**CMP 403- SOFTWARE ENGINERING**

**SUMMARY OF CHAPTER 6 & 7**

**GROUP MEMBERS**

1. AIYEWUMI DAVID OLUWOLE BHU/20/04/09/0011
2. ANOINTING EBUBE DAUDA BHU/20/04/05/0032
3. ORJI CHINONYEREM PEACE BHU/20/04/09/0007
4. CHUKWU DANIEL NONSO BHU/20/04/05/0010
5. AROMOLARAN ADENIKE ELIZABETH BHU/20/04/05/0136
6. JOLLY D JOSEPH BHU/20/04/05/0079
7. ABRAHAM BENJAMIN ORUME BHU/20/04/05/0145
8. AMBINKANME MBOSIRE MUSA BHU/20/04/05/0040
9. MAIGIDA GYENOM TALATU BHU/20/04/05/0013
10. DANJUMA KUYET DIVINE BHU/20/04/05/0011
11. BENCALVIN ISAAC AMBI BHU/20/04/05/0100

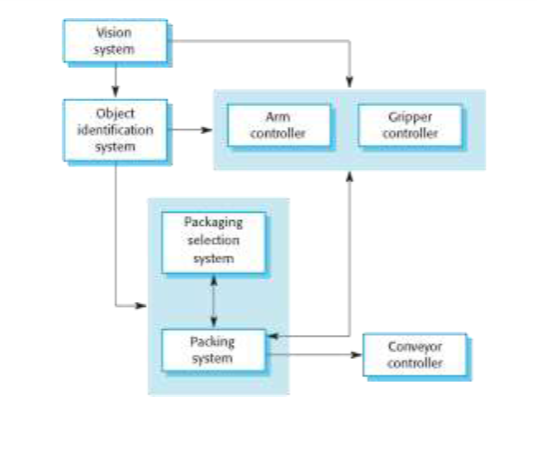
**CHAPTER 6**

**1. Definition and Purpose:**

Software architecture involves identifying subsystems, their framework for control, and communication.

Architectural design is an early stage in system design, connecting specification and design processes.

Figure 1.1: *The architecture of a packing robot control system.*



**2. Architectural Abstraction:**

* Architecture in the small deals with individual program structures.
* Architecture in the large involves complex enterprise systems distributed across multiple computers.

**3. Advantages of Explicit Architecture:**

* Facilitates stakeholder communication.
* Enables system analysis for non-functional requirements.
* Supports large-scale reuse, including product-line architectures.

**4. Architectural Representations:**

* Simple block diagrams are commonly used but criticized for lacking semantics.
* Box and line diagrams, though abstract, are useful for communication and project planning.

**5. Use of Architectural Models:**

* Facilitates discussion and communication during system design.
* Documents the designed architecture, depicting components, interfaces, and connections.

**6. Architectural Design Decisions:**

* Various decisions impact the non-functional characteristics of the system.
* Decisions include selecting generic application architecture, distribution strategy, architectural styles, structural approach, module decomposition, control strategy, evaluation method, and documentation format.

**7. Architecture Reuse:**

* Systems in the same domain often share similar architectures.
* Application product lines utilize a core architecture with variants.
* Systems can be designed around architectural patterns or styles.

**8. Architecture and System Characteristics:**

* Considerations for performance, security, safety, availability, and maintainability.

**9. Architectural Views:**

* Different views or perspectives are useful for designing and documenting system architecture.
* Multiple views, such as decomposition, runtime processes, and distribution of components, are necessary.

**10. 4 + 1 View Model:**

* A model with logical, process, development, and physical views, related through use cases or scenarios.

**11. Architectural Patterns:**

* Patterns are a means of representing, sharing, and reusing knowledge.
* Architectural patterns are stylized descriptions of good design practice.
* Patterns should indicate when they are and are not useful.

**12. Model-View-Controller (MVC) Pattern:**

* Separates presentation and interaction from system data.
* Comprises three logical components: Model, View, and Controller.
* Used when there are multiple ways to view and interact with data, especially when future requirements are unknown.
* Advantages include independent changes to data and its representation, supporting different presentations of the same data. However, it may involve additional code complexity for simple data models and interactions.

Figure 1.2: *The organization of the Model-View-Controller*

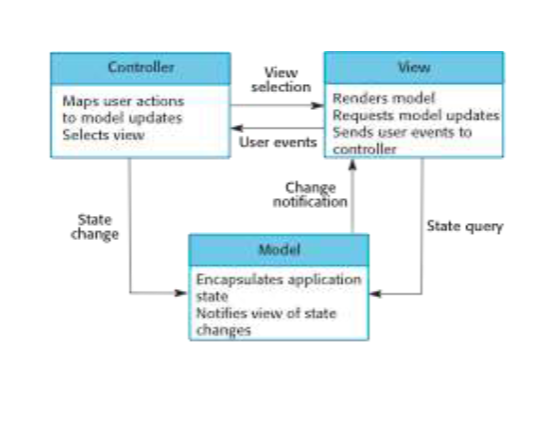
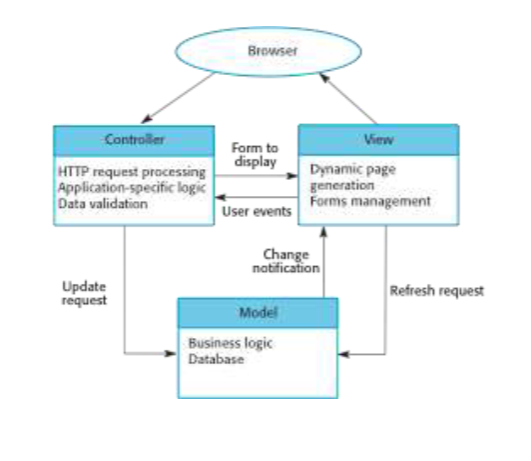


Figure 1.3: *Web application architecture using the MVC pattern*

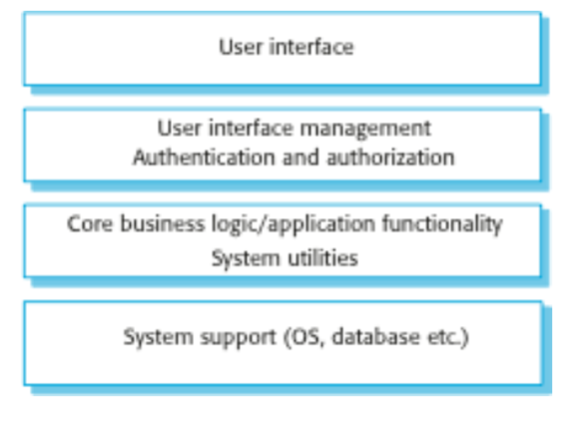


The document also includes diagrams illustrating the organization of the Model-View-Controller and a web application architecture using the MVC pattern.

**13. Layered Architecture:**

* Used for modeling subsystem interfaces, organizing the system into layers.
* Supports incremental development, where changes in one layer affect only the adjacent layer.
* May be artificial in practice.

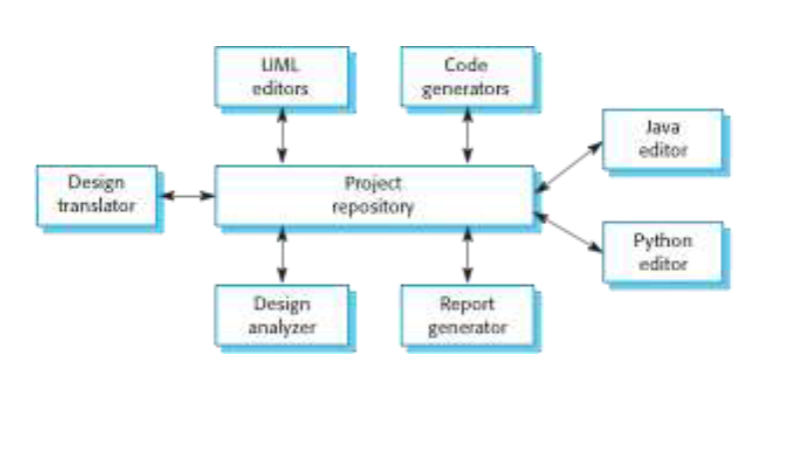
Figure 1.4: *A generic layered architecture*



**14. Repository Architecture:**

* Data exchange can be through a central repository or explicit passing between subsystems.
* Repository pattern manages all system data centrally, allowing components to access it indirectly.
* Suitable for systems generating large volumes of long-term data.

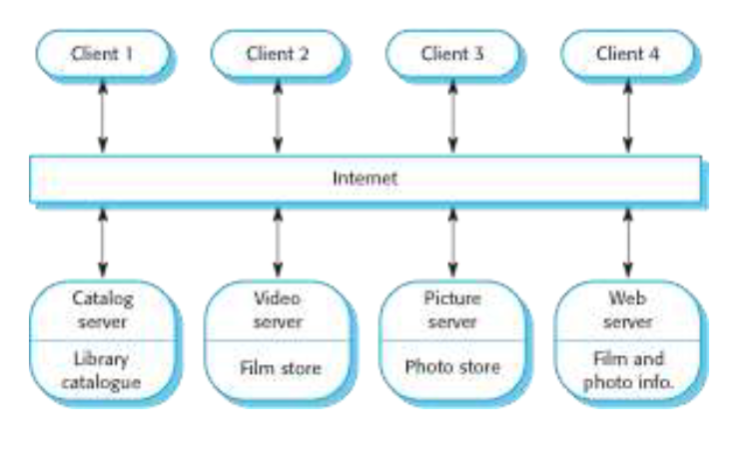
Figure 1.5: *A repository architecture for an IDE*



**15. Client-Server Architecture:**

* Organizes functionality into services delivered from separate servers, accessed by clients.
* Suitable for shared database access or variable system loads.
* Servers can be distributed, but each service is a single point of failure.

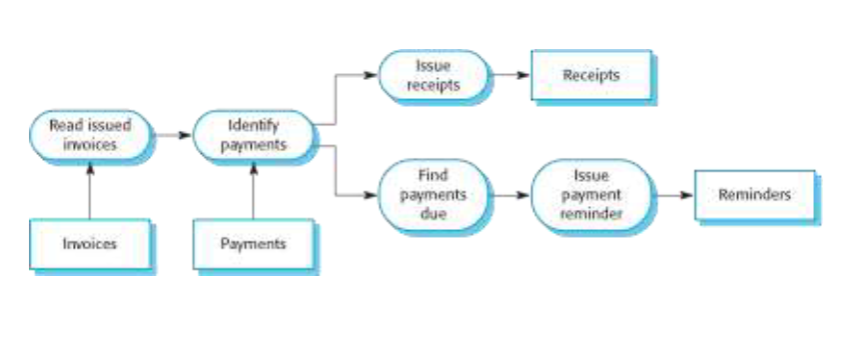
Figure 1.6: *A client-server architecture for a film library*



**16. Pipe and Filter Architecture:**

* Functional transformations are discrete components processing data sequentially.
* Commonly used in data processing applications, both batch- and transaction-based.
* Supports easy understanding and transformation reuse but requires agreed data format between components.

Figure 1.7: *An example of the pipe and filter architecture*



**17. Application Architectures:**

* Systems designed to meet organizational needs often share common architectures.
* Generic application architectures can be adapted for specific requirements.
* Used as a starting point for design, checklist, team organization, component assessment, and vocabulary for application types.

**18. Transaction Processing Systems:**

* Handle user requests for information or updates in a database.
* Transactions are coherent sequences of operations satisfying a goal.
* Users make asynchronous requests processed by a transaction manager.

**19. Transaction Processing Applications:**

* Presents the structure of transaction processing applications.
* Features the software architecture of an ATM system.

Figure 1.8: *The structure of transaction processing applications*

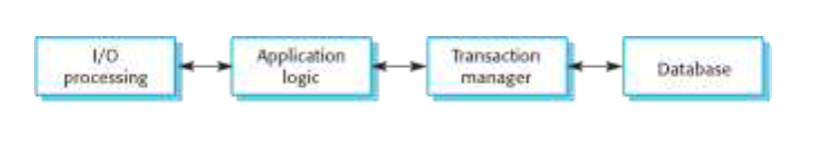
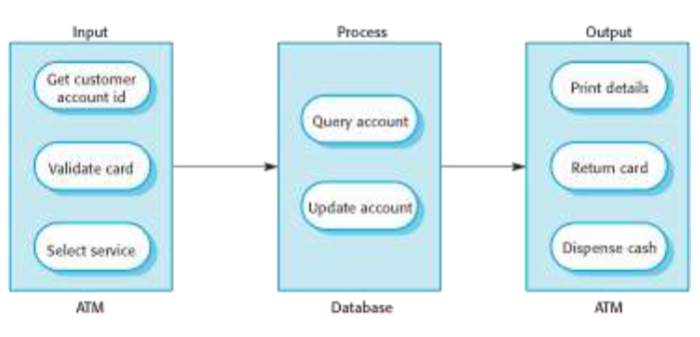


Figure 1.9: *The software architecture of an ATM system*



**20. Information Systems Architecture:**

* Information systems typically have a layered architecture, emphasizing transaction-based systems.
* Layers include user interface, user communications, information retrieval, and the system database.

**21. Web-based Information Systems:**

* Modern information and resource management systems are web-based.
* E-commerce systems are highlighted as examples, implementing a layered structure with a specific application layer for tasks like shopping cart functionality.

**22. Server Implementation:**

* Describes the implementation of web-based systems using multi-tier client-server architectures.
* Components include web servers for user communications, application servers for logic and data, and database servers for transaction management.

**23. Language Processing Systems:**

* Examines systems that accept natural or artificial language as input and generate alternative representations.
* Meta-case tools are mentioned, which process descriptions and rules to generate tools.

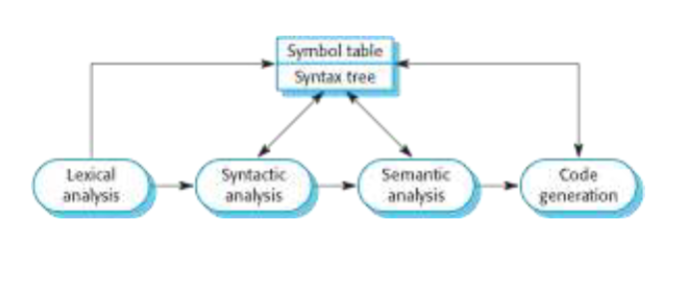
**24. Compiler Components:**

* Discusses the components of a language compiler, including lexical analyzers, symbol tables, syntax analyzers, syntax trees, semantic analyzers, and code generators.

**25. Pipe and Filter Compiler Architecture:**

* Illustrates a pipe and filter architecture for a compiler, emphasizing sequential processing stages.

Figure 1.10: *A pipe and filter compiler architecture*



**26. Key Points:**

* Highlights the importance of application system architecture models for understanding, comparison, validation, and assessing components for reuse.
* Defines transaction processing systems as interactive systems allowing remote database access 0and modification by multiple users.
* Describes language processing systems as tools for translation and execution of instructions in a specified language, including a translator and an abstract machine.

**CHAPTER 7**

**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.

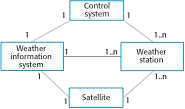
4. **Process Stages:**

* 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.

# Figure 1.1: *System context for the weather station*



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

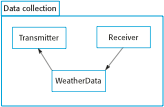
* Describes a use case involving the interaction between a weather station and a weather information system.
* Actors include the weather information system and the weather station.
* Stimulus, response, and comments provide detailed information about the use case.

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.

# Figure 1.2 *: Architecture of data collection system*



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.

**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.

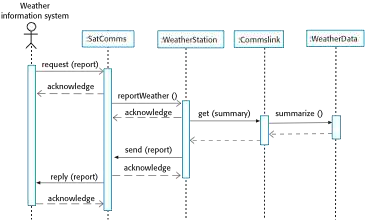
2. **Sequence Models:**

- Display object interactions sequence.

- Objects arranged horizontally, time represented vertically.

- Interactions represented by labeled arrows.

# Figure 1.3*: Sequence diagram describing data collection*



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 Concrete Subject and Concrete Observer.

- 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.

**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.