LAB MANUAL

School of Computing
SASTRA Deemed to be University
Thirumalaisamudram, Thanjavur

Course Code: **CSE315R02**
Course Name: **Software Engineering Practices**

Department: **CSE/IT/ICT**
Semester: **V/VI Semesters**

Prepared by

Dr. A. Umamakeswari (Professor)
Dr. A. Joy Christy (AP III)
Dr. B. Lakshmi (AP II)

LIST OF EXPERIMENTS

1. Prepare Software Requirement Specification (SRS) Report
2. Design Use case and Activity Diagrams using UMLet
3. Draw Class Diagram using UMLet
4. Draw Sequence and State Diagrams using UMLet
5. Perform Manual Testing using TestRail
6. Testing individual methods or functions within classes using Unit Testing 7.
Testing interactions between different classes using Integration Testing 8.
Exercise Regression Testing by incorporating changes in the previously tested

modules

9. Conduct code coverage testing using JaCoCo/Cobertura
10. Testing single method multiple times with different set of parameters using Parameterized Testing
11. Conduct concurrency testing to run test cases in parallel or simulating concurrent scenarios using Junit Pioneer
12. Perform web site testing using selenium IDE

Exp:1 Prepare Software Requirement Specification (SRS) Report – Tool: Text Editor (Roger Pressman Template)

Objectives:

- Understanding the IEEE standard of Software Requirements Specification
- Preparing Requirement Management Plan
- Identifying features
- Identifying Use Cases
- Identifying Stakeholders requirements
- Identifying functional requirements
- Identifying non-functional requirements

Pre-Requisite: IEEE SRS Template

Tool: Any open-source text editor

Requirements Management:

Requirements management is a systematic approach to finding, organizing, documenting and tracking the changing requirements of a software application or system. Requirements are capabilities and objectives to which the software or system being built must conform. Requirements are found in vision documents, problem reports, feature requests, business practices, designs, Quality Assurance plans, Test cases and Prototypes.

One way to help ensure the success of a project is to implement an effective requirements management process. Requirements management offers numerous benefits.

These include improved

- Predictability of a project's schedule and deliverables
- Cost effective project costs
- Software quality
- Team communication
- Improved compliance with standards and regulations (Capability Maturity Model (CMM), ISO 9000etc.,).

Implementing an effective requirements management process

The five essential steps used in managing requirements and achieving the project goals are 1. Identify requirements.

2. Organize and prioritize the requirements.
3. Analyze and understand the factors associated with the requirements.
4. Refine and expand requirements.
5. Manage changes to requirements.

1. Identify requirements:

It is very important to identify and manage requirements from the beginning of a project. Here are some factors to consider when setting priorities at the beginning of the requirements management process.

• How does the requirement add to product functionality, usability, reliability and performance? • Will the requirement being considered be worth the effort and schedule constraints? • Is the requirement feasible given the risks associated with it?

• If the requirement is implemented, how will it impact the ability to maintain the product?

2. Organize your requirements.

Look at the schedule to check whether there will be enough time to complete all high priorities? Setting expectations at this stage will help team stay on schedule and close to budget. Ask yourself these questions:

- What are the needs of the stakeholders?
- What are primary goals of this project?
- What key features are parts of the requirements list?
- What kinds of documents are needed to record and trace requirements from wish lists to test scripts?
- Determine which tasks are necessary to accomplish requirements, who is responsible for completing them and which requirements are parts of the project's critical path.

3. Understand and analyze the factors associated with the requirements. If the project has a long list of requirements, classify them in categories that make sense. Asking the following questions will help to analyze these factors more efficiently.

- How will each requirement benefit the customers?
- How much effort is necessary to accomplish the requirement?
- How will each requirement affect the budget and schedule?
- What are the risks associated with each requirement?

4. Refine and expand the requirements.

Refining requirements early on in the process will help prevent requirement errors later. The time, money and effort required to fix errors at the end of the process could cost one hundred times more than the time spent planning now.

5. Manage changes to the requirements.

Track the progress of requirements and trace changes to them. Monitor changes, analyze the impact of change and communicate this to the team.

Preparing SRS :

It needs identification of stakeholders, use cases and features. Use cases and features are both types of but they have different purposes and levels of detail.

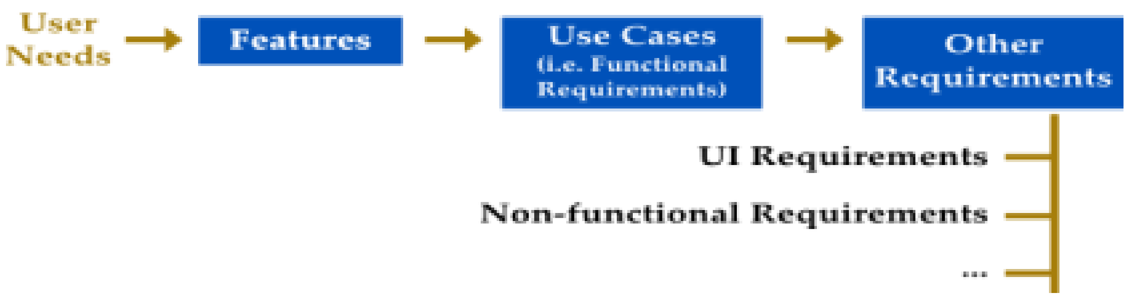
Features:

Features are high-level descriptions of the system’s capabilities or characteristics that are derived from the stakeholders’ needs. Features can be further refined into functional and non-functional requirements, which specify the behaviour and quality attributes of the system. A feature can be derived from one or more user needs, and a feature can be decomposed into one or more use cases.

In any project, the developer starts by understanding the user needs. Then the needs are translated into features. Then the use cases are used to describe functional requirements. Then other types of requirements such as non-functional requirements, UI requirements, etc. are defined. **Use cases:**

A use case tells how a user (actor) interacts with the system under a specific set of circumstances. Actors are the different people (or devices) that use the system within the context of the function and behaviour that is to be described. Actors represent the roles that people (or devices) play as the system operates.

A use case may be narrative text, an outline of tasks or interactions, a template-based description, or a diagrammatic representation. A use case depicts the software or system from the user’s point of view.


User Needs
Features
Use Cases (i.e. Functional Requirements)
Other Requirements
UI Requirements
Non-functional Requirements
...

Example:
In the wake of rapid urbanization, cities are facing multifaceted challenges ranging from traffic congestion, pollution, inadequate infrastructure, to inefficient resource management. To address these issues and propel towards sustainable development, there's an imperative need for the implementation of Smart City Systems. These systems integrate various technologies to optimize city operations, enhance quality of life, and foster economic growth. This system includes features like smart healthcare, governance, transportation, security surveillance, infrastructure, job opportunities, and amenities for comfortable living. This system requires iterative development, continuous improvement, and stakeholder collaboration throughout the development process. i) Prepare a detailed software requirements specification document for the given scenario. Smart City Systems Software Requirements Specification (SRS)

Table of Contents

Appendix C: Issues List

1. Introduction

1.1 Purpose

The purpose of this Software Requirements Specification (SRS) document is to outline the functional and non-functional requirements for the development of a Smart City System. This system aims to address urban challenges by integrating various technologies to optimize city operations, enhance the quality of life, and foster economic growth.

The system requires iterative development, continuous improvement, and stakeholder collaboration throughout its lifecycle.

1.2 Document Conventions

This document follows the conventions of the IEEE SRS template, including section headings, numbering, and format standards.

1.3 Intended Audience and Reading Suggestions

This document is intended for stakeholders involved in the development and implementation of the Smart City System, including city planners, government officials, software developers, and system architects.

1.4 Project Scope

The Smart City System can be implemented in any city to optimize city operations such as smart healthcare, governance, transportation, security surveillance, infrastructure, job opportunities, and amenities for comfortable living.

1.5 References

IEEE Std 830-1998, IEEE Recommended Practice for Software Requirements Specifications Smart City Reference Model- https://smartcities.gov.in/

2. Overall Description

2.1 Product Perspective

The Smart City System is an integrated solution that leverages IoT, AI, big data, and cloud computing to address urban challenges. It is designed to interact with various subsystems and existing city infrastructure to provide a cohesive and efficient urban management platform. 2.2 Product Features

Smart Healthcare: Remote health monitoring, telemedicine, and emergency response. Smart Governance: Digital services for citizens, e-governance portals, and public participation platforms.

Smart Transportation: Intelligent traffic management, public transport optimization, and real time information systems.

Smart Security: Surveillance systems, emergency alert systems, and public safety management. Smart Infrastructure: Energy-efficient buildings, smart grids, and water management systems. Job Opportunities: Platforms for job matching, skill development programs, and employment services.

Amenities for Comfortable Living: Smart housing, recreational facilities, and community services.

2.3 User Classes and Characteristics

Citizens: Primary beneficiaries who interact with various services provided by the system.

Government Officials: Administrators who manage and oversee the system's operations.

Service Providers: Entities providing healthcare, transportation, and other services integrated into the system.

Developers and Technicians: Personnel involved in the development, maintenance, and support of the system.

2.4 Operating Environment

The system will operate in urban environments, requiring robust infrastructure to support IoT devices, high-speed internet connectivity, and reliable power supply.

2.5 Design and Implementation Constraints

Integration with existing city infrastructure and services.
Compliance with local regulations and standards.

2.6 User Documentation

Comprehensive user manuals, online help systems, and training programs will be provided to ensure all users can effectively interact with the Smart City System.

2.7 Assumptions and Dependencies

Availability of necessary funding and resources.
Cooperation from various stakeholders, including government bodies, private sectors, and citizens.
Technological advancements and infrastructure improvements.

3. System Features

3.1 System Feature 1 : Smart Healthcare

Remote health monitoring and data analysis.
Integration with hospitals and emergency services.
User-friendly interfaces for patients and healthcare providers.

3.2 System Feature 2: Smart Governance

E-governance portals for digital services.
Platforms for citizen engagement and feedback.
Real-time data analytics for informed decision-making.

3.3 System Feature 3: Smart Transportation

Real-time traffic monitoring and management.
Public transport optimization with dynamic scheduling.
User applications for travel planning and updates.

3.4 System Feature 4: Smart Security

Surveillance systems with AI-based threat detection.
Emergency alert systems and response coordination.
Public safety management tools.

4. External Interface Requirements

4.1 User Interfaces

Web-based portals for citizens and administrators.
Mobile applications for on-the-go access.
Interactive kiosks in public areas.

4.2 Hardware Interfaces

Sensors and IoT devices for data collection.
Servers and data centers for processing and storage.
Communication networks for data transmission.

4.3 Software Interfaces

APIs for integration with third-party services.
Middleware for data aggregation and processing.
Databases for storing and retrieving information.

4.4 Communications Interfaces

Secure communication protocols for data transmission.
Integration with existing city communication infrastructure
Support for various communication standards.

5. Other Non-functional Requirements

5.1 : Performance Requirements

The system should be able to handle up to 5,000 concurrent users requests during peak times without performance degradation.
Response time for any action within the system should not exceed 2 seconds under normal load conditions.

5.2 : Availability Requirements

Availability: Both catalogue and databases are available to students 24/7.

5.3 Security Requirements

Send OTP - OTP is used to authorize users and to maintain sensitive data.

5.4 Reliability Requirements

Consistent performance and minimal downtime.

5.5 Usability Requirements:

Intuitive and user-friendly interfaces

5.6. Scalability Requirements:

The system should be Scalable to accommodate the growing population and expanding urban areas.

6. Other Requirements

Legal and regulatory compliance.

Environmental impact considerations.

Community engagement and feedback mechanisms.

Appendix A: Glossary

IoT: Internet of Things

AI: Artificial Intelligence

ML- Machine Learning

DP-Deep Learning

API: Application Programming Interface

Appendix B: Analysis Models

Use case diagrams

Appendix C: Issues List

Identified issues and challenges

Proposed solutions and mitigations

Result:

The SRS for the proposed system has been prepared by identifying features, use cases, functional and non functional requirements based on the SRS template.

Exp 2: Design Use case and Activity Diagrams using UMLet – Tool: UMLet (Open Source)

Objectives:

• Understanding and deriving the use cases for the given scenario

• Understanding objects and its actions of the software system

• Understanding the symbols of UML which are necessary to draw Use case and activity diagrams.

• Identifying actors, actions, prerequisite, post-requisite and constraints of each scenario. • Identifying and elaborate the flow of activities to be carried out for each use case • Elaboration of use cases (activity diagram)

• Preparing Use case and activity diagram using UMLet.

Pre-Requisite:

Knowledge of UML notations

Tool: UMLet

UMLet:

UMLet is a lightweight, open-source UML (Unified Modeling Language) diagramming tool designed for fast and intuitive creation of various UML diagrams. It provides a simple interface and a palette of diagramming elements that allow users to quickly sketch out software designs, system architectures, and business processes.

Key Features of UMLet:

- **User-Friendly Interface**: UMLet offers a straightforward drag-and-drop interface where users can easily place and connect diagram elements. To zoom the diagram +/- or Ctrl+mousewheel can be used.

- **Rich Palette of Symbols**: It includes a comprehensive palette of UML symbols and elements, such as actors, use cases, classes, sequences, and more, tailored for various types of UML diagrams.
- **Customization Options**: Users can customize diagram elements, add text annotations, and adjust properties like colors and fonts to enhance clarity and presentation.
- **Export Capabilities**: It supports exporting diagrams to various formats, including PDF, PNG, SVG, and LaTeX, facilitating sharing and integration into documentation and presentations. • **Platform Compatibility**: UMLet is platform-independent and runs on Java, making it accessible on Windows, macOS, and Linux systems.
- **Integration with IDEs**: It can be integrated with IDEs (Integrated Development Environments) like Eclipse through plugins, allowing seamless diagram creation within development workflows.

Procedure:

1. Open UMLet:
Launch the UMLet application on your computer.
2. Create a New Diagram:
Go to `File` in the menu bar, select `New`.
3. Choose Use Case Diagram:
From the options presented, select `Use Case Diagram`.
4. Locate the Palette:
The palette in UMLet contains all the tools (icons) needed to create use
case diagram. It is usually located on the right-hand side of the window.
The symbols can be added by double clicking it.
5. Add Actors:
Locate the "Actor" icon in the palette .Click on the "Actor" icon in the palette. Click anywhere on the diagram canvas to place the actor symbol. Edit the text using the properties window in the lower-right text panel .
6. Add Use Cases:
To add a use case to the diagram, Locate the "Use Case" icon in the palette . Click on the "Use Case" icon in the palette. Click anywhere on the diagram canvas to place the use case symbol. 7. Connect Actors and Use Cases:
To connect an actor to a use case (representing their interaction) the appropriate connection tool in the palette (often represented as a line or arrow) is selected.
Click on the connection tool in the palette , click on the actor symbol, then drag the mouse to the use case symbol.
Release the mouse button to create the connection.
8. Add Relationships:
Use the palette tools to add relationships between use cases, such as include, extend, or generalization relationships.
9. Arrange and Customize:
Arrange the actors, use cases, and relationships on the diagram canvas to create a clear representation of system's functionality.The user can change the forground and background colors of actors and use cases by right clicking the mouse from the actor or use case.
10. Save Your Diagram:
It is possible to save the use case diagram by going to `File` > `Save As` and choosing a location and file format (UMLet saves diagrams in `.uxf` format by default).
11.Export or Share:
The diagram can be exported to various formats such as PDF, PNG, or SVG for sharing or further use.

i)Use Case Diagram

As requirements are gathered, an overall vision of system functions and features begin to materialize. However, it is difficult to move into more technical software engineering activities until how these functions and features will be used by different classes of end users is known.

To accomplish this, developers and users can create a set of scenarios that identify a thread of usage for the system to be constructed. The scenarios, often called *use cases* provide a description of how the system will be used.

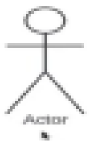
It is a visual representation of a system. It models the complete representation of the system. It depicts the view of different people about the system from their perspective. The use case diagram has 2 major elements

1. Actor
2. Use case

Actors represent the roles that people (or devices) play as the system operates. An actor is anything that communicates with the system or product and that is external to the system itself.The use cases are functional requirements of the system from different users' view.

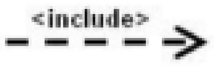
A use case diagram is a type of UML diagram that shows the interaction between actors and use cases in a system. A use case is a description of a specific functionality or scenario that the system can perform. The use cases tell the interaction of the system with outside world.

The following symbols are used to draw a use case in RationalRose

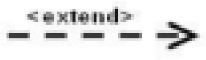
1.Actor 

- Someone that uses the system to achieve the goal. It may be a person, organization, another system or an external device.

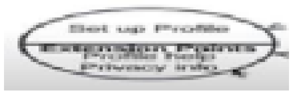
2.Use case - A scenario that tells the interaction of the user with the system.

3.include relationship(dotted lines) 

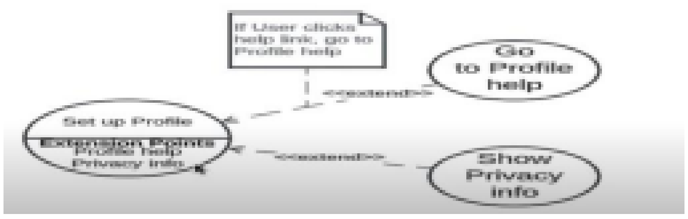
- The use case is mandatory and part of the base use case. It is represented by a dashed arrow in the direction of the included use case with the notation <<include>>. For example, a use case "Buy Book on WebSite" may include another use case "Make Payment" as a necessary step. The dotted dependency line is used.

4. extend relationship(dotted lines) 

-The use case is optional and comes after the base use case. It is represented by a dashed arrow in the direction of the base use case with the notation
<<extend>>. For example, a use case "Deposit Funds" may extend another use case "Calculate Bonus" if certain conditions are met.The dotted dependency line is used.

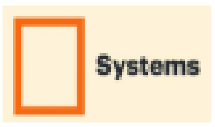
5. Extension points 

- The point at which an extending use case is added can be defined by means of an extension point. The dotted dependency line is used.



6. Unidirectional arrows - shows the association between actor and use case. 7. Generalization - Uses a triangle arrow head to represent a parent child relationship

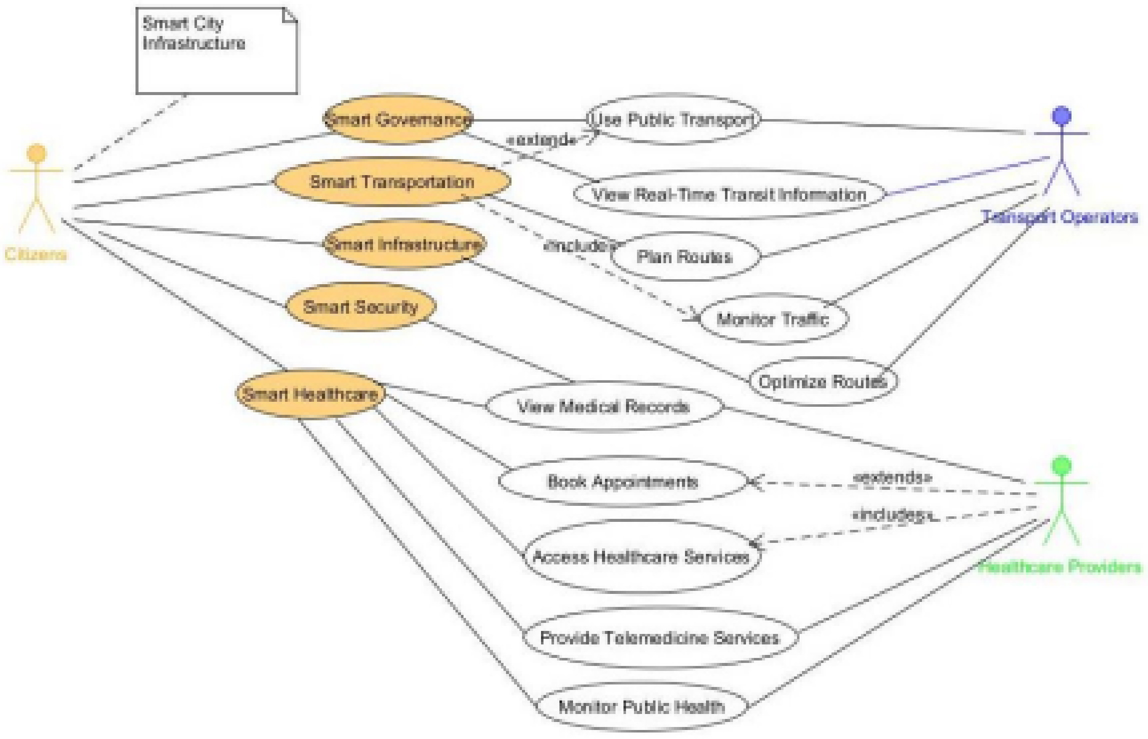
among the actors.

8. System 

- Uses a rectangle to refer to website, software component, business process or app

Example:

Use case diagram for implementation of Smart City Systems



Activity Diagram.

A UML activity diagram depicts the dynamic behaviour of a system or part of a system through the flow of control between actions that the system performs. It is similar to a flowchart except that an activity diagram can show concurrent flows.

An activity diagram can be used to represent features at a low level of abstraction, by showing the steps or tasks involved in each feature and their order of execution. An activity diagram can also show the conditions, loops and synchronization of the actions.

Activity diagrams are useful for capturing the dynamic aspects of a system, such as concurrency, synchronization, branching, and looping.

Use cases and activity diagrams are both used to model the behaviour of a system or a process. They are related in the following ways:

- Use cases describe the goals and tasks of the actors or users of the system, while activity diagrams show the flow of actions and events that lead to those goals.
- Activity Diagrams are used to elaborate each use case.

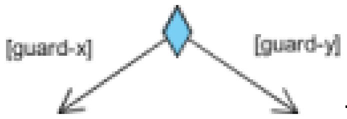
Swimlane Diagram:

Swimlanes are formed by dividing the diagram into strips or "lanes," each of which corresponds to one of the participants. All actions in one lane are done by the corresponding participant. A swimlane is a way to group activities performed by the same actor on an activity diagram or to group activities in a single thread.

It is also suitable for modeling how a collection of use cases coordinate to represent business workflows. It is used to

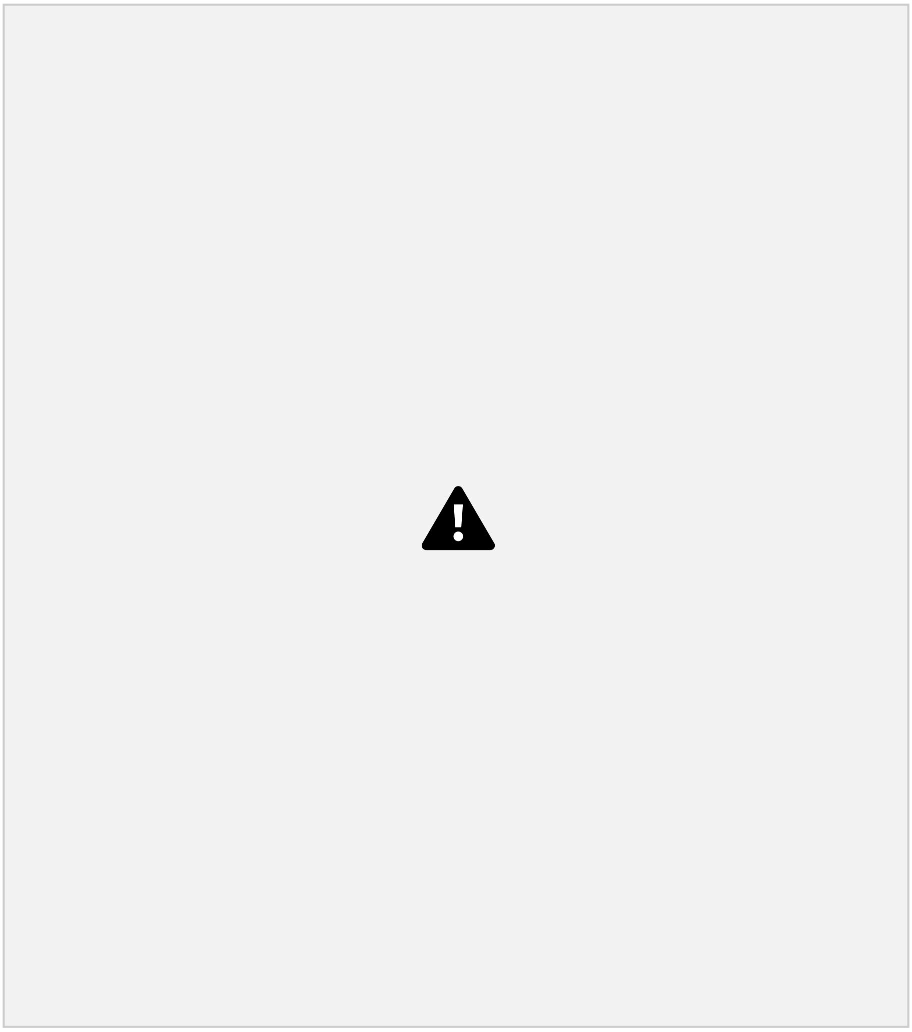
- Identify candidate use cases, through the examination of business workflows •

Identify pre- and post-conditions (the context) for use cases
• Model workflows between/within use cases
• Model complex workflows in operations on objects
• Model in detail complex activities in a high level activity Diagram
The following symbols are used to draw an activity diagram.

1. Initial node ->● .A solid black dot forms the *initial node* that indicates the starting point of the activity.
2. End node -> ◉. A black dot surrounded by a black circle is the *final node* indicating the end of the activity.
3. Decision Node-> 

-Represents a test condition to ensure that the control flow or object flow only goes down one path
4. Merge Node -> -Bring back together different decision paths that were created using a decision-node.
5. Fork Node/Sync Node **->** **-** A *fork* represents the separation of activities into two or more concurrent activities. It is drawn as a horizontal black bar with one arrow pointing to it and two or more arrows pointing out from it.
6. **Join Node** -> -Bring back together a set of parallel or concurrent flows of activities (or actions).
7. Activity > - used to represent a set of actions
8. Action Node**->** **-**A task to be performed(state)
9. Arrows-> -Arrows from one action node to another indicate the flow of control.
10. Swimlane- - to create partitions in activity diagrams

Activity Diagram for Implementation of Smart City Systems



Result: Once the stakeholders interactions with the system **have been** identified, **use case, activity and swim lane diagrams are drawn using UMLet.**

Exp 3: Draw Class Diagram using UMLet– Tool (Open Source)

Objectives:

- Understanding and deriving the objects for the given scenario
- Understanding properties and behaviour of objects
- Understanding the symbols of UML which are necessary to draw the class diagram. • Identifying the relation between use case and class diagram of converting actors in to objects and use cases to responsibilites or behaviour.
- Identifying the relation between activity and class diagram of converting activities to behaviour.

Pre-Requisite:

Knowledge of UML notations

Tool: UMLet

Class Diagram

Class diagram is a static diagram. It represents the static view of an application. Class diagram is not only used for visualizing, describing, and documenting different aspects of a system but also for constructing executable code of the software application.

Class diagram describes the attributes and operations of a class and also the constraints imposed on the system. The class diagrams are widely used in the modeling of object oriented systems because they are the only UML diagrams, which can be mapped directly with object-oriented languages.

Class diagram shows a collection of classes, interfaces, associations, collaborations, and constraints. It is also known as a structural diagram.

The class diagrams can be used

- In analysis and design of the static view of an application.

- To describe responsibilities of a system.
- As a base for component and deployment diagrams.
- To perform forward and reverse engineering.

The construction of the class diagram involves the below steps

1. Identification of different classes
2. Identification of attributes(members) with visibility labels
3. Identification of behaviors(member functions/methods) with visibility labels
4. Visibility of Class attributes and Operations
   + denotes public attributes or operations
   - denotes private attributes or operations
   # denotes protected attributes or operations
   ~ denotes package attributes or operations
5. Establishing relationships between classes- describe how classes are connected or interact with each other within a system. There are several types of relationships, each serving a specific purpose. Here are some common types of relationships in class diagrams:

i) **Inheritance (or Generalization):** A solid line with a hollow arrowhead that point from the child to the parent class. It depicts is-a relationship.

ii) Simple Association - A solid line connecting two classes. An **association** almost always implies that one object has the other object as a field/property/attribute .

iii) **Aggregation**: A special type of association. It represents a "part of" relationship(optional). A solid line with an unfilled diamond at the association end connected to the class of composite.

iv) **Composition**: A special type of aggregation(compulsory relation) where parts are destroyed when the whole is destroyed. A solid line with a filled diamond at the association connected to the class of composite.

v) **Dependency**: Exists between two classes if the changes to the definition of one may cause changes to the other (but not the other way around).A dashed line with an open arrow. For example, a class that uses another class as a parameter in one of its methods has a dependency on that class. To show dependency in a diagram, draw a dashed line with an open arrowhead pointing from the dependent class to the independent class.

vi) **Realization(**Interface Implementation)-Realization indicates that a class implements the features of an interface.

vii) Multiplicity - It tells how many objects of each class take part in the relationships and multiplicity can be expressed as:

- Exactly one - 1
- Zero or one - 0..1
- Many - 0..* or *
- One or more - 1..*

viii) Relationship Names - Names of relationships are written in the middle of the association line. It uses a small arrowhead to show the direction .

ix) Relationship – Roles- Roles are written at the end of an association line and describe the purpose played by that class in the relationship.

Class Diagram for Implementation of Smart City Systems

Result: Once the objects and their responsibilities **have been** identified, class diagram is **drawn using the tool UMLet .**

Exp 4: Draw Sequence and State Diagrams using UMLet– Tool (Open Source)

Objectives:

• Indicating how events cause transitions from object to object in the case of sequence diagram. • Identifying the events and their transition from one state to another.

Pre-Requisite:

Knowledge of UML notations

Tool: UMLet

Sequence Diagram.

This is a behavioral representation that indicates how events cause transitions from object to object. Once events have been identified by examining a use case, the modeler creates a sequence diagram—a representation of how events cause flow from one object to another as a function of time. The sequence diagram is a shorthand version of the use case. It represents key classes and the events that cause behavior to flow from class to class.

Sequence diagrams are useful for understanding the dynamic behavior of a system, such as how it responds to user inputs, how it communicates with other systems, or how it handles different scenarios. Sequence Diagram captures:

• The interaction that takes place in a collaboration that either realizes a use case or an operation. • High-level interactions between user of the system and the system, between the system and other systems, or between subsystems

Creating a Sequence Diagram

Sequence Diagrams show elements as they interact over time and they are organized according to object (horizontally) and time (vertically):

Sequence Diagrams are time focus and they show the order of the interaction visually by using the vertical axis of the diagram to represent time what messages are sent and when. **Steps to Create a Sequence Diagram**

1. **Identify the Objects**: Citizen, City Administration, Smart Transportation System, Data Analytics System, Service Providers.
2. **Draw the Objects**: Represent each object as a lifeline (vertical dashed lines) at the top of the diagram.
3. **Draw Messages/Interactions**: Use horizontal arrows to show the flow of messages between objects. Label each arrow with the message or function being called.
4. **Iterative and Continuous Improvement**: Show iterative feedback loops, especially from the Data Analytics System back to the Smart Transportation System.

Symbols used in Sequence Diagram

1.Actor - -a type of role played by an entity

2.Lifeline - -an individual participant in the Interaction.

3. Activations -- - A thin rectangle on a lifeline) represents the period during which an element is performing an operation.

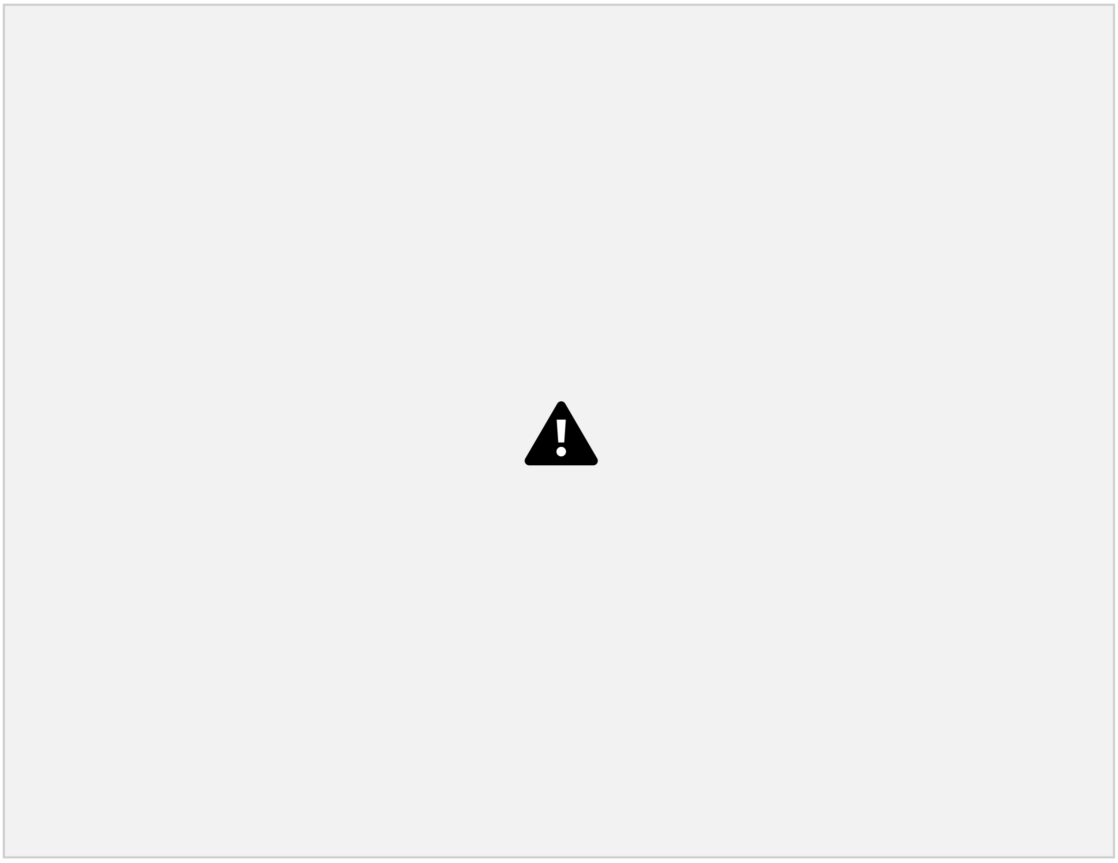
4. Call Message - - It is a kind of message that represents an invocation of operation

5. Return Message - - is a kind of message that represents the pass of information back to the caller of a corresponded former message.

6.**Self Message - -** It is a kind of message that represents the invocation of message of the same lifeline.

7. **Destroy Message- -** It is a kind of message that represents the request of destroying the lifecycle of target lifeline.

Sequence Diagram for Implementation of Smart Healthcare (in Smart City Systems)

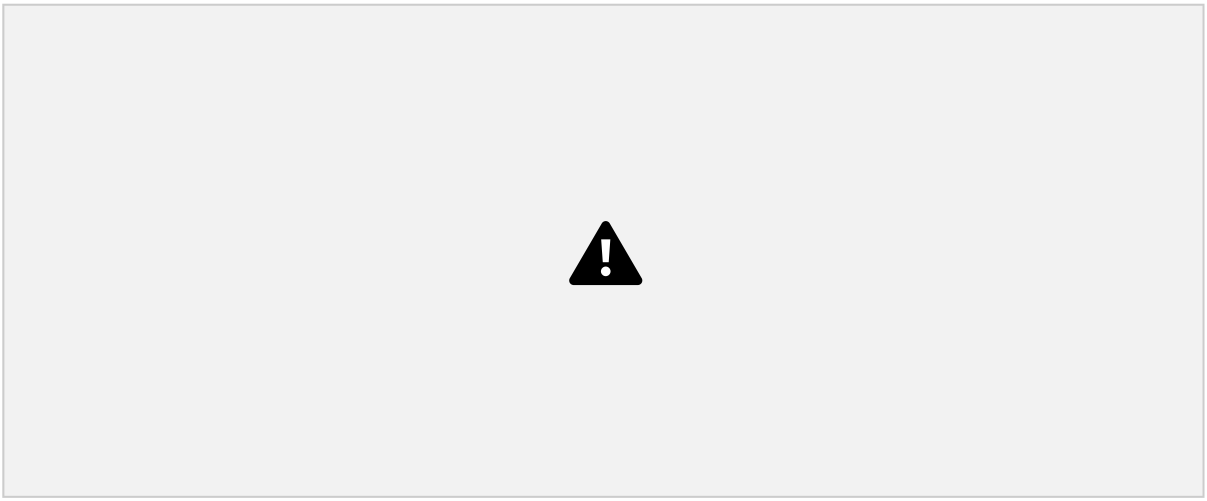


State Diagram:

State diagrams are a fundamental tool in software engineering used to model the behaviour of systems that exhibit dynamic behaviour or that involve various states and transitions. **Steps to Create a State Diagram:**

1. **Identify the System or Subsystem**: Determine the part of the software system to model with the state diagram. This could be a specific module, component, or feature of the system.
2. **Identify States**: Identify the distinct states that the system or subsystem can be in. States represent different conditions or modes that the system undergoes during its lifecycle.
3. **Define Events**: Identify events or triggers that cause the system to transition from one state to another. Events could be external stimuli, internal conditions, or changes in data. 4. **Draw the States**: Using a diagramming tool, draw the states as rounded rectangles. Label each state with a meaningful name that describes the condition or mode.
5. **Draw Transitions**: Draw arrows between states to represent transitions. Label each arrow with the event that triggers the transition. Ensure arrows are clear and properly directed (from the current state to the next state).
6. **Review and Validate**: Review the state diagram to ensure all states and transitions are accurately represented. Validate with stakeholders or subject matter experts to ensure completeness and correctness.
7. **Refine and Iterate**: Make refinements based on feedback and iterate on the diagram until it accurately reflects the behavior of the system or subsystem.

State Diagram for Implementation of Smart Transportation (in Smart City Systems)

Result:

Once the objects and their responsibilities, different states and events **have been** identified, sequence and state diagram are **drawn using the tool UMLet .**

Exp 5: Perform Manual Testing using TestRail

Objectives:

- To create and configure manual test plans and test cases
- To create and run suites or individual test cases

Tool: Testrail

Pre-requisite: Use cases and its equivalent test cases

Procedure:

Test Rail

TestRail is a test management tool that helps organize and manage testing efforts. The steps to perform manual testing using TestRail:

1. Setting Up TestRail
   i. Create a TestRail Account
   ii. Create a Project: Go to the 'Projects' tab and create a new project to store test cases, test suites, and test runs.
2. Creating Test Cases
   i. Navigate to the Test Cases Section: Select the project and go to the 'Test Cases' tab. ii. Add a New Test Case: Click on the 'Add Test Case' button.
   iii. Fill in Test Case Details: Enter the title, description, preconditions, steps, and expected results. The description should clearly define what is being tested.
   iv. Save the Test Case: Once all details are entered, save the test case.
3. Organizing Test Cases
   i. Create Test Suites: Test cases can be grouped into test suites for better organization. Go to the 'Test Suites & Cases' tab and create a new test suite.
   ii. Add Test Cases to Suites: Move or add test cases to the relevant test suite.

4. Creating a Test Run
   i. Navigate to the Test Runs & Results Section: Select your project and go to the 'Test Runs & Results' tab.
   ii. Create a New Test Run: Click on the 'Add Test Run' button.
   iii. Select Test Cases: Choose the test cases you want to include in this test run. iv. Configure the Test Run: Set the name, description, and any other relevant settings for the test run.
   v. Start the Test Run: Once configured, start the test run.
5. Executing Test Cases

i. Open a Test Run: Go to the 'Test Runs & Results' tab and select the test run created. ii. Execute Test Cases: For each test case in the test run, follow the steps and check the expected results against actual results.

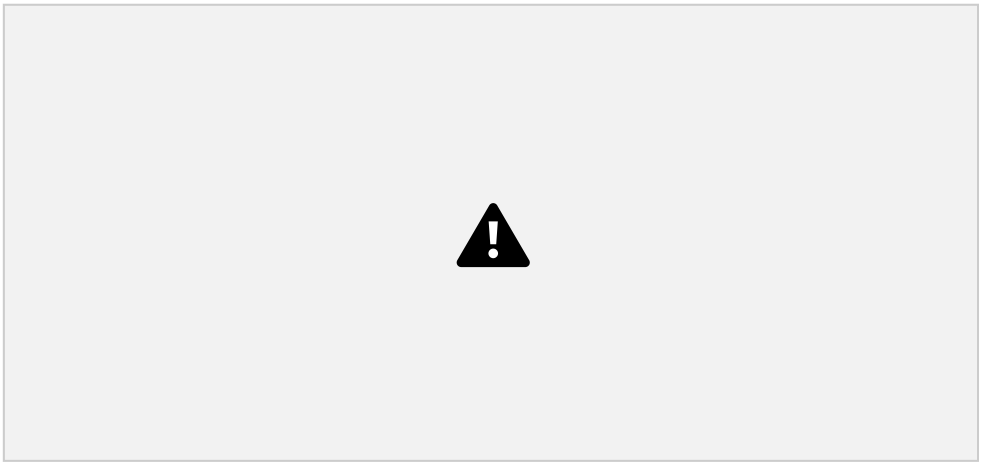
iii. Record Results: Mark the test case as passed, failed, or blocked based on the outcome. Add comments or attach files if necessary.

6. Reporting and Tracking

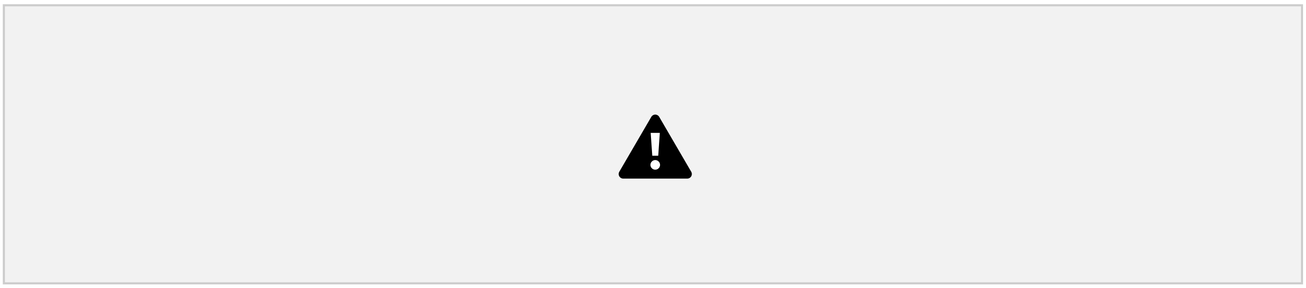
   i. View Test Run Summary: TestRail provides a summary of the test run showing the status of each test case.

   ii. Generate Reports: Use the 'Reports' tab to create detailed reports on the test run, test cases, and overall project status.

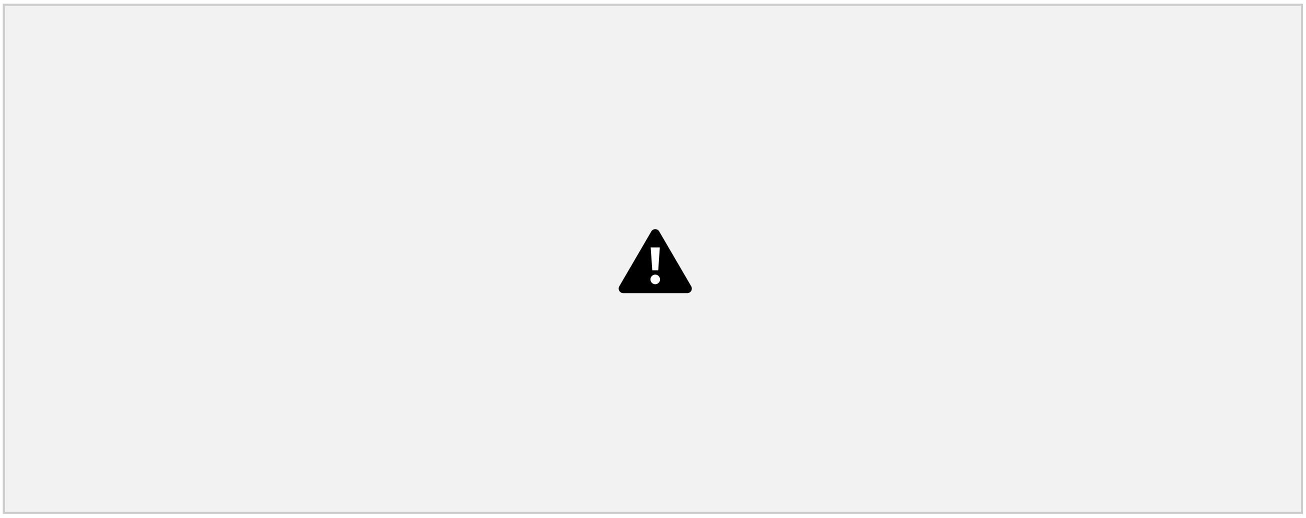
Creation of a Project Smart city



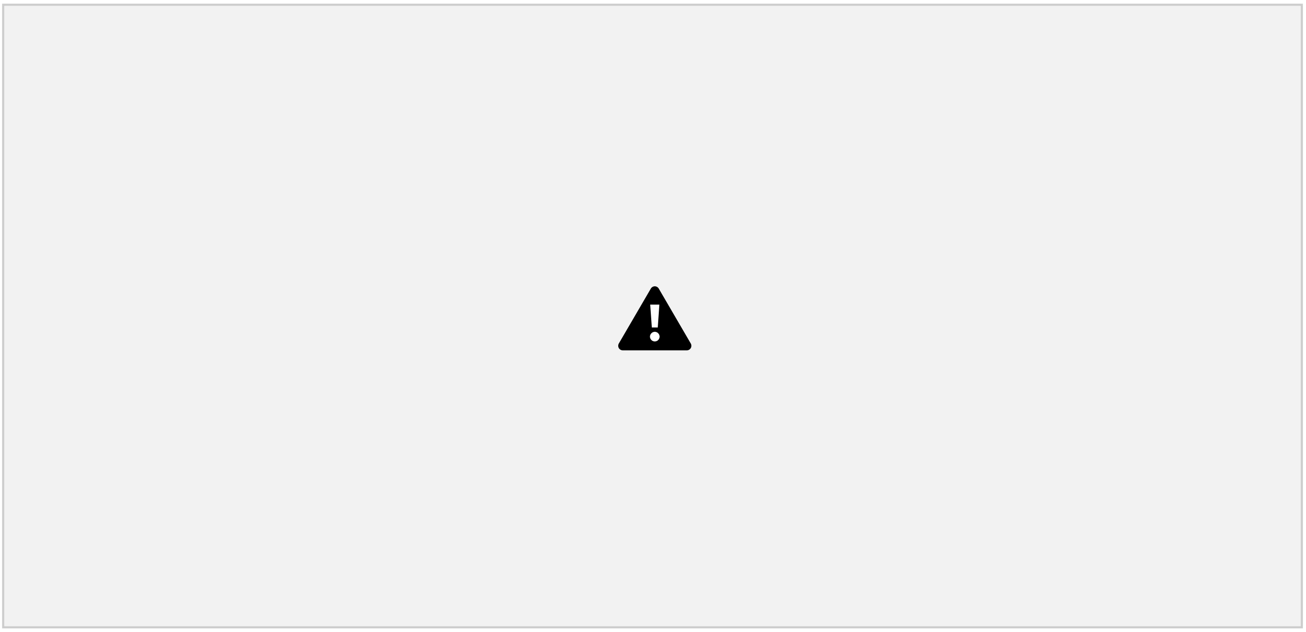
Creation of Test cases



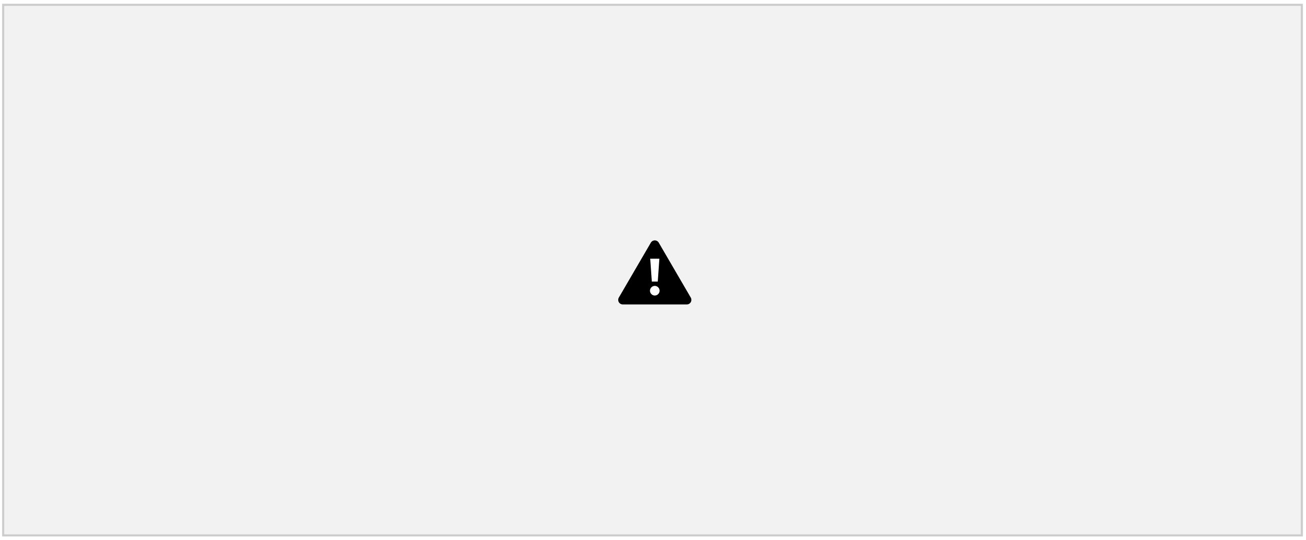
Creation of Test case Smart Healthcare



Status of Test cases before Execution:



Sample Output



Exp 6: Testing individual methods or functions within classes using Unit Testing Objectives: The students will ensure that the individual methods and functions within a java class work as expected by Identifying and fixing the bugs early in the development cycle **Tools:** Eclipse IDE, JUnit 5

Pre-Requisite:

- Knowledge of JAVA Programming
- Knowledge of Annotations for Junit testing
    - ➢ @Test annotation specifies the method is the test method.
    - ➢ @Test(timeout=100) annotation specifies the method will be failed if it takes longer than 100 milliseconds.
    - ➢ @BeforeClass annotation specifies the method will be invoked only once, before starting all the tests.
    - ➢ @Before annotation specifies the method will be invoked before each test.
    - ➢ @After annotation specifies the method will be invoked after each test.
    - ➢ @AfterClass annotation specifies the method will be invoked only once, after finishing all the tests.
- Knowledge of Assert Class for Junit Testing

  The common methods of Assert class are as follows:
    - ➢ void assertEquals(boolean expected,boolean actual): checks that two

primitives/objects are equal. It is overloaded.

- ➢ void assertTrue(boolean condition): checks that a condition is true.
- ➢ void assertFalse(boolean condition): checks that a condition is false.
- ➢ void assertNull(Object obj): checks that object is null.
- ➢ void assertNotNull(Object obj): checks that object is not null.

Procedure:

1. Install JAVA and Set the JAVA_HOME environment variable to the root directory location on your machine where Java is installed
2. Install Eclipse IDE and create a new JAVA project Calculator.java
3. Create a new pom.xml file in the root directory as the configuration file for Maven

```
<dependencies>
<!-- JUnit 5 dependencies -->
<dependency>
<groupId>org.junit.jupiter</groupId>
<artifactId>junit-jupiter-api</artifactId>
<version>5.8.0</version>
<scope>test</scope>
</dependency>
<dependency>
<groupId>org.junit.jupiter</groupId>
<artifactId>junit-jupiter-engine</artifactId>
<version>5.8.0</version>
<scope>test</scope>
</dependency>
</dependencies>
```

4. Open build.gradle file in the project's root directory, add the following dependencies to the dependencies section of build.gradle file and save the file

```
dependencies {
// JUnit 5 dependencies
testImplementation 'org.junit.jupiter:junit-jupiter-api:5.8.0'
testRuntimeOnly 'org.junit.jupiter:junit-jupiter-engine:5.8.0'
}
```

5. Write unit testing test cases using Junit
6. Run the test scenarios using test runner and view the test case results

Program:

Calculator.java

```
public class Calculator {
 public int add(int a, int b) {
 return a + b;
 }
 public int subtract(int a, int b) {
 return a - b;
 }
 public int multiply(int a, int b) {
 return a * b;
 }
 public int divide(int a, int b) {
 if (b == 0) {
 throw new IllegalArgumentException("Cannot divide by zero");  }
 return a / b;
}
}
```

CalculatorTest.java

```
import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
public class CalculatorTest {
private Calculator calculator;
@BeforeEach
void setUp() {
calculator = new Calculator();
}
@Test
void testAdd() {
int result = calculator.add(5, 3);
Assertions.assertEquals(8, result);
}
@Test
void testSubtract() {
int result = calculator.subtract(10, 4);
Assertions.assertEquals(6, result);
}
@Test
void testMultiply() {
int result = calculator.multiply(2, 5);
Assertions.assertEquals(10, result);
}
@Test
void testDivide() {
int result = calculator.divide(10, 2);
Assertions.assertEquals(5, result);
}
@Test
void testDivideByZero() {
Assertions.assertThrows(IllegalArgumentException.class, () -> {
calculator.divide(10, 0);
});
}
}
```

Sample Output:

Exp 7. Testing interactions between different classes using Integration Testing

Objectives:

The students will learn to verify the correct interaction between software components by identifying and resolving integration issues with an emphasize on seamless data flow and end-to-end system functionality

Tools: Eclipse IDE, JUnit 5

Pre-Requisite:

- Understanding of JAVA programming basics, including classes, objects, methods and inheritances
- Familiarity with object-oriented programming principles and concepts like encapsulation, polymorphism, and abstraction
- Knowledge of creating packages in java programs and creating multiple java files under one package

Procedure:

1. Open Eclipse and create a new Java Project: File -> New -> Java Project
2. Name the project as IntegrationTestingExample
3. Right click on the project and select Build Path -> Add Libraries
4. Choose Junit -> Next and select the Junit version (Junit5) and click Finish
5. Create necessary packages and classes
6. Right-click on the 'src' folder, select 'New->Junit Test Case' and Name the test class as 'ServiceIntegrationTest'
   a. In the test class, initialize objects and dependencies n a @Before or @BeforeClass method
7. Write test methods to test interactions between modules using assertions to verify expected outcomes
8. Right-click on the test class in the package explorer
   a. Select Run as -> Junit Test
9. Check Junit view in Eclipse for the test results
10. Interpret and fix any issues if the tests fail

Program:

ServiceA.java

```
package com.example.service;
public class ServiceA {
 public String process(String input) {
 return "Processed: " + input;
 }
}
```

ServiceB.java

```
package com.example.service;
public class ServiceB {
 private ServiceA serviceA;

 public ServiceB(ServiceA serviceA) {
 this.serviceA = serviceA;
 }
 public String process (String input) {
 return serviceA.process(input) + " and enhanced by ServiceB";
 }
}
```

ServiceIntegrationTest.java

```
package com.example.test;
```

```
import static org.junit.Assert.assertEquals;
import org.junit.Before;
import org.junit.Test;
import com.example.service.ServiceA;
import com.example.service.ServiceB;
public class ServiceIntegrationTest {
 private ServiceA serviceA;
 private ServiceB serviceB;
 @Before
 public void setUp() {
 // Initialize services
 serviceA = new ServiceA();
 serviceB = new ServiceB(serviceA); // ServiceB depends on ServiceA  }
 @Test
 public void testServiceIntegration() {
 // Arrange
 String input = "test input";
 String expectedOutput = "Processed: test input and enhanced by ServiceB";  //
Act
 String actualOutput = serviceB.process(input);
 // Assert
 assertEquals(expectedOutput, actualOutput);
 }
}
```

Sample Output:

```
JUnit 4
----------------------------------
com.example.test.ServiceIntegrationTest
 ✓ testServiceIntegration (passed)
----------------------------------
Total Tests: 1
Failures: 0
Ignored: 0
Time: 0.003 sec
```

Exp 8. Exercise Regression Testing by incorporating changes in the previously tested modules

Objectives:

The students will learn to verify recent changes in java code haven't inadvertently introduced defects or affected existing functionality to ensure software stability and reliability. The students will also learn to maintain the quality of software by validating the correctness and detecting regressions early in the development cycle

Tools: Eclipse IDE, JUnit 5

Pre-requisite:

- Eclipse IDE with Java Development Kit (JDK) installed and configured.
- Integration of JUnit framework into the Eclipse project either through manual setup or using build automation tools like Maven or Gradle.
- Defined JUnit test cases within the project that cover critical functionalities and scenarios affected by recent changes.
- Access to the source code of the application or module under test within the Eclipse project.

Procedure:

1. Open Eclipse and create a new Java Project: File -> New -> Java Project
2. Name the project as 'example'

3. Integrate Junit framework into your Eclipse project manually or through build automation tools like Maven or Gradle
4. Implement Item class with attributes namely ‘name’ and ‘price’ and methods ‘getName’ and ‘getPrice’
5. Implement the ‘ShoppingCart’ class to add or remove items, calculate total and apply discounts
6. Develop JUnit test cases (ShoppingCartTest) to verify the functionalities of the ShoppingCart class, including regression tests to detect unintended changes
7. Launch Eclipse and open your Java project containing the Item and ShoppingCart classes and the ShoppingCartTest JUnit test cases.
8. Open the ShoppingCartTest class in Eclipse to review existing JUnit test methods (testCalculateTotal, testCalculateTotalWithDiscount, testAddItem, testRemoveItem, etc.) and their expected outcomes.
9. Right-click on the ShoppingCartTest class in Eclipse.
10. Select Run As > JUnit Test to execute all existing test cases.
11. Ensure that all initial tests pass successfully without any failures.
12. Introduce new test methods or modify existing ones in ShoppingCartTest to simulate scenarios where recent changes might introduce regressions. Example: intentionally include failing assertions (assertEquals(expected, actual)) to verify correct behavior before and after changes.

Program:

ShoppingCart.java

```
package com.example;
import java.util.ArrayList;
import java.util.List;
public class ShoppingCart {
 private List<Item> items;
 public ShoppingCart() {
 this.items = new ArrayList<>();
 }
 public void addItem(Item item) {
 items.add(item);
 }
 public void removeItem(Item item) {
 items.remove(item);
 }
 public double calculateTotal() {
 double total = 0.0;
 for (Item item : items) {
 total += item.getPrice();
 }
 return total;
 }
 public double calculateTotalWithDiscount(double discountRate) {
double total = calculateTotal();
 return total * (1 - discountRate);
 }
 public List<Item> getItems() {
 return new ArrayList<>(items);
 }
}
```

Item.java

```
package com.example;
public class Item {
 private String name;
 private double price;
 public Item(String name, double price) {
 this.name = name;
 this.price = price;
 }
 public String getName() {
 return name;
 }
 public double getPrice() {
 return price;
 }
}
```

ShoppingCartTest.java

```
package com.example;
import static org.junit.Assert.*;
import org.junit.Before;
import org.junit.Test;
public class ShoppingCartTest {
 private ShoppingCart cart;
 @Before
 public void setUp() {
 cart = new ShoppingCart();
 Item item1 = new Item("Laptop", 1000.0);
 Item item2 = new Item("Mouse", 20.0);
 cart.addItem(item1);
 cart.addItem(item2);
 }
 @Test
 public void testCalculateTotal() {
 assertEquals(1020.0, cart.calculateTotal(), 0.001); // Expected total: 1020.0  }
 @Test
 public void testCalculateTotalWithDiscount() {
 assertEquals(918.0, cart.calculateTotalWithDiscount(0.1), 0.001); // 10% discount, expected
total: 918.0
 }
 @Test
 public void testAddItem() {
 Item newItem = new Item("Keyboard", 50.0);
 cart.addItem(newItem);
 assertTrue(cart.getItems().contains(newItem));
 }
 @Test
 public void testRemoveItem() {
 Item itemToRemove = new Item("Mouse", 20.0);
 cart.removeItem(itemToRemove);
 assertFalse(cart.getItems().contains(itemToRemove));
 }
// Regression test for ShoppingCart class after changes
@Test
public void testCalculateTotalWithDiscountRegression() {
// Intentionally failing assertion to simulate a regression
assertEquals(950.0, cart.calculateTotalWithDiscount(0.07), 0.001); // Expected total: 950.0  }
}
```

Sample Output:

```
Tests run: 4, Failures: 1, Errors: 0, Skipped: 0, Time elapsed: X.XXX sec <<< FAILURE!
testCalculateTotalWithDiscountRegression(com.example.ShoppingCartTest) Time elapsed: X.XXX
sec <<< FAILURE!
java.lang.AssertionError:
Expected :950.0
Actual :918.0
```

Exp 9. Conduct code coverage testing using JaCoCo/Cobertura

Objectives:

The students will learn to measure the extent to which the source code of a program is executed when a particular test suite runs like statement coverage, branch coverage, function coverage and path coverage to improve the effectiveness of tests and the reliability of the software **Tools:** Eclipse IDE, JUnit 5

Pre-requisite:

- Eclipse IDE with Java Development Kit (JDK) installed and configured.
- Installation of JaCoCo Plugin

Procedure:

1. Create a new JAVA project File-> New-> JAVA project->StringPalindromeProject
2. Create a new pom.xml file in the root directory as the configuration file for Maven

Maven build

Add JaCoCo Maven plugin to 'pom.xml'

```
<build>
 <plugins>
 <plugin>
 <groupId>org.jacoco</groupId>
 <artifactId>jacoco-maven-plugin</artifactId>
 <version>0.8.8</version>
 <executions>
 <execution>
 <goals>
 <goal>prepare-agent</goal>
 </goals>
 </execution>
 <execution>
 <id>report</id>
 <phase>test</phase>
 <goals>
 <goal>report</goal>
 </goals>
 </execution>
 </executions>
 </plugin>
 </plugins>
</build>
```

Gradle build

Open build.gradle file in the project's root directory, add the following dependencies to the dependencies section of build.gradle file and save the file plugins {

```
 id 'jacoco'
 }
jacoco {
 toolVersion = "0.8.8"
 }
test {
 finalizedBy jacocoTestReport
 }
jacocoTestReport {
 reports {
 xml.enabled = true
 html.enabled = true
 }
 }
```

3. Right-Click on 'src' folder -> New->Package 'com.javatechie.StringPalindrome' 4. Right-click on the com.javatechie.StringPalindrome package > New > Class->App 5. Create two methods 'isPalindrome' and 'reverse' with appropriate logic to check whether the given string is palindrome or not and reversing a string
6. Right-Click on 'com.javatechie.StringPalindrome' package->new->class->AppTest
7. Write appropriate test cases to test App.java program
8. Right-Click on the project->build path->Add libraries
9. Select Junit->next->Junit 5
10. Goto Help->Eclipse Marketplace->type JaCoCo in the search box
11. Find and install JaCoCo plugin
12. Right-Click on the project->Configure->Convert to Maven Project
13. Right-Click on the project->Run As->Mavent test
14. View the generated coverage report in the target/site/jacoco directory
15. Open 'index.html' in the directory to view detailed coverage information
16. Right-click on the 'AppTest' ->Coverage As->Junit Test

Program:

App.java

```
package com.javatechie.StringPalindrome;
public class App {
	public boolean isPalindrome(String input) {

		if (input == null) {
			throw new IllegalArgumentException("input shouldn't be null");
		}

		if (input.equals(reverse(input))) {
			return true;
		} else {
			return false;
		}
	}
	private String reverse(String input) {
		String rev = "";
		for (int i = input.length() - 1; i >= 0; i--) {
			rev = rev + input.charAt(i);
```

```
                }
                return rev;
        }
}
```
AppTest.java

```
package com.javatechie.StringPalindrome;
import static org.junit.Assert.assertEquals;
import org.junit.Test;
public class AppTest {
        String input1 = "noon";
        App app = new App();
        boolean expected = true;
        @Test
        public void isPlaindromeTest() {
                assertEquals(expected, app.isPalindrome(input1));
        }
        @Test
        public void isNotPlaindromeTest() {
                assertEquals(false, app.isPalindrome("abc"));
        }
        @Test(expected = IllegalArgumentException.class)
        public void isNotPlaindromeExceptionTest() {
                assertEquals(false, app.isPalindrome(null));
        }
}
```
Sample Output:

```
Tests run: 3, Failures: 0, Errors: 1, Skipped: 0, Time elapsed: X.XXX sec <<< FAILURE!
isNotPlaindromeExceptionTest(com.javatechie.StringPalindrome.AppTest) Time elapsed: X.XXX
sec <<< ERROR!
java.lang.IllegalArgumentException: input shouldn't be null
        at com.javatechie.StringPalindrome.App.isPalindrome(App.java:7)
        at
com.javatechie.StringPalindrome.AppTest.isNotPlaindromeExceptionTest(AppTest.java:20)
```
Exp 10. Testing single method multiple times with different set of parameters using Parameterized Testing

Objectives:

The students will learn to test the method with different data in one go and try out coding with lots of different inputs to catch any issues

Tools: Eclipse IDE, JUnit 5

Pre-requisite:

• Knowledge of JUnit for unit testing

• Understanding of JUnit concepts like annotations, assertions and test runners • Properly configured Eclipse IDE with Java Development Tools (JDT) plugin • Understanding different types of argument sources like @ValueSource, @EnumSource or @CsvSource

Procedure:

1. Create a new java class ‘PrimeNumberTest’
2. Add ‘@ParameterizedTest’ annotation to mark the test method as parameterized 3. Add ‘@CsvSource’ annotation to pass multiple sets of parameters to the test method 4. Use ‘assertEquals’ assertion to check if the actual result from ‘isprime’ method matches the expected result
5. Define ‘isPrime’ method to determine if a number is prime by checking divisibility from 2

upto square root of the number

6. Right-click on test class and select Run As->JUnit Test

Program:

```
import static org.junit.jupiter.api.Assertions.assertEquals;
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.CsvSource;
public class PrimeNumberTest {
@ParameterizedTest
@CsvSource({
"2, true",
"3, true",
"4, false",
"5, true",
"9, false",
"17, true"
})
void testIsPrime(int number, boolean expectedResult) {
assertEquals(expectedResult, isPrime(number));
}
boolean isPrime(int number) {
if (number <= 1) {
return false;
}
for (int i = 2; i <= Math.sqrt(number); i++) {
if (number % i == 0) {
return false;
}
}
return true;
}
}
```

Sample Output:

```
Tests run: 6, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: X.XXX sec
```

Exp 11. Conduct concurrency testing to run test cases in parallel or simulating concurrent scenarios using Junit Pioneer

Objectives:

The students will learn to apply concurrent and custom test execution to test multi-threaded code with comprehensive assertions and improved parameterized testing features. **Tools:** Eclipse IDE, JUnit 5

Pre-Requisite:

• Knowledge of JUnit for unit testing

• Understanding of JUnit concepts like annotations, assertions and test runners • Properly configured Eclipse IDE with Java Development Tools (JDT) plugin

Procedure:

1. Open Eclipse, Goto File->New->Project
2. Select ‘Maven Project’
3. Choose workplace location and select an archetype ‘maven-archetype-quickstart’
4. Fill the group and artifact id
5. Modify the following code in ‘pom.xml’

```
<dependencies>
<!-- JUnit 5 Dependency -->
```

```
        <dependency>
        <groupId>org.junit.jupiter</groupId>
        <artifactId>junit-jupiter-engine</artifactId>
        <version>5.7.0</version>
        <scope>test</scope>
        </dependency>
        <!-- JUnit Pioneer Dependency -->
        <dependency>
        <groupId>org.junit-pioneer</groupId>
        <artifactId>junit-pioneer</artifactId>
        <version>1.5.0</version>
        <scope>test</scope>
        </dependency>
       </dependencies>
```

6. Right-click on the project, select maven->update project
7. Implement ConcurrentQueue' class in the 'src/main/java' directory
8. Create a new test class 'ConcurrentQueueTest' in 'src/test/java' directory
9. Right-click on the 'ConcurrentQueueTest'
10. Select Run As->JUnit Test

Program:

ConcurrentQueue.java

```
import java.util.concurrent.ConcurrentLinkedQueue;
public class ConcurrentQueue<T> {
 private ConcurrentLinkedQueue<T> queue = new ConcurrentLinkedQueue<>();

 public void enqueue(T element) {
 queue.add(element);
 }
 public T dequeue() {
 return queue.poll();
 }
 public int size() {
 return queue.size();
 }
}
```

ConcurrentQueueTest.java

```
import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.junit.jupiter.api.Assertions.assertNotNull;
import org.junit.jupiter.api.Test;
import org.junitpioneer.jupiter.concurrent.Concurrent;
import org.junitpioneer.jupiter.concurrent.ConcurrentExecution;
public class ConcurrentQueueTest {
 @Test
 @Concurrent(threads = 10)
 public void testConcurrentEnqueue() throws InterruptedException {
ConcurrentQueue<Integer> queue = new ConcurrentQueue<>();  int
numberOfElements = 1000;
 // Concurrently enqueue elements
 ConcurrentExecution.run(() -> {
 for (int i = 0; i < numberOfElements; i++) {
 queue.enqueue(i);
 }
 }, 10);
 // Ensure all elements are enqueued
 assertEquals(numberOfElements * 10, queue.size());  }
 @Test
 @Concurrent(threads = 10)
 public void testConcurrentDequeue() throws InterruptedException {
ConcurrentQueue<Integer> queue = new ConcurrentQueue<>();  int
numberOfElements = 1000;
 // Enqueue elements
 for (int i = 0; i < numberOfElements * 10; i++) {
 queue.enqueue(i);
 }
 // Concurrently dequeue elements
 ConcurrentExecution.run(() -> {
 for (int i = 0; i < numberOfElements; i++) {
 Integer value = queue.dequeue();
 assertNotNull(value);
 }
 }, 10);
 // Ensure all elements are dequeued
 assertEquals(0, queue.size());
 }
 }
```

Sample Output:

```
Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: X.XXX sec
```

Exp 12: Perform web site testing using selenium IDE

Objectives:

The objectives include

- Record a test that contains navigation to a web site.
- Analyze the Web site.
- Check for broken links, orphan links
- Repair broken links and orphan links.

Tool: Selenium IDE

Pre-requisite: HTML Knowledge

Procedure:

Step 1: Install Selenium IDE

1. Install Selenium IDE Extension:
For Chrome: [Selenium IDE Chrome Extension](https://chrome.google.com/webstore/detail/selenium ide/mooikfkahbdckldjjndioackbalphokd)
For Firefox: [Selenium IDE Firefox Extension](https://addons.mozilla.org/en US/firefox/addon/selenium-ide/)
2. Launch Selenium IDE: After installing, open the extension from browser's toolbar.

Step 2: Create a New Project

i.Open Selenium IDE: Click the Selenium IDE icon in the browser.
ii.Create a New Project: Click the 'Create a new project' button.
iii..Name ther Project: Enter a name for the project and click 'OK'.

Step 3: Record a Test

i. Start Recording: Click the 'Record a new test in a new project' button.

ii. Enter Test Name: Provide a name for the test and click 'OK'.
iii.Enter URL: Enter the URL of the website the tester wants to test and click 'Start recording'.
iv. Perform Actions: Navigate through the website and perform actions that tester wants to test (e.g., clicking buttons, entering text).
v. Stop Recording: Once done, click the 'Stop recording' button.

Step 4: Edit the Recorded Test

i. View Recorded Steps: After stopping the recording, the tester can see the list of recorded steps in the Selenium IDE.
ii. Edit Steps: The user can edit the recorded steps if needed. For example, it is possible to change target elements, add assertions, or modify commands.
iii. Add Command: Right-click on a step and choose 'Insert new command'. iv.
Edit Command: Click on a command to edit it.

Step 5: Add Assertions

a. Add Assertions: Assertions are used to verify that the website behaves as expected. Right-click on a step and choose 'Add assertion'.
b. Choose Assertion Type: Select the type of assertion (e.g., `assertText`, `assertTitle`).
ii. Specify Target and Value: Specify the target element and the expected value for the assertion.

Step 6: Run the Test

i. Run the Test: Click the 'Run current test' button to execute the recorded test.
ii. View Results: Check the log and results to see if the test passed or failed. Step
7: Debug and Modify Tests
i. Debug Test: If a test fails, review the steps and log to understand the failure.
ii. 2. Modify Test: Make necessary changes to the test steps to fix any issues. iii.
Re-run Test: Run the test again to verify that the issue is resolved.

Step 8: Save the Project

i. Save Project: Click on 'File' -> 'Save' to save your project.
ii. Export Test: The tester can also export the test as a script (e.g., in JavaScript) by clicking on 'File' -> 'Export'.

Step 9: Schedule and Automate Tests

i. Use CI/CD: Integrate Selenium IDE with Continuous Integration/Continuous Deployment (CI/CD) tools like Jenkins to schedule and automate your tests.
ii. Run Tests on Multiple Browsers: Use Selenium WebDriver to run your tests on different browsers if needed.

Sample Output:

Implementation of website testing for SASTRA website