Answer Key

Q.1 Multiple Choice Questions

1. **c) Requirement Analysis**
2. **d) Hardware Model**
3. **c) The system should allow users to create accounts.**
4. **a) To ensure that the requirements are complete and correct.**
5. **c) Interviews**
6. **b) To control changes to requirements throughout the project lifecycle.**
7. **d) Testing Model**
8. **b) The interaction between objects in a system over time**
9. **b) Structural Model**
10. **a) Reduced development time and cost through automation.**
11. **b) To define the high-level structure and organization of the system.**
12. **b) Singleton**

Q.2 Solve the following:

A. Explain the different types of software requirements.

Software requirements are descriptions of the features and functionalities of the target system. Requirements convey the expectations of users. They can be classified into several types:

1. **Functional Requirements:** These describe *what* the system should do. They define the specific functions or services that the system is expected to provide.
    - Example: "The system shall allow users to create, update, and delete accounts."
2. **Non-Functional Requirements:** These describe *how* the system should perform. They specify the quality attributes, constraints, or characteristics of the system.
    - **Performance Requirements:** Response time, throughput, resource usage.
        - Example: "The system shall respond to user requests within 2 seconds."
    - **Security Requirements:** Confidentiality, integrity, availability.
        - Example: "The system shall protect user data using encryption."
    - **Usability Requirements:** Ease of use, learnability, accessibility.
        - Example: "The system shall provide a user-friendly interface with clear navigation."
    - **Reliability Requirements:** Availability, fault tolerance, recovery.
        - Example: "The system shall be available 99.9% of the time."
    - **Maintainability Requirements:** Ease of modification, extensibility.
        - Example: "The system shall be designed to allow for easy addition of new features."
    - **Portability Requirements:** Adaptability to different platforms or environments.
        - Example: "The system shall be compatible with Windows, macOS, and Linux operating systems."
3. **Domain Requirements:** These requirements are specific to the application domain of the software.
    - Example: "The system shall comply with HIPAA regulations for patient data privacy" (in a healthcare application).
4. **Interface Requirements:** These specify how the system interacts with external systems or components.
    - Example: "The system shall integrate with the existing CRM system using a REST API."

5. **Legal Requirements:** These include regulatory and compliance requirements.
    - Example: "The system must comply with GDPR data privacy regulations."

B. Describe the importance of traceability in requirements management.

Traceability in requirements management is the ability to link each requirement back to its origin and forward to its implementation and verification. It is crucial for several reasons:

1. **Ensuring Completeness:** Traceability helps ensure that all requirements are addressed during development. By tracing requirements from their source (e.g., stakeholder needs) to their implementation (e.g., code modules), it is possible to verify that nothing has been overlooked.
2. **Managing Change:** When requirements change, traceability allows developers to identify all the system components that are affected by the change. This helps in assessing the impact of the change and making informed decisions about how to implement it.
3. **Supporting Verification and Validation:** Traceability facilitates verification and validation by linking requirements to test cases. This ensures that each requirement is properly tested and that the system meets the specified needs.
4. **Improving Communication:** Traceability provides a clear and auditable record of the relationships between requirements, design elements, code, and tests. This improves communication among stakeholders, developers, and testers.
5. **Facilitating Reuse:** When developing new systems or enhancing existing ones, traceability helps identify reusable components and requirements from previous projects.
6. **Risk Management:** By tracing requirements to potential risks, developers can prioritize risk mitigation efforts and allocate resources accordingly.
7. **Compliance:** In regulated industries (e.g., healthcare, finance), traceability is often a mandatory requirement for demonstrating compliance with industry standards and regulations.

In summary, traceability is essential for managing the complexity of software projects, ensuring quality, and meeting stakeholder needs. It provides a roadmap for the entire development lifecycle, from initial requirements to final delivery.

Q.3 Solve the following:

A. Discuss various requirements elicitation techniques, highlighting their advantages and disadvantages.

Requirements elicitation is the process of discovering, gathering, and understanding the needs and constraints of stakeholders for a software system. Various techniques can be used, each with its own advantages and disadvantages:

1. **Interviews:**
    - *Description:* Directly asking stakeholders questions about their needs and expectations.
    - *Advantages:*
        - Provides rich, detailed information.
        - Allows for clarification and follow-up questions.
        - Can uncover hidden or unspoken needs.
    - *Disadvantages:*
        - Time-consuming and resource-intensive.
        - Can be difficult to schedule and coordinate.

        - Success depends on the interviewer's skills.
        - Stakeholder availability and willingness to participate.
2. **Questionnaires/Surveys:**
    - *Description:* Distributing a set of questions to a large group of stakeholders.
    - *Advantages:*
        - Efficient way to gather information from many people.
        - Cost-effective.
        - Can be analyzed statistically.
    - *Disadvantages:*
        - Limited opportunity for follow-up questions.
        - May not capture the nuances of stakeholder needs.
        - Low response rates can be a problem.
        - Difficult to design effective and unbiased questionnaires.
3. **Workshops/Focus Groups:**
    - *Description:* Bringing stakeholders together in a collaborative setting to discuss requirements.
    - *Advantages:*
        - Encourages brainstorming and idea generation.
        - Facilitates communication and consensus-building.
        - Can uncover conflicting requirements.
    - *Disadvantages:*
        - Can be dominated by certain individuals.
        - Requires skilled facilitation.
        - Difficult to manage large groups.
        - Logistical challenges in scheduling and coordinating.
4. **Brainstorming:**
    - *Description:* A group creativity technique designed to generate a large number of ideas in a short period of time.
    - *Advantages:*
        - Encourages innovative thinking.
        - Relatively quick and easy to conduct.
    - *Disadvantages:*
        - Can be unfocused if not properly managed.
        - May generate many irrelevant ideas.
5. **Use Cases:**
    - *Description:* Describing the interactions between users and the system to achieve specific goals.
    - *Advantages:*
        - Provides a clear and structured way to capture functional requirements.
        - Easy to understand by both developers and stakeholders.
        - Useful for test case generation.
    - *Disadvantages:*
        - May not capture non-functional requirements well.
        - Can be time-consuming to develop.
        - Requires a good understanding of user roles and goals.
6. **Prototyping:**
    - *Description:* Creating a preliminary version of the system to demonstrate functionality and gather feedback.

- *Advantages:*
  - Helps stakeholders visualize the system and identify potential problems.
  - Facilitates early feedback and iteration.
  - Can uncover hidden requirements.
- *Disadvantages:*
  - Can be time-consuming and expensive.
  - Stakeholders may have unrealistic expectations.
  - Prototype may not be representative of the final system.

7. **Document Analysis:**
   - *Description:* Reviewing existing documents (e.g., business plans, reports, user manuals) to identify requirements.
   - *Advantages:*
     - Provides a good starting point for requirements elicitation.
     - Can uncover existing knowledge and best practices.
     - Relatively inexpensive.
   - *Disadvantages:*
     - Documents may be outdated or incomplete.
     - May not reflect current stakeholder needs.
     - Can be time-consuming to analyze large volumes of documents.
8. **Observation:**
   - *Description:* Observing users in their natural environment to understand their tasks and needs.
   - *Advantages:*
     - Provides insights into how users actually perform their work.
     - Can uncover hidden or unspoken needs.
   - *Disadvantages:*
     - Can be time-consuming and intrusive.
     - Users may behave differently when they are being observed.
     - Requires skilled observers.

The choice of elicitation technique depends on the specific project context, the type of stakeholders involved, and the available resources. Often, a combination of techniques is used to obtain a comprehensive understanding of the requirements.

B. Explain the process of requirements validation and the different methods used for it.

Requirements validation is the process of ensuring that the gathered and documented requirements accurately reflect the stakeholders' needs and expectations and that they are complete, consistent, and feasible. It aims to answer the question: "Are we building the right product?". The validation process typically involves the following steps:

1. **Reviewing Requirements Documents:**
   - The requirements documents (e.g., SRS, use case specifications) are reviewed by stakeholders, developers, and testers to identify any errors, ambiguities, or omissions.
2. **Developing Validation Criteria:**
   - Clear and measurable validation criteria are defined for each requirement. These criteria specify how the requirement will be tested and verified.
3. **Performing Validation Activities:**

- Various validation activities are performed to assess the requirements against the validation criteria.

4. **Documenting Validation Results:**
    - The results of the validation activities are documented, including any issues identified and the actions taken to resolve them.
5. **Obtaining Sign-Off:**
    - Once the requirements have been validated and all issues have been resolved, stakeholders sign off on the requirements documents to indicate their agreement.

Different methods are used for requirements validation:

1. **Requirements Reviews:**
    - *Description:* A systematic examination of the requirements documents by a team of reviewers.
    - *Focus:* Identifying errors, ambiguities, inconsistencies, and omissions.
    - *Techniques:* Checklists, walkthroughs, inspections.
    - *Advantages:* Effective for identifying a wide range of problems.
    - *Disadvantages:* Can be time-consuming and requires skilled reviewers.
2. **Prototyping:**
    - *Description:* Creating a working model of the system to demonstrate functionality and gather feedback.
    - *Focus:* Validating the usability and feasibility of the requirements.
    - *Advantages:* Helps stakeholders visualize the system and identify potential problems early on.
    - *Disadvantages:* Can be time-consuming and expensive.
3. **Test Case Generation:**
    - *Description:* Developing test cases based on the requirements.
    - *Focus:* Validating the testability and completeness of the requirements.
    - *Advantages:* Ensures that the requirements are clear and measurable.
    - *Disadvantages:* Can be time-consuming and requires a good understanding of testing techniques.
4. **Model Validation:**
    - *Description:* Creating models of the system (e.g., UML diagrams) to visualize the requirements and identify potential problems.
    - *Focus:* Validating the consistency and completeness of the requirements.
    - *Advantages:* Helps to identify logical errors and inconsistencies.
    - *Disadvantages:* Requires specialized skills in modeling techniques.
5. **Acceptance Testing:**
    - *Description:* Having stakeholders test the system to ensure that it meets their needs and expectations.
    - *Focus:* Validating the overall satisfaction with the system.
    - *Advantages:* Provides a final check on the requirements before the system is deployed.
    - *Disadvantages:* Can be expensive and time-consuming.
6. **Automated Validation:**
    - *Description:* Using tools to automatically check requirements for consistency, completeness, and correctness.
    - *Focus:* Improving efficiency and accuracy of the validation process.
    - *Advantages:* Reduces manual effort and human error.
    - *Disadvantages:* Requires investment in specialized tools and training.

By using a combination of these methods, organizations can ensure that their requirements are thoroughly validated and that the resulting system meets the needs of its stakeholders.

Q.4 Solve any TWO of the following:

A. What is a Context Model? Explain how it helps in understanding the system's environment.

A context model is a diagram that defines the boundary between the system and its environment, illustrating the entities (actors, systems, or external components) that interact with the system. It shows how the system fits into its surrounding environment and helps to clarify the scope of the system. It is a high-level view that omits detailed internal workings.

How it helps in understanding the system's environment:

1. **Defines System Boundaries:** A context model clearly identifies what is inside the system and what is outside. This helps to define the scope of the project and avoid scope creep.
2. **Identifies External Entities:** It shows all the external entities that interact with the system. This helps to understand the system's dependencies and interfaces.
3. **Illustrates Interactions:** It depicts the interactions between the system and its external entities, such as data flows, control signals, and events. This helps to understand how the system exchanges information with its environment.
4. **Clarifies System Purpose:** By showing the context in which the system operates, the context model helps to clarify the system's purpose and objectives.
5. **Facilitates Communication:** It provides a common understanding of the system's environment for all stakeholders, including developers, users, and managers.

Example:

Consider an online library system. The context model might show the system interacting with:

- **Librarians:** Manage books and user accounts.
- **Users:** Search for and borrow books.
- **Payment Gateway:** Process payments for fines or subscriptions.
- **External Book Database:** Access information about available books.

The context model would visually represent these entities and their interactions with the online library system, making it easier to understand the system's role and boundaries.

B. Describe the use of UML class diagrams for structural modeling. Provide an example.

UML class diagrams are used for structural modeling to represent the static structure of a system. They show the classes, their attributes, and the relationships between them. Class diagrams are essential for visualizing, specifying, constructing, and documenting the structural aspects of a software system.

Key elements of a class diagram:

1. **Classes:** Represented by rectangles with three compartments: class name, attributes, and operations (methods).
2. **Attributes:** Data fields that define the characteristics of a class. Each attribute has a name and a type.
3. **Operations:** Methods that define the behavior of a class. Each operation has a name, parameters, and a return type.

4. **Relationships:** Connections between classes that represent different types of associations:
    - **Association:** A general relationship between two classes.
    - **Aggregation:** A "has-a" relationship, where one class is part of another. (e.g., a Department has Students) Represented using a hollow diamond.
    - **Composition:** A stronger form of aggregation, where the part cannot exist independently of the whole. (e.g., a Car has an Engine) Represented using a filled diamond.
    - **Inheritance (Generalization):** An "is-a" relationship, where one class inherits the attributes and operations of another. (e.g., a Dog is an Animal). Represented using a hollow triangle.
    - **Dependency:** A weaker form of relationship where one class uses another. Represented using a dashed arrow.

Example:

Consider a simple online shopping system. A UML class diagram could represent the following classes and their relationships:

```
Class: Customer
Attributes:
    - customerID: int
    - name: string
    - address: string
    - email: string
Operations:
    - register(): void
    - login(): boolean
    - placeOrder(): Order

Class: Order
Attributes:
    - orderID: int
    - orderDate: Date
    - totalAmount: double
Operations:
    - addLineItem(item: LineItem): void
    - calculateTotal(): double

Class: LineItem
Attributes:
    - quantity: int
    - price: double
Operations:
    - calculateSubtotal(): double

Class: Product
Attributes:
    - productID: int
    - name: string
    - description: string
    - price: double
    - stockQuantity: int
```

```
Operations:
    - updateStock(quantity: int): void
```

Relationships:

- Customer places one or more Orders (Association).
- Order contains one or more LineItems (Composition).
- LineItem refers to a Product (Association).

This class diagram provides a clear and concise representation of the structural elements of the online shopping system and their relationships. It helps developers understand the system's architecture and facilitates communication among stakeholders.

C. Explain the purpose of behavioral modeling and give examples of UML diagrams used for it.

Behavioral modeling is the process of describing how a system responds to events, stimuli, or conditions over time. It focuses on the dynamic aspects of the system, illustrating how the system behaves in different scenarios. The purpose of behavioral modeling is to:

1. **Understand System Dynamics:** To gain a clear understanding of how the system operates and responds to external stimuli.
2. **Specify System Behavior:** To define the expected behavior of the system in different situations.
3. **Validate System Design:** To verify that the system design meets the specified behavioral requirements.
4. **Communicate System Behavior:** To provide a clear and concise representation of the system's behavior for all stakeholders.
5. **Generate Test Cases:** To create test cases that cover all possible scenarios and ensure that the system behaves as expected.

UML diagrams used for behavioral modeling:

1. **Use Case Diagrams:**
    - *Purpose:* Describe the interactions between actors (users or external systems) and the system to achieve specific goals.
    - *Elements:* Actors, use cases, and relationships (e.g., association, include, extend).
    - *Example:* In an ATM system, use cases might include "Withdraw Cash," "Deposit Funds," and "Check Balance."
2. **Sequence Diagrams:**
    - *Purpose:* Illustrate the interactions between objects in a system over time.
    - *Elements:* Objects, lifelines, messages, and activation boxes.
    - *Example:* A sequence diagram for "Withdraw Cash" might show the interaction between the user, the ATM, the bank account, and the transaction log.
3. **State Diagrams (State Machine Diagrams):**
    - *Purpose:* Describe the different states of an object and the transitions between those states in response to events.
    - *Elements:* States, transitions, events, and actions.
    - *Example:* A state diagram for a "Door" object might include states such as "Open," "Closed," and "Locked," with transitions triggered by events like "open," "close," and "lock."
4. **Activity Diagrams:**

- *Purpose:* Model the flow of activities in a system, including parallel and conditional behavior.
- *Elements:* Activities, actions, control flows, decision nodes, fork nodes, and join nodes.
- *Example:* An activity diagram for "Order Processing" might show the sequence of activities involved in placing an order, such as "Select Items," "Enter Shipping Information," "Process Payment," and "Ship Order."

5. **Communication Diagrams (Collaboration Diagrams):**
    - *Purpose:* Similar to sequence diagrams, but focus on the relationships between objects rather than the sequence of messages.
    - *Elements:* Objects, links, and messages.
    - *Example:* A communication diagram for processing a payment might show the objects involved (e.g., Customer, Order, PaymentGateway) and the messages exchanged between them.

By using these UML diagrams, developers can effectively model the behavior of a system and ensure that it meets the specified requirements.

Q.5 Solve any TWO of the following:

A. Explain the concept of software architecture and its importance in software development.

Software architecture refers to the fundamental structure of a software system. It defines the system's components, their relationships, and the principles and guidelines governing their design and evolution. It's a blueprint that guides the development process. It is not about the detailed implementation but rather the high-level organization.

Importance of software architecture:

1. **Provides a Blueprint:** Architecture serves as a roadmap for development, ensuring that all team members have a shared understanding of the system's structure and how the different parts fit together.
2. **Facilitates Communication:** A well-defined architecture provides a common language and vocabulary for stakeholders, facilitating communication and collaboration.
3. **Supports Quality Attributes:** Architecture plays a critical role in achieving important quality attributes such as performance, security, reliability, scalability, and maintainability. Different architectural styles and patterns are better suited for different quality attributes.
4. **Reduces Complexity:** By breaking down the system into smaller, manageable components, architecture helps to reduce complexity and make the system easier to understand and maintain.
5. **Enables Reuse:** A well-designed architecture can promote the reuse of components and patterns across multiple projects, reducing development time and cost.
6. **Manages Risk:** Architecture helps to identify and mitigate potential risks early in the development process. By considering different architectural options and evaluating their trade-offs, developers can make informed decisions that minimize risk.
7. **Supports Evolution:** A flexible and adaptable architecture allows the system to evolve over time to meet changing requirements and new technologies.
8. **Cost Optimization:** Making architectural decisions early on can help avoid costly rework later in the development cycle.

In summary, software architecture is essential for building successful software systems. It provides a foundation for development, supports quality attributes, reduces complexity, enables reuse, manages risk, and supports evolution. Without a well-defined architecture, software projects are more likely to fail.

B. Describe different architectural styles, such as layered, client-server, and microservices, with their advantages and disadvantages.

Architectural styles are reusable solutions to commonly occurring software architecture problems. They provide a high-level structure and organization for a software system. Here are descriptions of three common architectural styles:

1. **Layered Architecture:**
    - *Description:* The system is organized into a hierarchy of layers, where each layer provides services to the layer above it and uses services from the layer below it. Common layers include presentation, business logic, and data access.
    - *Advantages:*
        - *Simplicity:* Easy to understand and implement.
        - *Maintainability:* Changes in one layer are less likely to affect other layers.
        - *Reusability:* Layers can be reused in other systems.
        - *Testability:* Each layer can be tested independently.
    - *Disadvantages:*
        - *Performance Overhead:* Each request must pass through multiple layers, which can impact performance.
        - *Tight Coupling:* Layers can become tightly coupled if not designed carefully.
        - "*Layer Skipping*": Sometimes layers are skipped, violating the architectural principles and leading to a monolithic design.
2. **Client-Server Architecture:**
    - *Description:* The system is divided into two parts: a client that requests services and a server that provides services. The client and server communicate over a network.
    - *Advantages:*
        - *Centralized Control:* The server can manage resources and enforce security policies.
        - *Scalability:* The server can be scaled to handle more clients.
        - *Accessibility:* Clients can access the server from anywhere on the network.
        - *Maintainability:* Server-side changes can be deployed without affecting clients.
    - *Disadvantages:*
        - *Single Point of Failure:* If the server fails, the entire system is unavailable.
        - *Network Dependency:* Performance depends on the network connection between the client and the server.
        - *Security Risks:* The server is a potential target for attacks.
        - *Bottlenecks:* The server can become a bottleneck if it is not properly designed.
3. **Microservices Architecture:**
    - *Description:* The system is composed of small, independent services that communicate over a network. Each service is responsible for a specific business function.
    - *Advantages:*
        - *Scalability:* Services can be scaled independently.
        - *Flexibility:* Services can be developed and deployed independently.
        - *Resilience:* If one service fails, the other services can continue to operate.
        - *Technology Diversity:* Different services can be implemented using different technologies.
    - *Disadvantages:*
        - *Complexity:* More complex to design and manage than monolithic architectures.

    - *Distributed Systems Challenges:* Requires careful handling of distributed transactions, data consistency, and network latency.
    - *Monitoring and Logging:* More difficult to monitor and log events across multiple services.
    - *Operational Overhead:* Requires more infrastructure and operational support.

The choice of architectural style depends on the specific requirements of the system, including the desired quality attributes, the complexity of the system, and the available resources.

C. What are architectural patterns? Explain the importance of any three architectural patterns.

Architectural patterns are reusable solutions to commonly occurring software architecture problems. They provide a proven template for structuring a system and solving recurring design challenges. They are more abstract than design patterns and address system-wide concerns.

Importance of architectural patterns:

1. **Proven Solutions:** Architectural patterns represent best practices that have been successfully used in many projects. By using these patterns, developers can avoid reinventing the wheel and reduce the risk of making costly mistakes.
2. **Improved Quality:** Architectural patterns help to achieve important quality attributes such as performance, security, reliability, and scalability.
3. **Reduced Complexity:** Architectural patterns provide a clear and consistent structure for the system, making it easier to understand and maintain.
4. **Faster Development:** By using architectural patterns, developers can speed up the development process and reduce the time to market.
5. **Better Communication:** Architectural patterns provide a common language and vocabulary for stakeholders, facilitating communication and collaboration.

Three important architectural patterns:

1. **Model-View-Controller (MVC):**
    - *Purpose:* Separates the application into three interconnected parts:
        - *Model:* Manages the data and business logic.
        - *View:* Displays the data to the user.
        - *Controller:* Handles user input and updates the model and view.
    - *Importance:* Improves maintainability, testability, and reusability. Allows for multiple views of the same data.
    - *Applicability:* Web applications, desktop applications, mobile applications.
2. **Microkernel (Plug-in):**
    - *Purpose:* Provides a minimal core system (microkernel) that can be extended with plug-ins.
    - *Importance:* Allows for flexible and extensible systems. New features can be added without modifying the core system.
    - *Applicability:* Operating systems, text editors, web browsers.
3. **Pipe and Filter:**
    - *Purpose:* Processes data through a series of filters connected by pipes. Each filter performs a specific transformation on the data.
    - *Importance:* Supports modularity, reusability, and parallel processing. Easy to add, remove, or modify filters.
    - *Applicability:* Data processing systems, image processing systems, compilers.

By using these and other architectural patterns, developers can build high-quality, maintainable, and scalable software systems.

Q.6 Solve any TWO of the following:

A. Explain the concept of design patterns and their role in software development.

Design patterns are reusable solutions to commonly occurring problems in software design. They are templates that can be adapted to solve specific design challenges in different contexts. They are not complete designs that can be directly implemented but rather descriptions or templates for how to solve a problem that can be used in many different situations.

Role of design patterns in software development:

1. **Provide Proven Solutions:** Design patterns represent best practices that have been successfully used in many projects. By using these patterns, developers can avoid reinventing the wheel and reduce the risk of making costly mistakes.
2. **Improve Code Quality:** Design patterns help to improve the quality of code by promoting modularity, reusability, and maintainability.
3. **Reduce Complexity:** Design patterns provide a clear and consistent structure for the system, making it easier to understand and maintain.
4. **Speed Up Development:** By using design patterns, developers can speed up the development process and reduce the time to market.
5. **Enhance Communication:** Design patterns provide a common language and vocabulary for developers, facilitating communication and collaboration.
6. **Promote Reusability:** Design patterns encourage the reuse of code and design knowledge across multiple projects.

In summary, design patterns are essential tools for software developers. They provide proven solutions to common design problems, improve code quality, reduce complexity, speed up development, enhance communication, and promote reusability.

B. Describe the different categories of design patterns (creational, structural, and behavioral) and provide an example of each.

Design patterns are typically categorized into three main types: creational, structural, and behavioral.

1. **Creational Patterns:**
    - *Purpose:* Deal with object creation mechanisms, trying to create objects in a manner suitable to the situation. They abstract the instantiation process, making the system more independent of how its objects are created, composed, and represented.
    - *Example:* Singleton, Factory Method, Abstract Factory, Builder, Prototype.
    - *Example Explanation:*
        - **Singleton:** Ensures that a class has only one instance and provides a global point of access to it.
2. **Structural Patterns:**
    - *Purpose:* Deal with the composition of classes or objects. They provide different ways to create a class structure, for example, using inheritance and composition, to create large object structures.
    - *Example:* Adapter, Bridge, Composite, Decorator, Facade, Flyweight, Proxy.

- *Example Explanation:*
  - **Adapter:** Allows classes with incompatible interfaces to work together by wrapping an interface around an existing class.

3. **Behavioral Patterns:**
   - *Purpose:* Deal with algorithms and the assignment of responsibilities between objects. They describe not just patterns of objects or classes but also the patterns of communication between them.
   - *Example:* Chain of Responsibility, Command, Interpreter, Iterator, Mediator, Memento, Observer, State, Strategy, Template Method, Visitor.
   - *Example Explanation:*
     - **Observer:** Defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

C. Choose any three specific design patterns, explain their purpose, and provide a scenario where they would be applicable.

1. **Factory Method (Creational):**
   - *Purpose:* Defines an interface for creating an object, but lets subclasses decide which class to instantiate. It defers instantiation to subclasses.
   - *Scenario:* A software company needs to create different types of products (e.g., cars, trucks, motorcycles) based on user input. Instead of hardcoding the object creation logic, a Factory Method can be used to allow subclasses (e.g., CarFactory, TruckFactory) to create the specific product types.
2. **Decorator (Structural):**
   - *Purpose:* Attaches additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.
   - *Scenario:* A coffee shop application needs to add different condiments (e.g., milk, sugar, chocolate) to a coffee object. Instead of creating a separate class for each combination of condiments, a Decorator pattern can be used to dynamically add the condiments to the coffee object.
3. **Observer (Behavioral):**
   - *Purpose:* Defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
   - *Scenario:* A stock market application needs to update multiple views (e.g., charts, tables, alerts) whenever the price of a stock changes. The stock object can act as the subject, and the views can act as observers. Whenever the stock price changes, the subject notifies all the observers, which then update their displays accordingly.