**Practical no. 9**

**Abhishek Gaur**

1. **Requirement Analysis**: Understand the functional and non-functional requirements of the software. This helps in defining the scope of testing.
2. **Identify Critical Features**: List down the features or functionalities of the software that are most critical to its intended purpose and business objectives.
3. **Boundary Value Analysis**: For each input field, identify the boundaries. Create test cases for:
   * Minimum value
   * Just above the minimum value
   * Maximum value
   * Just below the maximum value
   * Typical values
4. **Equivalence Partitioning**: Divide the input domain into equivalence classes and create test cases for each class. For instance, if an input field accepts values between 1 to 100, then test cases can be:
   * Any value from 1 to 100 (valid values)
   * Less than 1 (invalid values)
   * Greater than 100 (invalid values)
5. **Negative Testing**: Focus on invalid inputs. Test the software's behavior when it receives unexpected or incorrect data.
6. **Functional Testing**: Create test cases based on the functional requirements. Ensure that each requirement is tested individually and in combination with others.
7. **Integration Testing**: Test the interactions between different modules or components of the software. This includes checking data flow, interface compatibility, and system interactions.
8. **Performance Testing**: Validate the software's performance under load, stress, and normal conditions. Ensure that it meets the defined performance criteria.
9. **Security Testing**: Identify potential security vulnerabilities and create test cases to validate the software's security features. This includes testing for authentication, authorization, data encryption, etc.
10. **Usability Testing**: Evaluate the software's user interface, user experience, and overall usability. Ensure that it is intuitive, user-friendly, and meets the user's needs.
11. **Regression Testing**: After making changes or updates to the software, retest the existing functionalities to ensure that no new defects have been introduced.
12. **Error Handling and Recovery Testing**: Test the software's ability to handle errors gracefully and recover from failures. Create test cases for scenarios where unexpected errors occur.
13. **Compatibility Testing**: Validate the software's compatibility with different operating systems, devices, browsers, and other software components.
14. **Data Validations**: Test the software's data validation mechanisms. Ensure that it handles data correctly, prevents data corruption, and maintains data integrity.
15. **Localization and Internationalization Testing**: If the software is intended for a global audience, test its compatibility with different languages, cultures, and regional settings.
16. **Documentation Testing**: Validate the software's documentation, user manuals, help guides, and other instructional materials. Ensure that they are accurate, up-to-date, and provide clear guidance to users.

**Practical no.10**

**Abhishek Kumar Singh**

**Classes:**

1. **Identifying Classes**:
   * Review the requirements and functionalities described in the Case Study.
   * Identify nouns, verbs, and other key terms that represent entities, attributes, and actions within the system.
   * Consider the relationships and interactions between different components or entities.
2. **Classifying Classes**:
   * **Strong Classes**: These are the main entities or components that have a distinct identity and lifecycle within the system. They typically represent core functionalities or key entities.
   * **Weak Classes**: These are auxiliary or supporting classes that do not have a unique identity on their own and are dependent on strong classes for their existence. They often represent attributes or components that are part of a larger entity.
3. **Drawing the Class Diagram**:
   * Use standard UML notation to represent classes, attributes, methods, and relationships.
   * Identify associations, aggregations, compositions, and other relationships between classes.
   * Clearly define multiplicity, roles, and other constraints to specify the nature and cardinality of relationships.

**Strong Classes:**

1. **Distinct Identity**: Strong classes have a distinct identity and represent core entities or components within the system.
2. **High Cohesion**: Strong classes encapsulate related attributes and behaviors that are closely related and form a cohesive unit.
3. **Low Coupling**: Strong classes are loosely coupled with other classes, meaning they have minimal dependencies on external classes and changes to one class have minimal impact on others.
4. **Independent Lifecycle**: Strong classes have an independent lifecycle and can exist independently within the system.

**Weak Classes:**

1. **Dependent Identity**: Weak classes do not have a unique identity on their own and are dependent on strong classes for their existence.
2. **Low Cohesion**: Weak classes often represent auxiliary or supporting attributes or behaviors that are not closely related and do not form a cohesive unit on their own.
3. **High Coupling**: Weak classes are often tightly coupled with strong classes, meaning they have a high degree of dependency on external classes and changes to one class can have a significant impact on others.
4. **Dependent Lifecycle**: Weak classes have a dependent lifecycle and cannot exist independently within the system without the associated strong classes.

