CHAPTER 7

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*1-20

\*\*Design and Implementation in Software Engineering:\*\*

1. \*\*Software Design and Implementation:\*\*

- This stage involves developing an executable software system.

- Design and implementation activities are intertwined.

- Design is a creative process identifying software components and their relationships.

- Implementation is the realization of the design into a program.

2. \*\*Build or Buy:\*\*

- Off-the-shelf systems (COTS) are available for adaptation to user requirements.

- Buying existing systems can be faster and more cost-effective.

- Design process focuses on configuring features of the purchased system to meet requirements.

3. \*\*Object-Oriented Design Process:\*\*

- Structured object-oriented design involves developing different system models.

- Significant effort is required for model development and maintenance, especially for large systems by different groups.

- Design models serve as crucial communication mechanisms.

4. \*\*Process Stages:\*\*

- Various object-oriented design processes exist, and common activities include defining context, designing system architecture, identifying principal objects, developing design models, and specifying object interfaces.

- Illustrated using the example of designing a wilderness weather station.

5. \*\*System Context and Interactions:\*\*

- Understanding relationships between software and its environment is crucial.

- Establishing system boundaries helps decide what features are within the system and what is in associated systems.

- Context and interaction models aid in demonstrating relationships with the environment.

6. \*\*Use Case Description - Report Weather:\*\*

- Describes a use case involving the interaction between a weather station and a weather information system.

- Actors include the weather information system and the weather station.

- Stimulus, response, and comments provide detailed information about the use case.

7. \*\*Architectural Design:\*\*

- Involves identifying major components, their interactions, and organizing them using architectural patterns like layered or client-server models.

- Example: The weather station comprises independent subsystems communicating through a common infrastructure.

8. \*\*Object Class Identification:\*\*

- Identifying object classes is challenging and relies on designer skill, experience, and domain knowledge.

- Approaches include grammatical, tangible things, behavioral, and scenario-based analyses.

- Iterative process with likely refinements.

9. \*\*Weather Station Description and Object Classes:\*\*

- The weather station includes software-controlled instruments for data collection and transmission.

- Object class identification is based on tangible hardware and data in the system.

- Identified classes include ground thermometer, anemometer, barometer, weather station, and weather data.

10. \*\*Design Models:\*\*

- Design models illustrate objects, object classes, and relationships.

- Static models depict the static structure in terms of object classes and relationships.

- Dynamic models describe dynamic interactions between objects.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 21 – 40

\*\*Examples of Design Models:\*\*

1. \*\*Subsystem Models:\*\*

- Logical groupings of objects into coherent subsystems.

- Shown using packages and encapsulation in UML.

- Actual organization may differ.

2. \*\*Sequence Models:\*\*

- Display object interactions sequence.

- Objects arranged horizontally, time represented vertically.

- Interactions represented by labeled arrows.

3. \*\*State Machine Models:\*\*

- Show object responses to service requests and state transitions.

- Useful high-level models for system or object runtime behavior.

- Not required for all objects; avoids unnecessary detail.

4. \*\*Interface Specification:\*\*

- Object interfaces must be precisely specified for parallel design.

- Designers hide interface representation within objects.

- UML uses class diagrams for interface specification.

\*\*Key Points:\*\*

- Software design and implementation are interleaved; detail level depends on system type and development approach (plan-driven or agile).

- Object-oriented design involves designing system architecture, identifying objects, describing design using various object models, and documenting component interfaces.

- Object-oriented design produces static (class, generalization, association) and dynamic (sequence, state machine) models.

- Component interfaces must be precisely defined for usage by other objects. UML interface stereotype may be used.

\*\*Design Patterns:\*\*

- Design pattern is a way to reuse abstract knowledge about a problem and its solution.

- Consists of name, problem description, solution description, and consequences.

- Abstract enough for reuse in different settings.

- Utilizes object-oriented characteristics like inheritance and polymorphism.

\*\*The Observer Pattern:\*\*

- \*\*Description:\*\* Separates display of an object's state from the object itself; allows multiple displays to be provided.

- \*\*Problem:\*\* Multiple displays of state needed.

- \*\*Solution:\*\* Abstract objects Subject and Observer, and Concrete objects ConcreteSubject and ConcreteObserver.

- \*\*Consequences:\*\* Minimal coupling between objects; optimizations for display performance are impractical.

\*\*Design Problems:\*\*

- Recognizing design problems that can be solved by patterns:

i. Notifying objects about state changes (Observer pattern).

ii. Simplifying interfaces of related objects (Façade pattern).

iii. Providing a standard way to access elements in a collection (Iterator pattern).

iv. Allowing for runtime extension of existing class functionality (Decorator pattern).

\*\*Implementation Issues:\*\*

- Focus on non-programming implementation issues:

i. \*\*Reuse:\*\* Modern software is developed by reusing existing components or systems.

ii. \*\*Configuration Management:\*\* Tracking different versions of each software component in a configuration management system.

iii. \*\*Host Target Development:\*\* Developing on one computer (host system) and executing on another (target system).

\*\*Reuse:\*\*

- From the 1960s to the 1990s, software was mostly developed from scratch.

- Increasingly unviable due to costs and schedule pressure.

- Emergence of an approach based on the reuse of existing software, widely used for business and scientific software.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 41 – 56

\*\*Reuse Levels:\*\*

1. \*\*Abstraction Level:\*\*

- Reuse knowledge of successful abstractions in software design.

2. \*\*Object Level:\*\*

- Directly reuse objects from a library rather than writing code.

3. \*\*Component Level:\*\*

- Reuse collections of objects and object classes in application systems.

4. \*\*System Level:\*\*

- Reuse entire application systems.

\*\*Reuse Costs:\*\*

1. \*\*Search and Assessment:\*\*

- Time spent looking for reusable software and assessing suitability.

2. \*\*Acquisition:\*\*

- Costs of buying reusable software, especially for large off-the-shelf systems.

3. \*\*Adaptation and Configuration:\*\*

- Costs of adapting and configuring reusable software components or systems.

4. \*\*Integration:\*\*

- Costs of integrating reusable software elements with each other and new code.

\*\*Configuration Management:\*\*

- Process managing changes to a changing software system.

- Supports system integration, providing controlled access, change tracking, and component compilation and linking.

\*\*Configuration Management Activities:\*\*

1. \*\*Version Management:\*\*

- Keeps track of different versions of software components.

2. \*\*System Integration:\*\*

- Helps define versions used to create each system version.

3. \*\*Problem Tracking:\*\*

- Allows reporting and tracking of bugs and issues.

\*\*Host-Target Development:\*\*

- Software developed on one computer (host) but runs on a separate machine (target).

- Development platform and execution platform may differ in architecture and installed software.

\*\*Development Platform Tools:\*\*

- Integrated compiler and syntax-directed editing system.

- Language debugging system.

- Graphical editing tools (e.g., UML model editors).

- Testing tools (e.g., Junit for automated tests).

- Project support tools for organizing code in different projects.

\*\*Integrated Development Environments (IDEs):\*\*

- Set of software tools supporting various software development aspects within a common framework and UI.

- Specific IDEs for programming languages (e.g., Java).

\*\*Component/System Deployment Factors:\*\*

- Considerations for deploying components:

i. Specific hardware architecture.

ii. High availability may require deployment on multiple platforms.

iii. Communication traffic between components may influence deployment proximity.

\*\*Open Source Development:\*\*

- Approach where source code is published, and volunteers participate in development.

- Rooted in the Free Software Foundation's ideology of freely available source code.

\*\*Open Source Systems:\*\*

- Examples include Linux, Java, Apache web server, and MySQL database.

\*\*Open Source Issues:\*\*

- Considerations:

i. Use of open source components.

ii. Adoption of an open source approach for development.

\*\*Open Source Business:\*\*

- More product companies use an open source approach.

- Business model relies on selling support rather than the software product itself.

\*\*Open Source Licensing:\*\*

- Open source licenses:

i. \*\*GPL (GNU General Public License):\*\*

- Reciprocal license; using GPL software necessitates making the software open source.

ii. \*\*LGPL (GNU Lesser General Public License):\*\*

- Variant allowing linking to open source code without publishing linked components' source.

iii. \*\*BSD (Berkley Standard Distribution) License:\*\*

- Non-reciprocal license; no obligation to republish changes.

\*\*License Management:\*\*

- Establish a system for maintaining information about used open source components.

- Understand different licenses and component evolution pathways.

- Educate about open source, have auditing systems, and participate in the community.

\*\*Key Points:\*\*

- Consider reusing existing software in various forms during software development.

- Configuration management is vital for teams cooperating in software development.

- Most software development involves host-target development, using an IDE on a host machine.

- Open source development involves making source code publicly available, fostering collaboration and improvement.