies significantly based on the project's scale, methodology, and chosen technologies.

INTRODUCTION TO SOFTWARE DESIGN AND IMPLEMENTATION

Software Design

- Software design is a highly creative activity focused on identifying software components and their relationships based on customer requirements. It's about conceiving *how* a problem will be solved.
  - Can be formally documented (e.g., using UML).
  - Often informal, existing in a programmer's mind or as quick sketches.
  - Always a process, even if not explicitly detailed.

Implementation

- Implementation is the process of realizing the design as an executable program. It's the tangible creation of the software system.
  - Closely linked with design; implementation issues influence design choices.
  - Choice of programming language impacts design documentation utility (e.g., UML more useful for Java/C# than Python).
  - Agile methods emphasize informal design and programmer autonomy in implementation.

BUILD VS. BUY DECISION

One of the most critical early decisions in any software project is whether to develop a new system from scratch or acquire an existing off the shelf solution.

Build (Custom Development)

- Involves creating application software from the ground up using conventional programming languages.
  - Offers complete customization and control.
  - Typically more time-consuming and expensive.
  - Requires extensive design modeling and implementation.

Buy (Off-the Shelf Solutions)

- Utilizing existing, pre-built application systems that can be adapted and tailored.
  - Often cheaper and faster to implement.
  - Design process shifts to configuration and customization.
  - Common for systems like medical records, ERP, or CRM.

OBJECT-ORIENTED DESIGN USING UML

- Object-Oriented Design (OOD) is a development methodology that constructs a system as a collection of interacting **objects**.
  - Each object is an instance of a **class**, which acts as a blueprint defining the object's internal **state** (data, attributes) and **behavior** (operations, methods).
- A fundamental principle of OOD is **encapsulation**, meaning an object's state is private and can only be modified through its public operations. This creates modular, self-contained entities that are easier to understand, develop, and maintain because changes to one object's implementation are less likely to affect others.
- The Unified Modeling Language (UML) provides a standardized visual notation to express an object-oriented design, facilitating communication among developers and stakeholders.
- The process of developing a detailed OOD from a concept involves several key, often iterative, steps:
- Understand and define the context and external interactions.
- Design the system architecture.
- Identify the principal objects.
- Develop design models.
- Specify interfaces.

SYSTEM CONTEXT AND INTERACTIONS

- The initial phase of OOD is to establish the system's boundaries and its relationship with the external environment. This is crucial for defining the system's scope and how it will communicate with external entities (users, other systems, hardware).
- Two complementary UML models are used here:
  - ✓a **structural** view (System Context Model) and
  - ✓a **dynamic** view (Interaction Model).

SYSTEM CONTEXT AND INTERACTIONS

A. System Context Model (Structural View)

- This model identifies all external systems and actors that interact with the system being designed. It is typically represented with a simple block diagram, often a UML diagram using associations.
- Example (Figure 7.1): The context diagram for the weather station shows it interacts with three external entities:
  - **Weather Information System:** The central system that receives and processes weather data.
  - **Onboard Satellite System:** The hardware system used to establish a communication link.
  - **Control System:** A system used to send commands and control the weather station remotely.

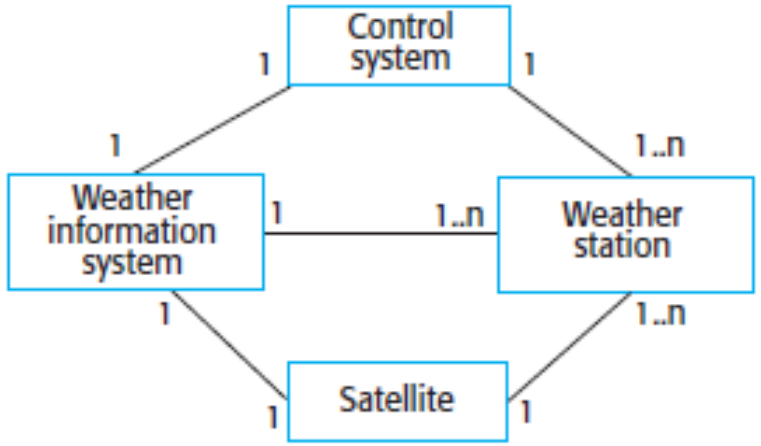

Figure 7.1: System context for the weather station

SYSTEM CONTEXT AND INTERACTIONS

B. Interaction Model (Dynamic View) - Use Case Model

- While the context model shows *who* the system interacts with, the use case model shows *what* interactions (goals) it performs.
- A use case describes a sequence of actions a system performs to yield an observable result for an **actor** (an external entity).
- Example (Figure 7.2): The use case diagram for the weather station identifies its main goals:
  - Report weather
  - Report status
  - Update
  - Shutdown
  - Configure
  - Restart

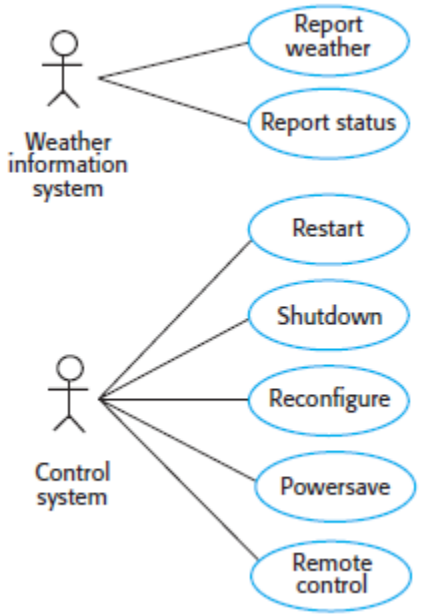

Figure 7.2: Weather station use cases

ARCHITECTURAL DESIGN

- Designing system architecture involves identifying the major structural components and the patterns that define how they interact. The goal is to create a robust, scalable, and maintainable high-level structure for the system.

Architectural Pattern Selection:

- The design must be informed by general architectural principles and domain-specific knowledge. A common approach is to use a known architectural pattern.
- Example - (Figure 7.3): The weather station software is decomposed into several key subsystems (Fault manager, Configuration manager, Communications, Data collection, Power manager, Instruments).

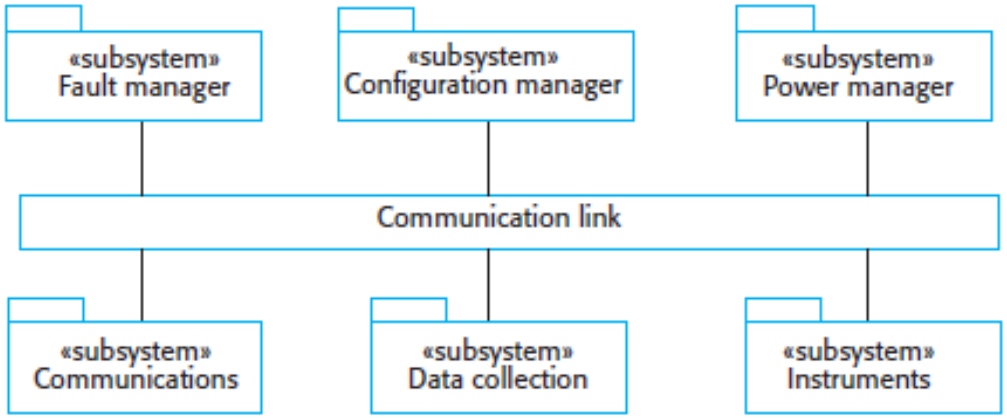

Figure 7.3: High-level architecture of weather station

ARCHITECTURAL DESIGN

- **Subsystem Decomposition** (Figure 7.4): The architecture of a single subsystem, Data collection, is further detailed. It shows internal objects (Transmitter, Receiver, WeatherData) and their relationships. This demonstrates how a high-level architectural component is itself designed using object-oriented principles.

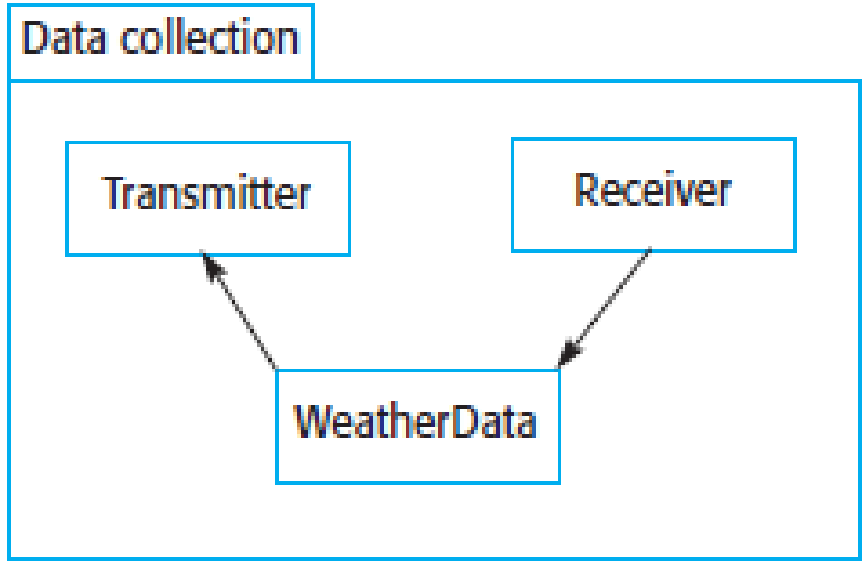

Figure 7.4: Architecture of data collection system

OBJECT CLASS IDENTIFICATION

- Object class identification is the critical process in object-oriented design (OOD) where designers discover and define the fundamental building blocks - the classes of the system.
- A class is a blueprint that defines the data (attributes) and behavior (operations) of the objects that will be created from it. This process transforms requirements and system descriptions into a structured software model.
- The process begins with the information gathered from earlier stages, particularly **use case descriptions**. These descriptions are a rich source for identifying potential objects, attributes, and operations.

OBJECT CLASS IDENTIFICATION

From Use Cases to Initial Candidates

- **Analyzing the "Report Weather" Use Case (Figure 7.2):**
  - **Nouns/Noun Phrases:** "weather station," "summary," "weather data," "instruments," "ground temperatures," "air pressures," "wind speeds," "rainfall," "collection period," "satellite communication link," "weather information system." These suggest potential **classes** or **attributes**.
  - **Verbs/Verb Phrases:** "sends," "collected," "establishes," "requests," "transmission." These suggest potential **operations** or **services**.
- From this analysis, it becomes obvious that the system will need at least three types of classes:
  - **Classes representing physical instruments** (e.g., thermometer, barometer).
  - **A class representing the summary of data** (to handle max, min, avg calculations).
  - **A high-level class encapsulating system interactions** (to handle the request and response).

OBJECT CLASS IDENTIFICATION

Established Techniques for Discovery

Over time, formal techniques have been developed to make this process more systematic:

1. **Grammatical Analysis (Abbott, 1983):**
   - This is the method demonstrated above. It involves parsing the natural language description of the system. *This is a good starting point but requires refinement, as not all nouns become classes.*
2. **Conceptual Category Analysis (Wirfs-Brock et al., 1990):**
   - This method suggests looking for classes in specific categories:
     - **Tangible Entities:** Physical things (e.g., GroundThermometer, Anemometer).
     - **Roles:** Functions performed (e.g., a DataController).
     - **Events:** Things that happen (e.g., DataRequestEvent).
     - **Interactions:** Transactions (e.g., CommunicationLink).
   - *This helps ensure a comprehensive search beyond obvious physical objects.*
3. **Scenario-Based Analysis (Beck and Cunningham, 1989):**
   - This involves walking through use case scenarios step-by-step and asking, for each action:
     - "What object is responsible for performing this action?"
     - "What information does that object need?"
     - *This technique is excellent for assigning responsibilities among objects and defining their operations.*

OBJECT CLASS IDENTIFICATION

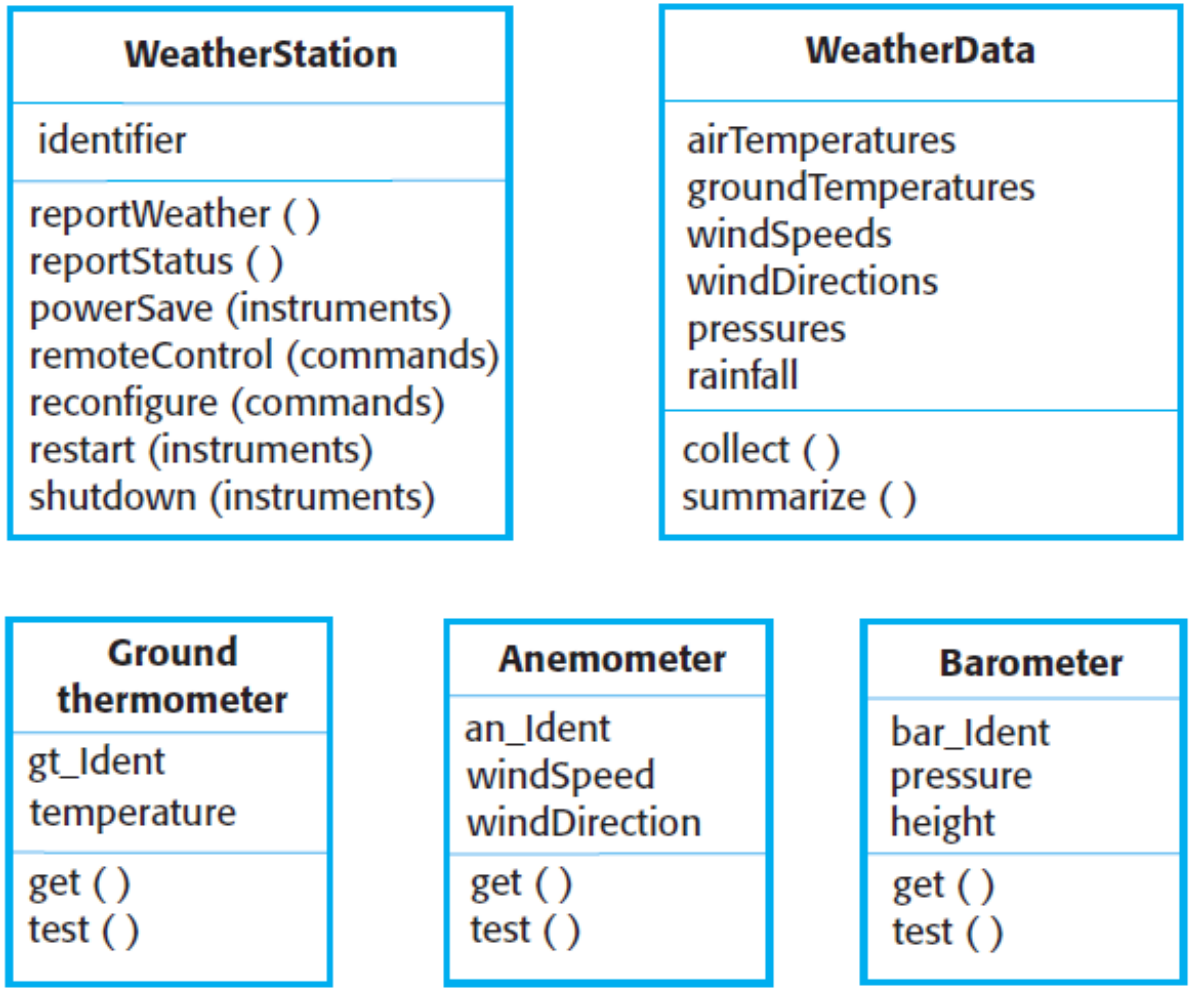

Figure 7.5: Weather Station Objects

DESIGN MODELS

Design models show the objects or object classes in a system and the relationships between them. They serve as a crucial bridge between system requirements and implementation, being abstract enough to avoid unnecessary detail yet detailed enough to guide programmers.

1. Purpose and Level of Detail:

The required level of detail depends on the development process.

- **Agile/Co-located Teams:** Abstract models or whiteboard sketches may be sufficient, with details resolved through informal discussion during implementation.
- **Plan-based/Distributed Teams:** More detailed and precise models are necessary to ensure a common understanding and facilitate communication between teams that may be in different locations.

2. Types of UML Design Models:

Two fundamental kinds of models:

- **Structural Models:** Describe the static structure of the system using object classes and their relationships (e.g., inheritance, associations). An example is a **Subsystem Model** (like Figure 7.4), which groups logically related objects into packages.
- **Dynamic Models:** Describe the dynamic, runtime interactions between objects. Three particularly useful types are highlighted:
  - **Sequence Models:** Show the sequence of object interactions over time.
  - **State Machine Models:** Show how a single object or subsystem changes state in response to events.

DESIGN MODELS

Sequence Models

- **Purpose:** To model the flow of messages between objects for a specific interaction, typically derived from a use case.
- **Example (Fig 7.6):** This sequence diagram details the "Report weather" use case.
  - It reads from top to bottom, showing the chronological order of messages.
  - It distinguishes between asynchronous messages (stick arrowhead, like request(report)) where the sender doesn't wait, and synchronous messages (solid arrowhead, like get(summary)) where the sender waits for a reply.
  - It identifies key objects (SatComms, WeatherStation, Commslink, WeatherData) and their responsibilities in the interaction, clarifying how the overall behavior is achieved through collaboration.

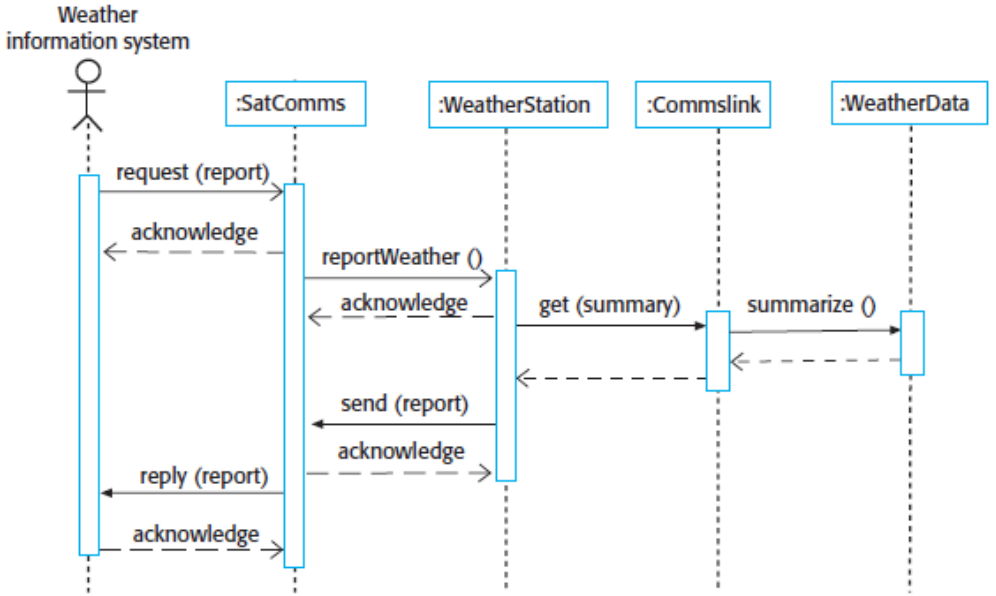

Figure 7.6: Sequence diagram describing data collection

DESIGN MODELS

State Machine Models

- **Purpose:** To model the behavior of a single object or subsystem by showing its various states and the events that cause transitions between those states.
- **Example (Fig 7.7):** This state diagram for the WeatherStation object shows its lifecycle.
  - It starts in the Shutdown state and transitions to others (Running, Collecting, Summarizing, Transmitting) based on events like reportWeather(), clock signals, or remoteControl().
  - It effectively summarizes the system's complex behavior in response to multiple stimuli, showing which operations are valid in which state (e.g., you can't reconfigure while Running).

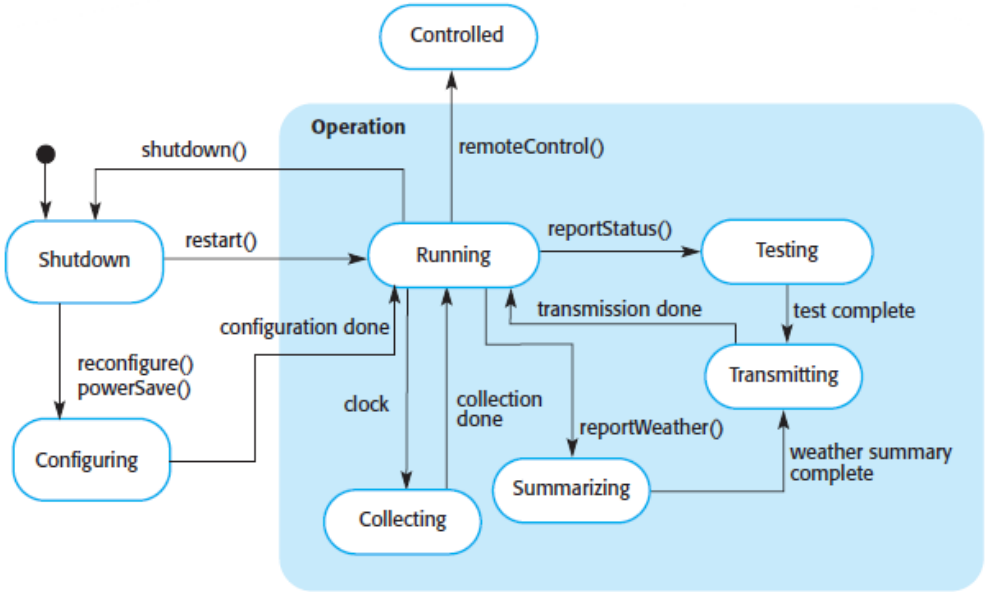

Figure 7.7: Weather station state diagram

INTERFACE SPECIFICATION

Interface specification is a critical design activity that enables parallel development by defining clear contracts that components must adhere to.

What is an Interface?

- An interface defines the **signatures** (names, parameters) and **semantics** (meaning, behavior) of the services provided by an object or a group of objects, without specifying any implementation details.
- It hides the internal data representation, making the design more maintainable. For example, changing how a data structure is implemented internally doesn't affect other objects that use its interface.

UML Notation and Principles:

- Interfaces are specified in UML using a class-like rectangle with the «interface» stereotype. Notably, they have **no attributes section**; only operations are defined.
- There is not a simple 1:1 relationship between objects and interfaces. A single object can implement multiple interfaces, and a single interface can provide a view onto a group of objects.

INTERFACE SPECIFICATION

The figure shows two interfaces for the weather station:

- «interface» Reporting**:**

  Defines operations for generating reports (weatherReport, statusReport). These likely map directly to operations in the WeatherStation object.

- «interface» Remote Control**:**

  Defines operations for instrument control (startInstrument, stopInstrument, etc.). These four operations are shown mapping onto a *single* method (remoteControl()) in the WeatherStation object. The specific command would be encoded in a string parameter, demonstrating how a coarse-grained object method can support a finer-grained interface.

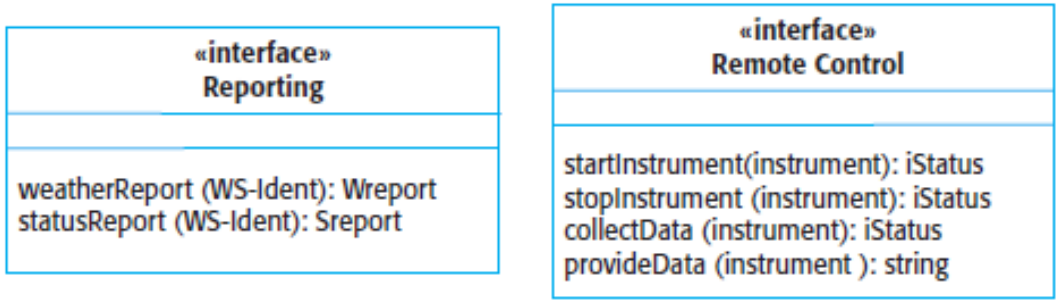

Figure 7.8: Weather Station Interfaces

DESIGN PATTERNS

In software engineering, a **design pattern** is a general, reusable solution to a commonly occurring problem within a given context in software design. It is not a finished piece of code but a **description or template** for how to solve a problem that can be used in many different situations.

The Role and Impact of Patterns

- Patterns have had a profound impact on object-oriented design, serving two key purposes:
- **Tested Solutions:** They provide reliable, proven solutions to recurring design problems.
- **Design Vocabulary:** They create a shared language for developers to communicate complex design ideas efficiently. For example, saying "we used a Singleton" instantly conveys a specific design approach to another designer.

Using Patterns in Design

- Using patterns effectively involves a process of recognition and application:
  - **Recognize the Problem:** During design, you encounter a problem (e.g., "I need to notify several objects about a change in state").
  - **Identify a Pattern:** You recall that a known pattern (e.g., the Observer pattern) provides a solution to this class of problem.
  - **Adapt the Template:** You then adapt the pattern's abstract solution template to the specific context of your system.
- This approach promotes **high-level concept reuse** rather than code reuse.

DESIGN PATTERNS

The Role and Importance of Patterns

- **Reuse of Expertise:** Allows less experienced developers to leverage the knowledge of experts.
- **Improved Communication:** Provides a standard terminology that simplifies discussion about design.
- **Documentation:** Patterns provide a higher-level, more intuitive description of a system's design.
- **Maintainability:** Patterns often lead to more flexible and modular designs that are easier to change.

IMPLEMENTATION ISSUES

- Implementation, the stage where software design is translated into a working system, involves far more than just writing code. It encompasses critical engineering decisions and processes that ensure the final product is robust, maintainable, and delivered efficiently.
- Three of the most significant implementation issues in modern software engineering are:
  1. **Reuse**
  2. **Configuration Management**
  3. **Host-Target Development**.

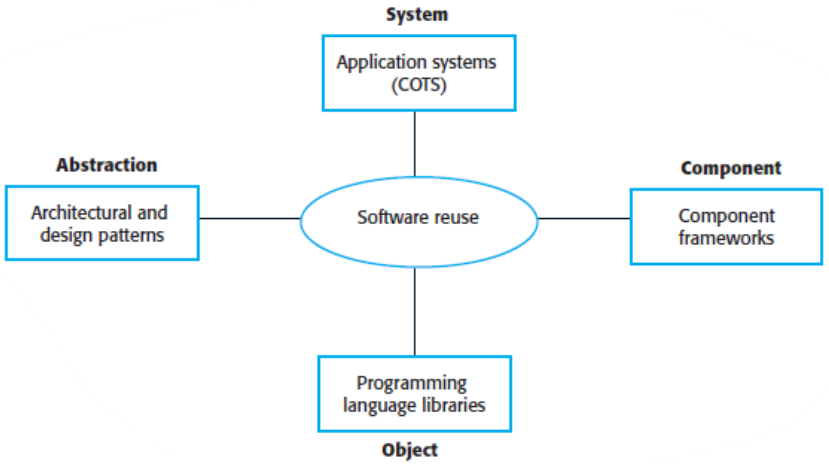

Figure 7.9: Software reuse

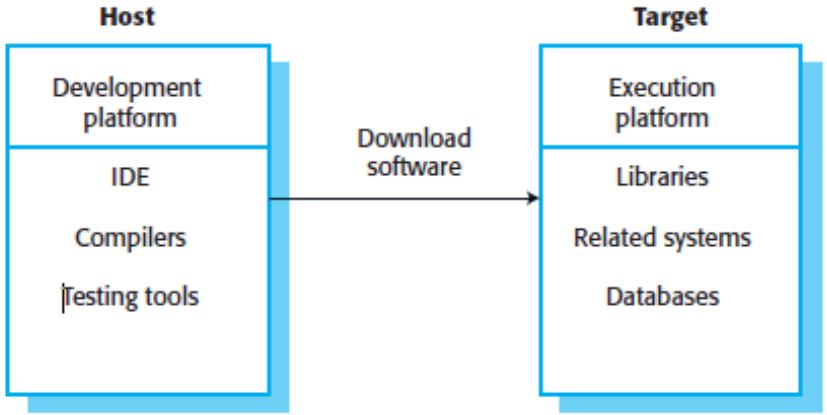

Figure 7.10: Host-target development

IMPLEMENTATION ISSUES

1. Reuse

The paradigm of building every system from scratch is largely obsolete. Modern software development is predominantly **reuse-based**, leveraging existing software assets to accelerate development, reduce costs, and improve reliability.

- **Levels of Reuse:** Reuse occurs at multiple levels of abstraction:
  - **Abstraction Level:** Reusing knowledge and patterns rather than code itself
    e.g., using well-known **Architectural and Design Patterns**
  - **Object Level:** Directly reusing objects from **Programming Language Libraries**
    e.g., using a StringUtils library in Java instead of writing string manipulation code
  - **Component Level:** Reusing larger, cohesive collections of objects that provide related services
    e.g., using a **Component Framework** like React or Angular to build a user interface
  - **System Level:** Reusing entire **Application Systems (COTS - Commercial Off-The-Shelf)**. This involves configuring and integrating large systems (like an ERP or CRM) to meet organizational needs, often creating new systems by combining several others.

IMPLEMENTATION ISSUES

2. **Configuration Management**

Software development is a process of constant change. **Configuration Management (CM)** is the discipline of managing these changes in a controlled way, ensuring team collaboration is smooth and the integrity of the software is maintained. It is essential for any multi-person project.

- Configuration Management encompasses four key activities:
  - **Version Management:** Tracks every change made to every component, preventing developers from overwriting each other's work. Tools like **Git** and **Subversion** are central to this, allowing teams to work concurrently and merge changes.
  - **System Integration (Build Management):** Defines how to assemble specific versions of components into a complete, working system. This process is often automated to ensure consistent and repeatable builds.
  - **Problem Tracking (Issue Tracking):** Manages the reporting, assignment, and resolution of bugs and other issues. Systems like **Bugzilla** or **Jira** provide visibility into who is fixing what and what remains to be done.
  - **Release Management:** Plans and controls the distribution of new software versions to customers, ensuring that releases are packaged and delivered correctly.

IMPLEMENTATION ISSUES

3. Host-Target Development

The **host-target model** separates the **development platform** (the host) from the **execution platform** (the target). The host system is where developers work, equipped with tools like **IDEs, compilers, and testing tools**. The target system is where the software will ultimately be deployed and run. These platforms can be the same type but are often different, especially in embedded systems.

- **Challenges and Solutions:**
  - **Testing:** A major challenge is testing software destined for a different target. Solutions include:
    - **Simulators:** Software that mimics the target hardware on the host system, allowing developers to test without physical hardware. These are crucial for embedded systems development but can be expensive to create.
    - **Cross-Compilation:** Compiling code on the host to produce executable code for the target architecture.
    - **Direct Deployment:** Transferring the built software to the target hardware for testing, which can be slow and may involve licensing issues for required middleware.

IMPLEMENTATION ISSUES

- **Integrated Development Environments (IDEs):** Host development relies heavily on IDEs like **Eclipse** or IntelliJ IDEA. These provide a framework that integrates all necessary tools—editing, compiling, debugging, testing, and configuration management—into a single environment, significantly boosting productivity.
- **Deployment Decisions:** For distributed systems, implementation involves deciding *where* to deploy components across various target platforms. Decisions are based on:
  - **Hardware/Software Requirements** of the component.
  - **Availability Requirements** (e.g., deploying redundant components for fault tolerance).
  - **Communication Latency** (placing frequently communicating components on the same or nearby platforms).

OPEN-SOURCE DEVELOPMENT

- Open-source development is a software development methodology where the source code of a system is made publicly available, and volunteers are invited to participate in its development process.
- It represents a significant shift from the traditional proprietary model, emphasizing collaboration, transparency, and community-driven innovation.

Significance and Prevalence

- Open-source software is fundamental to modern computing and forms the backbone of the internet and software engineering. Key examples include:
- **Operating Systems:** Linux (dominant on servers), Android (on mobile devices).
- **Web Servers:** Apache.
- **Development Tools:** Java programming language, Eclipse IDE.
- **Databases:** MySQL.
  Major corporations like IBM and Oracle actively support and build upon open-source products, validating its importance in the industry.

ORIGINS AND PRINCIPLES OF OPEN-SOURCE DEVELOPMENT

Origins

- The philosophical foundation of open-source development originated from the Free Software Movement.
- The movement advocated against proprietary (closed) source code, arguing that code should not be kept secret.

Core Principles

- **Source Code Accessibility:** Source code must always be made available for anyone to:
  - **Study:** Examine and understand how the software works.
  - **Modify:** Change and adapt the software to meet their own needs.
  - **Distribute:** Share the original or modified versions with others.
- **Expansion via the Internet:** The open-source model leveraged the Internet to scale participation, recruiting a large, global community of volunteer developers.
- **User-Developer Community:** A key characteristic is that many of the contributors are also users of the software, creating a direct feedback loop.
- In practice, while anyone can contribute, successful projects are typically governed by a **core group** of trusted developers. This group:
  - Maintains control over the official version of the software.
  - Is responsible for approving changes.
  - Ensures the overall quality and coherence of the project.

OPEN-SOURCE DEVELOPMENT

Benefits and Advantages

- **Cost:** The software is typically free to acquire and download.
- **Reliability:** Widely used open-source systems are exceptionally reliable. A large user base means bugs are discovered and fixed rapidly by the community, often faster than in proprietary software where users must wait for an official patch.
- **Support and Longevity:** While official documentation and support may have a cost, the open nature of the code assures users that they will never lose access to it, even if the original developer ceases operations. This reduces vendor lock-in.

OPEN-SOURCE DEVELOPMENT

Open-Source Licensing

- A fundamental misconception is that "open source" means "no rules." Legally, the code's owner (an individual or company) can place restrictions on its use through licenses. Understanding these licenses is crucial to avoid legal complications. The three primary license models are:
- **GNU General Public License (GPL):** A **reciprocal** or "copyleft" license. This is a restrictive license. If you use GPL-licensed code in your software, your entire software must also be released as open source under the GPL.
- **GNU Lesser General Public License (LGPL):** Less restrictive than GPL. You can write components that *link to* LGPL code (e.g., via a library) without making your components open source. However, any modifications to the LGPL-licensed code itself must be published.
- **Berkley Standard Distribution (BSD) License / MIT License: Non-reciprocal** and very permissive. You can use, modify, and integrate the code into proprietary, closed-source software that is sold, as long as you acknowledge the original creator. This is the most business-friendly license.