LIST OF EXPERIMENTS

S. No.	Name of the Experiment
1.	Introduction to Linux Operating System(Pre-requisite)
2.	Understand the background and driving forces for taking an Agile Approach to Software Development
3.	Case study of project management using traditional software development model
4.	Comparative study of open source agile tools
5.	Installation and use of open source agile tools for software development
6.	Project Planning and Tracking System using extreme programming(XP)
7.	Implementation of SOLID principles using any programming language
8.	Software testing using agile tools
9.	Test Driven Development using XUnit.
10.	Testing web apps using selenium
11.	Study and use of Configuration Management tools
12.	Study and use of DevOps Automation Tools

Programme Outcomes (POs):

Engineering Graduates will be able to:

1. **Engineering knowledge**: Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.
2. **Problem analysis**: Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.
3. **Design/development of solutions**: Design solutions for complex engineering problems anddesign system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.
4. **Conduct investigations of complex problems**: Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.
5. **Modern tool usage**: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.
6. **The engineer and society**: Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.
7. **Environment and sustainability**: Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.
8. **Ethics**: Apply ethical principles and commit to professional ethics and responsibilities and norms ofthe engineering practice.
9. **Individual and team work**: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.
10. **Communication**: Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.
11. **Project management and finance**: Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.
12. **Life-long learning**: Recognize the need for, and have the preparation and ability to engage independent and life-long learning in the broadest context of technological change.

LABORATORY OUTCOMES

The practical/exercises in this section are psychomotor domain Learning Outcomes (i.e. subcomponents of the COs), to be developed and assessed to lead to the attainment of the competency.

LO-1: Understand the background and driving forces for taking an Agile Approach to Software Development

LO-2: Make use of different open source agile tools

LO-3: Apply Design principle and Refactoring to achieve agility

LO-4 : Implement Test Driven Development using XUnit

LO-5: Test web apps using selenium

LO-6: Make use of configuration management tools for project

1. Lab Exercise

Exercise No 1: (2 Hours) – 1 Practical

Aim: - Introduction to Linux Operating System (Pre-requisite)

Objectives:

1. Student will able to install Linux operating system
2. Students should able to use commands and utilities of Linux for their day to day work.
3. Student will able to write simple shell scripts

THEORY:

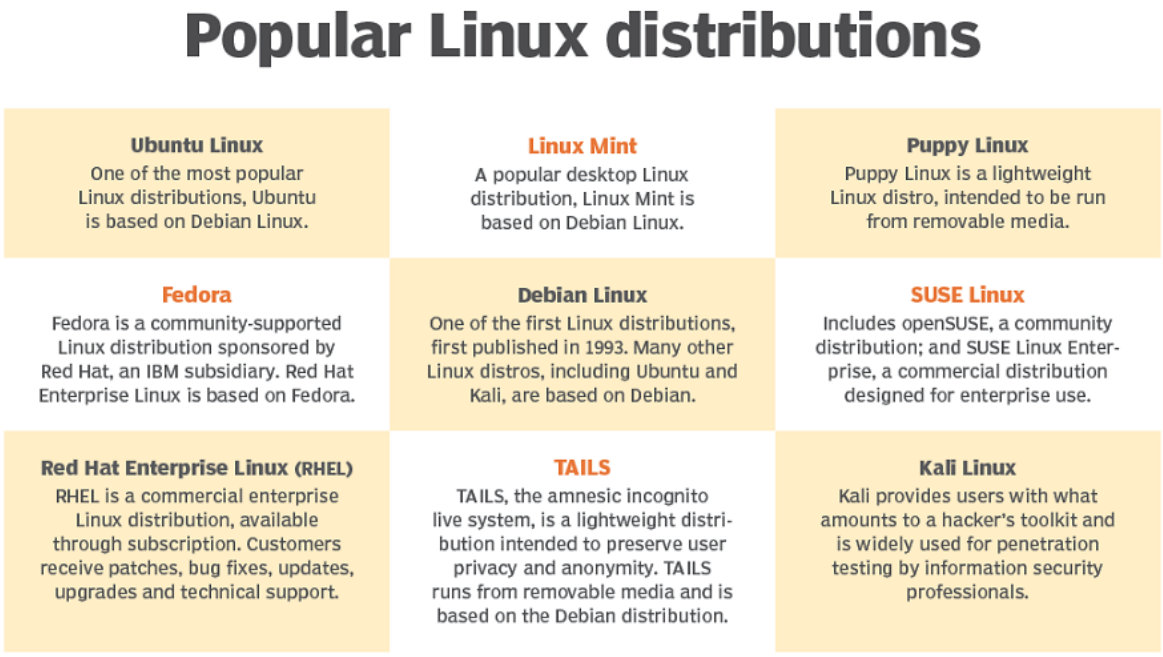
Introduction:

Linux is a Unix-like, open source and community-developed operating system (OS) for computers, servers, mainframes, mobile devices and embedded devices. It is supported on almost every major computer platform, including x86, ARM and SPARC, making it one of the most widely supported operating systems. Every version of the Linux OS manages hardware resources, launches and handles applications, and provides some form of user interface. The enormous community for developers and wide range of distributions means that a Linux version is available for almost any task, and Linux has penetrated many areas of computing.

In Sept 1991, Linus Torvalds, a second year student of Computer Science at the University of Helsinki, developed the preliminary kernel of Linux, known as Linux version 0.0.1. It was put to the Internet and received enormous response from worldwide software developers. By December came version 0.10. Still Linux was little more than in skeletal form.

Linux Distributions:

Since its initial development, Linux has adopted the copy left stipulations of the Free Software Foundation which originated the GNU GPL. The GPL says that anything taken for free and modified must be distributed for free. In practice, if Linux or other GNU-licensed components are developed or modified to create a new version of Linux, that new version must be distributed for free.



Installation of Linux Operating System:

Linux will be installed on any machine. It can be installed on standalone machine or server. Following are different installation methods of Linux operating system.

- Sun Installation Assistant (SIA)
- CD/DVD Media
- Network or PXE
- Remote KVMS

STRUCTURE OF A LINUXSYSTEM:

It consists of three parts.

a)UNIX kernel

b) Shells

c) Tools and Applications

UNIX KERNEL: Kernel is the core of the UNIX OS. It controls all tasks, schedule all Processes and carries out all the functions of OS. Decides when one programs tops and another starts.

SHELL: Shell is the command interpreter in the UNIX OS. It accepts command from the user and analyses and interprets them

Outcome:

To learn the installation process of Linux distribution and perform basic operations using commands and utilities.

CONCLUSIONS:

Students will able to install any Linux distribution on the laptop or desktop. Students will be able to develop simple shell scripts and execute them.

Journal Write-up

- History of Linux operating System
- Installation of Linux
- Virtualization
- Booting process and run levels
- File System & Directory structure
- Commands and utilities
- User and group management
- File permissions
- Shell programming
- Package management in Linux
- Editors in Linux
- Conclusion

2. Lab Exercise

Exercise No 2: (2 Hours) – 1 Practical

Aim: - Understand the background and driving forces for taking an Agile Approach to Software Development

Objectives:

1. Difference between agile software development model and waterfall model.
2. Why Agile is better?
3. Understanding the Agile Manifesto
4. Discussing Important Characteristics that make agile approach best suited for Software Development.

THEORY:

Agile software development is a group of software development methods in which requirements and solutions evolve through collaboration between selforganizing, cross-functional teams. It promotes adaptive planning, evolutionary development, early delivery, continuous improvement, and encourages rapid and flexible response to change. The Manifesto for Agile Software Development, also known as the Agile Manifesto, first introduced the term agile in the context of software development in 2001.

Manifesto for Agile Software Development

We are uncovering better ways of developing
software by doing it and helping others do it.

Through this work we have come to value:
Individuals and interactions over processes and tools
Working software over comprehensive documentation
Customer collaboration over contract negotiation
Responding to change over following a plan

That is, while there is value in the items on

the right, we value the items on the left more.

Agile principles

The Agile Manifesto is based on 12 principles:

1. Customer satisfaction by rapid delivery of useful software
2. Welcome changing requirements, even late in development
3. Working software is delivered frequently (weeks rather than months)
4. Close, daily cooperation between business people and developers
5. Projects are built around motivated individuals, who should be trusted
6. Face-to-face conversation is the best form of communication (co-location)
7. Working software is the principal measure of progress
8. Sustainable development, able to maintain a constant pace
9. Continuous attention to technical excellence and good design
10. Simplicity—the art of maximizing the amount of work not done—is essential
11. Self-organizing teams
12. Regular adaptation to changing circumstance

What's wrong With Traditional Approaches?

In 1970, Dr. Winston Royce presented a paper entitled "Managing the Development of Large Software Systems," which criticized sequential development. He asserted that software should not be developed like an automobile on an assembly line, in which each piece is added in sequential phases, each phase depending on the previous. Dr. Royce recommended against the phase based approach in which developers first gather all of a project's requirements, then complete all of its architecture and design, then write all of the code, and so on. Royce specifically objected to the lack of communication between the specialized groups that complete each phase of work. It's easy to see the problems with the waterfall method. It assumes that every requirement can be identified before any design or coding occurs. Could you tell a team of developers everything that needed to be in a software product before any of it was up and running? Or would it be easier to describe your vision to the team if you could react to functional software? Many software developers have learned the answer to that question the hard way: At the end of a project, a team might have built the software it was asked to build, but, in the time it took to create, business realities have changed so

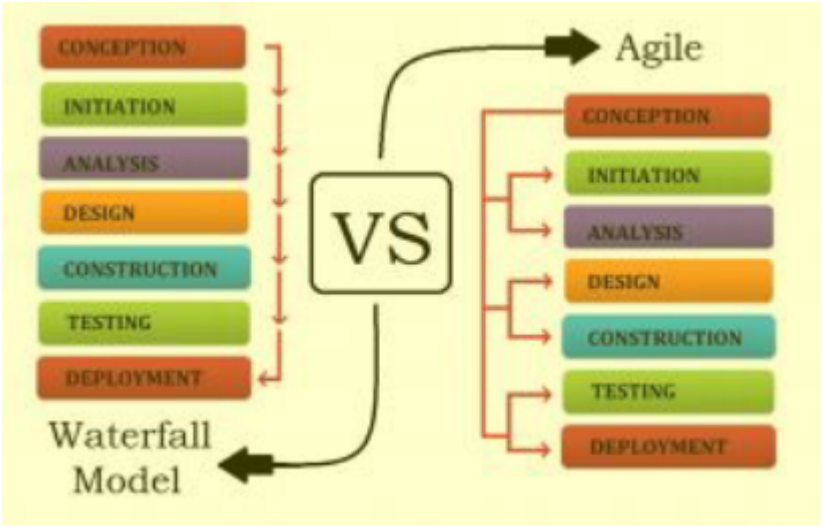
dramatically that the product is irrelevant. Your company has spent time and money to create software that no one wants. Couldn't it have been possible to ensure the end product would still be relevant before it was actually finished? Today very few organizations openly admit to doing waterfall or traditional command and control. But those habits persist.

Why Agile?

Agile development provides opportunities to assess the direction throughout the development lifecycle. This is achieved through regular cadences of work, known as Sprints or iterations, at the end of which teams must present a potentially shippable product increment. By focusing on the repetition of abbreviated work cycles as well as the functional product they yield, agile methodology is described as "iterative" and "incremental." In waterfall, development teams only have one chance to get each aspect of a project right. In an agile paradigm, every aspect of development — requirements, design, etc. — is continually revisited. When a team stops and reevaluates the direction of a project every two weeks, there's time to steer it in another direction. This "inspect-and-adapt" approach to development reduces development costs and time to market. Because teams can develop software at the same time they're gathering requirements, "analysis paralysis" is less likely to impede a team from making progress. And because a team's work cycle is limited to two weeks, stakeholders have recurring opportunities to calibrate releases for success in the real world. Agile development helps companies build the right product. Instead of committing to market a piece of software that hasn't been written yet, agile empowers teams to continuously replan their release to optimize its value throughout development, allowing them to be as competitive as possible in the marketplace. Agile development preserves a product's critical market relevance and ensures a team's work doesn't wind up on a shelf, never released.

Difference between agile software development model and waterfall model

It is worth mentioning here that the Waterfall model is the primitive model type and has been implemented in the development phase time after time. Hence in the due course if time developers found many drawbacks in this model which were later rectified to form various other development models.



Advantages of the Agile Methodology

1. The Agile methodology allows for changes to be made after the initial planning. Rewrites to the program, as the client decides to make changes, are expected.
2. Because the Agile methodology allows you to make changes, it's easier to add features that will keep you up to date with the latest developments in your industry.
3. At the end of each sprint, project priorities are evaluated. This allows clients to add their feedback so that they ultimately get the product they desire.
4. The testing at the end of each sprint ensures that the bugs are caught and taken care of in the development cycle. They won't be found at the end.
5. Because the products are tested so thoroughly with Agile, the product could be launched at the end of any cycle. As a result, it's more likely to reach its launch date.

CONCLUSIONS:

Both the Agile and waterfall methodologies have their strengths and weaknesses. The key to deciding which is right for you comes down to the context of the project. Is it going to be changing rapidly? If so, choose Agile. Do you know exactly what you need? Good. Then maybe waterfall is the better option. Or better yet? Consider taking aspects of both methodologies and combining them in order to make the best possible software development process for your project.

3. Lab Exercise

Exercise No 3: (2 Hours) – 1 Practical

Aim: - Case study of project management using traditional software development model

Objectives:

1. Students should able to understand traditional software development model
2. Students should able to understand challenges of traditional software development model

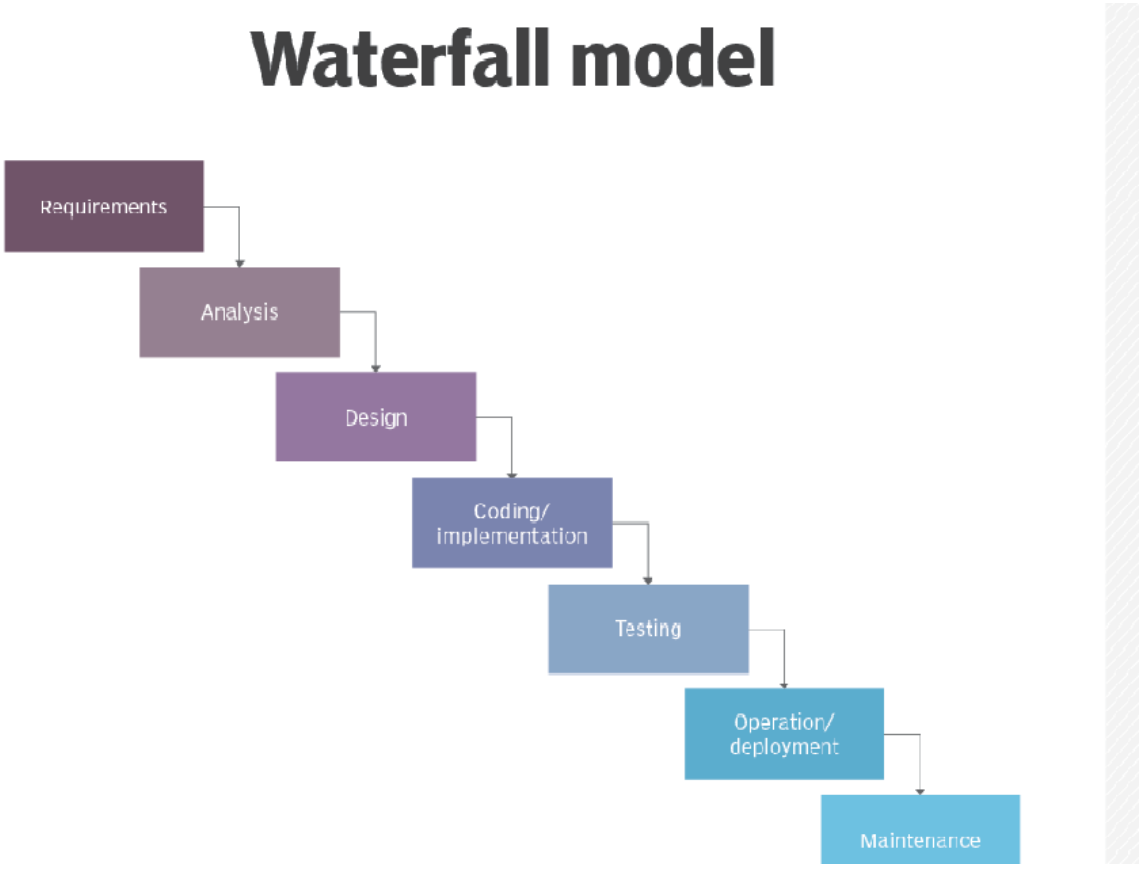
THEORY:

In this fast-moving world, project management has become one of the most important pillars that are helping businesses run without any glitch in their processes. Conflict Management is also a very big part of project management. Both small- and large-scale organizations around the world depend on project management systems to deliver their products/services successfully. Whether it is team workflow management or timing, these tools help to ensure that the processes flow in a hassle-free manner while achieving the desired goals. Despite the presence of different project management approaches, Agile is considered one of the most practical and flexible software development mechanisms that exist today. It is capable of executing a variety of tasks.

What is Traditional Project Management?

The traditional Project Management (waterfall) approach is linear where all the phases of a process occur in sequence. Its concept depends on predictable tools and experience. Each and every project follows the same life cycle which includes the stages such as feasibility, planning, designing, building, testing, production, and support, as shown in the figure above.

The entire project is planned upfront without any scope for changing requirements. This approach assumes that time and cost are variables and requirements are fixed. The rigidity of this method is the reason why it is not meant for large projects and leaves no scope for changing the requirements once the project development starts.



What is Agile Project Management?

When a traditional system focuses on upfront planning where factors like cost, scope, and time are given importance, Agile management gives prominence to teamwork, customer collaboration, and flexibility. It is an iterative approach that focuses more on incorporating customer feedback and continuous releases with every iteration of a software development project.

The basic concept behind Agile software development is that it delves into evolving changes and collaborative effort to bring out results rather than a predefined process. Adaptive planning is perhaps the top feature of Agile and one that makes it a favorite among project

managers,worldwide.

Scrum and Kanban are two of the most widely used Agile frameworks. They are very well known for encouraging decision-making and preventing time consumption on variables that are bound to change. It stresses customer satisfaction and uses available teams to fast-track software development at every stage.

Waterfall vs. Agile: How To Choose the Right Methodology for Your Project

There are several factors to consider when you are choosing between Waterfall and Agile. Here are a few questions to consider:

Does your project require strict regulations or requirements? Waterfall is better suited for projects with regulations or requirements because each phase's deliverables and strict procedures ensure that they are met. For instance, the Department of Defense and the aerospace industry are a couple of industries that would more likely use Waterfall over Agile, since the requirements are a safety factor. Dr. Chris Mattmann, Chief Technology and Innovation Officer (CTIO) at NASA Jet Propulsion Laboratory, told Forbes Advisor that "agile methodology is used more for IT companies, [companies] that fail fast and move fast, types of places where you can proceed in parallel in different phases."

When choosing between Agile and Waterfall, consider how involved the project owners or stakeholders will be in the project. Agile is better suited for projects where stakeholders are closely involved every step of the way. Waterfall is a more structured project management method and does not lend itself to the same type of flexibility.

Bottom Line

To summarize, Agile and Waterfall are two different management methodologies best suited for different types of projects. If you clearly understand the project outcomes from the beginning, Waterfall may be the best fit. Waterfall is a better method when a project must meet strict regulations as it requires deliverables for each phase before proceeding to the next one.

Alternatively, Agile is better suited for teams that plan on moving fast, experimenting with direction and don't know how the final project will look before they start. Agile is flexible and requires a collaborative and self-motivated team, plus frequent check-ins with business owners and stakeholders about the progress

CONCLUSIONS:

By studying this assignment student will able compare and contrast SDLC waterfall model and agile project management.

4. Lab Exercise

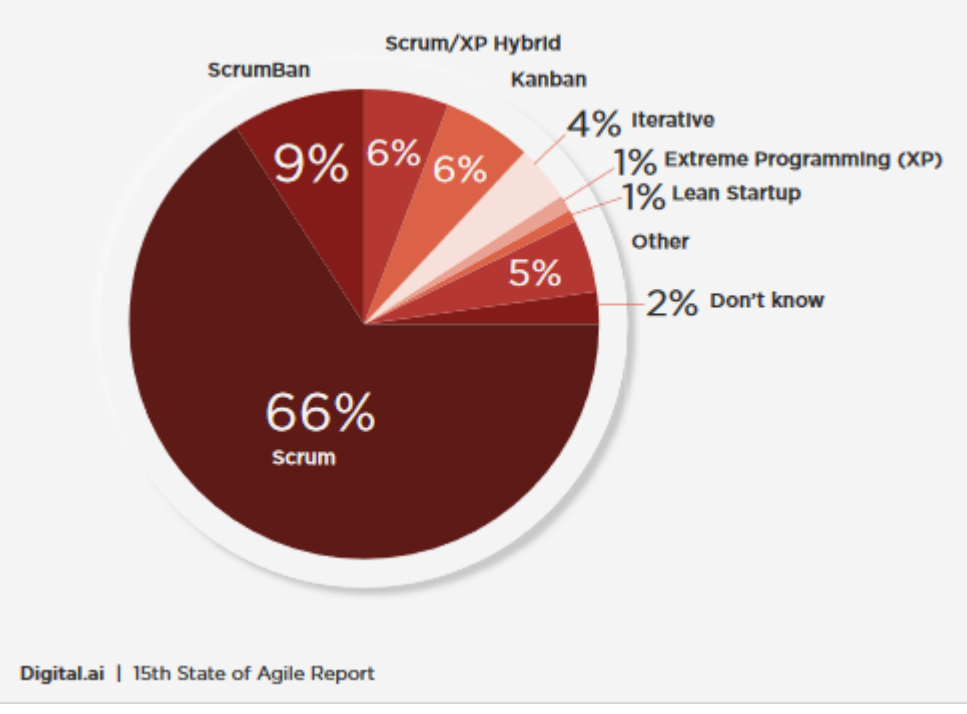
Exercise No 4: (2 Hours) – 1 Practical

Aim: - Comparative study of open source agile tools.

THEORY:

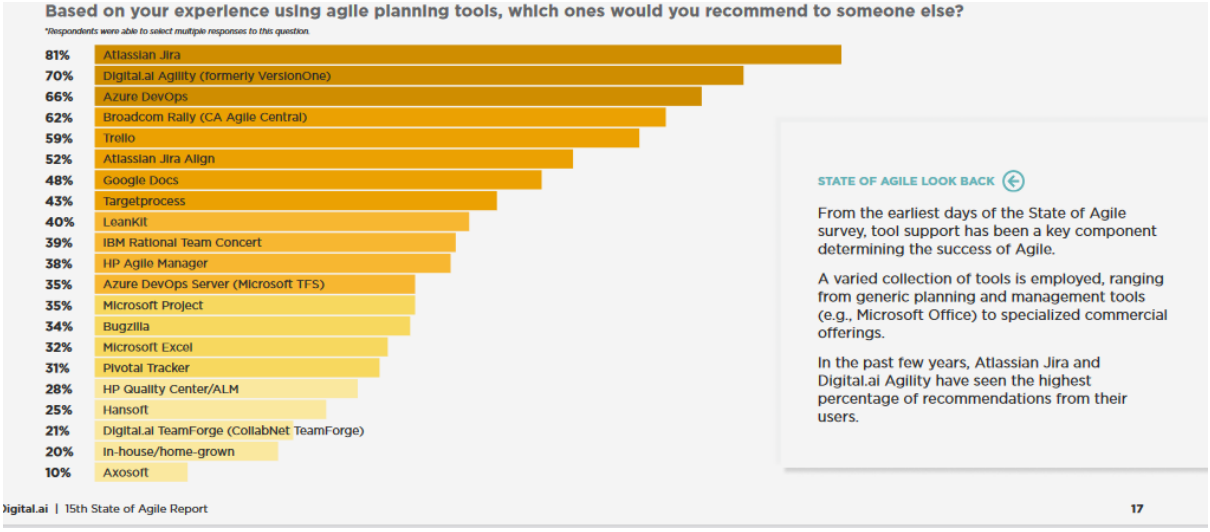
Tools are of paramount importance in automating software engineering tasks; although the Agile Manifesto prefers "*individuals and their interactions over processes and tools*", some agile development activities make no exception and can be automated effectively and successfully. In process frameworks like Scrum or similar ones some activities are in fact quite structured and need specific tool support. Hence, it is interesting to study the combination of specific agile practices with OSS tools.

Following figure shows the current rate of Agile methodologies used. Scrum is aimed at providing an agile approach for managing software projects while increasing the probability of successful development of software, whereas XP focuses more on the project level activities of implementing software.



Agile Tooling:

Agile practitioners rely on a wide variety of tools to support Agile techniques and practices. However, the survey indicated that two tools, Atlassian Jira (81%) and Digital.ai Agility (70%), were the most highly recommended to colleagues by their users.



open source project management tools :

Agile is a framework of behaviors and approaches that encourage “just in time” production that enables project teams to complete projects on time, with higher quality.

Agile encourages that a little of everything, including design, development, and testing. Portions of the project are done at the same time—as opposed to the traditional approach to projects, where one phase is closed and completed before the next begins. Agile encourages short, frequent feedback loops and embraces changes to requirements. In traditional project management (sometimes known as Waterfall), feedback is usually not collected until the very end of the project and changes are discouraged.

Agile Project Management has been around for decades, yet thousands of companies and managers have difficulties when transitioning from traditional to Agile when managing projects. Following is a list of some free Agile tools for project managers to help them improve decision making time.

1. Icescrum

Created by a French company called Kagilum (KAnban-aGILe-scrUM) SAS in 2011, Icescrum is a free and open source solution for teams of any size. This tool is compatible with Linux, Windows, and Mac—and integrates with a variety of popular apps like Box, Excel, Dropbox, GitHub, Google Drive, Slack, and more.

Cost: free download. The cloud version starts at $35 USD per month

2. Taiga

Awarded Best Agile Tool 2015 by the Agile awards, and Top 11 PM Tools 2016 by Opensource.com, Taiga offers better functionality than many paid tools, including backlogs, sprints, Kanban boards, and QA, along with the ability to import from Trello, Jira, Asana, and Github.

Cost: Public projects are completely free; private projects start at $19 a month.

3. Hansoft

Hansoft's fast and intuitive interface, powerful find/report tools, and a host of other features can run different project management methods depending on your project. Hansoft is scalable and configurable since it was inspired by the Project Management Institute (PMI), Scaled Agile Framework (SAFe), and Large Scale SCRUM (LeSS). Its core feature critical for planning and tracking is the backlog, which can be refined by any custom attribute to your deliverables.

Cost: Completely free for up to 2 users. It allows unlimited projects and programs.

4. ScrumDesk

ScrumDesk is a project management tool. ScrumDesk is an online application for Agile product management and Scrum project management for teams utilizing Scrum or Kanban. With ScrumDesk, we hope to give Agile teams the ability to oversee product development from conception to completion.

Small or medium-sized teams that are working on one or more projects concurrently are the main emphasis of ScrumDesk. The tool should offer excellent transparency for the team's daily work and the backlog. ScrumDesk encompasses all aspects of a business. It assists product owners in quickly understanding the backlog and determining priorities.

5. KanbanTool

The Kanban Tool is a program for using the Kanban methodology to manage tasks and projects. The Kanban technique is based on two principles: a graphic, clear picture of the workflow and a limitation on the number of functions that can be carried out concurrently. Kanban Tool is the original program for setting up your Kanban board and tracking your work process step by step. As a result, clients and team members can effortlessly communicate in real time by exchanging tasks, information, and comments at any time and location.

6. VersionOne Lifecycle

VersionOne is a cloud-based Agile application lifecycle management (ALM) tool that aids enterprises in involving stakeholders, tracking progress, and reporting on diverse software portfolios, programs, and projects. It offers features of Agile development tools connected to Scrum and Kanban methodologies, such as project boards with targets, problems, and faults, spring plans, project road mapping, release planning, and test management.

7. Kanbanize

Agile project management software called Kanbanize creates a virtual workspace by merging business automation and Kanban-style features. You may use this Agile solution to manage your software development projects, programs, tasks, and portfolios because it was designed with scalability.

8. Yodiz

Yodiz is an easy-to-use yet complete Agile tool that aids in managing Agile projects of all levels of difficulty.

The best and most effective approaches to complete story-related Agile tasks are outlined by Yodiz, which simplifies the management of Agile projects.

By using well-planned and practical approaches to provide a great user experience, it lessens the difficulty of handling project and team-related data

How to Choose Which Tool Is Right for You?

Choosing the appropriate Agile tool to match your business needs and methodology is crucial. Your decision-making process for a solution that supports your team and their Agile methodology can be aided by the five features listed below:

- Encourage communication and collaboration
- Accountability and History Tracking
- Central Searchable Storage
- Scaling Ability
- Analytical Services

CONCLUSIONS:

Open source project management tools plays important role while learning the Agile methodology and DevOps. Students will be familiar with the working mechanisms of these tools. Students will able to compare the features of Agile tools and select the appropriate for the project.

5. Lab Exercise

Exercise No 4: (2 Hours) – 1 Practical

1. **Aim: Installation and use of open source agile tools for software development**
   Case Study: Integration of ZenHub with GitHub

Objective:

1. Create a free GitHub account.
2. Create a GitHub repository.
3. Create a free ZenHub account.
4. Install the optional browser extension for ZenHub.

THEORY:

Zenhub is an agile project management and product roadmaps solution, natively integrated into GitHub. With Zenhub installed, your team stays lean and agile: you can plan sprints, create epics, and visualize your workflow without leaving GitHub. Zenhub is available either inside GitHub via our browser extension or as a full featured standalone web app. Zenhub allows you to keep project tracking and productivity reporting where it belongs —close to code. ZenHub was founded in 2014.

GitHub, Inc., is an Internet hosting service for software development and version control using Git. It provides the distributed version control of Git plus access control, bug tracking, software feature requests, task management, continuous integration, and wikis for every project. At a high level, GitHub is a website and cloud-based service that helps developers store and manage their code, as well as track and control changes to their code.

Lab 1: Get set up in ZenHub

Execrcise 1 : Create a free GitHub account

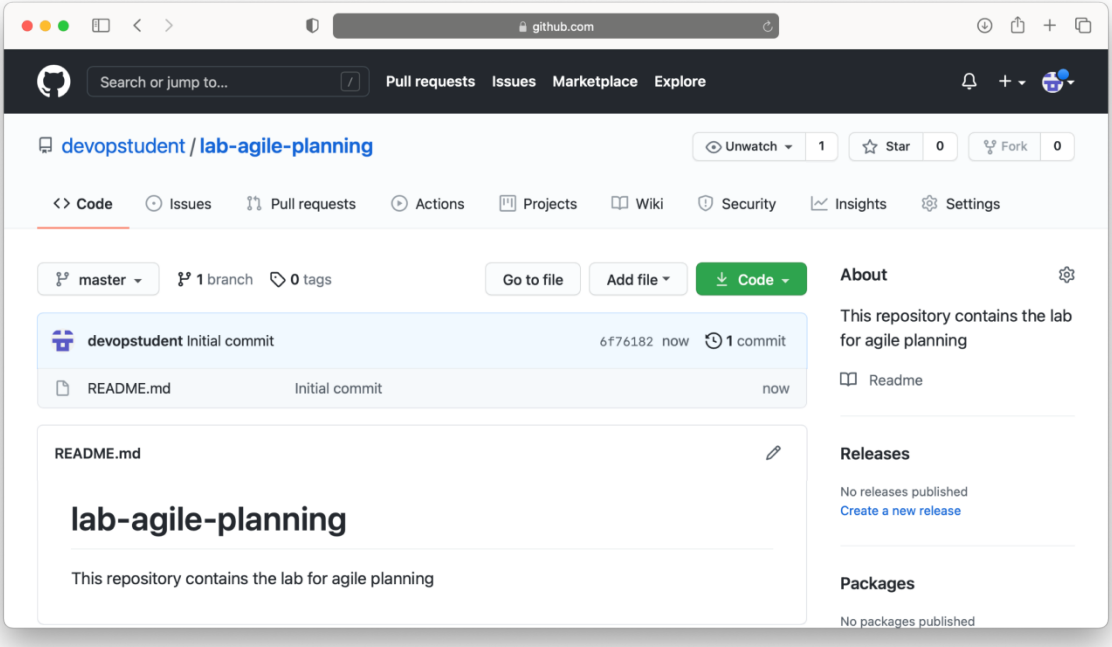
In this exercise, you will create a free GitHub account, if you don't already have one. ZenHub requires GitHub so this first step is a prerequisite to getting a ZenHub account.

1. Go to GitHub.com and sign up for a free GitHub account.
2. Enter your email address and press the **Sign up for GitHub** button. If you have an account, press the **Sign in** button and login.

Exercise 2 : Create a GitHub repository

In this exercise, you will create a repository to hold the issues and kanban board for your plan.

1. If you just created your GitHub account, create your first repository by pressing the **Create repository** button, then go to step **3**.
2. If you already have a GitHub account, sign into GitHub and on your account page, press the **New** repository button.
3. Name your repository **lab-agile-planning** and give it a good description like: *This repository contains the lab for agile planning* and make sure the **Public** option is selected.
4. Scroll down on that page and select **Add a README file** and then press the **Create repository** button to create the new repository.
5. You should now have a new repository called **lab-agile-planning** that we will use with ZenHub. It should look similar to the screenshot below:



Exercise 3 : Create free ZenHub account

In this exercise, you will create a free ZenHub account to use with GitHub. ZenHub is an application that integrates with GitHub. In order to use it, you must sign up for a free account.

1. Go to www.zenhub.com and press the Try for free button. Even though it says "*Start your free 14-day trial with ZenHub*" you can continue to use ZenHub for free on open source projects.
2. On the next page, you need to enter your Email ID and a strong Password to create a new ZenHub account.

Note: Alternate way, press the **Sign up with Google** button if you have a google account already and want to use that on ZenHub and then login using your Google/Gmail credentials.

3. Enter whatever you want for this ZenHub survey and press **Submit**
4. On the next page, enter your organization name.
5. Next, you will be asked to create a workspace. Assign the name "**Development**" to your new ZenHub workspace.
6. Go to your kanban Board, and click on **Connect your GitHub account** button.
7. Since you are already signed into GitHub from the previous steps, you should be presented with a page that will allow you to authorize ZenHubIO to access your GitHub account.
8. (optional) If you are not signed into GitHub, you will be prompted to sign in to GitHub. That will bring you to a page where you must use your GitHub credentials.
9. If you have two-factor authentication enabled on your GitHub account, you must enter your authorization code now.
10. Accept ZenHub's privacy policy
11. On your kanban board, click on **Add repositories** to add add repositories to Development workspace.
12. On the next page, click on **Add repos** under connect repositories.
13. Select **lab-agile-planning** repository that you created in the previous steps, then click on **Add**.
14. This will place you in your kanban board for the `lab-agile-planning` repository.

Exercise 4 : (Optional) Browser extension for Chrome or Firefox

In this optional exercise, you will download a browser extension for ZenHub.

If you use Chrome or Firefox, you can download a browser extension for ZenHub that will add a ZenHub tab while you are using GitHub so that you don't have to go to zenhub.com to view your kanban board.

1. Download a browser extension for ZenHub here: www.zenhub.com/extension. Once installed, it will add a ZenHub tab while you are using GitHub. This is purely a convenience. The capabilities of both the browser extension and the web zenhub.com site are the same.

Lab 2: Create an issue template in GitHub

Exercise 1 : Create an issue template in GitHub

In this exercise, you will create an issue template in GitHub. This only needs to be done once for each new repository that you create.

1. Go to GitHub.com and select the `lab-agile-planning` repository that you created in Lab 1.
2. From the repository page, select **Settings**.
3. Scroll down to the **Features** section and select **Set up templates.**
4. From the dropdown list labeled **Add template** (1), select **Custom template** (2).
5. Next to the **Custom issue template** entry, press the **Preview and edit** button.
6. Press the **pencil** icon to edit the template.
7. Copy the following markdown for the story template content:

````
**As a** [role]
**I need** [function]
**So that** [benefit]

### Details and Assumptions
* [document what you know]

### Acceptance Criteria

  ```gherkin
  Given [some context]
  When [certain action is taken]
  Then [the outcome of action is observed]
````
````

```

8. Edit the **Template name** to be: `User Story`, give it an appropriate **description**, and paste the contents of the above markdown into the **Template content**
9. Scroll to the top of the page and press the **Propose changes** button.
10. Press the **Commit changes** button to commit the change to your repository.
11. You should now have a new folder in your repository called `.github/ISSUE_TEMPLATES`, which will contain your new user story template.
12. You now have an issue template that you can use for all of your GitHub repositories that you need to write stories for to use in ZenHub. When we create issues in future labs, this template will guide you through what information is needed to create your user story.

### Lab 3: Assemble your Product Backlog

# Exercise 1 : Create new user stories using GitHub issues

In this exercise, you will create the following user stories using ZenHub:

**Stories from the Lesson:**

Title: Need a service that has a counter

- As a User, I need a service that has a counter, So that I can keep track of how many times something was done.

Title: Must allow multiple counters

- As a User, I need to have multiple counters, So that I can keep track of several counts at once.

Title: Must persist counter across restarts

- As a Service Provider, I need the service to persist the last known count, So that users don't lose track of their counts after the service is restarted.

Title: Counters can be reset
```

- As a System Administrator, I need the ability to reset the counter, So that I can redo counting from the start.

New Requirements:

- Deploy service to the cloud.
- Need the ability to remove a counter.
- Need the ability to update a counter to a new value.

You will use ZenHub to enter these stories as issues in GitHub using the template that you created in Lab 2.

Goto app.zenhub.com and sign in with your GitHub account.

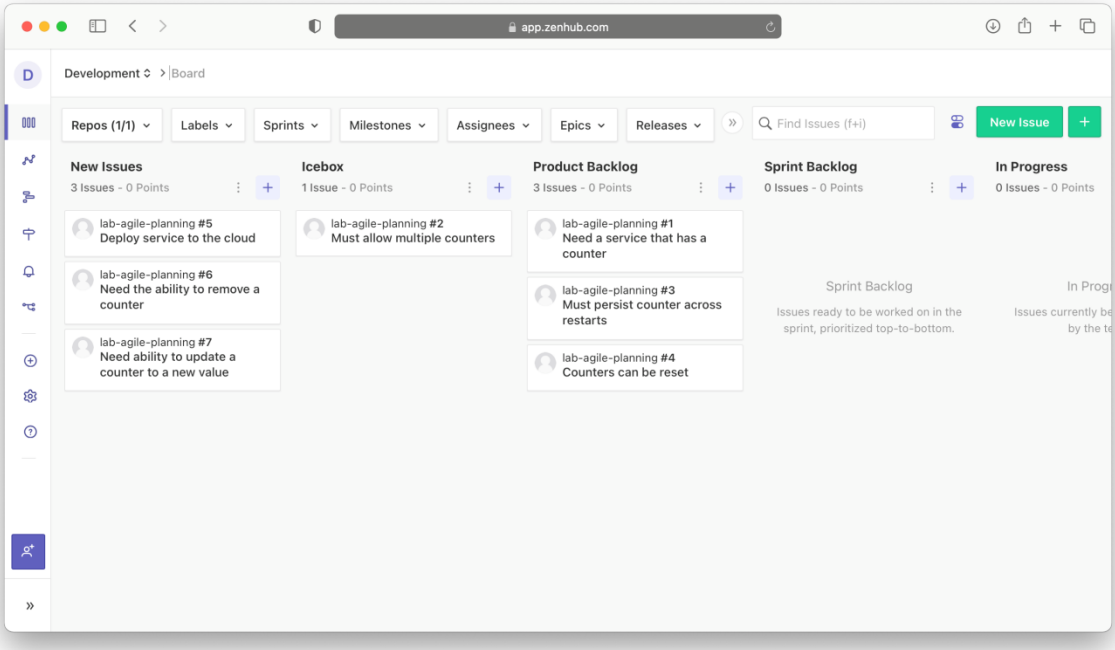
2. From your kanban board view, select **New Issue**.
3. From the **Template** dropdown, select **User Story**.
4. Enter the title for the first story **Need a service that has a counter** and fill out just the user story section (i.e., As a, I need, So that) for now
5. Scroll to the bottom of the page and press **Submit new Issue**.
6. Press the **X** icon in the upper right corner to close the new issue. Note that it now shows up in your **New Issues** pipeline.
7. Continue adding stories until all seven stories are created and your kanban board looks like this. Note that the first four stories are given to you from the lesson. You will need to create your own role, function, and benefit for the last three stories.

Exercise 2 : Prioritize the product backlog

In this exercise, you will move issues between pipelines to recreate the kanban board from the video lesson **Building the Product Backlog**. This will simulate an initial starting point for our next lab on backlog refinement. Please note that you can move the issues between pipelines by simply dragging and dropping them from one pipeline to teh other.

1. Move the **Need a service that has a counter** story to the top of the **Product Backlog** pipeline.
2. Move the **Must allow multiple counters** story to the **Icebox** pipeline.
3. Move the **Must persist counter across restarts** story to the bottom of the **Product Backlog** pipeline.
4. Move the **Counters can be reset** story to the bottom of the **Product Backlog** pipeline.
5. Leave the remaining stories in the **New Issues** pipeline for now. We will move them in a later lab.

At the completion of this exercise, your kanban board should look like this:



Lab 4: Refine your Product Backlog

In this lab, you will follow the steps of conducting a backlog refinement meeting. You will be the product owner getting the product backlog ready for your next sprint planning meeting. This involves grooming the stories we created in the last lab to make them *sprint ready*.

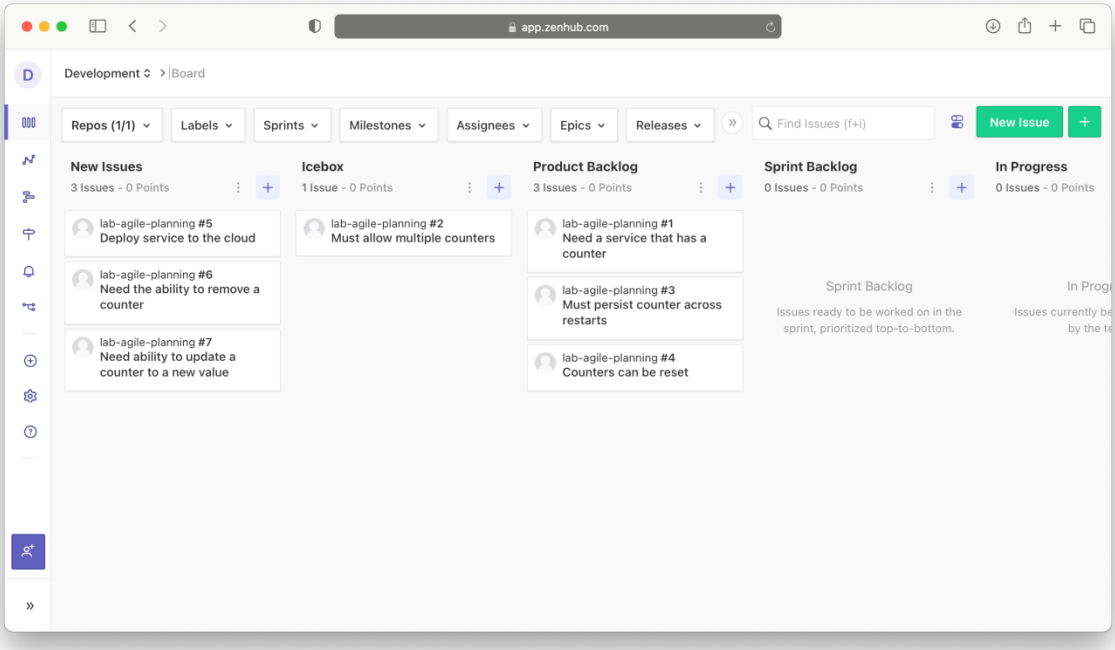
Objectives

After completing this lab, you will be able to:

1. Triage new issues.
2. Make stories sprint ready.
3. Create new labels in GitHub.
4. Add labels to stories.

Initial State

At the completion of Lab 3, your kanban board should look like this:



New Issues:

- Deploy service to the cloud
- Need the ability to remove a counter
- Need ability to update a counter to a new value

Icebox:

- Must allow multiple counters

Product Backlog:

- Need a service that has a counter
- Must persist counter across restarts
- Counters can be reset

Exercise 1 : Triage new issues

In this exercise, you will take all of the stories in the **New Issues** pipeline and move them to an appropriate pipeline or reject them.

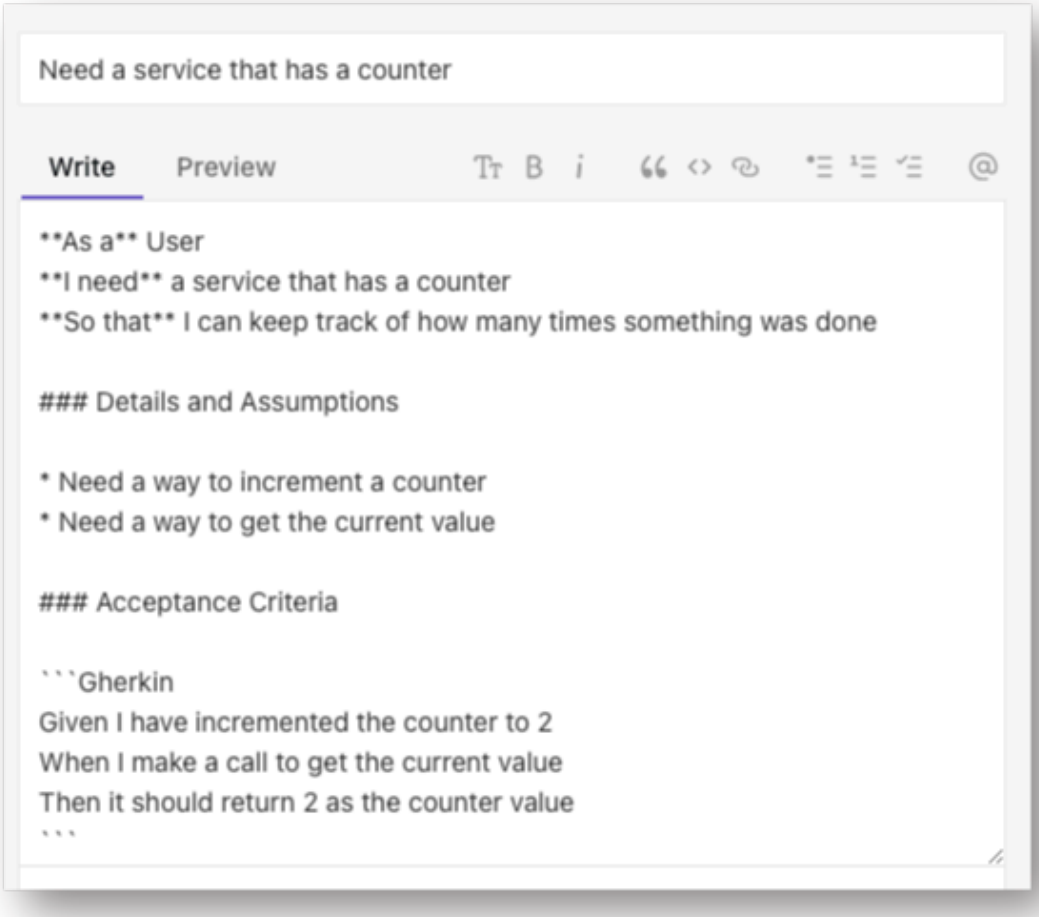
Go to app.zenhub.com and sign in with your GitHub account and bring up your kanban board.

1. The first new issue is **Deploy service to the cloud**. We want to do that after adding persistence, so move that to the **Product Backlog** pipeline under **Must persist counter across restarts**.
2. The next new issue is **Need the ability to remove a counter**. We only have one counter and we wouldn't want to remove that, so let's move that to the **Icebox** after **Must allow multiple counters**.
3. The last new issue is **Need ability to update a counter to a new value**. We might want to do that as an enhancement after we can reset the counter, so let's move that to the **Product Backlog** after **Counters can be reset**
4. have now completed new issue triage and can start making the stories in the **Product Backlog** pipeline sprint ready.

Exercise 2 : Make stories sprint ready

In this exercise, you will add more details to the stories in the **Product Backlog** that you feel might make it into the next sprint. You will be provided with the details for two of the stories. You must provide the details for the other three.

1. Select the first story at the top of the **Product Backlog** pipeline to open it. Then press the **Edit** button to edit the issue
2. Edit the **Details and Assumptions** to let developers know what we know, and edit the **Acceptance Criteria** to make sure that everyone understands what the definition of "*done*" is. Make your story look like this one:



3. When you are finished editing, press the **Update** button to save the edits
4. Close the window by pressing the **X** icon
5. Edit the **Must persist counter across restarts** story in the same way
6. Edit the **Deploy service to the cloud** story and make up your own **Details and Assumptions**, and **Acceptance Criteria**.
7. Edit the **Counters can be reset** story and make up your own details, assumptions, and acceptance criteria.
8. Edit the **Need ability to update a counter to a new value** story and make up your own details, assumptions, and acceptance criteria.

At the completion of this exercise, your kanban board should have sufficient details in all of the stories in the **Product Backlog** to make them sprint ready.

Exercise 3 : Create new labels in GitHub

In this exercise, you will create a new label in GitHub called `technical debt` to flag those stories that provide no visible customer value but must be completed in order to continue development.

1. Go to github.com and select your **lab-agile-planning** repository
1. From your repository page, select the **Issues** tab
2. From the issues page, select the **Labels** button
    2. From the labels page, select the **New label** button.
    3. In the new label section: (1) set the **Label name** to `technical debt`, (2) set the **Color** to `yellow` (`#FBCA04`), then (3) press the **Create label** button.
    4. You should now see a yellow `technical debt` label that we can use to annotate our stories

Exercise 4 : Add labels to stories

In this exercise, you will go back to ZenHub and add labels to the stories in the **Product Backlog** to further make them sprint ready. You will also use our new label called technical debt to flag those stories that provide no visible customer value but must be completed to continue development.

1. Go to ZenHub and select the first story at the top of the **Product Backlog** pipeline to open it. Then press the **Gear** icon next to **Labels** to assign a label.
2. Our first story is an enhancement to our product. From the label menu, select **enhancement** to reflect this
3. Click anywhere off of the labels menu to close it. You should now see that the label **enhancement** has been assigned to this story.
4. Select each of the following stories in the **Product Backlog** pipeline and assign them the corresponding labels.

Story Title	Label
Must persist counter across restarts	enhancement
Deploy service to the cloud	technical debt
Counters can be reset	enhancement
Need ability to update a counter to new value	enhancement

Lab 5: Build the Sprint Plan from your Product Backlog

In this lab, you will create a sprint plan from your product backlog. This is normally done during the sprint planning meeting with the entire team, so for this exercise we will have to simulate that meeting.

Objectives

After completing this lab, you will be able to:

1. Set up sprints.
2. Assign estimates to stories.
3. Assign a sprint to stories.
4. Build your sprint backlog.

New Issues:

- None

Icebox:

- Must allow multiple counters
- Need the ability to remove a counter

Product Backlog:

- Need a service that has a counter
- Must persist counter across restarts
- Deploy service to the cloud
- Counters can be reset
- Need ability to update a counter to a new value

Exercise 1 : Setup Sprints

In this exercise, you will setup your sprints. ZenHub will set up three (3) sprints by default to get you started. It wil then create new sprints for you as needed automatically.

1. Go to app.zenhub.com and sign in with your GitHub account and bring up your kanban board.
2. Click the **+** icon and select **Set up Sprints for your team** from the dropdown menu.
3. In the window that pops up, disable the switch labeled "Move unfinished Issues to the next sprint". This type of automation breeds bad habits and is not very agile, which is why I

suggest it be disabled. You should understand why Issues are unfinished and determine if they are still needed in the next sprint as priorities may have changed

1. Pick a date range of 2 weeks for the sprint duration, where the current date that you are working on this lab is within that range. In this example we selected a Monday to the following Friday which is two weeks later. You will see this selection in blue. Notice that ZenHub hints that it will create two future sprints as well (in grey). Press the **Create sprints** button to create the sprints.

You now have a sprint that can be used for sprint planning in the next exercise.

Exercise 2 : Create a Sprint Plan

In this exercise, you will create a sprint plan. We will assign estimated story points and a sprint, and move the stories from the **Product Backlog** into the **Sprint Backlog** to build our plan.

1. Select the top story *Need a service that has a counter* from the **Product Backlog** to open it.
2. You discussed the story with the team, and your developers agree that this is a large story worth **8** story points, so set the **Estimate** to **8**.
3. Click on the **Gear** icon next to **Sprints** and select the first sprint from the dropdown list to assign the story to that sprint. You may notice that you can assign a story to more than one sprint. This is not very agile! If your story is bigger than a sprint then it's too big and should be broken down into smaller "sprint-sized" stories
4. Your story should look like the one below. Click the **X** to close the story window and get back to the kanban board
5. The development team has determined that adding an 8 point story to the sprint is acceptable. Drag the story from the **Product Backlog** to the **Sprint Backlog**.
6. Your kanban board should now look like the one below. Notice that the story point estimate and sprint dates are annotations on the story.
7. The sprint planning meeting has proceeded nicely. In discussions with the development team, they have estimated the next two stories in the **Product Backlog** and determined that they can both fit in the current sprint. Select each of the following stories in the **Product Backlog**, assign them the corresponding story points and the same *Sprint*, and drag them to the **Sprint Backlog** in the same order.

Story Title	Story Points
Must persist counter across restarts	5
Deploy service to the cloud	5

8. At the end of this step, your sprint plan should look like this:
9.
10. Based on the teams velocity, the development team has decided there are enough stories in the sprint, but there is some time left in the sprint planning meeting to estimate more stories. Add the following estimates to the stories in the **Product Backlog**.

Story Title	Story Points
Counters can be reset	3
Need ability to update a counter to new value	5

Lab 6: Move stories from In Progress to Done

In this lab, you will follow the daily workflow of moving stories from the sprint backlog to the In Progress pipeline, assigning them to yourself to work on them, moving them to Review/QA, and moving them to the Done pipeline.

Objectives

After completing this lab, you will be able to:

1. Assign stories to yourself.
2. Move stories to **In Progress** to work on them.
3. Move stories to **Review/QA** for team review.
4. Move stories to **Done** once they are completed.

New Issues:

- None

Icebox:

- Must allow multiple counters
- Need the ability to remove a counter

Product Backlog:

- Counters can be reset
- Need ability to update a counter to a new value

Sprint Backlog:

- Need a service that has a counter
- Must persist counter across restarts
- Deploy service to the cloud

Exercise 1 : Daily Workflow

In this exercise, you will simulate the daily workflow of a developer on an Agile team. You will start by moving a story from the top of the **Sprint Backlog** to the **In Progress** pipeline and assign it to yourself. Then you will simulate completing the story, asking for a review, and finally moving it to done.

1. Go to app.zenhub.com and sign in with your GitHub account and bring up your kanban board.
2. The sprint has started and you are ready to work on your first story. Select the story at the top of the **Sprint Backlog** to open it and read it.
3. After reading it, you decide that this is something you have the skills to work on, so you assign it to yourself.
4. Once assigned, press the **X** to close the window
5. Back at the kanban board, move the story you just assigned to yourself from the **Sprint Backlog** by dragging it to the **In Progress** pipeline.
6. Everyone now knows that you are working on this story. You would normally create a branch in GitHub to start working on your code. For this lab, just make sure that your kanban board looks like the one below. (*Note: Your avatar may look different.*)
7. Once you finish working on the story, it's time to request a review. It's always a good idea to get two sets of eyes on all work products. If you checked code into GitHub, this is the step where you would make a pull request to merge your code into the `master` branch. Move your story from **In Progress** to **Review/QA**
8. While you are waiting for a review, you decide to start working on another story. Take the next story off of the top of the **Sprint Backlog**, read it to make sure that you have the skills to implement it, (*Hint: You do.*) assign it to yourself, and move it to **In Progress**
9. Your pull request on your initial story has been approved and the review process is complete. Move the story "*Need a service that has a counter*" from the **Review/QA** pipeline to the **Done** pipeline.
10. You have completed work on your second story and made another pull request. Move the story "*Must persist counter across restarts*" from **In Progress** to **Review/QA** to request a review
11. Take the last story, "*Deploy service to the cloud*", off of the top of the **Sprint Backlog**, assign it to yourself, and move it to **In Progress**.
12. The review of your second story is complete. Move the story "*Must persist counter across restarts*" from **Review/QA** to **Done**
13. The sprint has ended and we have run out of time to complete our last story, "*Deploy service to the cloud*", which is still in progress. We will see how to deal with this in a future lab. Leave it where it is for now.

Lab 7: Setup a Burndown Chart for Your Plan

In this lab, you will set up a burndown chart for your plan. In most cases, the burndown chart is created automatically by ZenHub if you have set the start and end dates of your sprint correctly. There is still some setup to do to customize it to show stories that are done, but not closed. These changes are not permanent and will need to be made whenever you view your burndown charts.

Objectives

After completing this lab, you will be able to:

1. Understand how to set up your burndown chart.
2. See the data that is contained in a burndown chart.
3. Judge your progress in the sprint.

Exercise 1 : Set up your burndown chart

In this exercise, you will set up your burndown chart to show the status of the **Done** pipeline stories instead of the default **Closed** stories.

Since a burndown chart shows the story points completed vs. the time left in a sprint, it will be difficult to simulate in a one-day lab. Hopefully, you created a sprint that included the date that you performed the previous lab on the topic of daily execution. For best results, it is recommended that you wait a few days after that lab before performing this lab so that the passage of time will be reflected on the burndown chart.

1. Go to app.zenhub.com and sign in with your GitHub account. Bring up your kanban board and click the **>>** icon in the lower left-hand corner to open the side menu bar.
2. From the side menu bar, select **Reports**
3. From the reports menu, select **Burndown Report**
4. Make sure the current sprint is selected. You can change the sprint using the dropdown list. Since we only have one sprint, the current sprint should already be selected

5. Notice that the burndown chart isn't burning down. We just have a straight line across the top. This is because, by default, the burndown measures closed stories. Since we want to keep our stories in the **Done** pipeline before the sprint review, select the **Burn Pipelines** dropdown to change this behavior
6. From the **Burn Pipelines** drop down, select **Done** to show the status of the stories that are done
7. You can now see the status of the sprint in the burndown chart. Since you completed your stories in a single lab, there is one big drop in the progress. Normally this would be more gradual. Notice that we are well below the dotted line, which is a projection of where we should be at this time.

CONCLUSIONS:

Students will able to perform different Scrum operations using ZenHub and GitHub.

6. Lab Exercise

Exercise No 4: (2 Hours) – 1 Practical

1. Aim: - Implementation of SOLID principles using any programming language

.**THEORY:**

Objective:

1. Understanding different design principles
2. Understanding refactoring Theory

What is software architecture? The answer is multitiered. At the highest level, there are the architecture patterns that define the overall shape and structure of software applications1 . Down a level is the architecture that is specifically related to the purpose of the software application. Yet another level down resides the architecture of the modules and their interconnections. This is the domain of design patterns2 , packakges, components, and classes. It is this level that we will concern ourselves with in this chapter. Our scope in this chapter is quite limitted. There is much more to be said about the principles and patterns that are exposed here. Architecture and Dependencies What goes wrong with software? The design of many software applications begins as a vital image in the minds of its designers. At this stage it is clean, elegant, and compelling. It has a simple beauty that makes the designers and implementers itch to see it working. Some of these applications manage to maintain this purity of design through the initial development and into the first release. But then something begins to happen. The software starts to rot. At first it isn't so bad. An ugly wart here, a clumsy hack there, but the beauty of the design still shows through. Yet, over time as the rotting continues, the ugly festering sores and boils accumulate until they dominate the design of the application. The program becomes a festering mass of code that the developers find increasingly hard to maintain. Eventually the sheer effort required to make even the simplest of changes to the application becomes so high that the engineers and front line managers cry for a redesign project. Such redesigns rarely succeed. Though the designers start out with good intentions, they find that they are shooting at a moving target. The old system continues to evolve and change, and the new design must keep up. The warts and ulcers accumulate in the new design before it ever makes it to its first release. On that fateful day, usually much

later than planned, the morass of problems in the new design may be so bad that the designers are already crying for another redesign.

Symptoms of Rotting Design:

There are four primary symptoms that tell us that our designs are rotting. They are not orthogonal, but are related to each other in ways that will become obvious. they are: rigidity, fragility, immobility, and viscosity.

Rigidity:

Rigidity is the tendency for software to be difficult to change, even in simple ways. Every change causes a cascade of subsequent changes in dependent modules. What begins as a simple two day change to one module grows into a multiweek marathon of change in module after module as the engineers chase the thread of the change through the application. When software behaves this way, managers fear to allow engineers to fix non-critical problems. This reluctance derives from the fact that they don't know, with any reliability, when the engineers will be finished. If the managers turn the engineers loose on such problems, they may disappear for long periods of time. The software design begins to take on some characteristics of a roach motel -- engineers check in, but they don't check out. When the manager's fears become so acute that they refuse to allow changes to software, official rigidity sets in. Thus, what starts as a design deficiency, winds up being adverse management policy.

Fragility:

Closely related to rigidity is fragility. Fragility is the tendency of the software to break in many places every time it is changed. Often the breakage occurs in areas that have no conceptual relationship with the area that was changed. Such errors fill the hearts of managers with foreboding. Every time they authorize a fix, they fear that the software will break in some unexpected way. As the fragility becomes worse, the probability of breakage increases with time, asymptotically approaching 1. Such software is impossible to maintain. Every fix makes it worse, introducing more problems than are solved.Such software causes managers and customers to suspect that the developers have lost control of their software. Distrust reigns, and credibility is lost.

Immobility:

Immobility is the inability to reuse software from other projects or from parts of the same project. It often happens that one engineer will discover that he needs a module that is similar to one that another engineer wrote. However, it also often happens that the module in question has too much baggage that it depends upon. After much work, the engineers discover that the work and risk required to separate the desirable parts of the software from the undesirable parts are too great to tolerate. And so the software is simply rewritten instead of reused.

Viscosity:

Viscosity comes in two forms: viscosity of the design, and viscosity of the environment. When faced with a change, engineers usually find more than one way to make the change. Some of the ways preserve the design, others do not (i.e. they are hacks.) When the design preserving methods are harder to employ than the hacks, then the viscosity of the design is high. It is easy to do the wrong thing, but hard to do the right thing. Viscosity of environment comes about when the development environment is slow and inefficient. For example, if compile times are very long, engineers will be tempted to make changes that don't force large recompiles, even though those changes are not optiimal from a design point of view. If the source code control system requires hours to check in just a few files, then engineers will be tempted to make changes that require as few check-ins as possible, regardless of whether the design is preserved. These four symptoms are the tell-tale signs of poor architecture. Any application that exhibits them is suffering from a design that is rotting from the inside out. But what causes that rot to take place?

S.O.L.I.D. Principles

When you start asking the question of how, it's a little like looking at a marathon race and wondering how you end up at the finish line. Obviously, for a marathon you arrive at the finish line by running one step at a time. Software development lets you move one step at a time toward your object-oriented goals, as well. The steps are composed of additional principles and implementation goals, such as those outlined in the SOLID acronym:

- S: Single Responsibility Principle (SRP)
- O: Open-Closed Principle (OCP)
- L: Liskov Substitution Principle (LSP)
- I: Interface Segregation Principle (ISP)

- D: Dependency Inversion Principle (DIP)

Originally compiled by Robert C. Martin in the 1990s, these principles provide a clear pathway for moving from tightly coupled code with poor cohesion and little encapsulation to the desired results of loosely coupled code, operating very cohesively and encapsulating the real needs of the business appropriately. The Open-Closed Principle indicates how a system can be extended by modifying the behavior of individual classes or modules, without having to modify the class or module itself. This helps you create well-encapsulated, highly cohesive systems. The Liskov Substitution Principle also helps with encapsulation and cohesion. This principle says that you should not violate the intent or semantics of the abstraction that you are inheriting from or implementing. The Interface Segregation Principle helps to make your system easy to understand and use. It says that you should not force a client to depend on an interface (API) that the client does not need. This helps you develop well-encapsulated, cohesive set of parts. The Dependency Inversion Principle helps you to understand how to correctly bind your system together. It tells you to have your implementation detail depend on the higher-level policy abstractions, and not the other way around. This helps you to move toward a system that is coupled correctly, and directly influences that system's encapsulation and cohesion. The Single Responsibility Principle says that classes, modules, etc., should have one and only one reason to change. This helps to drive cohesion into a system and can be used as a measure of coupling as well.

Refactoring

Refactoring is the process of changing a computer program's source code without modifying its external functional behavior in order to improve some of the nonfunctional attributes of the software. It is a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behaviour. Advantages include improved code readability and reduced complexity to improve the maintainability of the source code, as well as a more expressive internal architecture or object model to improve extensibility. Its heart is a series of small behavior preserving transformations. Each transformation (called a 'refactoring') does little, but a sequence of transformations can produce a significant restructuring. Since each refactoring is small, it's less likely to go wrong. The system is also kept fully working after each small refactoring, reducing the chances that a system can get seriously broken during the restructuring. By continuously improving the design of code, we

make it easier and easier to work with. If you get into the hygienic habit of refactoring continuously, you'll find that it is easier to extend and maintain code.

Why refactor?

The purpose of refactoring is to improve the quality, clarity and maintainability of your code. Simple really. But also, refactoring can be a great lesson in understanding an unfamiliar code base. Think about it, if you inherit a poorly designed code base that you've not seen before and you now need to either fix a bug or add a new feature, then implementing the code necessary would be a lot easier once you had refactored it to be in a more stable, maintainable and ultimately 'understandable' state. Otherwise you would be forced to retro fit your new code on top of a poorly designed foundation and that would be the start of a very unhappy relationship.

When should you refactor?

You'll usually find the time you start refactoring the most is when you are fixing bugs or adding new features. For example, you typically first need to understand the code that has already been written (regardless of whether it was you who wrote it originally or someone else). The process of refactoring helps you better understand the code, in preparation for modifying it. But don't fall into the trap of thinking that refactoring is something you set aside time for, or only consider at the start/end of a project. It's not. Refactoring should be done in small chunks throughout the entire life cycle of the project. As the great Uncle Bob once said: leave a module in a better state than you found it ...what this suggests is that refactoring is essential to your daily coding process.

Tests

Before we get started, it's important to mention that you should have tests in place when you're refactoring. You can refactor without tests, but realise that without tests to back you up then you can have no confidence in the refactoring you are implementing. Refactoring can result in substantial changes to the code and architecture but still leave the top layer API the same. So while you're refactoring remember the old adage... program to an interface, not an implementation We want to avoid changing a public API where ever possible (as that's one of the tenets of refactoring). If you don't have tests then I recommend you write some (now)... don't worry, I'll wait. Remember, the process of writing tests (even for an application you don't know) will help solidify your understanding and expectations of the

code you're about to work on. Code should be tested regularly while refactoring to ensure you don't break anything. Keep the 'red, green, refactor' feedback loop tight. Tests help confirm if your refactoring has worked or not. Without them you're effectively flying blind. So although I won't explicitly mention it below when discussing the different refactoring techniques, it is implied that on every change to your code you should really be running the relevant tests to ensure no broken code appears.

Refactoring Techniques

There are many documented refactoring techniques and I do not attempt to cover them all, as this post would end up becoming a book in itself. So I've picked what I feel are the most common and useful refactoring techniques and I try my best to explain them in a short and concise way. I've put these techniques in order of how you might approach refactoring a piece of code, in a linear, top to bottom order. This is a personal preference and doesn't necessarily represent the best way to refactor. Final note: with some of the techniques I have provided a basic code example, but to be honest some techniques are so simple they do not need any example. The Extract Method is one such technique that although really useful and important, providing a code example would be a waste of time and space.

CONCLUSIONS:

The purpose of design principles and refactoring is to improve the quality, clarity and maintainability of your code.

7. Lab Exercise

Exercise No 4: (2 Hours) – 1 Practical

Aim: - Test Driven Development using XUnit

Objective

1. Understand Test driven Development
2. Understanding Junit framework:

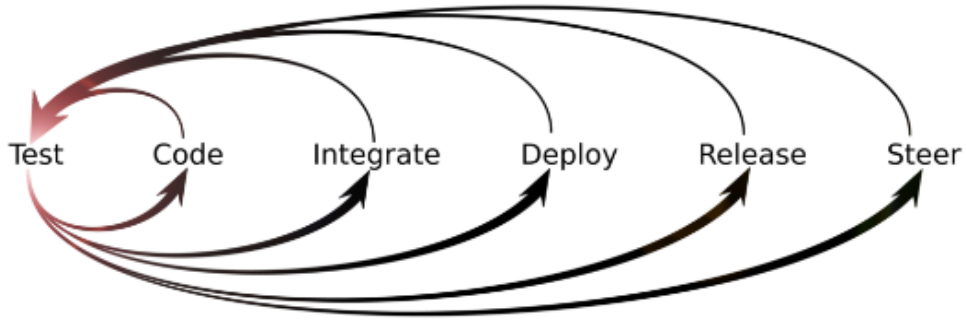
THEORY:

Test-driven development (TDD) is a software development process that relies on the repetition of a very short development cycle: first the developer writes an (initially failing) automated test case that defines a desired improvement or new function, then produces the minimum amount of code to pass that test, and finally refactors the new code to acceptable standards. Kent Beck, who is credited with having developed or 'rediscovered' the technique, stated in 2003 that TDD encourages simple designs and inspires confidence.

Test-driven development is related to the test-first programming concepts of extreme programming, begun in 1999, but more recently has created more general interest in its own right.

Programmers also apply the concept to improving and debugging legacy code developed with older techniques.

Test Driven Development Cycle:



1. Add a Test

In test-driven development, each new feature begins with writing a test. To write a test, the developer must clearly understand the feature's specification and requirements. The developer can accomplish this through use cases and user stories to cover the requirements and exception conditions, and can write the test in whatever testing framework is appropriate to the software environment. It could be a modified version of an existing test. This is a differentiating feature of test-driven development versus writing unit tests after the code is written: it makes the developer focus on the requirements before writing the code, a subtle but important difference.

2.Run all tests and see if the new one fails

This validates that the test harness is working correctly, that the new test does not mistakenly pass without requiring any new code, and that the required feature does not already exist. This step also tests the test itself, in the negative: it rules out the possibility that the new test always passes, and therefore is worthless. The new test should also fail for the expected reason. This step increases the developer's confidence that the unit test is testing the correct constraint, and passes only in intended cases.

3.Write some code

The next step is to write some code that causes the test to pass. The new code written at this stage is not perfect and may, for example, pass the test in an inelegant way.

That is acceptable because it will be improved and honed in Step 5. At this point, the only purpose of the written code is to pass the test; no further (and therefore untested) functionality should be predicted nor 'allowed for' at any stage.

. **4.Run Tests**

If all test cases now pass, the programmer can be confident that the new code meets the test requirements, and does not break or degrade any existing features. If they do not, the new code must be adjusted until they do.

5. Refactor Code

The growing code base must be cleaned up regularly during test-driven development. New code can be moved from where it was convenient for passing a test to where it more logically belongs. Duplication must be removed. Object, class, module, variable and method names should clearly represent their current purpose and use, as extra functionality is added. As features are added, method bodies can get longer and other objects larger. They benefit from being split and their parts carefully named to improve readability and maintainability, which will be increasingly valuable later in the software lifecycle. Inheritance hierarchies may be rearranged to be more logical and helpful, and perhaps to benefit from recognized design patterns. There are specific and general guidelines for refactoring and for creating clean code.By continually re-running the test cases throughout each refactoring phase, the developer can be confident that process is not altering any existing functionality.

The concept of removing duplication is an important aspect of any software design. In this case, however, it also applies to the removal of any duplication between the test code and the production code—for example magic numbers or strings repeated in both to make the test pass in Step 3.

Repeat

Starting with another new test, the cycle is then repeated to push forward the functionality. The size of the steps should always be small, with as few as 1 to 10 edits between each test run. If new code does not rapidly satisfy a new test, or other tests fail unexpectedly, the programmer should undo or revert in preference to excessive debugging. Continuous helps by providing revertible checkpoints. When

using external libraries it is important not to make increments that are so small as to be effectively merely testing the library itself, unless there is some reason to believe that the library is buggy or is not sufficiently feature-complete to serve all the needs of the software under development.

Development Style

There are various aspects to using test-driven development, for example the principles of "keep it simple, stupid" (KISS) and "You aren't gonna need it" (YAGNI). By focusing on writing only the code necessary to pass tests, designs can often be cleaner and clearer than is achieved by other methods. In Test-Driven Development by Example, Kent Beck also suggests the principle "Fake it till you make it".

To achieve some advanced design concept such as a design pattern, tests are written that generate that design. The code may remain simpler than the target pattern, but still pass all required tests. This can be unsettling at first but it allows the developer to focus only on what is important.

Writing the tests first: The tests should be written before the functionality that is to be tested. This has been claimed to have many benefits. It helps ensure that the application is written for testability, as the developers must consider how to test the application from the outset rather than adding it later. It also ensures that tests for every feature get written. Additionally, writing the tests first leads to a deeper and earlier understanding of the product requirements, ensures the effectiveness of the test code, and maintains a continual focus on software quality. When writing feature-first code, there is a tendency by developers and organisations to push the developer on to the next feature, even neglecting testing entirely. The first TDD test might not even compile at first, because the classes and methods it requires may not yet exist. Nevertheless, that first test functions as the beginning of an executable specification.

Each test case fails initially: This ensures that the test really works and can catch an error. Once this is shown, the underlying functionality can be implemented. This has led to the "test-driven development mantra", which is "red/green/refactor," where red means fail and green means pass. Test-driven development constantly repeats the steps of adding test cases that fail, passing them, and refactoring. Receiving the expected test results at each stage reinforces the developer's mental model of the code,

boosts confidence and increases productivity.

Keep the unit small

For TDD, a unit is most commonly defined as a class, or a group of related functions often called a module. Keeping units relatively small is claimed to provide critical benefits, including:

- Reduced debugging effort – When test failures are detected, having smaller units aids in tracking down errors.
- Self-documenting tests – Small test cases are easier to read and to understand.

Advanced practices of test-driven development can lead to Acceptance test-driven development (ATDD) and Specification by example where the criteria specified by the customer are automated into acceptance tests, which then drive the traditional unit test-driven development (UTDD) process. This process ensures the customer has an automated mechanism to decide whether the software meets their requirements. With ATDD, the development team now has a specific target to satisfy – the acceptance tests – which keeps them continuously focused on what the customer really wants from each user story.

Test structure

Effective layout of a test case ensures all required actions are completed, improves the readability of the test case, and smooths the flow of execution. Consistent structure helps in building a self-documenting test case. A commonly applied structure for test cases has (1) setup, (2) execution, (3) validation, and (4) cleanup.

- Setup: Put the Unit Under Test (UUT) or the overall test system in the state needed to run the test.
- Execution: Trigger/drive the UUT to perform the target behavior and capture all output, such as return values and output parameters. This step is usually very simple.
- Validation: Ensure the results of the test are correct. These results may include explicit outputs captured during execution or state changes in the UUT & UAT.
- Cleanup: Restore the UUT or the overall test system to the pre-test state. This restoration permits another test to execute immediately after this one.

Individual best practices

- Separate common set up and teardown logic into test support services utilized by the appropriate test cases.
- Keep each test oracle focused on only the results necessary to validate its test.
- Design time-related tests to allow tolerance for execution in non-real time operating systems. The common practice of allowing a 5-10 percent margin for late execution reduces the potential number of false negatives in test execution.
- Treat your test code with the same respect as your production code. It also must work correctly for both positive and negative cases, last a long time, and be readable and maintainable.
- Get together with your team and review your tests and test practices to share effective techniques and catch bad habits. It may be helpful to review this section during your discussion.

CONCLUSIONS:

Students will able to develop simple program using TDD.