Lab Session 01

OBJECT: **Exploration of UML Class Diagram using StarUML**

THEORY

The Unified Modeling Language (UML) is a visual modeling language that enables system builders to create blueprints that capture their visions in a standard, easy-to-understand way, and provides a mechanism to effectively share and communicate these visions with others. The purpose of the diagrams is to present multiple views of a system; this set of multiple views is called a model. UML model describes what a system is supposed to do. It doesn't tell how to implement the system.

Class Diagram

A class is a category or group of things that have the same attributes and the same behaviors. A rectangle is the icon that represents the class. It's divided into three areas. The uppermost area contains the name, the middle area holds the attributes, and the lowest area holds the operations.

Procedure for Creating Class

In order to create class,

1. Click [Toolbox] -> [Class] -> [Class] button.
2. And click at the position where class will be placed in the [main window].
3. A new class is created on the diagram and class quick dialog is opened. 4. At the quick dialog, enter the class name and press [Enter] key.

Procedure for Adding Attribute

There are three methods to add attribute to class.

- using quick dialog
- using model in the [main window] or the [model explorer]
- using [collection editor] In the case of using quick dialog,

1. Double-click class.
2. Press [Add Attribute] button at the quick dialog, and you can add attribute.

In the case of using model,

1. Select class in the [main window] or in the [model explorer].
2. Right-click the selected class, select [Add] -> [Attribute] popup menu, and you can do.

In the last case,

1. Select [Collection Editor...] popup menu.
2. Or click button in [attributes] property on properties window. 3. At [attribute] tab of the [collection editor], you can add attribute by using button.

Procedure for Adding Operation

There are three methods to add operation to class.

- using quick dialog
- using model in the [main window] or the [model explorer]
- using [collection editor]

In the case of using quick dialog,

1. Double-click class and class quick dialog is shown.
2. Press [Add Operation] button at the quick dialog, and you can add operation.

In the case of using model, select class in the [main window] or in the [model explorer], right-click the selected class, select [Add] -> [Operation] popup menu, and you can do.

In the last case,

1. Select [Collection Editor...] popup menu. 2. At [operations] tab of the [collection editor], you can add operation by using button.

Procedure for Adding Parameter to Operation

In order to add parameter to operation,

1. Select operation in the [model explorer], select [Add] -> [Parameter] popup menu, and new parameter will be added.
2. Or select operation in the [model explorer], select [Collection Editor...] popup menu.
3. Or click button in [Parameters] property on properties window.
4. At the [Parameters] tab of the [collection editor], you can add parameter by using button.

Hospital Management Scenario

Kabeer Ahmed SE-19028

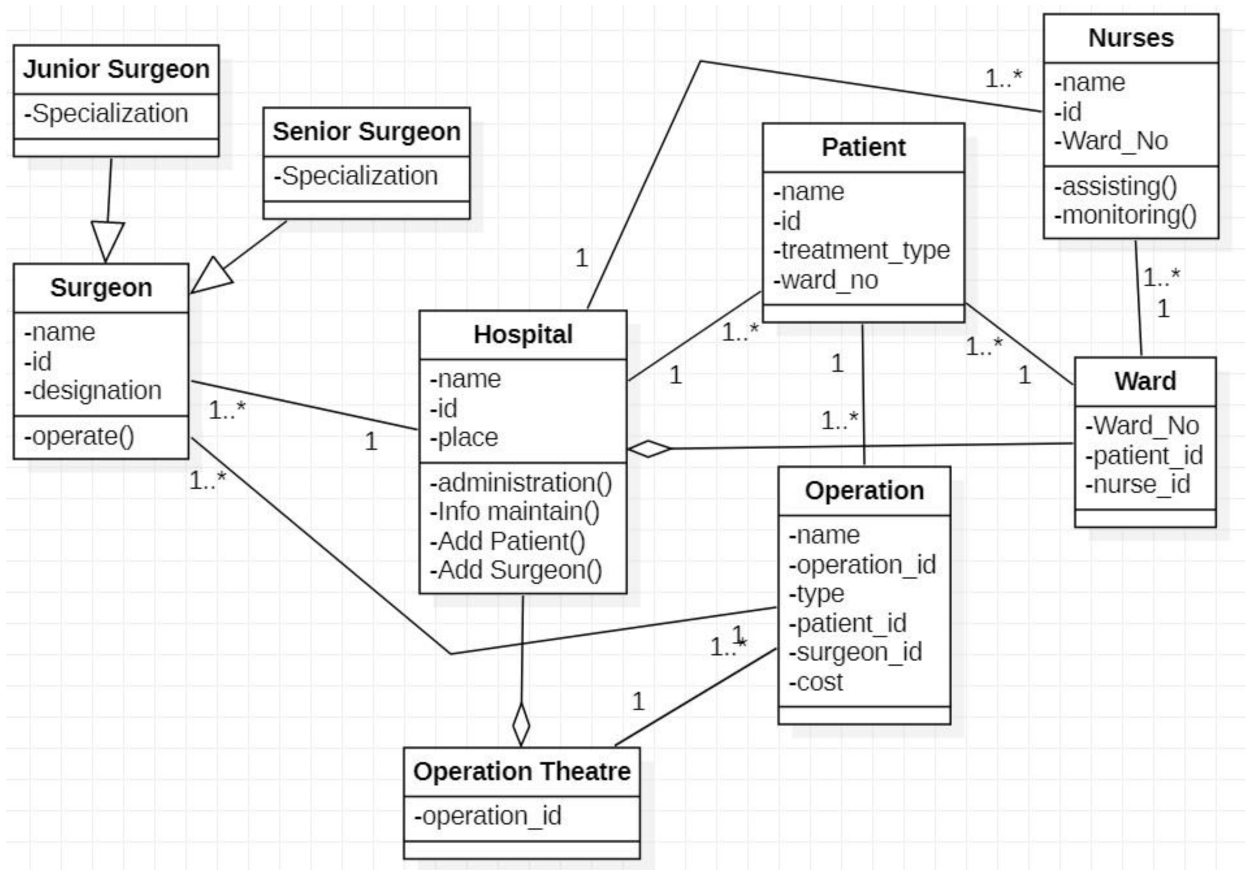
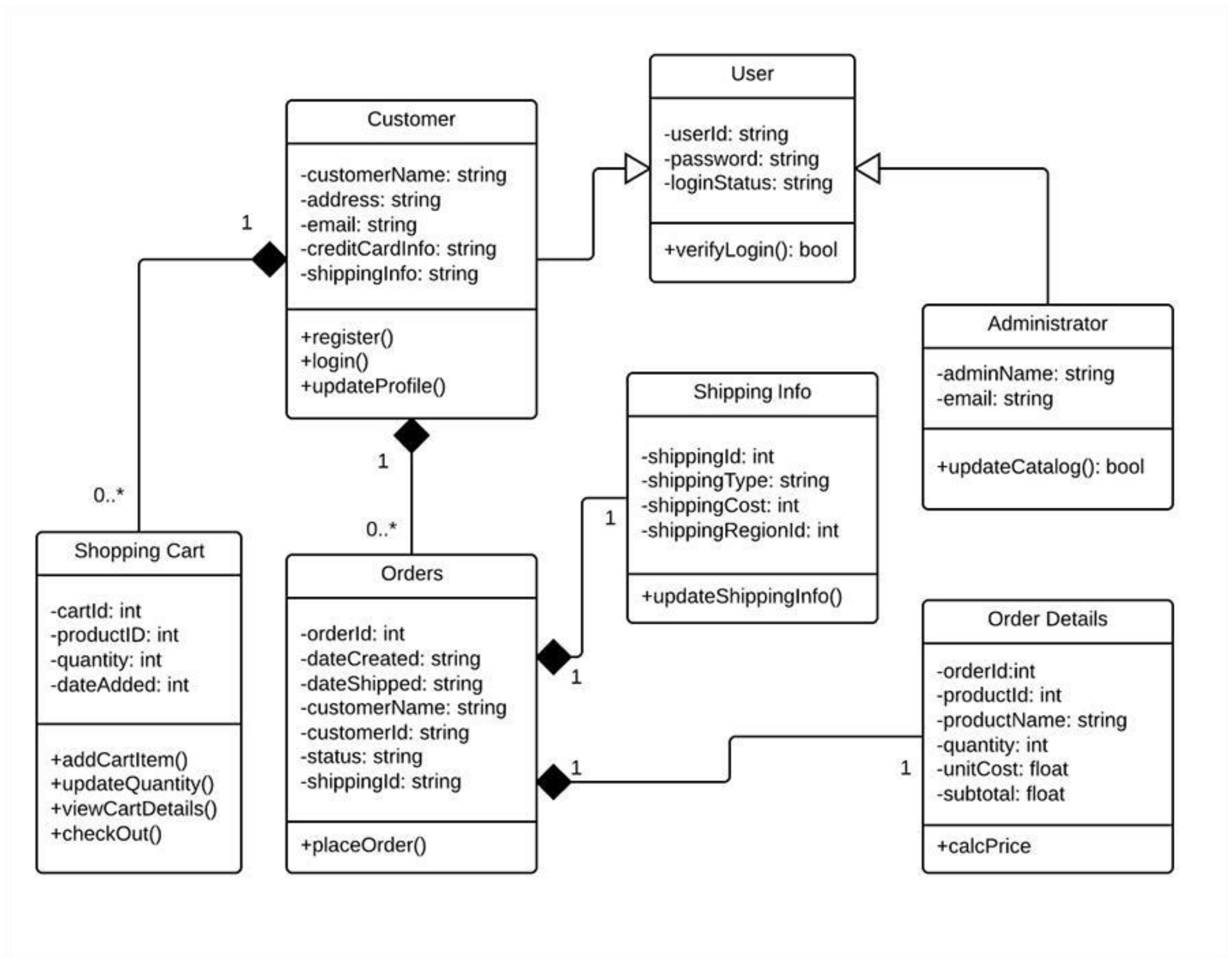


Fig. 1.1 Hospital Management System – Class Diagram

EXERCISE

1. Create a class diagram for online shopping system (Attach printout of the class diagram created during lab session).

Kabeer Ahmed SE-19028



Lab Session 02

OBJECT: **Study UML Use Case Diagram via StarUML**

THEORY

The use case view of the system addresses the understandability and usability of the system. This view looks at actors and use cases along with their interactions. The diagrams in this view are use case diagrams, sequence diagrams and collaboration diagrams.

Use Case Diagram

You can create a UML use case diagram in StarUML to summarize how users (or actors) interact with a system, such as a software application. An actor can be a person, an organization, or another system. Use case diagrams show the expected behavior of the system. They don't show the order in which steps are performed.

- Defining the system boundary determines what is considered external or internal to the system.
- An actor represents a role played by an outside object. One object may play several roles and, therefore, is represented by several actors.
- An association illustrates the participation of the actor in the use case.
- A use case is a set of events that occurs when an actor uses a system to complete a process. Normally, a use case is a relatively large process, not an individual step or transaction.

Use Case Diagram Creation

To create a Use Case Diagram:

- Select first an element where a new Use Case Diagram to be contained as a child.
- Select Model | Add Diagram | Use Case Diagram in Menu Bar or select Add Diagram | Use Case Diagram in Context Menu.

Use Case Subject

To create a Use Case Subject:

- Select Use Case Subject in Toolbox.
- Drag on the diagram as the size of Use Case Subject.

You can use QuickEdit for Model Element.

Actor

To create an Actor:

- Select Actor in Toolbox.

- Drag on the diagram as the size of Actor.

You can use QuickEdit for Actor by double-click or press Enter on a selected Actor.

Name Expression: Edit name expression.
Visibility: Change visibility property.
Add Note: Add a linked note.
Add Constraint: Add a constraint.
Add Attribute (Ctrl+Enter)**:** Add an attribute.
Add Operation (Ctrl+Shift+Enter)**:** Add an operation.
Add Sub-Actor: Add a sub-actor.
Add Super-Actor: Add a super actor.
Add Associated Use Case: Add an associated use case

Use Case

To create a Use Case:

- Select Use Case in Toolbox.
- Drag on the diagram as the size of Use Case.

You can use QuickEdit for Use Case by double-click or press Enter on a selected Use Case.

Name Expression: Edit name expression.
Visibility: Change visibility property.
Add Note: Add a linked note.
Add Constraint: Add a constraint.
Add Extension Point (Ctrl+Enter)**:** Add an extension point.
Add Associated Actor: Add an associated actor.
Add Included Use Case: Add an included use case.
Add Extended Use Case: Add an extended use case.

Extension Point

To add an Extension Point:

- Select a Use Case.
- Select Model | Add | Extension Point in Menu Bar or Add | Extension Point in Context Menu.

You can use QuickEdit for Extension Point by double-click or press Enter on a selected Extension Point.

Name Expression: Edit name expression.
Visibility: Change visibility property.
Add (Ctrl+Enter)**:** Add one more extension point in the below. **Delete**
(Ctrl+Delete)**:** Delete the extension point
Move Up (Ctrl+Up)**:** Move the extension point up.

Move Down (Ctrl+Down)**:** Move the extension point down.
Include

To create an Include:
- Select Include in Toolbox.
- Drag from a Use Case and drop on another Use Case (to be included).

You can use QuickEdit for Relationship.

Extend

To create an Extend:
- Select Extend in Toolbox.
- Drag from a Use Case (to be extended) and drop on another Use Case.

You can use QuickEdit for Relationship.

Online Shopping System Scenario

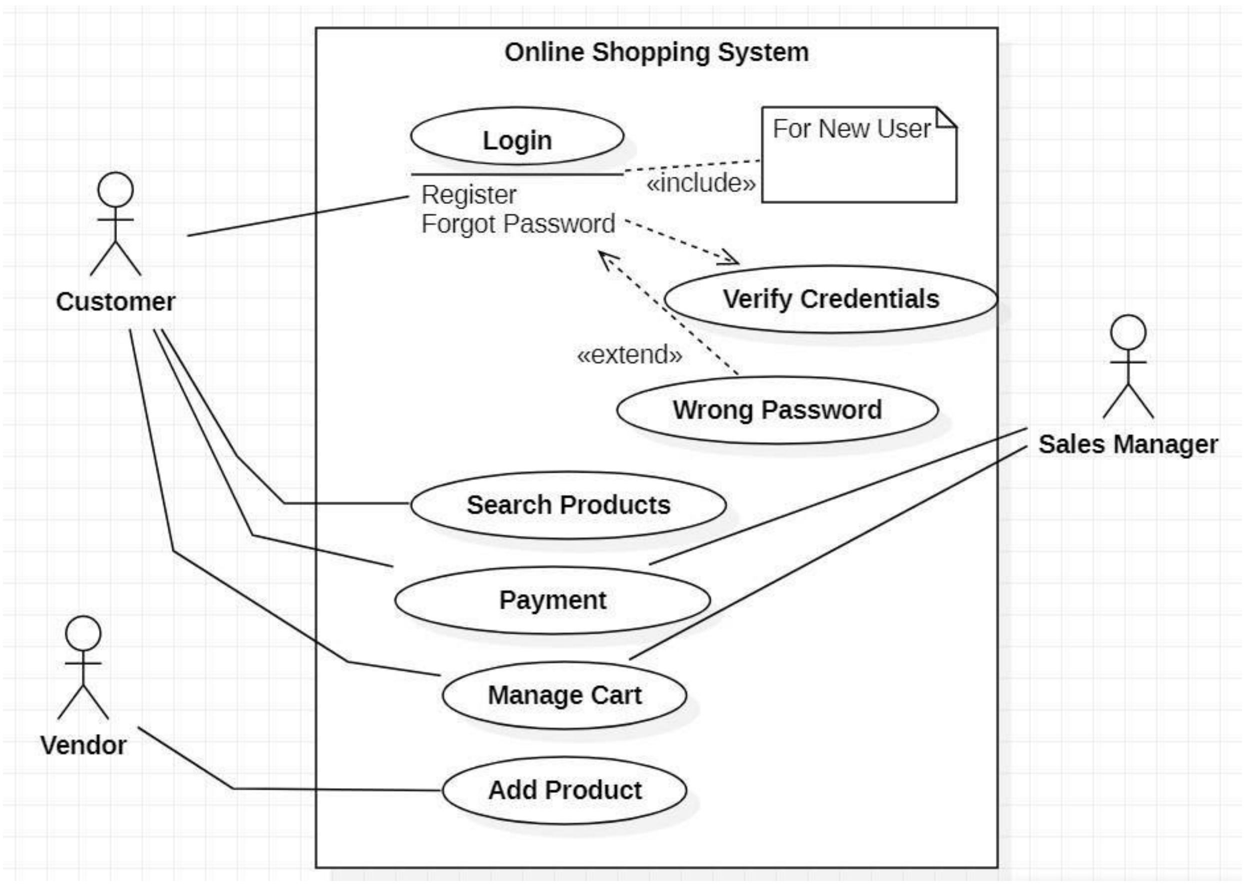
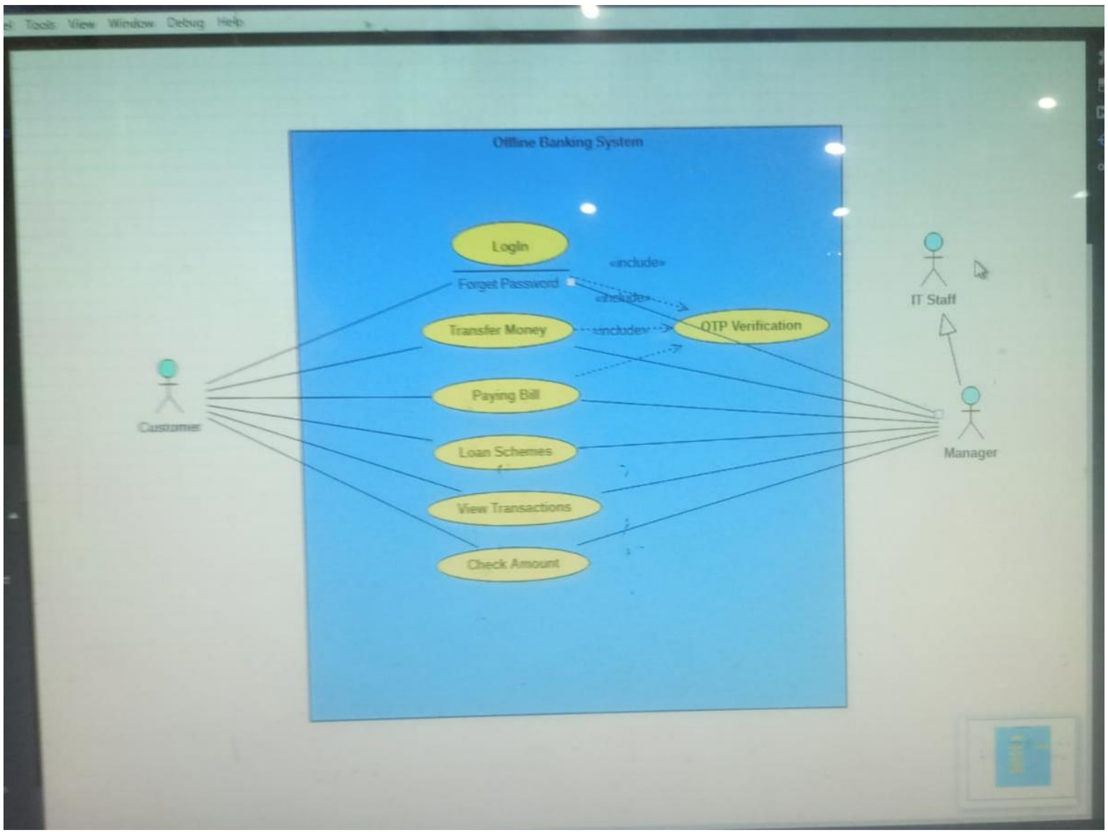


Fig. 2.1 Online shopping system (Use Case Diagram)

Kabeer Ahmed SE-19028

EXERCISE

1. Create the use case diagram representing the Banking System/Online Banking. Show all the actors and their relationship with different use cases. Also show extended and include use cases where appropriate (Attach printout of the use case diagram created during lab session).



Lab Session 03

OBJECT: <u>**Project Planning through the use of Microsoft Project**</u>

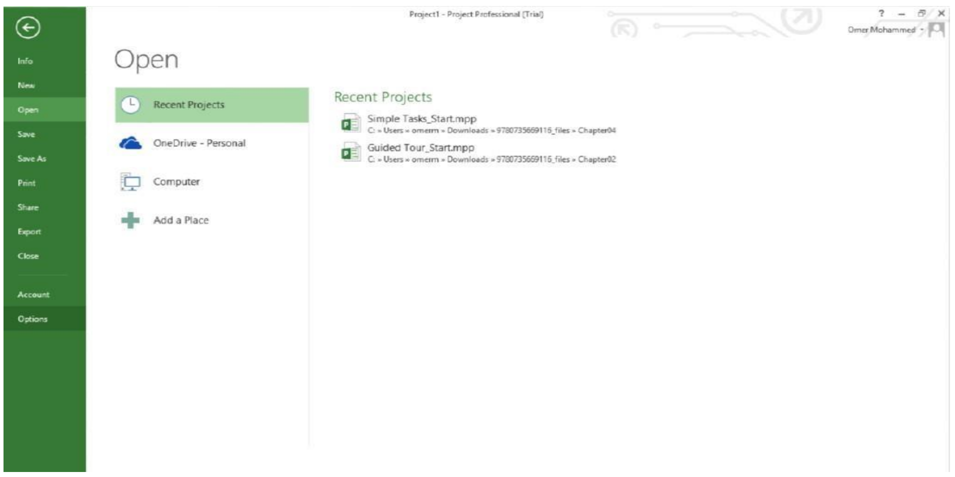
THEORY

Project management involves the planning and organization of a company's resources to move a specific task, event, or duty towards completion. It can involve a one-time project or an ongoing activity, and resources managed include personnel, finances, technology, and intellectual property. Generally speaking, the project management process includes the following stages: planning, initiation, execution, monitoring, and closing. From start to finish, every project needs a plan that outlines how things will get off the ground, how they will be built, and how they will finish.
A plan is a detailed action-oriented, experience and knowledge-based exercise which considers all elements of strategy, scope, cost, time, resources, quality and risk for the project. Scheduling is the science of using mathematical calculations and logic to generate time effective sequence of task considering any resource and cost constraints. Schedule is part of the Plan. In Project Management Methodology, schedule would only mean listing of a project's milestones, tasks/activities, and deliverables, with start and finish dates. The schedule is linked with resources, budgets and dependencies.
MS Project does more than just create a schedule it can establish dependencies among tasks, it can create constraints, it can resolve resource conflicts, and it can also help in reviewing cost and schedule performance over the duration of the project. So, it does help in more than just creating a Schedule.

MS Project – Settings

Following are the default settings you will have when you first install MS Project on your computer.

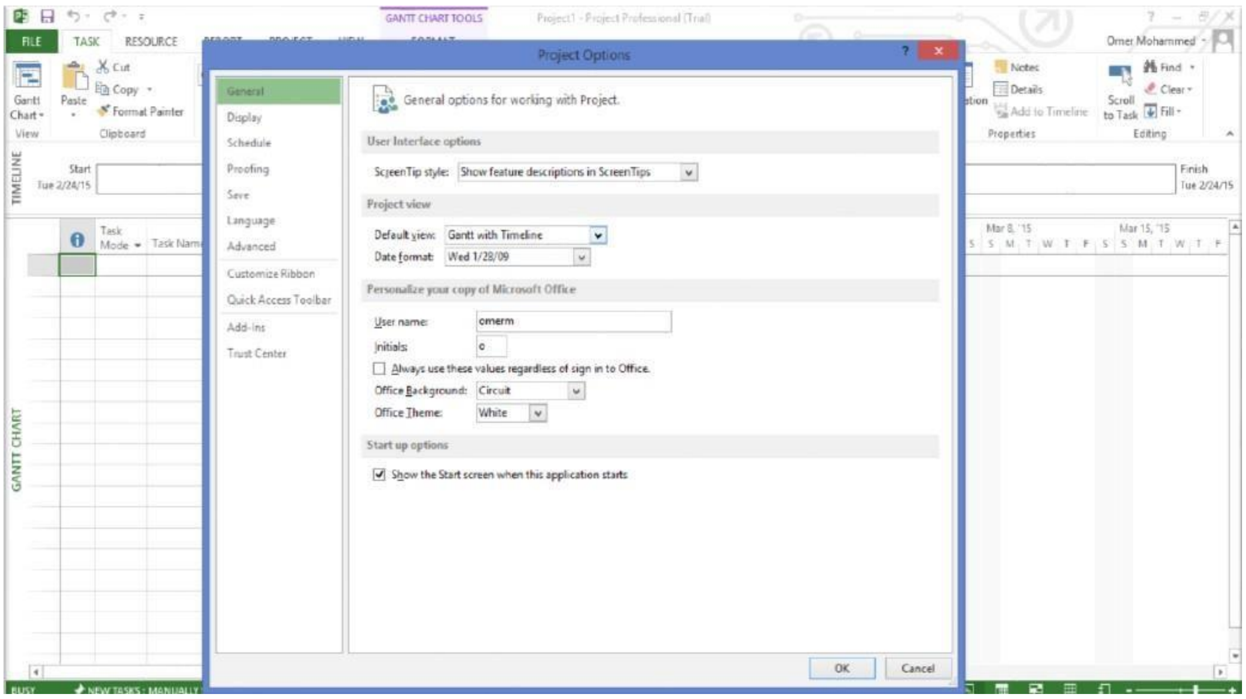
Step 1: File -> Options -> General tab -> Project view -> Default view.



Simple Tasks_Start.mpp
C: » Users » omerm » Downloads » 9780735669116_files » Chapter04
Guided Tour_Start.mpp
C: » Users » omerm » Downloads » 9780735669116_files » Chapter02

Fig. 3.1 Settings of MS Project File

Select “Gantt with Timeline” from the dropdown box.


Project Options

General options for working with Project.

User Interface options

ScreenTip style: Show feature descriptions in ScreenTips

Project view

Default view: Gantt with Timeline

Date format: Wed 1/28/09

Personalize your copy of Microsoft Office

User name: omerm

Initials: o

☐ Always use these values regardless of sign in to Office.

Office Background: Circuit

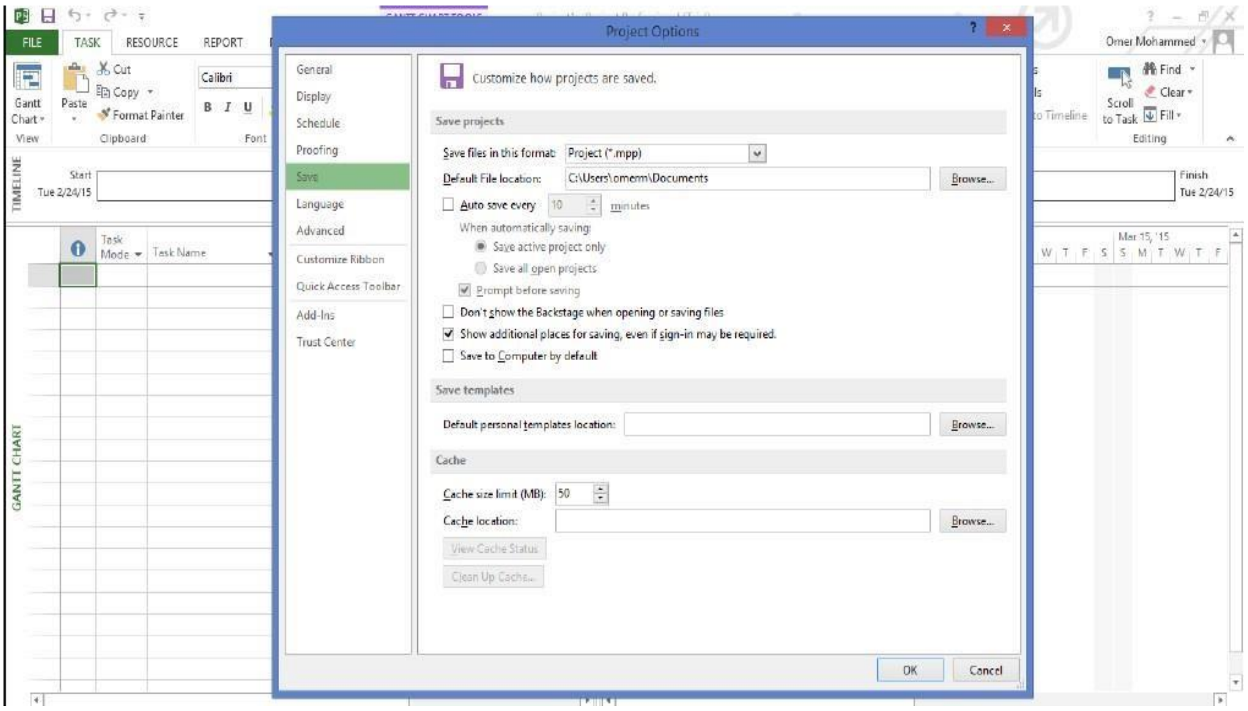
Office Theme: White

Start up options

☑ Show the Start screen when this application starts

Fig. 3.2 Gantt Chart View Selection

Step 2: File -> Options -> Save tab -> Save projects -> Save Files In this format. Select Project (*.mpp).


Project Options

Customize how projects are saved.

Save projects

Save files in this format: Project (*.mpp)

Default File location: C:\Users\omerm\Documents

☐ Auto save every 10 minutes

When automatically saving:

● Save active project only

○ Save all open projects

☑ Prompt before saving

☐ Don't show the Backstage when opening or saving files

☑ Show additional places for saving, even if sign-in may be required.

☐ Save to Computer by default

Save templates

Default personal templates location:

Cache

Cache size limit (MB): 50

Cache location:

Fig. 3.3 Project File Extension for Project Save Option

MS Project – Introduction

Project creates budgets based on assignment work and resource rates. As resources are assigned to tasks and assignment work estimated, the program calculates the cost, equal to the work times the rate, which rolls up to the task level and then to any summary task, and finally to the project level.

Each resource can have its own calendar, which defines what days and shifts a resource is available. Microsoft Project is not suitable for solving problems of available materials (resources) constrained production. Additional software is necessary to manage a complex facility that produces physical goods.

Project Management

MS Project is feature rich, but project management techniques are required to drive a project effectively. A lot of project managers get confused between a schedule and a plan. MS Project can help you in creating a Schedule for the project even with the provided constraints.

From the perspective of Project Management Methodology, a Plan and Schedule are not the same. A plan is a detailed action-oriented, experience and knowledge-based exercise which considers all elements of strategy, scope, cost, time, resources, quality and risk for the project. Scheduling is the science of using mathematical calculations and logic to generate time-effective sequence of task considering any resource and cost constraints. Schedule is part of the Plan. In Project Management Methodology, schedule would only mean listing of a project's milestones, tasks/activities, and deliverables, with start and finish dates. The schedule is linked with resources, budgets and dependencies.

MS Project – Getting Started MS Project UI

Windows 7: Click on Start menu, point to All Programs, click Microsoft Office, and then click Project.

Windows 8: On the Start screen, tap or click Project.

Windows 10: Click on Start menu -> All apps -> Microsoft Office -> Project.

The following screen is the Project’s start screen. Here you have options to open a new plan, some other plans, and even a new plan template.

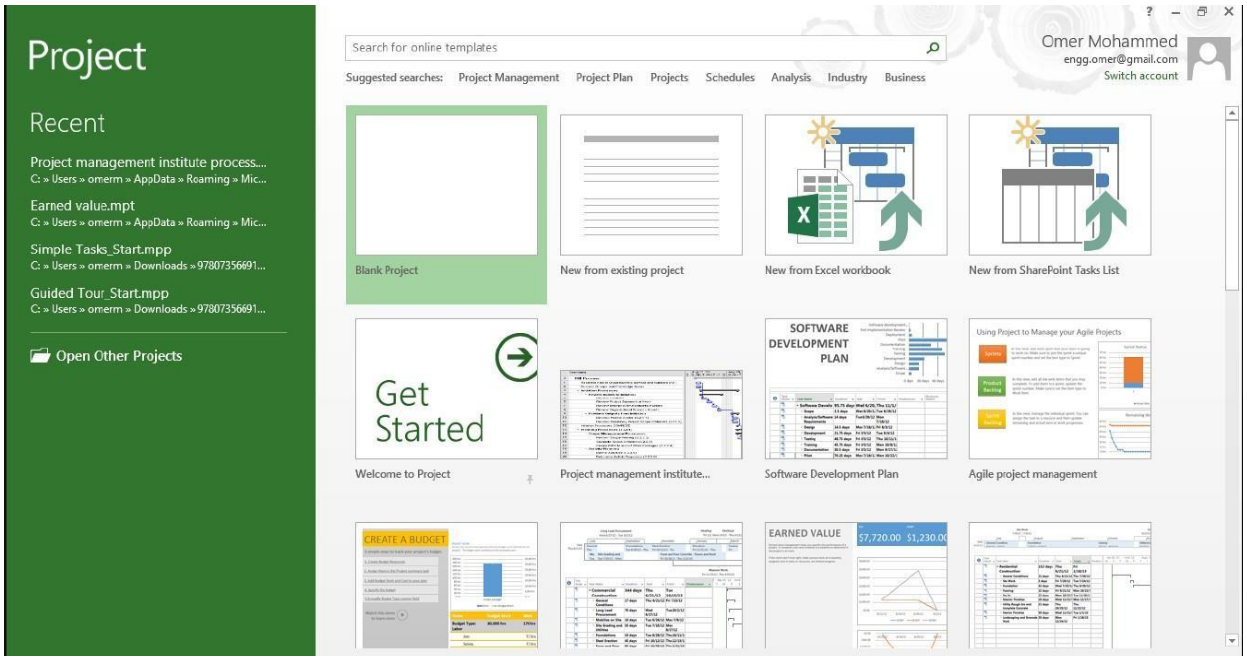

Project
Recent
Project management institute process....
Earned value.mpt
Simple Tasks_Start.mpp
Guided Tour_Start.mpp
Open Other Projects
Search for online templates
Suggested searches: Project Management Project Plan Projects Schedules Analysis Industry Business
Omer Mohammed
engg.omer@gmail.com
Switch account
Blank Project
New from existing project
New from Excel workbook
New from SharePoint Tasks List
Get Started
Welcome to Project
Project management institute...
Software Development Plan
Agile project management
CREATE A BUDGET
EARNED VALUE

Fig. 3.4 MS Project Start Window

Click the Blank Project Tab. The following screen pops up.

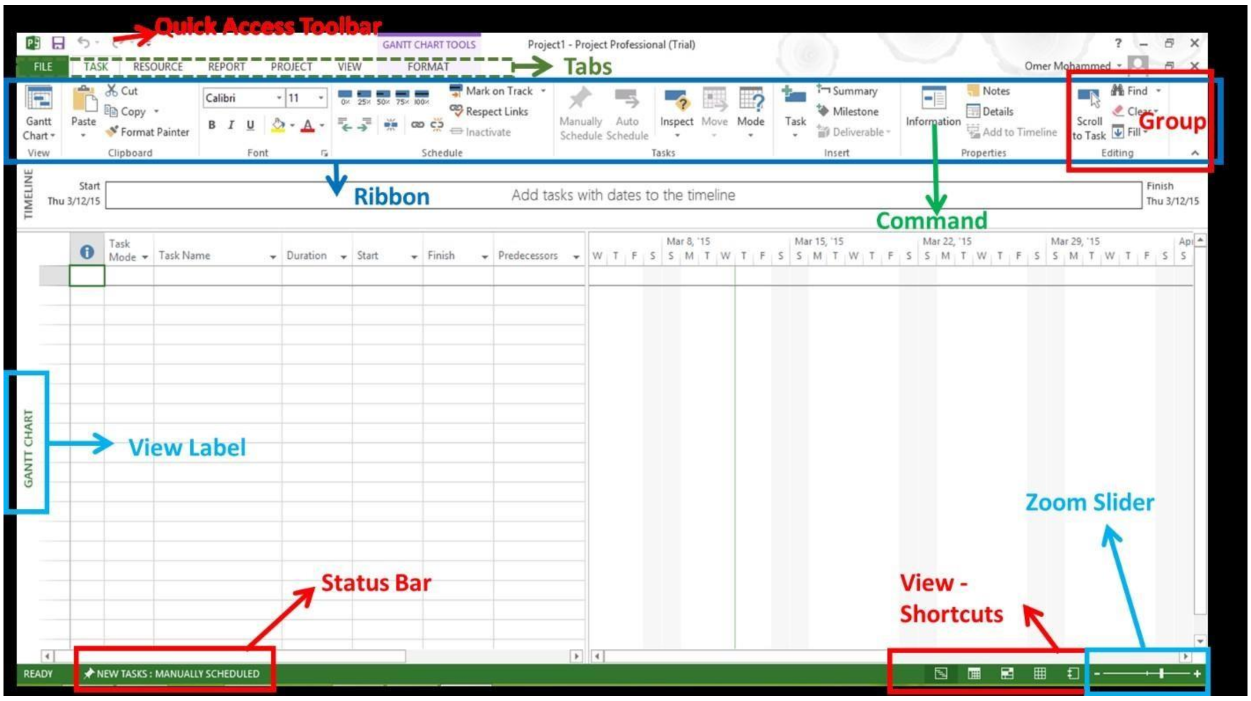


Fig. 3.5 Overview of MS Project Window

The screen should have the MS Project interface displayed. The major part of this interface is:

Quick Access Toolbar: A customizable area where you can add the frequently used commands.

Tabs on the **Ribbon**, **Groups**: With the release of Microsoft Office 2007 came the "Fluent User Interface" or "Fluent UI", which replaced menus and customizable toolbars with a single "Office menu", a miniature toolbar known as "quick-access toolbar" and what came to be known as the ribbon having multiple tabs, each holding a toolbar bearing buttons and occasionally other controls. Toolbar controls have heterogeneous sizes and are classified in visually distinguishable Groups. Groups are collections of related commands. Each tab is divided into multiple groups.

Commands: The specific features you use to perform actions in Project. Each tab contains several commands. If you point at a command, you will see a description in a tooltip.

View Label: This appears along the left edge of the active view. Active view is the one you can see in the main window at a given point in time. Project includes lots of views like Gantt Chart view, Network Diagram view, Task Usage view, etc. The View label just tells you about the view you are using currently. Project can display a single view or multiple views in separate panes.

View Shortcuts: This lets you switch between frequently used views in Project.

Zoom Slider: Simply zooms the active view in or out.

Status Bar: Displays details like the scheduling mode of new tasks (manual or automatic) and details of filter applied to the active view.

Details of Views and Tasks Views

Views allow you to examine your project from different angles based on what information you want displayed at any given time. You can use a combination of views in the same window at the same time.

Project Views are categorized into two types:

- Task Views (5 types)
- Resource Views (3 types)

The Project worksheet is located in the main part of the application window and displays different information depending on the view you choose. The default view is the Gantt Chart View.

Tasks

Goals of any project need to be defined in terms of tasks. There are four major types of tasks:

1. Summary tasks - contain subtasks and their related properties.
2. Subtasks - are smaller tasks that are a part of a summary task.
3. Recurring tasks - are tasks that occur at regular intervals.
4. Milestones - are tasks that are set to zero duration and are like interim goals in the project.

Entering Tasks and Assigning Task Duration

Click in the first cell and type the task name. Press enter (<-) to move to the next row. By default, estimated duration of each task is a day. But there are very few tasks that can be completed in a day's time. You can enter the task duration as and when you decide upon a suitable estimate.
Double-clicking a task or clicking on the Task Information button on the standard toolbar opens the Task Information box. You can fill in more detailed descriptions of tasks and attach documents related to the task in the options available in this box.
To enter a milestone, enter the name and set its duration to zero. Project represents it as a diamond shape instead of a bar in the Gantt chart.
To copy tasks and their contents, click on the task ID number at the left of the task and copy and paste as usual.
You can enter Recurring tasks by clicking on Insert > Recurring task and filling in the duration and recurrence pattern for the task.
Any action you perform on a summary task - deleting it, moving or copying it apply to its subtasks too.

Links

Tasks are usually scheduled to start as soon as possible i.e. the first working day after the project start date.

Dependencies

Dependencies specify the manner in which two tasks are linked. Because Microsoft Project must maintain the manner in which tasks are linked, dependencies can affect the way a task is scheduled. In a scenario where there are two tasks, the following dependencies exist:

Finish to Start – Task 2 cannot start until task 1 finishes.
Start to Finish – Task 2 cannot finish until task 1 starts.
Start to Start – Task 2 cannot start until task 1 starts.
Finish to Finish – Task 2 cannot finish until task 1 finishes.

Tasks can be linked by following these steps:

1. Double-click a task and open the Task Information dialog box.
2. Click the predecessor tab.
3. Select the required predecessor from the drop-down menu.
4. Select the type of dependency from drop down menu in the Type column.
5. Click OK.

The Split task button splits tasks that may be completed in parts at different times with breaks in their duration times.

Constraints

Certain tasks need to be completed within a certain date. Intermediate deadlines may need to be specified. By assigning constraints to a task you can account for scheduling problems. There are about 8 types of constraints and they come under the flexible or inflexible category. They are: **As**

Late As Possible – Sets the start date of your task as late in the Project as possible, without pushing out the Project finish date.

As Soon As Possible – Sets the start date of your task as soon as possible without preceding the project start date.
Must Finish On – Sets the finish date of your task to the specified date.
Must Start On – Sets the start date of your task to the specified date.
Start No Earlier Than – Sets the start date of your task to the specified date or later.
Start No Later Than – Sets the start date of your task to the specified date or earlier.
Finish No Earlier Than – Sets the finish date of your task to the specified date or later.
Finish No Later Than – Sets the finish date of your task to the specified date or earlier.

Resources

Resources are of two types

- work resources - material resources.

Work resources complete tasks by expending time on them. They are usually people and equipment that have been assigned to work on the project.
Material resources are supplies and stocks that are needed to complete a project.
A new feature in Microsoft Project is that it allows you to track material resources and assign them to tasks.

Entering Resource Information in Project

The Assign Resources dialog box is used to create list of names in the resource pool.
To enter resource lists:

1. Click the Assign Resources button on the Standard Tool bar or Tools > Resources > Assign resources.
2. Select a cell in the name column and type a response name. Press Enter <-
3. Repeat step 2 until you enter all the resource names.
4. Click OK.

Resource names cannot contain slash (/), brackets [ ] and commas (,). Another way of defining your resource list is through the Resource Sheet View.

1. Display Resource Sheet View by choosing View > Resource Sheet or click the Resource Sheet icon on the View bar.
2. Enter your information. (To enter material resource use the Material Label field).

To assign a resource to a task:

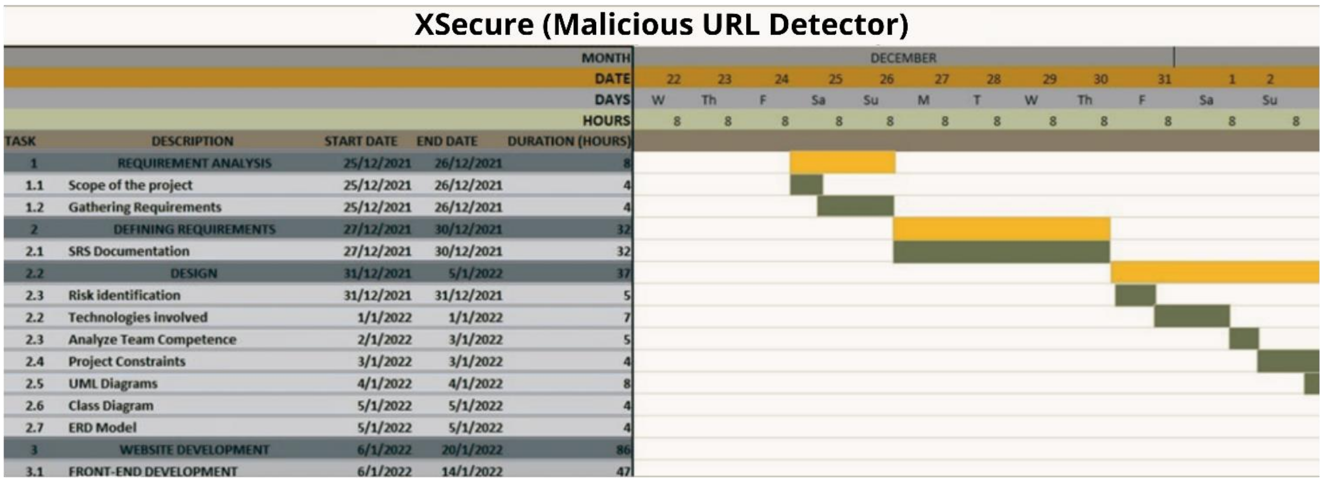
1. Open the Gantt Chart View and the Assign Resources Dialog box.
2. In the Entry Table select the tasks for which you want to assign resources.

Kabeer Ahmed SE-19028

3. In the Assign Resources Dialog box, select the resource (resources) you want to assign.
4. Click the Assign button.

EXERCISE

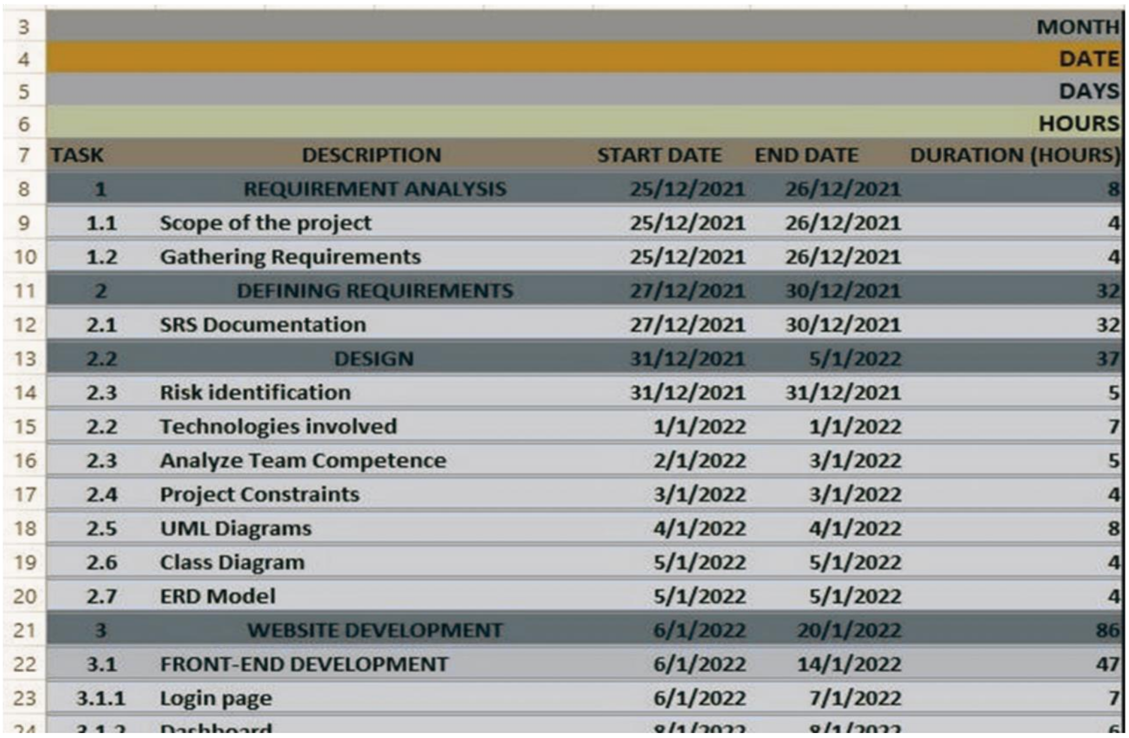
1. Create the activity plan with complete tasks and their durations of the semester project assigned to you in the course. Also, show the allocated resources to each of the tasks (Attach printout of the sheets).



XSecure (Malicious URL Detector)

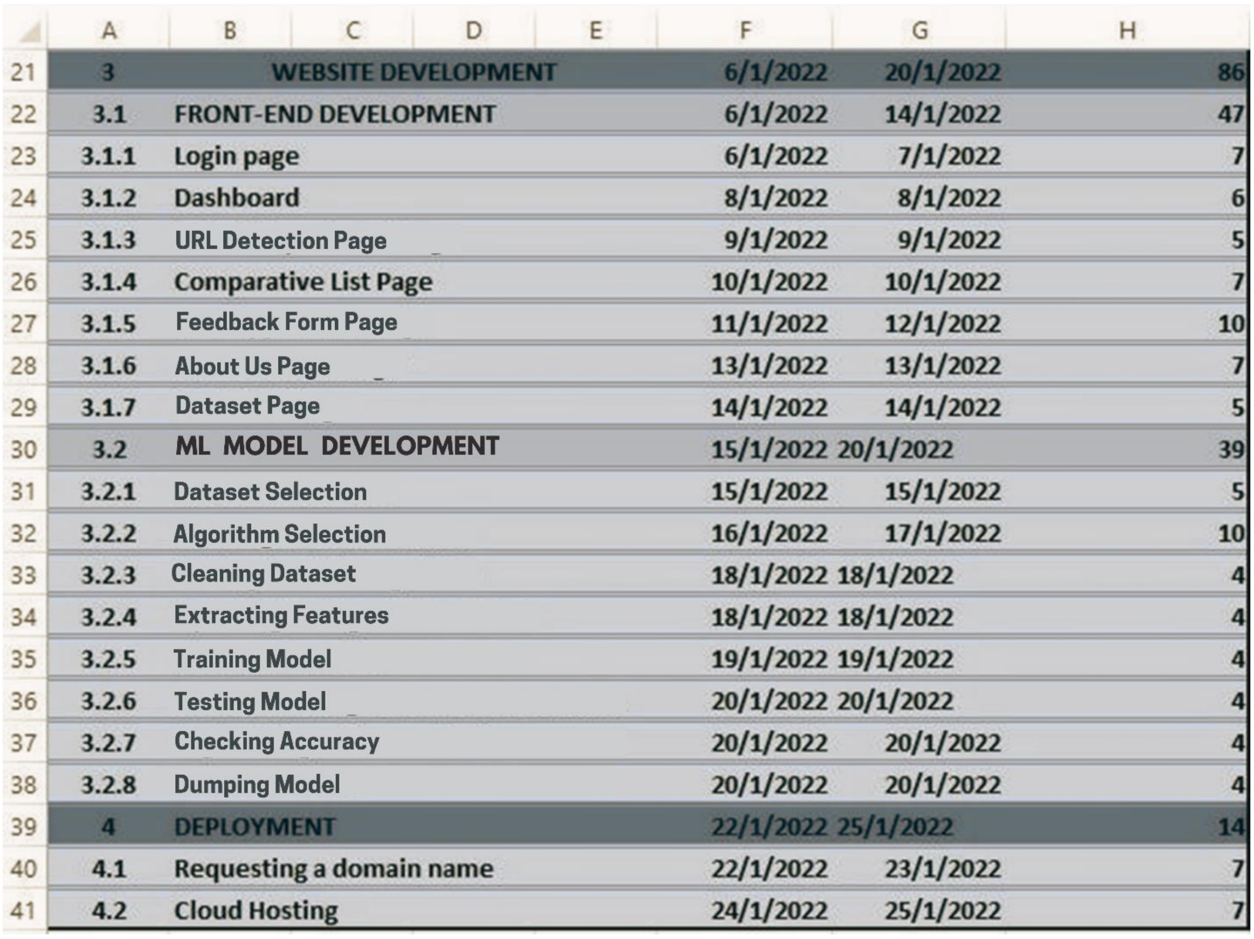
MONTH	DECEMBER											
DATE	22	23	24	25	26	27	28	29	30	31	1	2
DAYS	W	Th	F	Sa	Su	M	T	W	Th	F	Sa	Su
HOURS	8	8	8	8	8	8	8	8	8	8	8	8

TASK	DESCRIPTION	START DATE	END DATE	DURATION (HOURS)
1	REQUIREMENT ANALYSIS	25/12/2021	26/12/2021	8
1.1	Scope of the project	25/12/2021	26/12/2021	4
1.2	Gathering Requirements	25/12/2021	26/12/2021	4
2	DEFINING REQUIREMENTS	27/12/2021	30/12/2021	32
2.1	SRS Documentation	27/12/2021	30/12/2021	32
2.2	DESIGN	31/12/2021	5/1/2022	37
2.3	Risk identification	31/12/2021	31/12/2021	5
2.2	Technologies involved	1/1/2022	1/1/2022	7
2.3	Analyze Team Competence	2/1/2022	3/1/2022	5
2.4	Project Constraints	3/1/2022	3/1/2022	4
2.5	UML Diagrams	4/1/2022	4/1/2022	8
2.6	Class Diagram	5/1/2022	5/1/2022	4
2.7	ERD Model	5/1/2022	5/1/2022	4
3	WEBSITE DEVELOPMENT	6/1/2022	20/1/2022	86
3.1	FRONT-END DEVELOPMENT	6/1/2022	14/1/2022	47

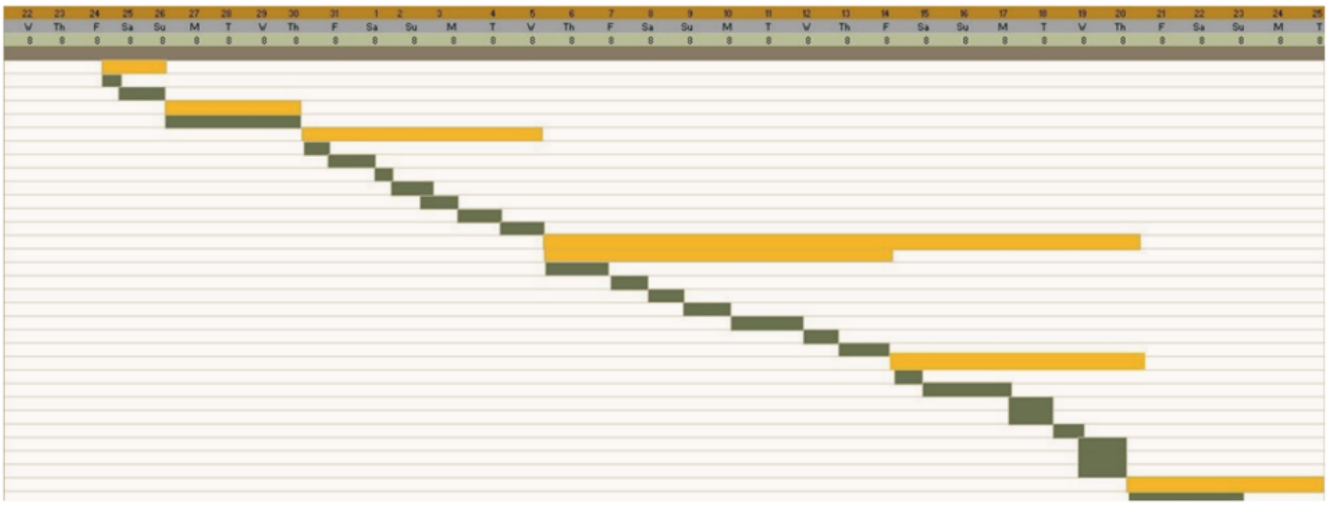


3					MONTH
4					DATE
5					DAYS
6					HOURS
7	TASK	DESCRIPTION	START DATE	END DATE	DURATION (HOURS)
8	1	REQUIREMENT ANALYSIS	25/12/2021	26/12/2021	8
9	1.1	Scope of the project	25/12/2021	26/12/2021	4
10	1.2	Gathering Requirements	25/12/2021	26/12/2021	4
11	2	DEFINING REQUIREMENTS	27/12/2021	30/12/2021	32
12	2.1	SRS Documentation	27/12/2021	30/12/2021	32
13	2.2	DESIGN	31/12/2021	5/1/2022	37
14	2.3	Risk identification	31/12/2021	31/12/2021	5
15	2.2	Technologies involved	1/1/2022	1/1/2022	7
16	2.3	Analyze Team Competence	2/1/2022	3/1/2022	5
17	2.4	Project Constraints	3/1/2022	3/1/2022	4
18	2.5	UML Diagrams	4/1/2022	4/1/2022	8
19	2.6	Class Diagram	5/1/2022	5/1/2022	4
20	2.7	ERD Model	5/1/2022	5/1/2022	4
21	3	WEBSITE DEVELOPMENT	6/1/2022	20/1/2022	86
22	3.1	FRONT-END DEVELOPMENT	6/1/2022	14/1/2022	47
23	3.1.1	Login page	6/1/2022	7/1/2022	7
24	3.1.2	Dashboard	8/1/2022	8/1/2022	6

Kabeer Ahmed SE-19028



	A	B	C	D	E	F	G	H
21	3	WEBSITE DEVELOPMENT				6/1/2022	20/1/2022	86
22	3.1	FRONT-END DEVELOPMENT				6/1/2022	14/1/2022	47
23	3.1.1	Login page				6/1/2022	7/1/2022	7
24	3.1.2	Dashboard				8/1/2022	8/1/2022	6
25	3.1.3	URL Detection Page				9/1/2022	9/1/2022	5
26	3.1.4	Comparative List Page				10/1/2022	10/1/2022	7
27	3.1.5	Feedback Form Page				11/1/2022	12/1/2022	10
28	3.1.6	About Us Page				13/1/2022	13/1/2022	7
29	3.1.7	Dataset Page				14/1/2022	14/1/2022	5
30	3.2	ML MODEL DEVELOPMENT				15/1/2022	20/1/2022	39
31	3.2.1	Dataset Selection				15/1/2022	15/1/2022	5
32	3.2.2	Algorithm Selection				16/1/2022	17/1/2022	10
33	3.2.3	Cleaning Dataset				18/1/2022	18/1/2022	4
34	3.2.4	Extracting Features				18/1/2022	18/1/2022	4
35	3.2.5	Training Model				19/1/2022	19/1/2022	4
36	3.2.6	Testing Model				20/1/2022	20/1/2022	4
37	3.2.7	Checking Accuracy				20/1/2022	20/1/2022	4
38	3.2.8	Dumping Model				20/1/2022	20/1/2022	4
39	4	DEPLOYMENT				22/1/2022	25/1/2022	14
40	4.1	Requesting a domain name				22/1/2022	23/1/2022	7
41	4.2	Cloud Hosting				24/1/2022	25/1/2022	7



Lab Session 04

OBJECT: <u>**Familiarization with EiffelStudio**</u>

THEORY

EiffelStudio is an Integrated Development Environment (IDE), a software application that provides comprehensive facilities to computer programmers for software development, powered by the Eiffel language. EiffelStudio has a comprehensive suite of tools and services that enable programmers to produce correct, reliable, and maintainable software and control the development process.

Setup and Installation

Download the product as per your operating system.

System Requirements

Table. 4.1: System Requirements for Eiffelstudio (Windows)

Computer/Processor	x86 or x86-64
Operating System	Windows 7, Windows 8, Windows 8.1 and Windows 10
C Compiler	Microsoft Visual Studio 2010 or greater, or using MinGW (gcc) included in the EiffelStudio delivery
Memory	4GB of RAM
Hard Disk	1GB of free space

Installing EiffelStudio from the Web

After downloading the EiffelXX.msi installation package, right click on it and select Install. This will launch the installation procedure. Follow the steps indicated in the dialogs to complete the installation.
EiffelStudio is the central tool of Eiffel Software's implementation of Eiffel, letting you design, develop, debug, document, measure, maintain, test, revise and expand systems using the full power of object technology and Design by Contract.

Launching EiffelStudio under Window

On Windows, you can launch EiffelStudio from the Start Menu by following the path:

Start --> Programs --> EiffelStudio Version --> EiffelStudio

where Version is the version number, e.g., 6.5. Alternatively, you can double-click the icon that the installation procedure will have added to your desktop (if you have selected that option during installation).

If this is the first time you are using EiffelStudio, you may get a dialog asking for an unlock code or inviting you to register the product.

Compiling and Executing a System

EiffelStudio first comes up with a window and a dialog on top of it; the dialog looks like this (from here on the look-and-feel will be slightly different on platforms other than Windows, but the content will be the same):

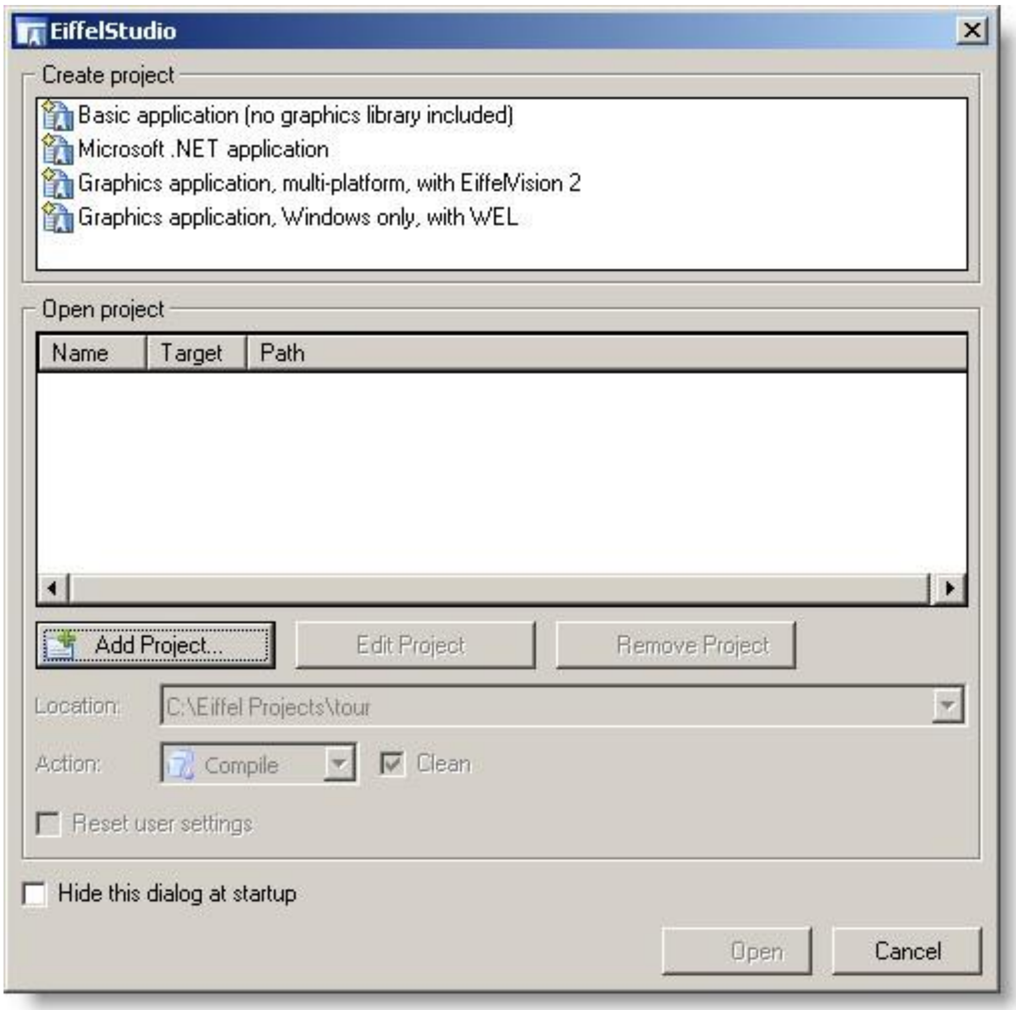

EiffelStudio
Create project
Basic application (no graphics library included)
Microsoft .NET application
Graphics application, multi-platform, with EiffelVision 2
Graphics application, Windows only, with WEL
Open project
Name
Target
Path
Add Project...
Edit Project
Remove Project
Location:
C:\Eiffel Projects\tour
Action:
Compile
Clean
Reset user settings
Hide this dialog at startup
Open
Cancel

Fig. 4.1 Stratup Window of EiffelStudio

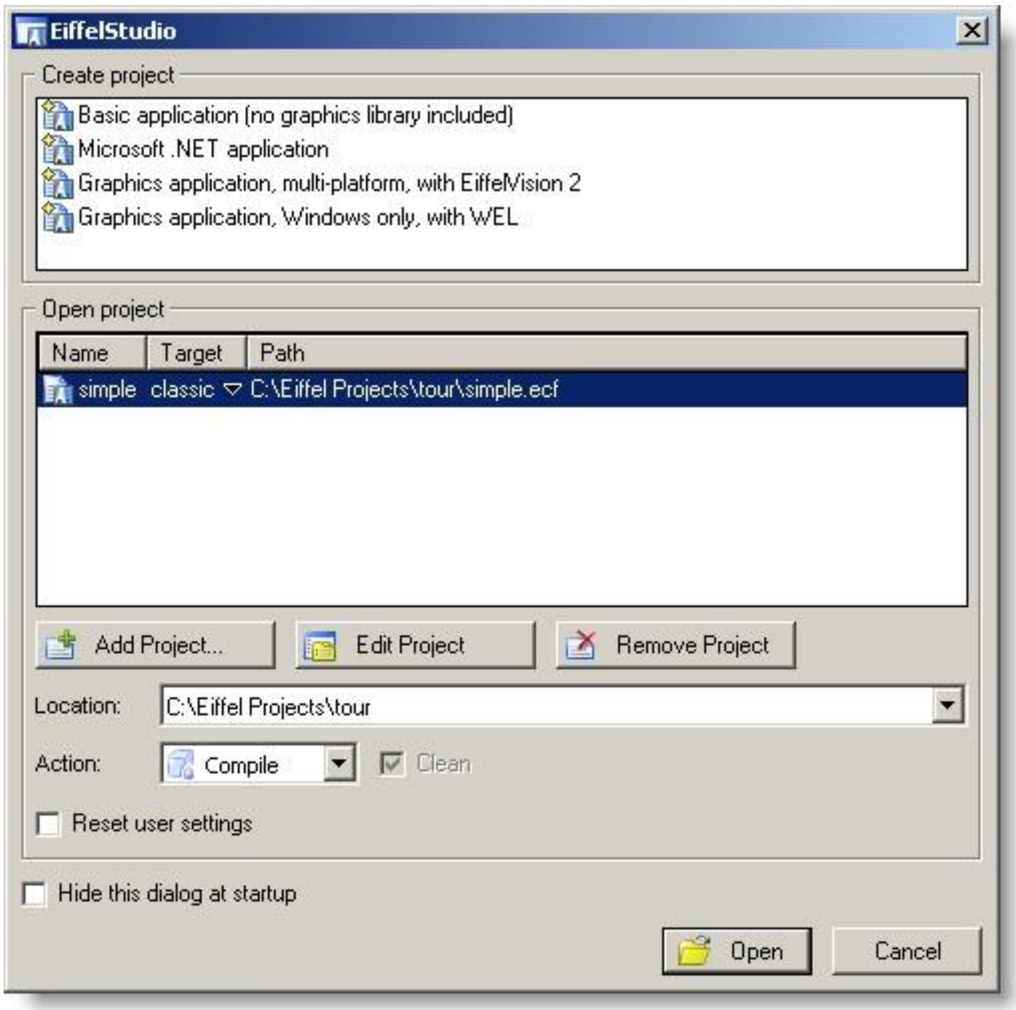
As this is our first project we want to "*Add Project...*". We could also

"*Create project*", which would let you select one among the common schemes -- Basic Application, Graphical Windows Application, Graphical Multi-platform Application, Microsoft .NET Application -- and set up everything for you.

"*Open project*", which would let you open a previously added project.

In future sessions you'll probably use "*Create project*" for a new project, as it takes care of generating a root class and configuration file for you, and Open project" to open an existing project.

Right now, you first have to add the project, so click on the *Add Project*... button. This brings up a File Explorer inviting you to select an ECF file.
In the directory "*YOURDIR*", (either *$ISE_EIFFEL\examples\studio\tour* or the copy that you have made). The "*.ecf*" file is an Eiffel Configuration File which contains the information necessary for construction of an Eiffel project. So, use the File Explorer to find and select the file.



Basic application (no graphics library included)
Microsoft .NET application
Graphics application, multi-platform, with EiffelVision 2
Graphics application, Windows only, with WEL

Name	Target	Path
simple	classic	C:\Eiffel Projects\tour\simple.ecf

Location: C:\Eiffel Projects\tour
Action: Compile [✓ Clean]
Reset user settings
Hide this dialog at startup

Fig. 4.2 Open Exiting Project in EiffelStudio

Click the button labeled "*Open*" to confirm. This starts compilation of your project.
During Eiffel compilation, you can observe the successive compilation steps, in the Outputs tool. The bulk of our little project is the EiffelBase library, which the EiffelStudio installation procedure has precompiled; as a result, there are only a few extra classes to compile, and the process is almost instantaneous on a state-of-the-art computer.

After Eiffel compilation completes you will see the message

```
Eiffel Compilation Succeeded
```

in the Outputs tool.

At this stage your project has finished compiling.

Executing the System

Find and click the Run button (▶) on the toolbar at the top of the EiffelStudio window.

EXERCISE

1. Explore the syntax of Eiffel language. What is the syntax for adding variables and methods in the program? (Show using basic "Hello World" program as performed in lab session).

LAB # 04:

APPLICATION CLASS:

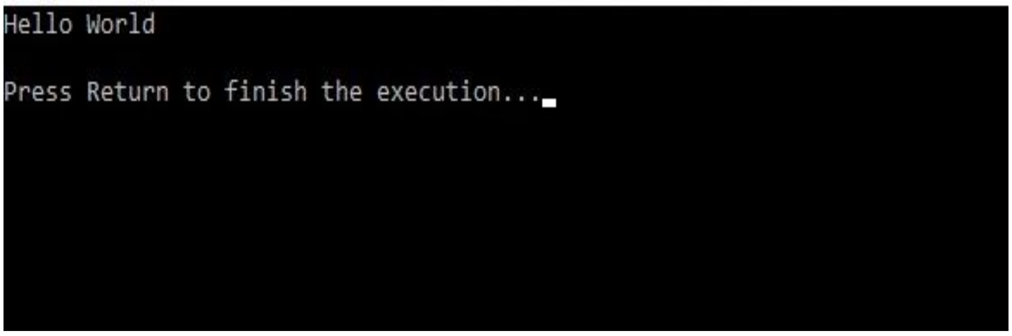
Code: note

```
    description: "session1 application root class"     date:
"$Date$"
         revision: "$Revision$"

class
         APPLICATION

inherit
         ARGUMENTS_32

create
         make

feature {NONE} -- Initialization

         make

                   -- Run application.
           do
                   --| Add your code here
                   create s.set_name ("Hello World")
                        print ( s.name + "%N")
           end

feature
         s: STRINGCLASS

end
```

STRINGCLASS: CODE:

```
note
                description: "Summary description for {STRINGCLASS}."
                author: ""
                date: "$Date$"
                revision: "$Revision$"
class
                STRINGCLASS

create
                set_name
feature
                Name: STRING

feature
                set_name (n: STRING)
                 require
                        name_exist: not n.is_empty
                 do
                        name := n
                 ensure
                        name_set: name = n
                 end
invariant
                name_exist: not name.is_empty

end
```

OUTPUT:



```
Hello World

Press Return to finish the execution...
```

Lab Session 05

OBJECT: <u>Apply Design by Contract (DbC) using EiffelStudio</u>

THEORY

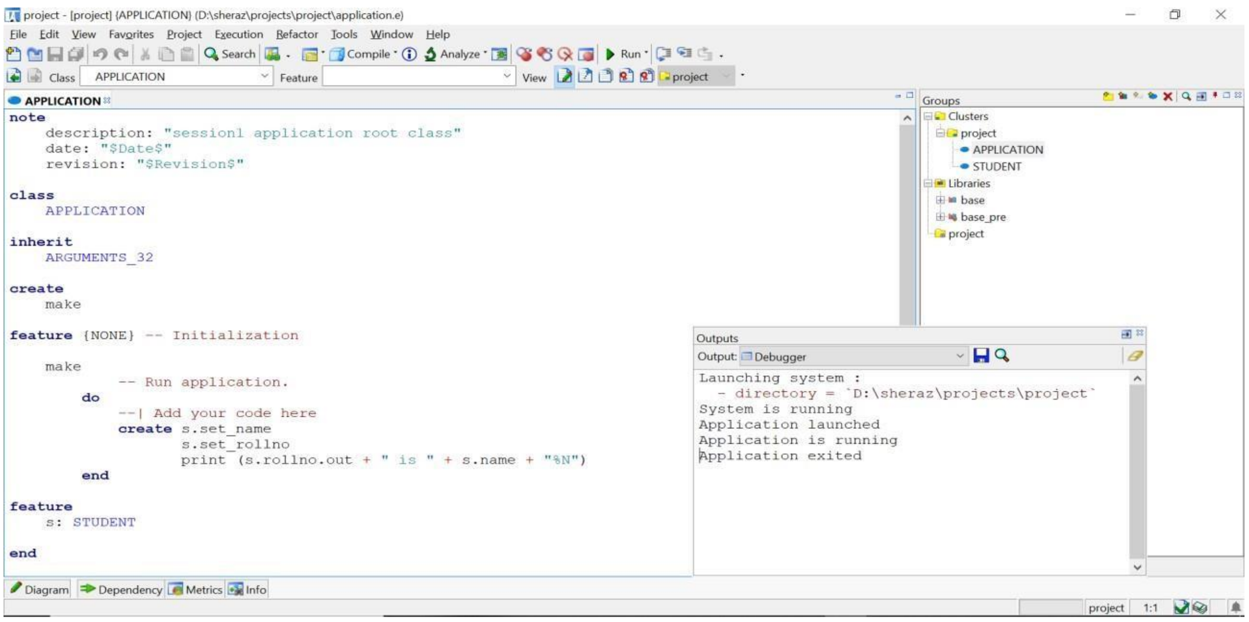
The term “Design by Contract” was introduced by Bertrand Meyer as an approach to defining formal specifications for software components. Many developers now refer to it as contract programming instead. The central notion is that a software component provides a contract, or obligation, for the services that it will provide, and that by using the component a client agrees to the terms of that contract. This concept is founded upon Hoare logic, a formal system developed by Tony Hoare for reasoning about the correctness of programs.

The main principles of this model are threefold:

1. **<u>Preconditions:</u>** the client is obligated to meet a function's required preconditions before calling a function. If the preconditions are not met, then the function may not operate correctly.
2. **<u>Postconditions:</u>** the function guarantees that certain conditions will be met after it has finished its work. If a postcondition is not met, then the function did not complete its work correctly.
3. **<u>Class Invariant:</u>** constraints that every instance of the class must satisfy. This defines the state that must hold true for the class to operate according to its design.

Students Details Scenario

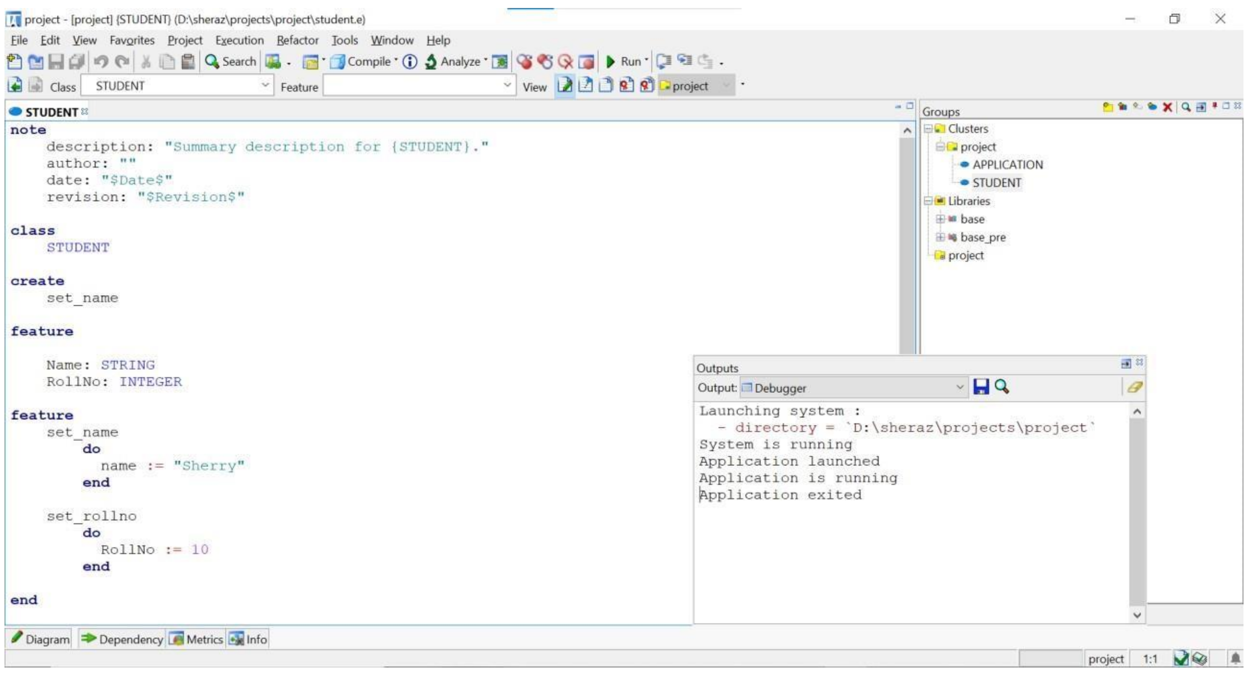
Code without DbC Implementation


project - [project] {APPLICATION} (D:\sheraz\projects\project\application.e)
File Edit View Favorites Project Execution Refactor Tools Window Help
Search Compile Analyze Run
Class APPLICATION Feature View project
Groups
Clusters
project
APPLICATION
STUDENT
Libraries
base
base_pre
project
Outputs
Output: Debugger
Diagram Dependency Metrics Info

```
note
    description: "session1 application root class"
    date: "$Date$"
    revision: "$Revision$"

class
    APPLICATION

inherit
    ARGUMENTS_32

create
    make

feature {NONE} -- Initialization

    make
            -- Run application.
        do
            --| Add your code here
            create s.set_name
                   s.set_rollno
                   print (s.rollno.out + " is " + s.name + "%N")
        end

feature
    s: STUDENT

end
```

```
Launching system :
  - directory = `D:\sheraz\projects\project`
System is running
Application launched
Application is running
Application exited
```

Fig. 5.1 Application Class (Before DbC)

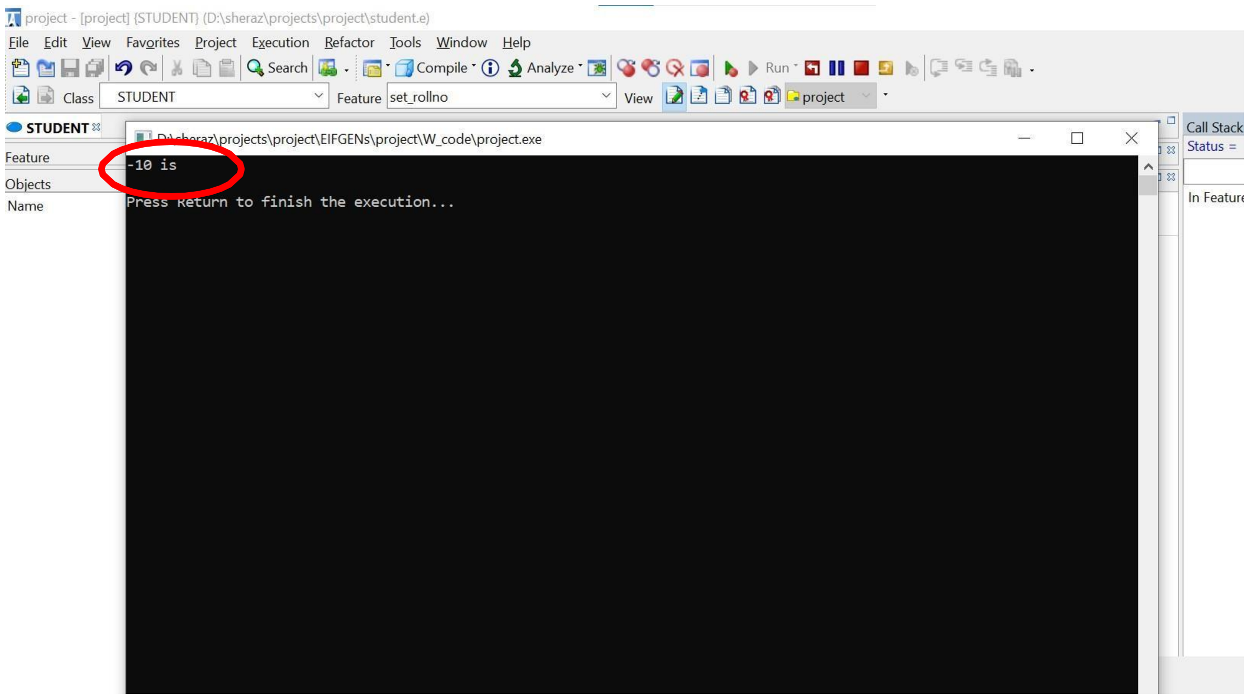

project - [project] {STUDENT} (D:\sheraz\projects\project\student.e)
File Edit View Favorites Project Execution Refactor Tools Window Help
Class STUDENT Feature View project
Groups Clusters project APPLICATION STUDENT Libraries base base_pre project
Diagram Dependency Metrics Info

```
note
    description: "Summary description for {STUDENT}."
    author: ""
    date: "$Date$"
    revision: "$Revision$"

class
    STUDENT

create
    set_name

feature

    Name: STRING
    RollNo: INTEGER

feature
    set_name
        do
            name := "Sherry"
        end

    set_rollno
        do
            RollNo := 10
        end

end
```

```
Launching system :
  - directory = `D:\sheraz\projects\project`
System is running
Application launched
Application is running
Application exited
```

Fig. 5.2 Student Class (Before DbC)

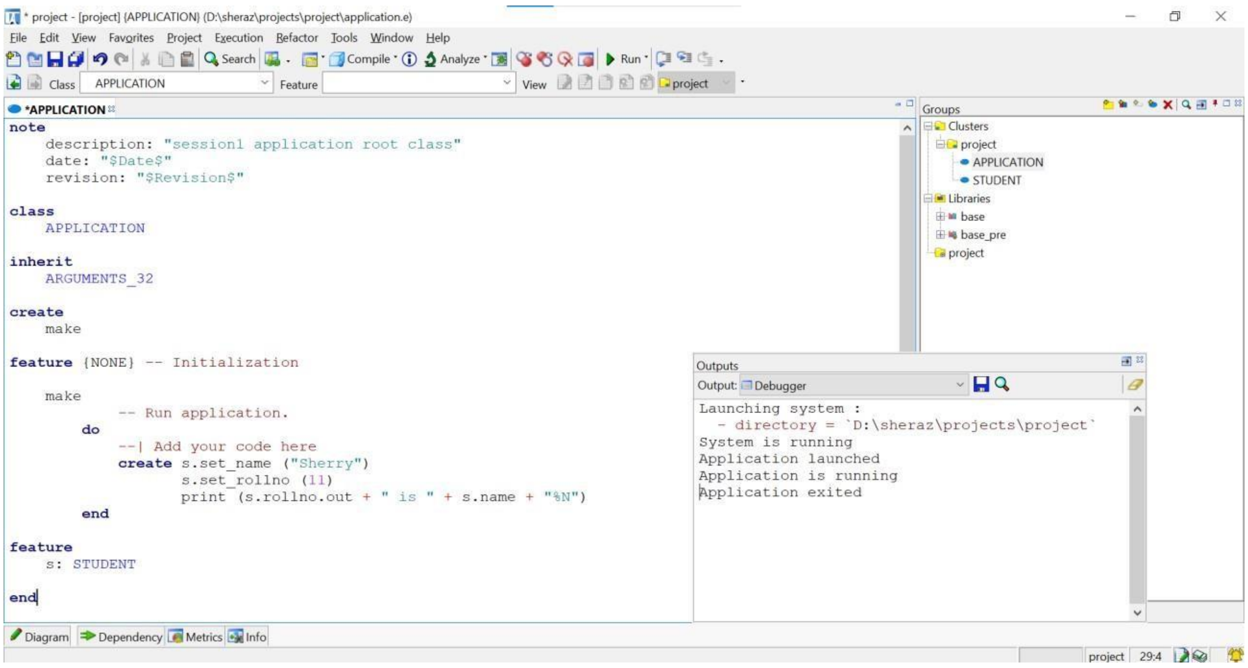
As there is no precondition and postondition defined in the code so any kind of invalid input would also be acceptable in the program in this condition. For example, student name NULL i.e. left the student name field empty and a negative value of roll number would also be accpeted by the program as show in Fig. 5.3.


project - [project] {STUDENT} (D:\sheraz\projects\project\student.e)
File Edit View Favorites Project Execution Refactor Tools Window Help
Class STUDENT Feature set_rollno View project
STUDENT
Feature
Objects
Name
Call Stack
Status =
In Feature
D:\sheraz\projects\project\EIFGENs\project\W_code\project.exe

```
-10 is

Press Return to finish the execution...
```

Fig. 5.3 Output of the Program (Before DbC)

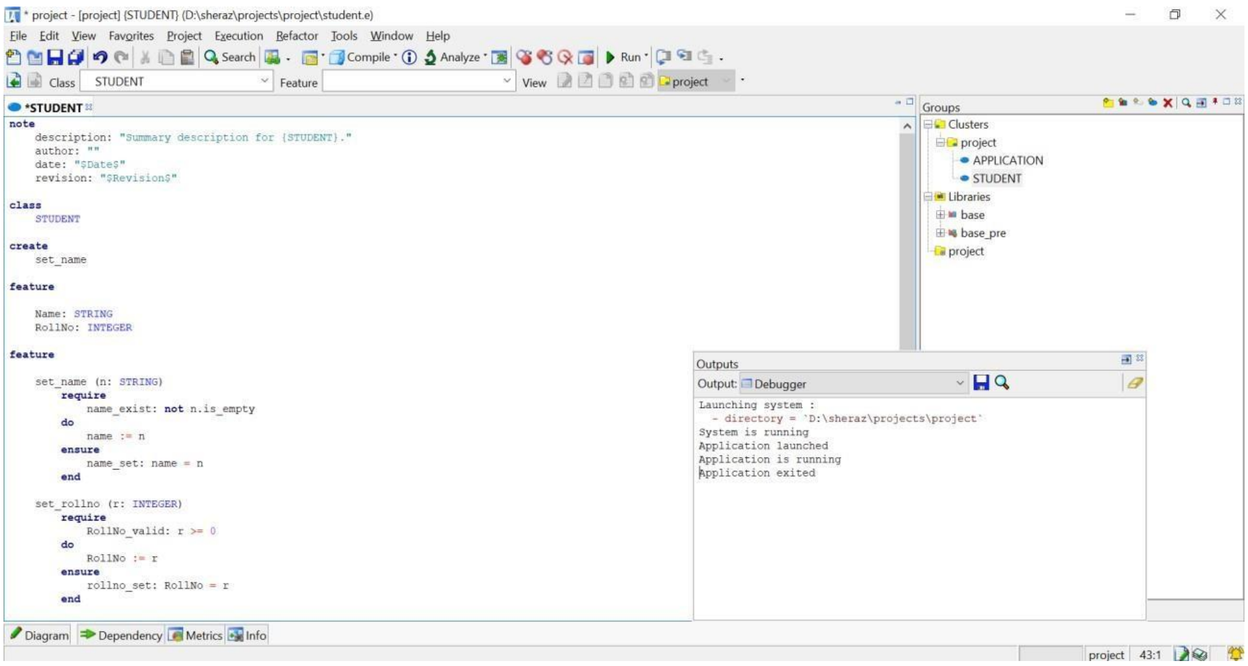
Code with DbC Implementation



```
note
    description: "session1 application root class"
    date: "$Date$"
    revision: "$Revision$"

class
    APPLICATION

inherit
    ARGUMENTS_32

create
    make

feature {NONE} -- Initialization

    make
            -- Run application.
        do
            --| Add your code here
            create s.set_name ("Sherry")
                   s.set_rollno (11)
                   print (s.rollno.out + " is " + s.name + "%N")
        end

feature
    s: STUDENT

end
```

```
Launching system :
  - directory = `D:\sheraz\projects\project`
System is running
Application launched
Application is running
Application exited
```

Fig. 5.4 Application Class (After DbC)



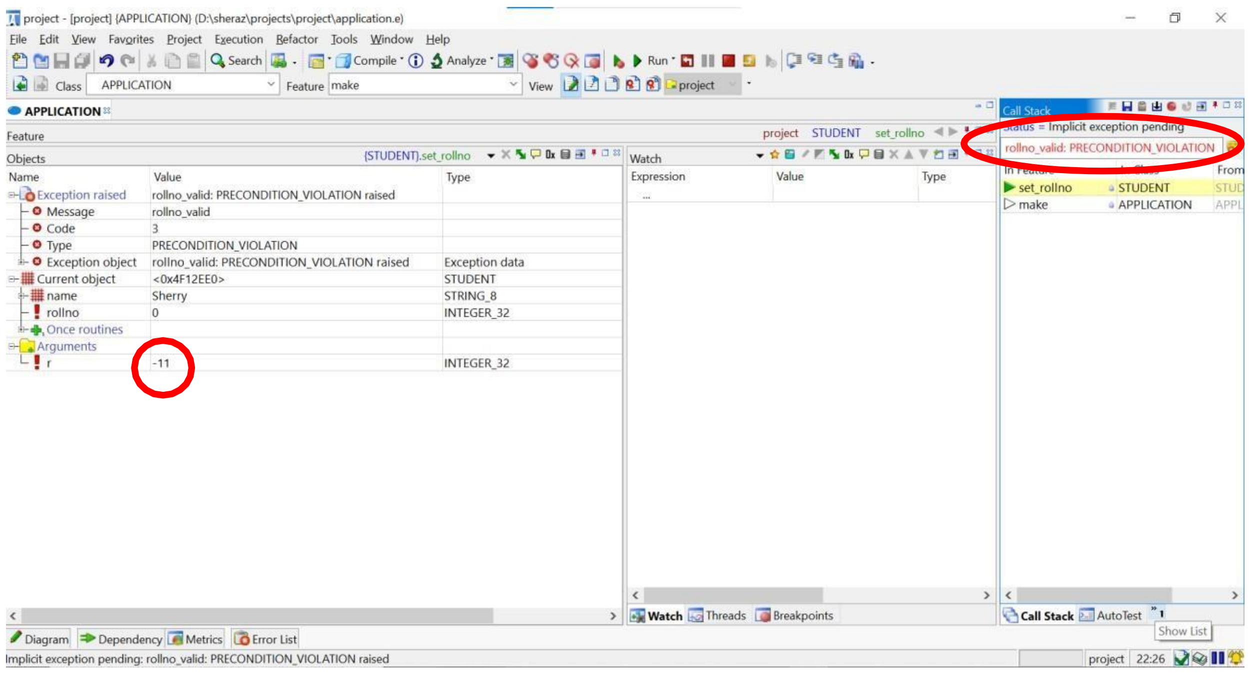
```
note
    description: "Summary description for {STUDENT}."
    author: ""
    date: "$Date$"
    revision: "$Revision$"

class
    STUDENT

create
    set_name

feature

    Name: STRING
    RollNo: INTEGER

feature

    set_name (n: STRING)
        require
            name_exist: not n.is_empty
        do
            name := n
        ensure
            name_set: name = n
        end

    set_rollno (r: INTEGER)
        require
            RollNo_valid: r >= 0
        do
            RollNo := r
        ensure
            rollno_set: RollNo = r
        end
```

```
Launching system :
  - directory = `D:\sheraz\projects\project`
System is running
Application launched
Application is running
Application exited
```

Fig. 5.5 Student Class (After DbC)

Here, in Student Class, new keywords can be seen. These keywords "*require*" and "*ensure*" represents the precondition and postcondition respectively. The statement after the *require* keyword represnts the precondition that needs to be met and the statement after keyword :ensure:

gurantees the generation of correct output. Here, now if we give the invalid input like earlier, condition violation would be generated.


project - [project] {APPLICATION} (D:\sheraz\projects\project\application.e)

Class APPLICATION Feature make

APPLICATION

Objects {STUDENT}.set_rollno

Name	Value	Type
Exception raised	rollno_valid: PRECONDITION_VIOLATION raised	
Message	rollno_valid	
Code	3	
Type	PRECONDITION_VIOLATION	
Exception object	rollno_valid: PRECONDITION_VIOLATION raised	Exception data
Current object	<0x4F12EE0>	STUDENT
name	Sherry	STRING_8
rollno	0	INTEGER_32
Once routines		
Arguments		
r	-11	INTEGER_32

project STUDENT set_rollno

Call Stack

Status = Implicit exception pending

rollno_valid: PRECONDITION_VIOLATION

In Feature	In Class	From
set_rollno	STUDENT	STUD
make	APPLICATION	APPL

Implicit exception pending: rollno_valid: PRECONDITION_VIOLATION raised

Fig. 5.6 DbC Violation

Here, it can be observed that precondition violation is generated as the value of roll number is given as negative number which is invalid. This is how, the program is prevented from generating invalid outputs as each student must have name, any student cannot have student field empty. Similarly, the roll numbe rof any student cannot be a negative number.

EXERCISE

1. Schedule the time of a class using Eiffel program. Apply Design by Contract on time so that no invalid time can be assigned to class. Hours can be either in 24 hours format or 12 hours format.

MIN HOURS:

APPLICATION CLASS:

```
note
        description: "SCDlab2 application root class"
date: "$Date$"   revision: "$Revision$"

class
        APPLICATION

inherit
        ARGUMENTS_32
```

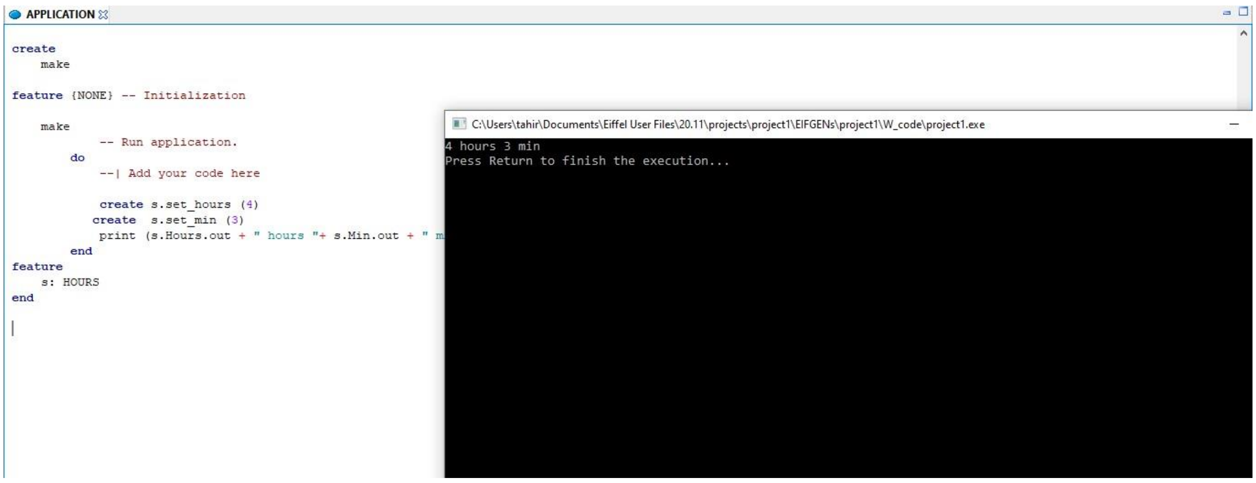
Kabeer Ahmed SE-19028

```
create
            make

feature {NONE} -- Initialization

            make
                                    -- Run application.
                        do
                                    --| Add your code here

     create s.set_hours (5)      create  s.set_min
(30)
                                    print (s.Hours.out + " hours "+ s.Min.out + " min "+"%N" )
                        end
feature
            s: HOURS
end
HOUR CLASS:
note
            description: "Summary description for {HOURS}."
            author: ""
date: "$Date$"
            revision: "$Revision$"

class
            HOURS
create
            set_hours
create
            set_min
feature
            Hours: INTEGER
Min: INTEGER feature

              set_hours (h: INTEGER_32)
require
                        Hours_valid: h >= 1 and h <= 12
              do
                        Hours := h
ensure
                        hours_set: Hours = h
end
              set_min(m: INTEGER_32)
require
                        Min_valid: m>=1 and m<=60
                        do                                    Min:=m
                        ensure
min_set: Min=m                        end
```

Kabeer Ahmed SE-19028

```
invariant
            hours_valid: Hours >= 2 and Hours <= 12
min_valid: Min >= 2 and Min <= 59 end
```

OUTPUT:


APPLICATION

```
create
    make

feature {NONE} -- Initialization

    make
            -- Run application.
        do
            --| Add your code here

            create s.set_hours (4)
           create  s.set_min (3)
            print (s.Hours.out + " hours "+ s.Min.out + " m
        end
feature
    s: HOURS
end
```

C:\Users\tahir\Documents\Eiffel User Files\20.11\projects\project1\EIFGENs\project1\W_code\project1.exe

```
4 hours 3 min
Press Return to finish the execution...
```

Lab Session 06

OBJECT: **Refactor the code in EiffelStudio**

THEORY

Code refactoring is defined as the process of restructuring computer code without changing or adding to its external behavior and functionality. The changes in existing source code preserve the software's behavior and functionality because the changes are so tiny that they are unlikely to create or introduce any new errors. Experts say that the goal of code refactoring is to turn dirty code into clean code, which reduces a project's overall technical debt. Dirty code is an informal term that refers to any code that is hard to maintain and update, and even more difficult to understand and translate.
This is the idea behind technical debt: if code is as clean as possible, it is much easier to change and improve in later iterations – so that your future self and other future programmers who work with the code can appreciate its organization.

Code Refactoring in EiffelStudio

Refactoring used to be a manual process, but new refactoring tools for common languages mean you can speed up the process a little bit. Still, it can be helpful to understand what the tool is actually doing.
EiffelStudio also has built-in refactoring tool which can be accessed from the Refactor tab in the software. Also, the Refactor Bar can also be added through View -> Toolbars -> Refactor Bar.

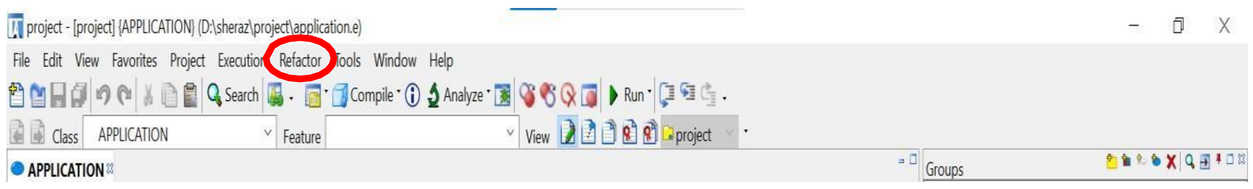

project - [project] (APPLICATION) (D:\sheraz\project\application.e)
File Edit View Favorites Project Execution Refactor Tools Window Help
Search Compile Analyze Run
Class APPLICATION Feature View project
APPLICATION
Groups

Fig. 6.1 Refactor Tab

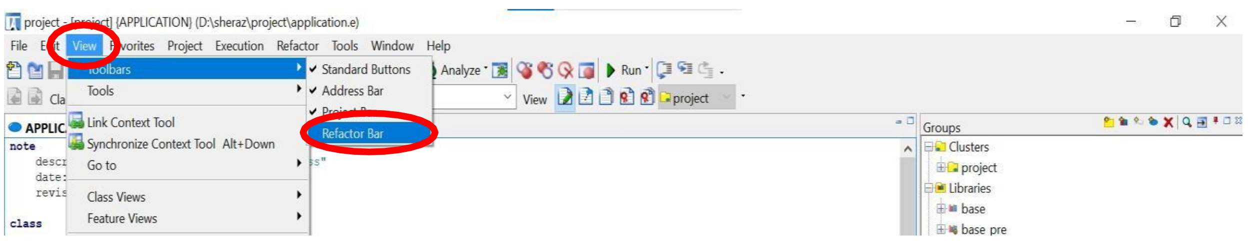


Fig. 6.2 Adding the Refactor Bar

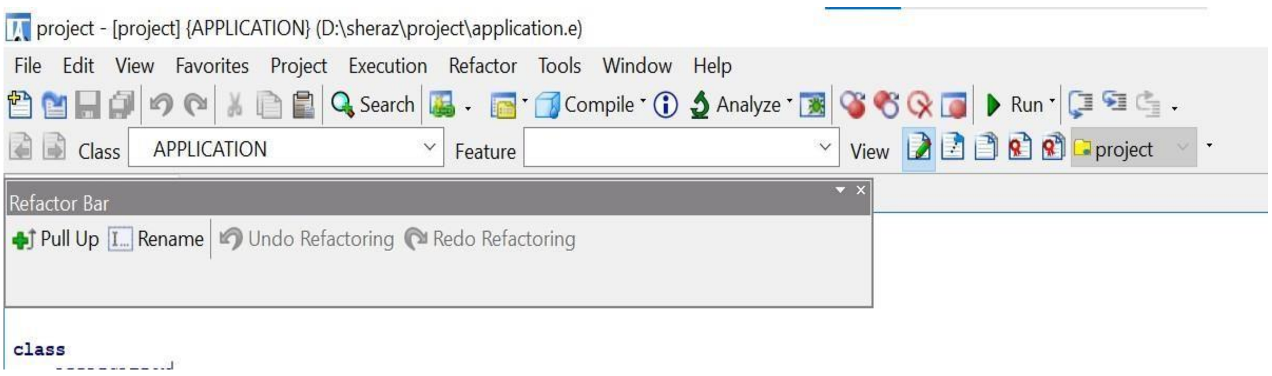

project - [project] {APPLICATION} (D:\sheraz\project\application.e)
File Edit View Favorites Project Execution Refactor Tools Window Help
Search Compile Analyze Run
Class APPLICATION Feature View project
Refactor Bar
Pull Up Rename Undo Refactoring Redo Refactoring
class

Fig. 6.3 Refactor Bar

Refactoring actions include renaming classes and features, and relocating features to an ancestor class. Refactoring actions start with a compilation and also end with a compilation. Refactoring has a separate undo functionality which allows you to undo a refactoring action as long as no changes have been made to the classes that have been refactored.

Rename Class

1. Start the **Rename** refactoring action by either:
    a. Selecting **Refactor->Rename** from the context menu associated with the class you want to rename.
    b. Picking the class and dropping its pebble into the **Rename** hole () on the refactoring toolbar.
2. After a compilation the **Refactoring: Class Rename** dialog appears. You can enter a new name.
3. You can choose between these options:
    a. **Compiled Classes:** only to the refactoring in compiled classes.
    b. **All Classes:** do the refactoring in all classes, even in uncompiled classes.
    c. **Rename File:** rename the file of the class to correspond with the class name, if another file with this name already exists the file will not be renamed and a warning will be given.
    d. **Replace Name in Comments:** replace usage of the class name in comments that follow the **{CLASSNAME}** syntax.
    e. Replace Name in Strings: replace usage of the class name in strings that follow the **{CLASSNAME}** syntax.
    f. **Reuse Existing Name:** allow two or more classes to have the same name during refactoring (note: the compiler will still report an error, if there are two or more classes conflicting with same name, but this option could be use to merge two classes).
4. Click **OK**.

Rename Feature

1. Start the **Rename** feature refactoring action by either:
    a. Selecting **Refactor->Rename** from the context menu associated with the feature you want to rename.
    b. Picking the feature and dropping its pebble into the **Rename** hole (I...) on the refactoring toolbar.
2. After a compilation the **Refactoring: Feature Rename** dialog appears. You can enter a new name.
3. You can choose between these options:
    a. **Replace Name in Comments:** replace usage of the feature name in comments that follow the **featurename** syntax.
    b. **Replace Name in Strings:** replace usage of the feature name in strings that follow the **featurename** syntax.
    c. **Reuse ExistingName:** allow two or more features to have the same name during refactoring (note: the compiler will still report an error, if there are two or more features with same name, but this option could be used to merge two features).
4. Click **OK**.

In lab, we have performed the **Refactoring: Feature Rename** on **Students Details Scenario** shown in previous lab session.

EXERCISE

1. In the exercise of previous session, create two classes; one with 24 hours format and other with 12 hours format. After that, use the “reuse existing name” option for renaming the class and show the results.

LAB # 06: (Q#01)
APPLICATION CLASS:
Code: (Before refactoring)

```
note
 description: "SCDlab2 application root class"
date: "$Date$"
 revision: "$Revision$"

class
 APPLICATION

inherit
```

Kabeer Ahmed SE-19028

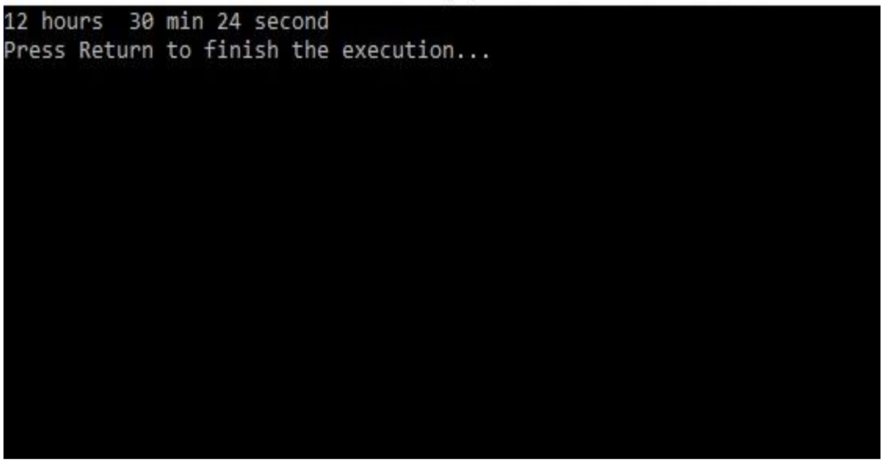
```
 ARGUMENTS_32

create
make

feature {NONE} -- Initialization

 make
                    -- Run application.
          do
                    --| Add your code here

     create s.set_hours (12)
s.set_min (30)
                       s.set_sec (24)
                    print (s.Hours.out + " hours " + s.Min.out + " min " + s.Sec.out +" second " +
"%N" )
          end
feature  s:
HOURS
end
```

<u>HOUR CLASS:</u>
<u>CODE: (Before Refactoring)</u>

```
note
          description: "Summary description for {HOURS}."
          author: ""
          date: "$Date$"
          revision: "$Revision$"

class
          HOURS
create
          set_hours
create
          set_min
create
          set_sec
feature
          Hours: INTEGER
          Min: INTEGER
          Sec: INTEGER
feature

            set_hours (h: INTEGER)
            require
```

```
                    Hours_valid: h >= 1 and h <= 12
do
                    Hours := h
ensure
                    hours_set: Hours = h
            end

            set_min (m: INTEGER)
            require
                    Min_valid: m >= 1 and m <= 59
            do              Min := m
            ensure
          min_set: Min = m
            end

            set_sec (s: INTEGER)
            require
                    Sec_valid: s >= 1 and s <= 60
            do              Sec := s
            ensure
          sec_set: Sec = s
            end
invariant
          Hours_valid: Hours >= 1 and Hours <= 12
          Min_valid: Min >= 1 and Min <= 59
          Sec_valid: Sec >= 1 and Sec <= 60 end
```

OUTPUT:



```
12 hours  30 min 24 second
Press Return to finish the execution...
```

APPLICATION CLASS: **Code:**
(After Refactoring) note

```
        description: "SCDlab2 application root class"
        date: "$Date$"
        revision: "$Revision$"

class
        APPLICATION

inherit
        ARGUMENTS_32

create make
        featur
        e
        {NON
        E} --
        Initiali
        zatio
        n

        make
                        -- Run application.
                do
                        --| Add your code here

    create s.set_hours (12)          s.set_min
(30)
                          s.set_sec (24)
  print (s.Hours.out + " hours " + s.Min.out + " min " + s.Sec.out +" second " + "%N" )
                end
feature
        s: HOURS_24_FORMAT
end
```

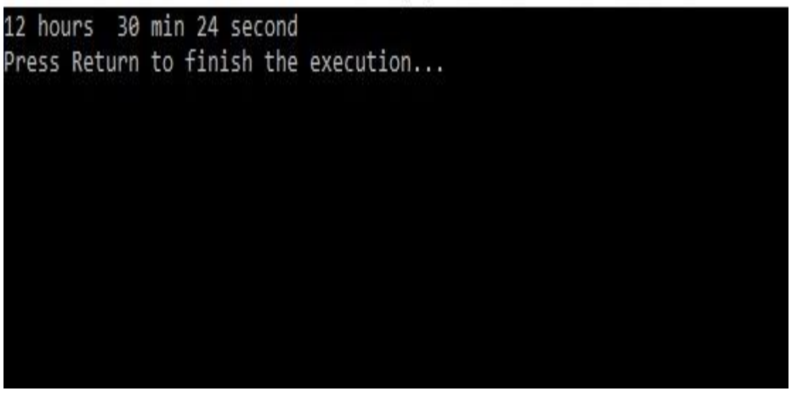
Kabeer Ahmed SE-19028

HOUR CLASS: (After Refactoring Named As "HOURS_24_FORMAT")
CODE: (After Refactoring)

```
note
        description: "Summary description for {HOURS_24_FORMAT}."
        author: ""      date: "$Date$"
        revision: "$Revision$"

class
        HOURS_24_FORMAT
create
        set_hours
create
        set_min
create
        set_sec feature
        Hours: INTEGER
        Min: INTEGER
        Sec: INTEGER
feature

          set_hours (h: INTEGER)
          require
                 Hours_valid: h >= 1 and h <= 24
          do
                 Hours := h
ensure                  hours_set:
Hours = h
          end

          set_min (m: INTEGER)
        require
                 Min_valid: m >= 1 and m <= 59
          do             Min := m
          ensure
        min_set: Min = m
          end

          set_sec (s: INTEGER)
          require
                 Sec_valid: s >= 1 and s <= 60
          do             Sec := s
          ensure
        sec_set: Sec = s
          end
invariant
        Hours_valid: Hours >= 1 and Hours <= 24
```

```
Min_valid: Min >= 1 and Min <= 59
Sec_valid: Sec >= 1 and Sec <= 60 end
```

OUTPUT:



```
12 hours  30 min 24 second
Press Return to finish the execution...
```

2. Rename the Hours, Minutes and Seconds features with hr, min and sec. Show the results

3. Again, choose the "reuse existing name" for renaming the feature and verify that the program is still giving the correct output and not throwing any error.

LAB # 06: (Q#02 & Q#03)
APPLICATION CLASS:
Code: (Before refactoring)

```
note
 description: "SCDlab2 application root class"
date: "$Date$"
 revision: "$Revision$"

class
 APPLICATION

inherit
 ARGUMENTS_32

create
make

feature {NONE} -- Initialization
```

```
 make
                    -- Run application.
          do
                    --| Add your code here

    create s.set_hours (12)
s.set_min (30)
                       s.set_sec (24)
                    print (s.Hours.out + " hours " + s.Min.out + " min " + s.Sec.out +" second " +
"%N" )
          end
feature
 s: HOURS_24_FORMAT
end
```

<u>HOUR_24_FORMAT CLASS:</u>
<u>CODE: (Before Refactoring)</u>

```
note
          description: "Summary description for {HOURS_24_FORMAT}."
          author: ""       date: "$Date$"
          revision: "$Revision$"

class
          HOURS_24_FORMAT
create
          set_hours
create
          set_min
create
          set_sec
feature
          Hours: INTEGER
          Min: INTEGER
          Sec: INTEGER
feature

            set_hours (h: INTEGER)
            require
                    Hours_valid: h >= 1 and h <= 24
            do
```

```
                    Hours := h
ensure                    hours_set:
Hours = h
            end

            set_min (m: INTEGER)
require
                    Min_valid: m >= 1 and m <= 59
            do              Min := m          ensure
                    min_set: Min = m       end

            set_sec (s: INTEGER)
require
                    Sec_valid: s >= 1 and s <= 60
            do
          Sec := s
ensure
                    sec_set: Sec = s
end
invariant
          Hours_valid: Hours >= 1 and Hours <= 24
          Min_valid: Min >= 1 and Min <= 59
          Sec_valid: Sec >= 1 and Sec <= 60 end
```

<u>APPLICATION CLASS:</u> <u>Code:</u>
<u>(After Refactoring)</u> note

```
          description: "SCDlab2 application root class"
          date: "$Date$"
          revision: "$Revision$"

class
          APPLICATION

inherit
          ARGUMENTS_32

create
          make
```

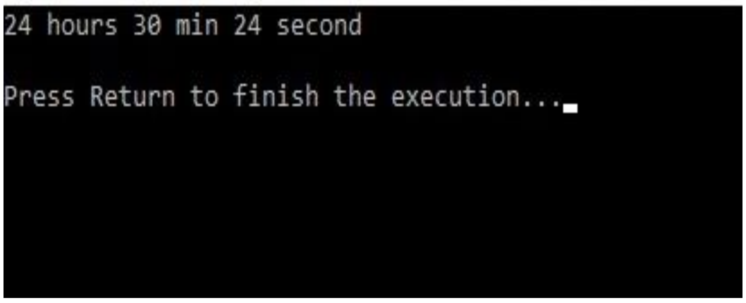
```
feature {NONE} -- Initialization

	make
			-- Run application.
		do
			--| Add your code here

		   create s.set_hours (12)
			 s.set_min (30)
			   s.set_sec (24)
			  print (s.hr.out + " hours " + s.Min.out + " min " + s.Sec.out +" second " +
"%N" )
		end
feature
	s: HOURS_24_FORMAT
end
```

HOUR_24_FORMAT CLASS:
CODE: (After Refactoring)

```
note
	description: "Summary description for {HOURS_24_FORMAT}."
	author: ""	date: "$Date$"
	revision: "$Revision$"

class
	HOURS_24_FORMAT create
	set_hours
create
	set_min
create
	set_sec
feature
	hr: INTEGER
	Min: INTEGER
	Sec: INTEGER
feature

	set_hours (h: INTEGER)
		require
			Hours_valid: h >= 1 and h <= 24
		do
			hr := h
	ensure
			hours_set: hr = h
		end
```

```
    set_min (m: INTEGER)
        require
            Min_valid: m >= 1 and m <= 59
        do
            Min := m
    ensure
            min_set: Min = m
        end

    set_sec (s: INTEGER)
        require
            Sec_valid: s >= 1 and s <= 60
        do
            Sec := s
    ensure
            sec_set: Sec = s
    end

invariant
    Hours_valid: hr >= 1 and hr <= 12
    Min_valid: Min >= 1 and Min <= 59
    Sec_valid: Sec >= 1 and Sec <= 60

end
```

OUTPUT:



```
24 hours 30 min 24 second

Press Return to finish the execution...
```

Kabeer Ahmed SE-19028

Lab Session 07

OBJECT: <u>**Manual Test Creation in EiffelStudio**</u>

THEORY

Developers test software in the hope that the testing process will expose faults in the software they've developed. Most developers also realize that no amount of testing will ever prove software to be bug free. So, while testing is a virtuous activity that we dare not neglect, we are wise to temper our expectation of the practical value of testing.
A test is designed to exercise a software element given certain inputs and execution state. The state is observed after the test execution to see if the software element has behaved in a manner that is consistent with its specification.
Some of today's development methods require tests to be written before the software elements they test. Then the tests are included as a part of the software specification. But tests can only reflect a very small subset of the possible execution cases. Testing can never replace a comprehensive software specification.
The great advantage you have with Eiffel, of course, is that the specification for a software element exists in its contract. With contract checking enabled at run time, the running software's behavior is constantly monitored against the contract's expectations.

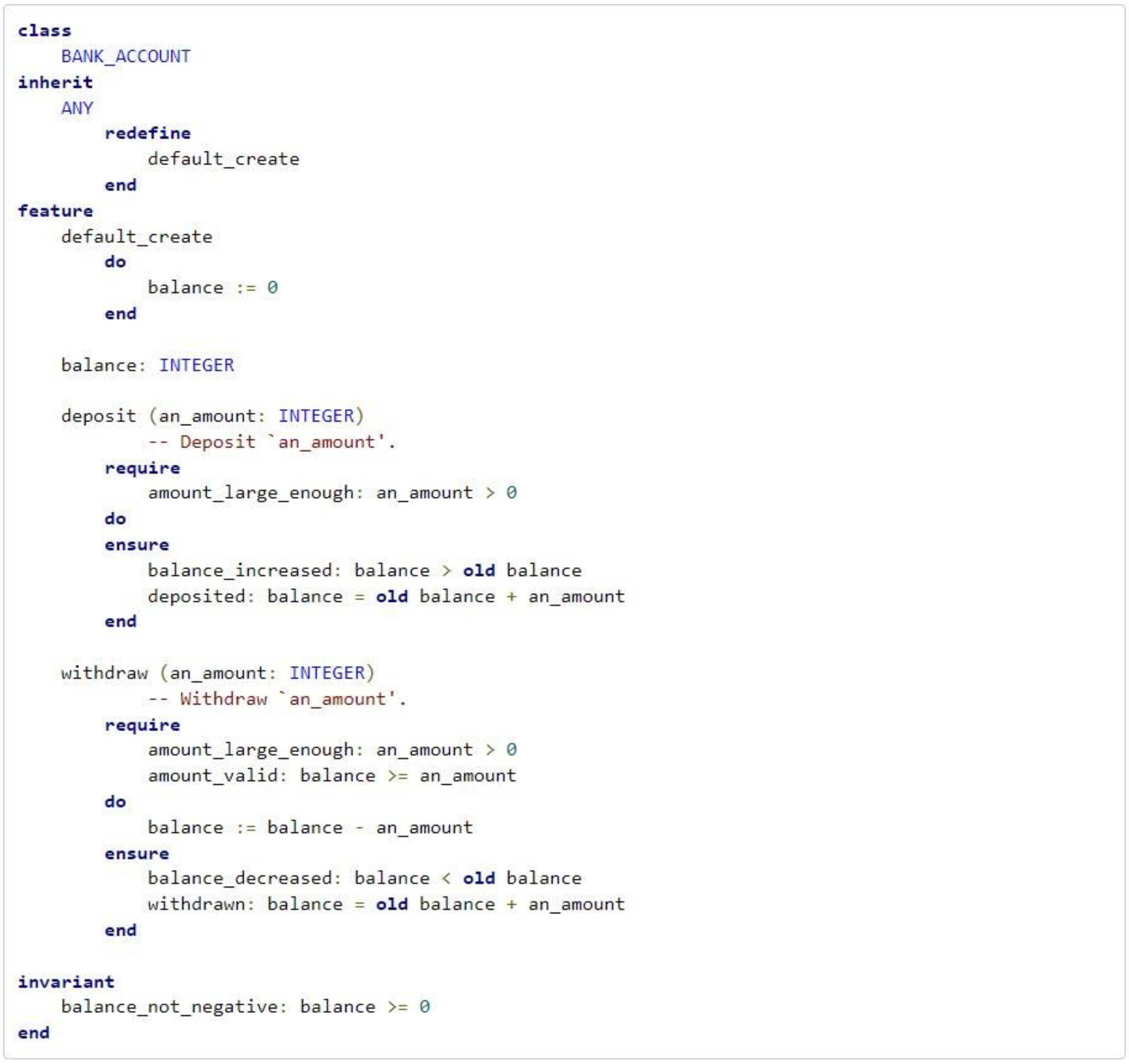
Create a Manual Test

For developing our manual test, let's use a simple system that contains a class modeling bank account. Here are two classes that will make up our system. The first, APPLICATION will be the root class of our system. APPLICATION really only serves to declare an attribute of type BANK_ACCOUNT, which is the class we will write a test against. APPLICATION looks like this:

```
class
    APPLICATION

inherit
    ARGUMENTS

create
    make

feature {NONE} -- Initialization

    make
            -- Run application.
        do
            create my_account
        end

    my_account: BANK_ACCOUNT

end
```

Fig 7.1 Application Class

And here's the class BANK_ACCOUNT:



```
class
    BANK_ACCOUNT
inherit
    ANY
        redefine
            default_create
        end
feature
    default_create
        do
            balance := 0
        end

    balance: INTEGER

    deposit (an_amount: INTEGER)
            -- Deposit `an_amount'.
        require
            amount_large_enough: an_amount > 0
        do
        ensure
            balance_increased: balance > old balance
            deposited: balance = old balance + an_amount
        end

    withdraw (an_amount: INTEGER)
            -- Withdraw `an_amount'.
        require
            amount_large_enough: an_amount > 0
            amount_valid: balance >= an_amount
        do
            balance := balance - an_amount
        ensure
            balance_decreased: balance < old balance
            withdrawn: balance = old balance + an_amount
        end

invariant
    balance_not_negative: balance >= 0
end
```

Fig. 7.2 Bank_Account Class

Getting to the AutoTest Interface

If the AutoTest interface is not on a tab next to Clusters, Features, and Favorites, you can invoke it by following the menu path:

```
View --> Tools --> AutoTest
```

Depending upon your version and platform, the AutoTest interface should look about like this:

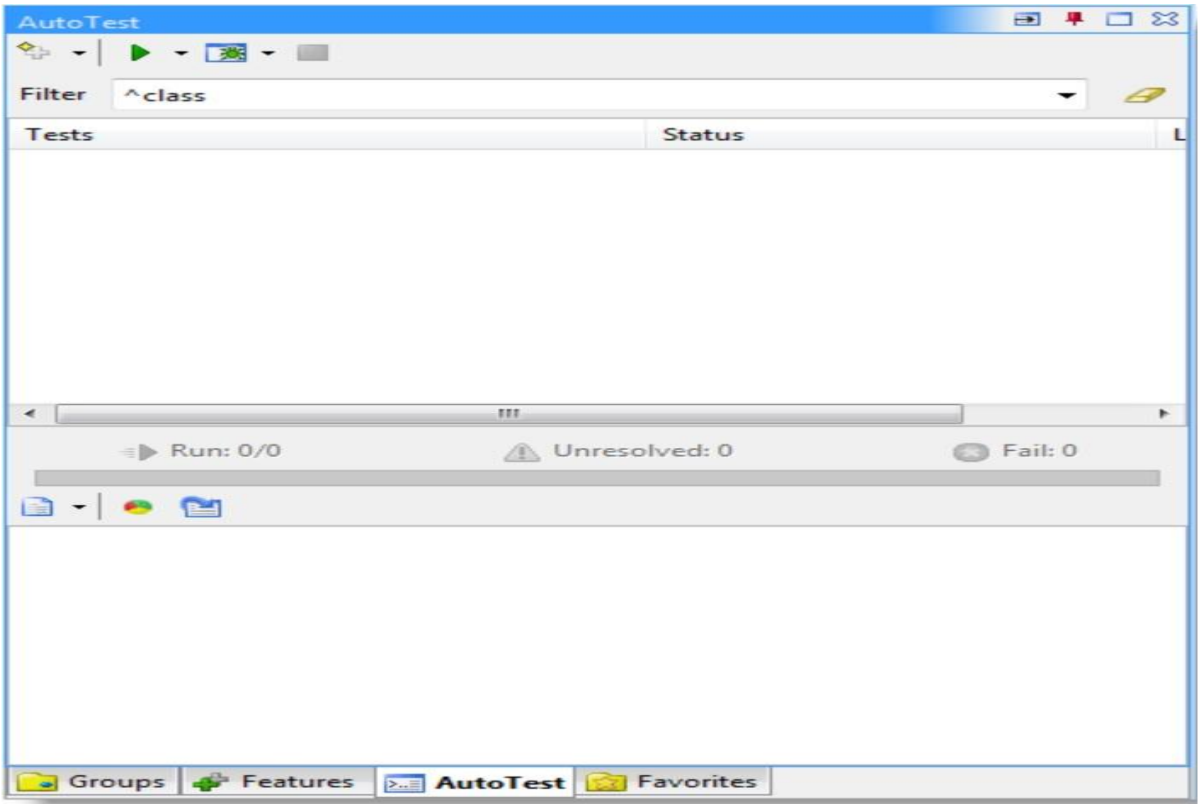


Fig. 7.3 Manual Test Window

Creating a New Test

To begin the process of creating a new test, click the Create New Test button ( ) on the interface's tool bar. When you click this button, by default AutoTest will set you up to create a new Manual test. To choose a different test type, click the small triangle to the right of the Create New Test button and you'll be presented with a drop-down menu of choices:

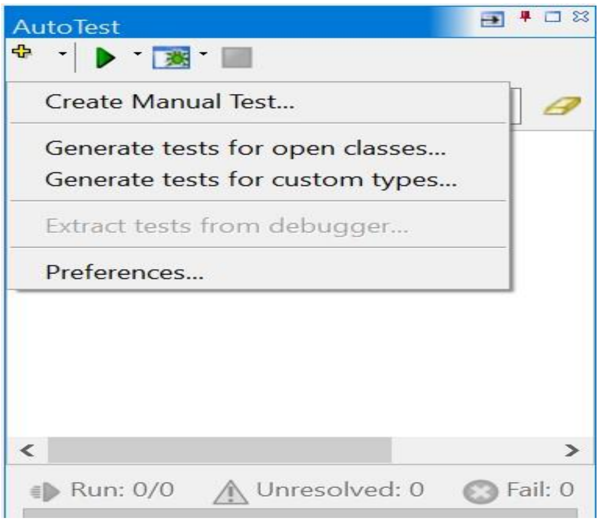
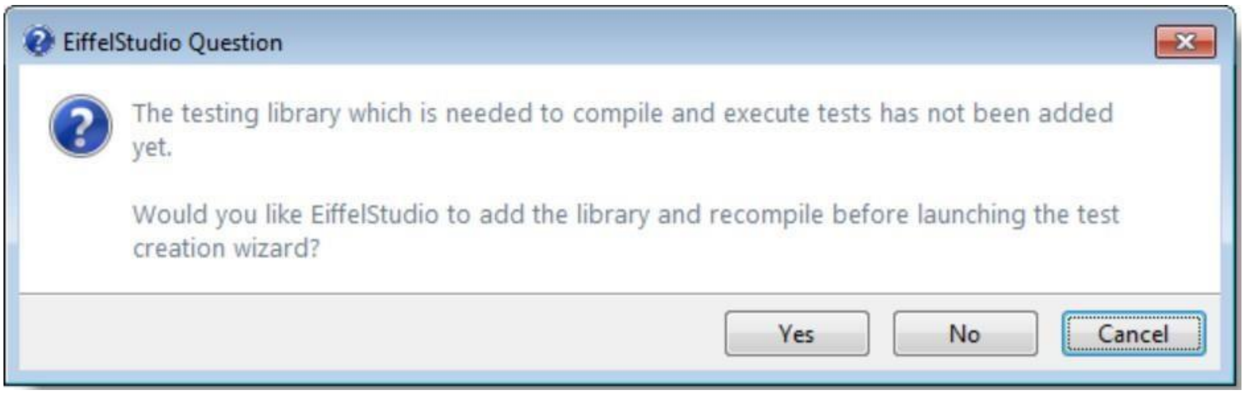

AutoTest
Create Manual Test...
Generate tests for open classes...
Generate tests for custom types...
Extract tests from debugger...
Preferences...
Run: 0/0
Unresolved: 0
Fail: 0

Fig. 7.4 Manual Test Creation

For now, let's select Create Manual Test.

If this is the first time you've used the testing tool for this project, it is likely that you will be presented with a dialog box asking if you want to add the testing library classes to your project and recompile:



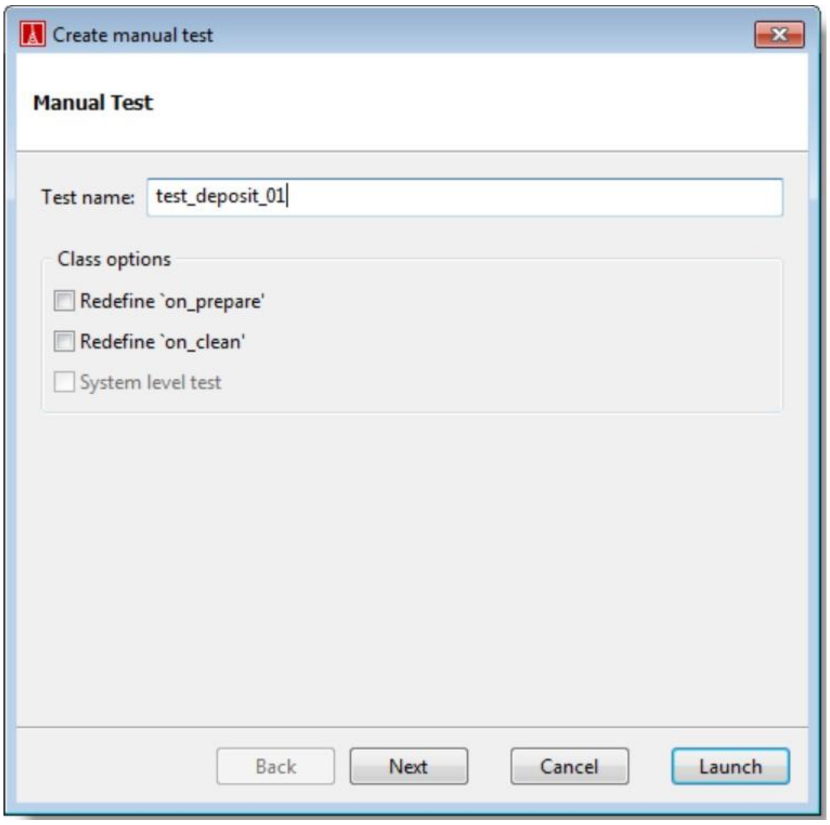
The testing library which is needed to compile and execute tests has not been added yet.

Would you like EiffelStudio to add the library and recompile before launching the test creation wizard?

Fig. 7.5 Libraries needs to be Compiled for first Time

The Manual Test Pane

After the compile completes, then the first pane of the New Eiffel Test Wizard appears. It's the Manual Test pane and should look like this:



Test name: test_deposit_01

☐ Redefine `on_prepare'
☐ Redefine `on_clean'
☐ System level test

Fig. 7.6 Test Pane

Another thing to notice before we click **Next**, is that at this point we could click **Launch**. **Launch** will immediately try to create the test with the information it has available. The idea is that if you are creating several similar tests, you can change the test routine name and leave the rest of the information as you had entered it on a previous test. This keeps you from having to traverse the wizard panes entering the same information repeatedly.
But in our case, we need to use the subsequent wizard panes, so let's click **Next**, to go to the next one.

The Tags Pane

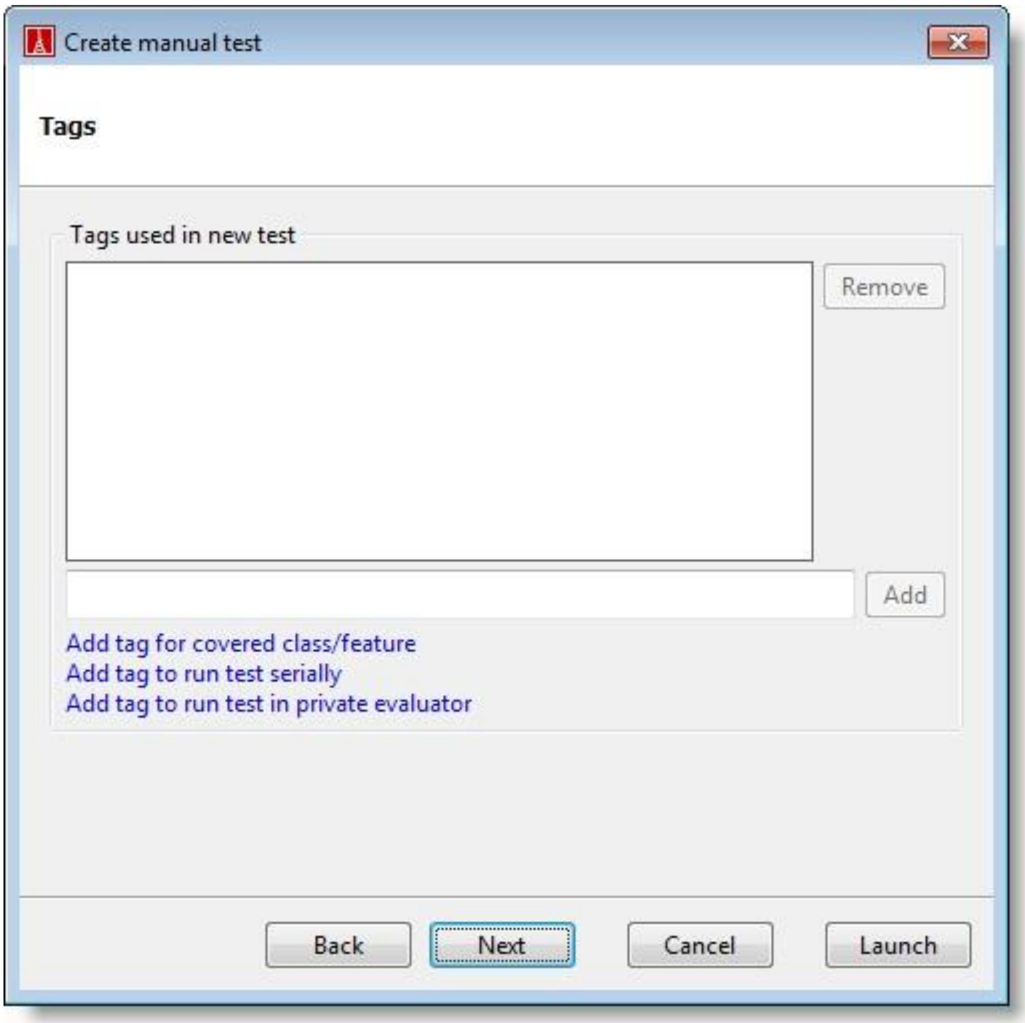

Create manual test
Tags
Tags used in new test
Remove
Add
Add tag for covered class/feature
Add tag to run test serially
Add tag to run test in private evaluator
Back
Next
Cancel
Launch

Fig. 7.7 Tag Pane

With this pane, you identify tags for your test that allow you to manage your test set more easily in the future. For this test, we will include only a tag that identifies the class and feature covered by the test. To do this we click **Add tag for covered class/feature**. When we do, we are presented with a dialog in which we can choose a class and feature.

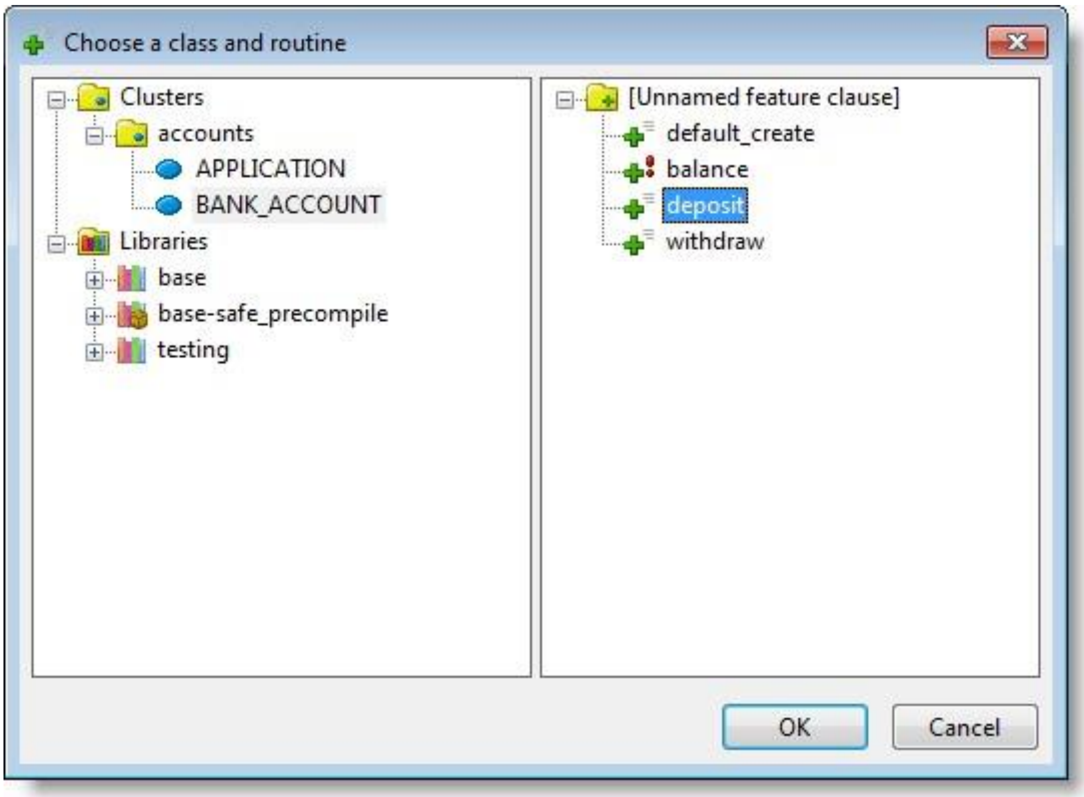
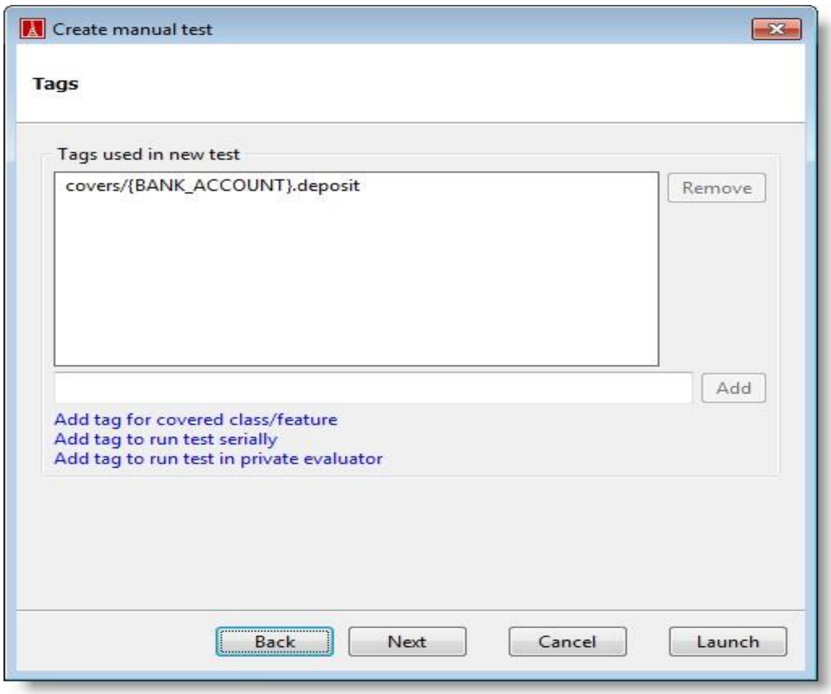


Fig 7.8 Class/Routine for Implementation of Test

We'll choose class BANK_ACCOUNT and feature deposit, click **OK**.
Now you should see the coverage tag in the list of **Tags used in new test**.

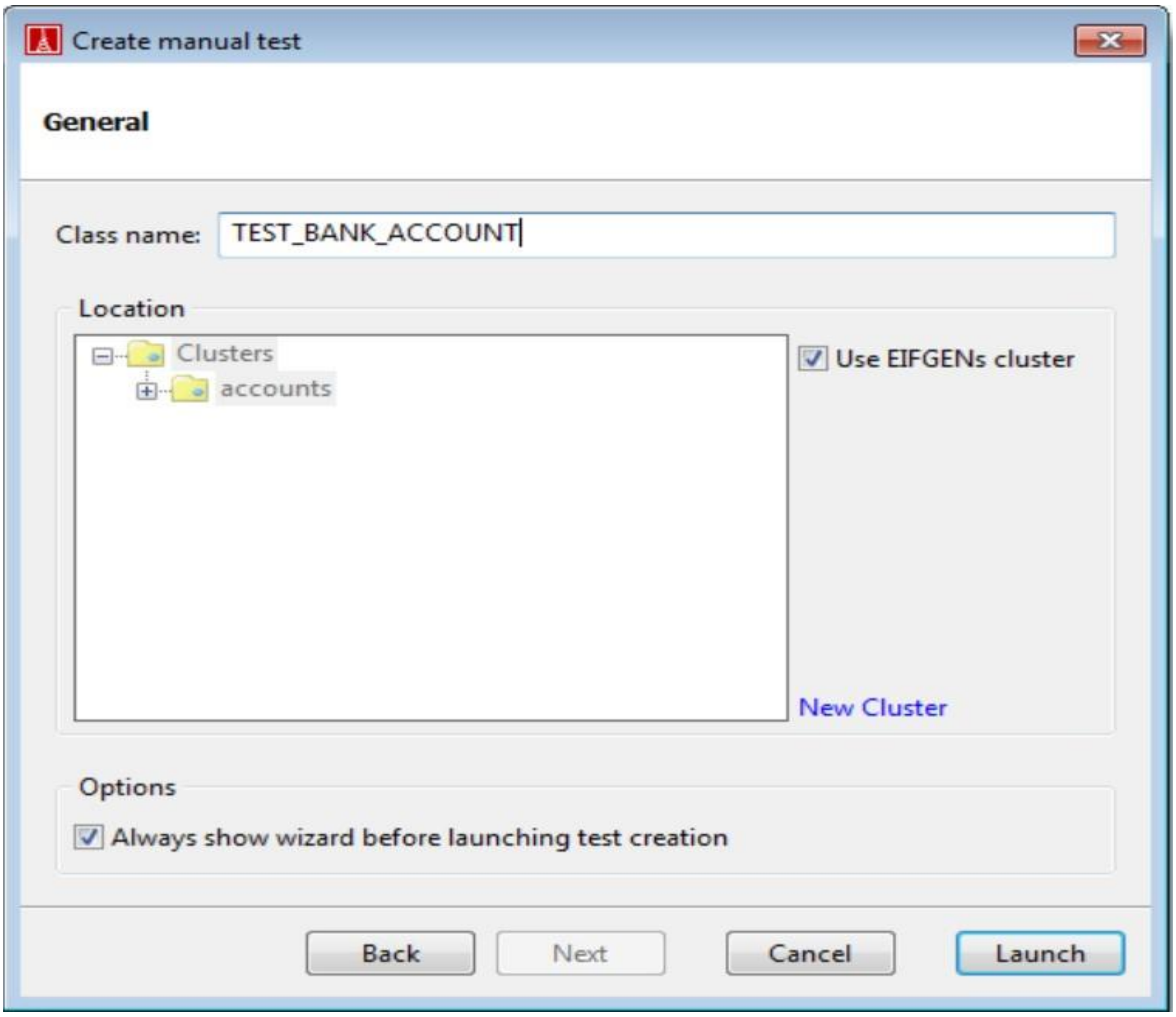

Create manual test
Tags
Tags used in new test
Remove
Add
Add tag for covered class/feature
Add tag to run test serially
Add tag to run test in private evaluator
Back
Next
Cancel
Launch

covers/{BANK_ACCOUNT}.deposit

Fig. 7.9 Chosen Feature

That takes care of adding our coverage tag, so let's click **Next** to go to the next wizard pane, the **General** pane.

The General Pane



Class name: TEST_BANK_ACCOUNT

☑ Use EIFGENs cluster

☑ Always show wizard before launching test creation

Fig. 7.10 General Pane

We will use this wizard pane to name our test class and let AutoTest know where we want the test class to reside. You can give a test class any name you wish, as long as it doesn't conflict with another class name in your system. If you try to type in a class name that already exists, the wizard will let you know right away by changing the text color to red. There is a convention that has arisen around test class names. If possible, make the test class name the name of the target class, prefixed with TEST_. So in our case, we want to build a test against a feature of the BANK_ACCOUNT class, so we will name our test class TEST_BANK_ACCOUNT.

Now, for the question of where the tests should be kept.

By default, tests will be stored in a subdirectory of the EIGENs directory that is generated by the

Eiffel compiler. Because it's the default, it's the quickest, easiest way to house tests. But it may not be the best for you in the long run. For example, if you manually delete the EIFGENs directory, which is occasionally necessary, you will lose your tests.
You could include them in the same cluster as some of your application classes. But there are some advantages to keeping the test classes in a **test cluster** separate from your target classes. For example, it will be easier for you to deliver your application or library classes if the testing classes aren't mixed with your domain classes. A **test cluster** is just a cluster of classes that EiffelStudio and AutoTest expect to contain test classes. So, in our case, let's create a new testing cluster as a subcluster of the cluster in which the classes APPLICATION and BANK_ACCOUNT reside.
First, uncheck the box labeled Use **EIFGENs cluster**.
Notice the **New cluster** link on the General pane. We click that link to add a new test cluster. The **Add Cluster** dialog box appears:

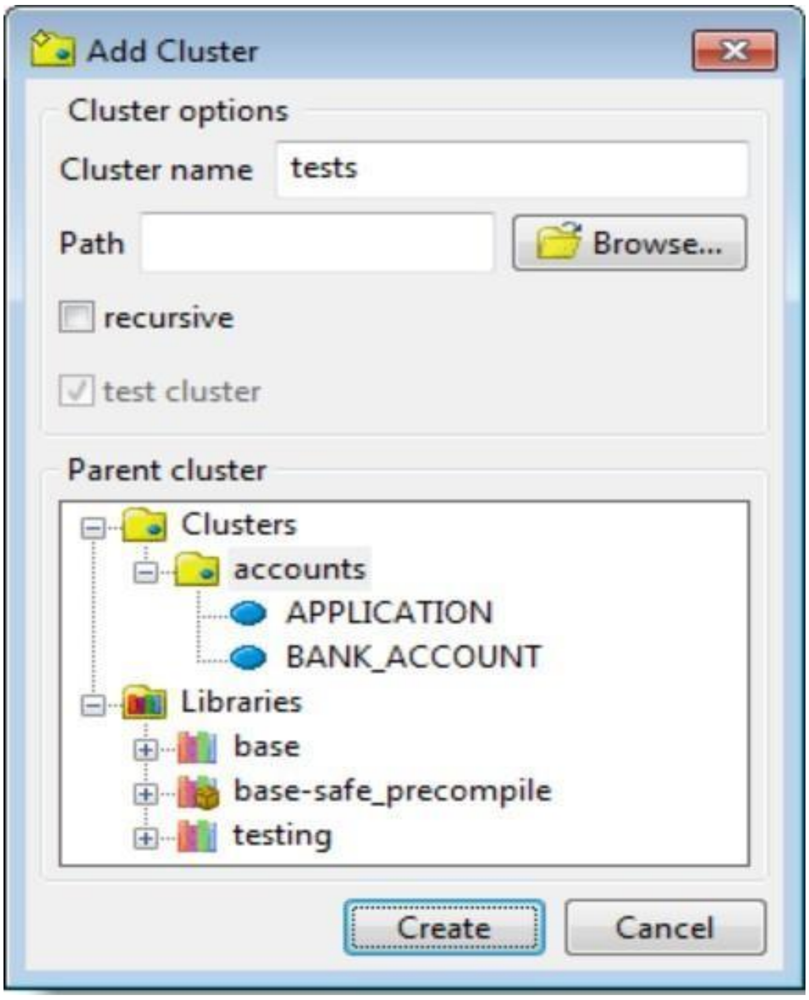
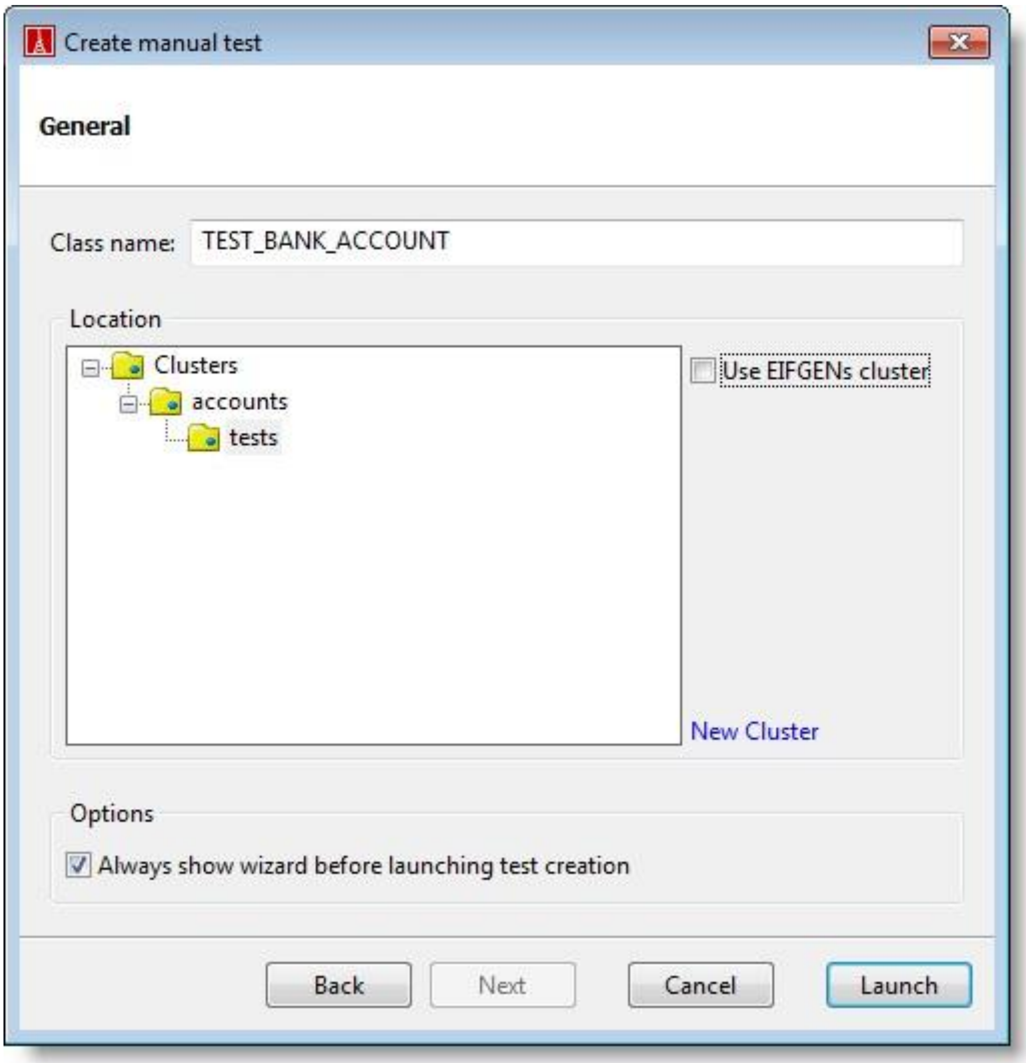

Add Cluster
Cluster options
Cluster name: tests
Path:
[ ] recursive
[✓] test cluster (disabled)
Parent cluster
- Clusters
  - accounts
    - APPLICATION
    - BANK_ACCOUNT
- Libraries
  - base
  - base-safe_precompile
  - testing

Fig. 7.11 Test Location and Cluster

We can name our test cluster tests, the default, and make it a subcluster to our root cluster accounts. Notice that there is a **test cluster** check box on the dialog. It is checked and disabled, so at this point in the wizard you would always create a test cluster. Let's also check the box labeled **recursive**. Once the test cluster is created, we're back to the General pane which now looks like this:



General

Class name: TEST_BANK_ACCOUNT

Location

- Clusters
  - accounts
    - tests

[ ] Use EIFGENs cluster

Options

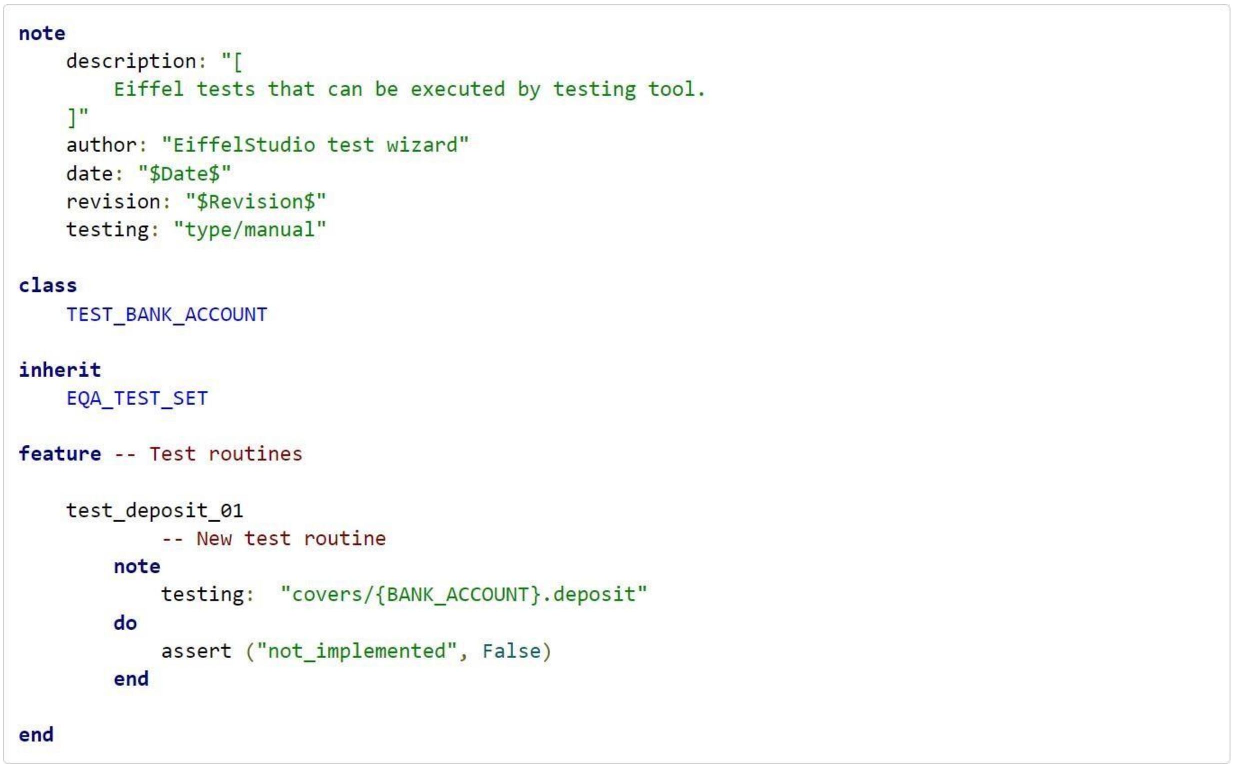
[✓] Always show wizard before launching test creation

Fig. 7.12 Test Class Creation

At this point we have provided all the information necessary for AutoTest to create the shell for a manual test on the deposit feature of the BANK_ACCOUNT class. So, now we click **Launch**, and AutoTest creates our test set and test.

Writing a Test

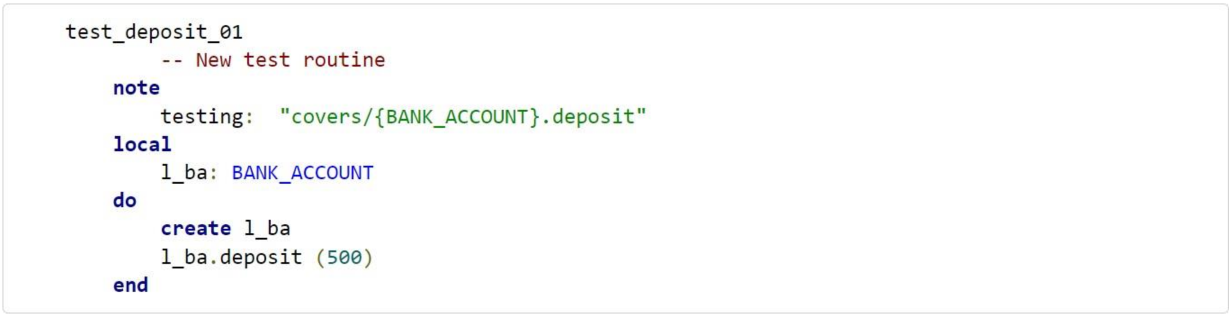
Let's look at the class TEST_BANK_ACCOUNT:



```
note
    description: "[
        Eiffel tests that can be executed by testing tool.
    ]"
    author: "EiffelStudio test wizard"
    date: "$Date$"
    revision: "$Revision$"
    testing: "type/manual"

class
    TEST_BANK_ACCOUNT

inherit
    EQA_TEST_SET

feature -- Test routines

    test_deposit_01
            -- New test routine
        note
            testing:  "covers/{BANK_ACCOUNT}.deposit"
        do
            assert ("not_implemented", False)
        end

end
```

Fig. 7.13 Test Class (Default)

We can see that the feature test_deposit_01 exists, but doesn't really test anything. So, let's change that. We'll alter test_deposit_01 so that it creates an instance of BANK_ACCOUNT and then makes a deposit to that account. So, test_deposit_01 now looks like this:



```
    test_deposit_01
            -- New test routine
        note
            testing:  "covers/{BANK_ACCOUNT}.deposit"
        local
            l_ba: BANK_ACCOUNT
        do
            create l_ba
            l_ba.deposit (500)
        end
```

Fig. 7.14 Test Class (Edited)

Now we have created and written a manual test using AutoTest.

EXERCISE

1. Create a manual test for withdraw feature in the same program performed in lab session. Show the results with default BANK_ACCOUNT and then after correcting the errors in the BANK_ACCOUNT class.

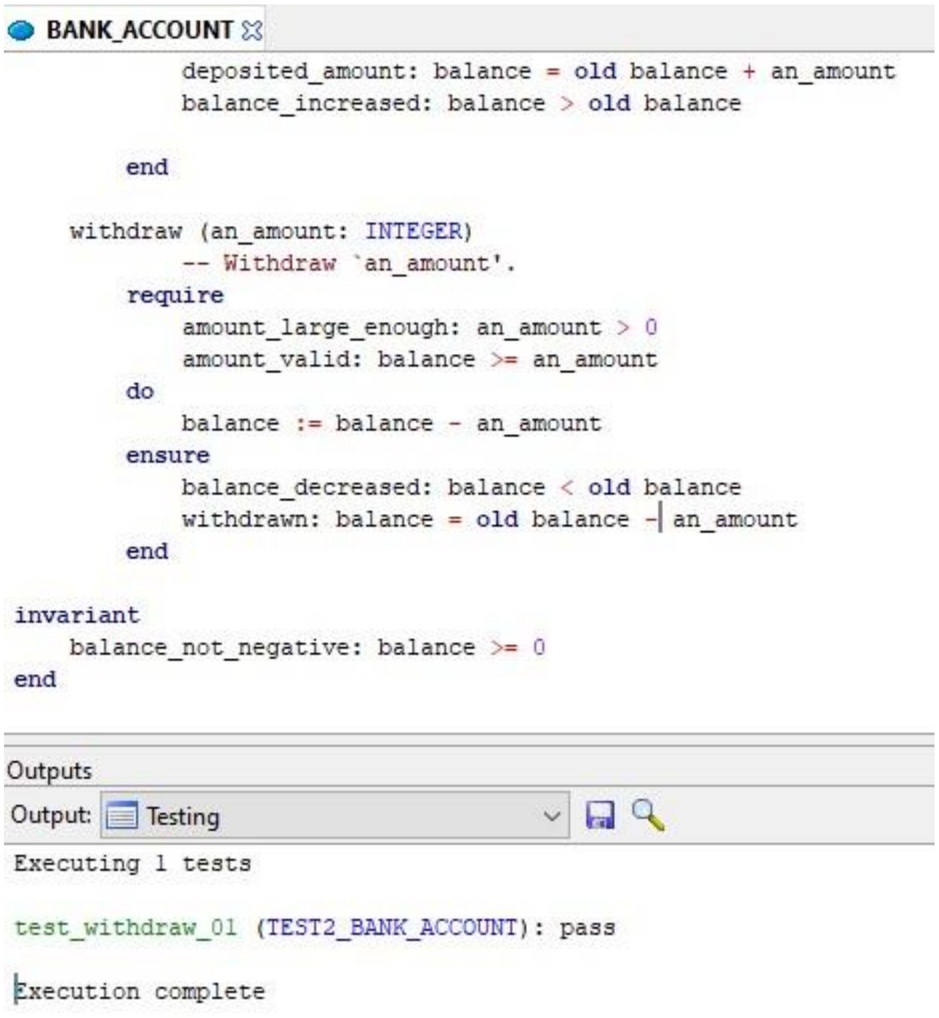
BANK_ACCOUNT WITHDRAW: class
   BANK_ACCOUNT

Kabeer Ahmed SE-19028

```
inherit
ANY
        redefine
default_create
        end feature
default_create
do
            balance :=0
        end

    balance: INTEGER

    deposit (an_amount: INTEGER)
-- Deposit `an_amount'.
        require
            amount_large_enough: an_amount > 0
do          ensure
            deposited: balance = old balance + an_amount
balance_increased: balance > old balance

        end

    withdraw (an_amount: INTEGER)
-- Withdraw `an_amount'.
        require
            amount_large_enough: an_amount > 0
            amount_valid: balance >= an_amount
do
            balance := balance - an_amount
ensure
            balance_decreased: balance < old balance
            withdrawn: balance = old balance - an_amount
end

invariant
    balance_not_negative: balance >= 0 end
```

Kabeer Ahmed SE-19028

OUTPUT:



BANK_ACCOUNT

```
            deposited_amount: balance = old balance + an_amount
            balance_increased: balance > old balance

        end

    withdraw (an_amount: INTEGER)
            -- Withdraw `an_amount'.
        require
            amount_large_enough: an_amount > 0
            amount_valid: balance >= an_amount
        do
            balance := balance - an_amount
        ensure
            balance_decreased: balance < old balance
            withdrawn: balance = old balance - an_amount
        end

invariant
    balance_not_negative: balance >= 0
end
```

Outputs

Output: Testing

```
Executing 1 tests

test_withdraw_01 (TEST2_BANK_ACCOUNT): pass

Execution complete
```

Lab Session 08

OBJECT: <u>Design Principles in Software Construction and Development</u>

THEORY

Design patterns represent the best practices used by experienced object-oriented software developers. Design patterns are solutions to general problems that software developers faced during software development. These solutions were obtained by trial and error by numerous software developers over quite a substantial period of time.

Singleton Pattern

Singleton pattern is one of the simplest design patterns in Java. This type of design pattern comes under creational pattern as this pattern provides one of the best ways to create an object. This pattern involves a single class which is responsible to create an object while making sure that only single object gets created. This class provides a way to access its only object which can be accessed directly without need to instantiate the object of the class.

Implementation

We're going to create a *SingleObject* class. *SingleObject* class have its constructor as private and have a static instance of itself.
SingleObject class provides a static method to get its static instance to outside world. *SingletonPatternDemo*, our demo class will use *SingleObject* class to get a *SingleObject* object.

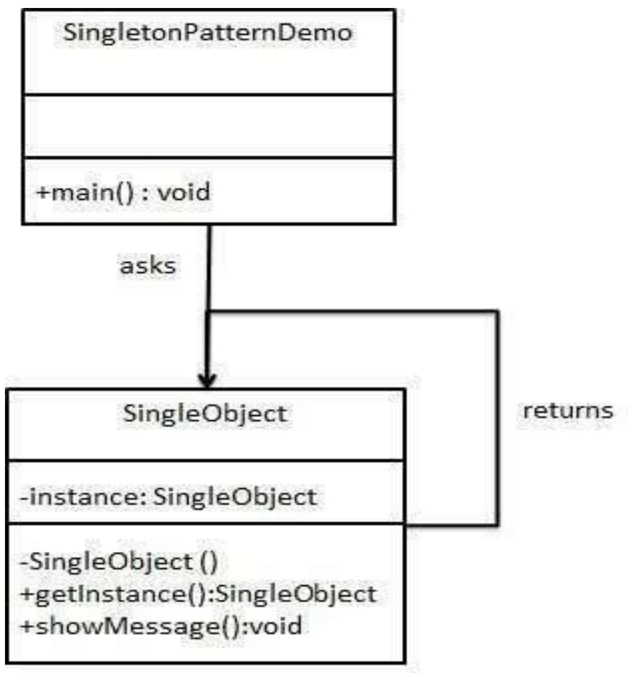
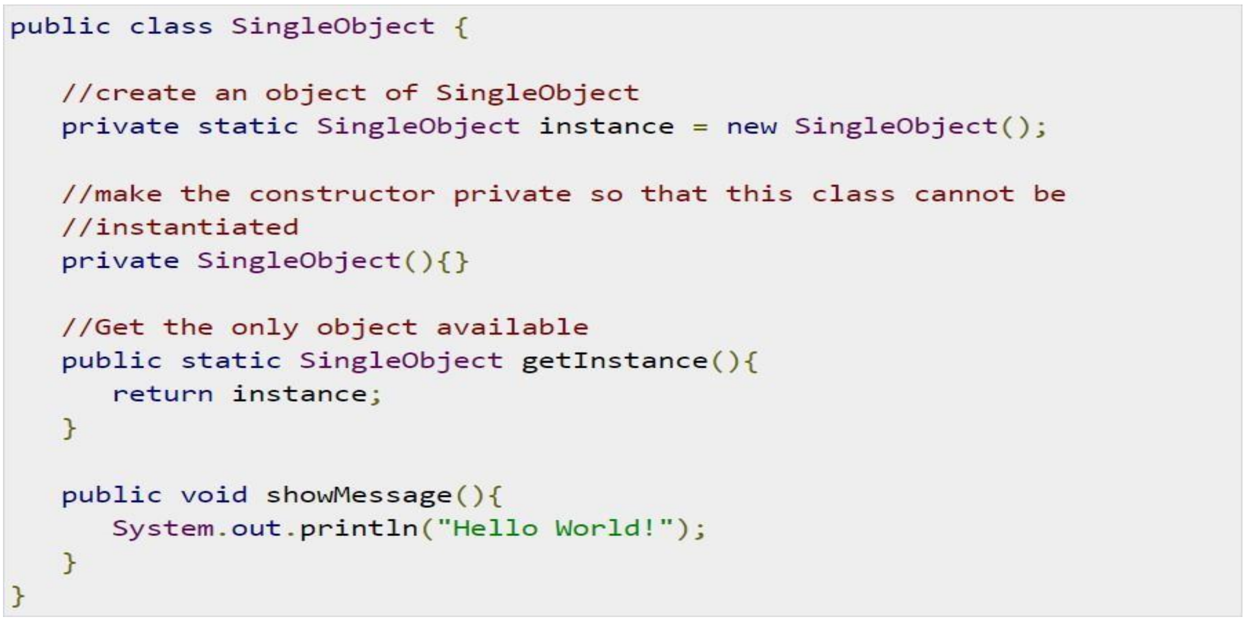


Fig 8.1 Singleton Pattern

Step 1

Create a Singleton Class.

SingleObject.java



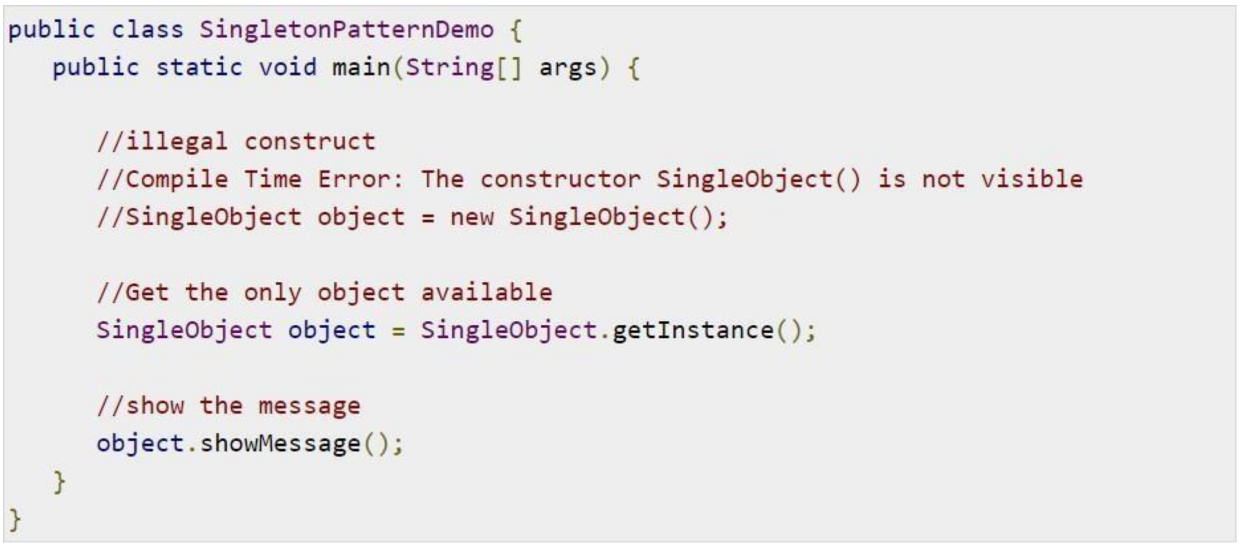
```java
public class SingleObject {

   //create an object of SingleObject
   private static SingleObject instance = new SingleObject();

   //make the constructor private so that this class cannot be
   //instantiated
   private SingleObject(){}

   //Get the only object available
   public static SingleObject getInstance(){
      return instance;
   }

   public void showMessage(){
      System.out.println("Hello World!");
   }
}
```

Fig. 8.2 SingleObject Class

Step 2

Get the only object from the singleton class.

SingletonPatternDemo.java



```java
public class SingletonPatternDemo {
   public static void main(String[] args) {

      //illegal construct
      //Compile Time Error: The constructor SingleObject() is not visible
      //SingleObject object = new SingleObject();

      //Get the only object available
      SingleObject object = SingleObject.getInstance();

      //show the message
      object.showMessage();
   }
}
```

Fig. 8.3 SingletonPatternDemo Class

Kabeer Ahmed SE-19028

Verification of Single Object

Following changes can be made to verify single object creation.

SingleObject.java

```
public class SingleObject {

   //create an object of SingleObject
   private static SingleObject instance = new SingleObject();
private int x;
   //make the constructor private so that this class cannot be
   //instantiated
   private SingleObject(){}

   //Get  the  only  object  available  public
   static SingleObject getInstance(){ return
   instance;
   }
   public void setx(int
   a){ x=a; }
   public int getx(){ return
       x;
   }

   public void showMessage(){
      System.out.println("Hello World!");
   }
}
```

SingletonPatternDemo.java

```
public  class  SingletonPatternDemo  {  public
   static void main(String[] args) {

      //illegal construct
      //Compile Time Error: The constructor SingleObject() is not visible
      //SingleObject object = new SingleObject();

      //Get the only object available
      SingleObject object1 = SingleObject.getInstance(); SingleObject
      object2 = SingleObject.getInstance(); object1.setx(3);

System.out.println(object2.getx()); //show
      the message
```

```
    //object.showMessage();
  }
}
```

EXERCISE

1. Create a function that perform some arithmetic operation on “x” value set by setx() function and observe the result verifying single object creation.

LAB # 08:
CODE:
SingleObject.java

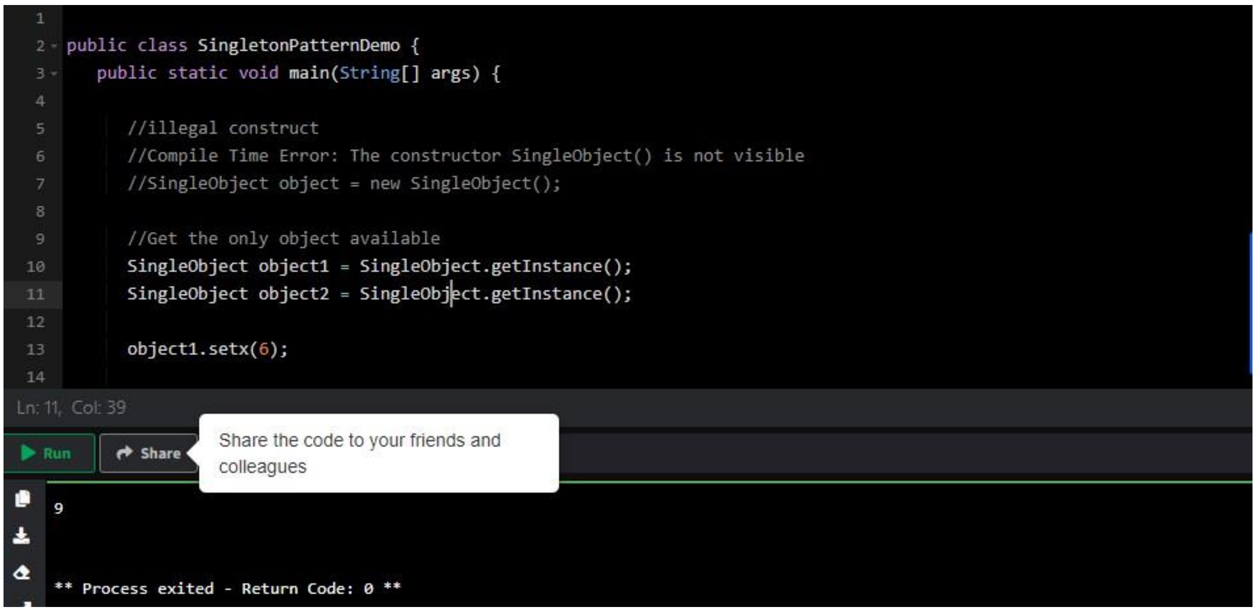
```
public class SingleObject {

   //create an object of SingleObject
   private static SingleObject instance = new SingleObject(); private
int x;
   //make the constructor private so that this class cannot be
   //instantiated
   private SingleObject(){}

   //Get the only object available    public
static SingleObject getInstance(){
      return instance;
   }
   public void setx(int a){
       x=a;
   }
   public int getx(){
       return x+3;
   }
   public int sum( int a){
       return a;
   }

   public void showMessage(){
System.out.println("Hello World!");
   }
}
```

Kabeer Ahmed SE-19028

SingletonPatternDemo.java

```
public class SingletonPatternDemo {
public static void main(String[] args) {

      //illegal construct
      //Compile Time Error: The constructor SingleObject() is not visible
      //SingleObject object = new SingleObject();

      //Get the only object available
      SingleObject object1 = SingleObject.getInstance();
      SingleObject object2 = SingleObject.getInstance();

      object1.setx(6);



System.out.println(object2.getx());
      //show the message
      //object.showMessage();
   }
}
```

OUTPUT



```
public class SingletonPatternDemo {
   public static void main(String[] args) {

      //illegal construct
      //Compile Time Error: The constructor SingleObject() is not visible
      //SingleObject object = new SingleObject();

      //Get the only object available
      SingleObject object1 = SingleObject.getInstance();
      SingleObject object2 = SingleObject.getInstance();

      object1.setx(6);

```

```
9


** Process exited - Return Code: 0 **
```

Lab Session 09

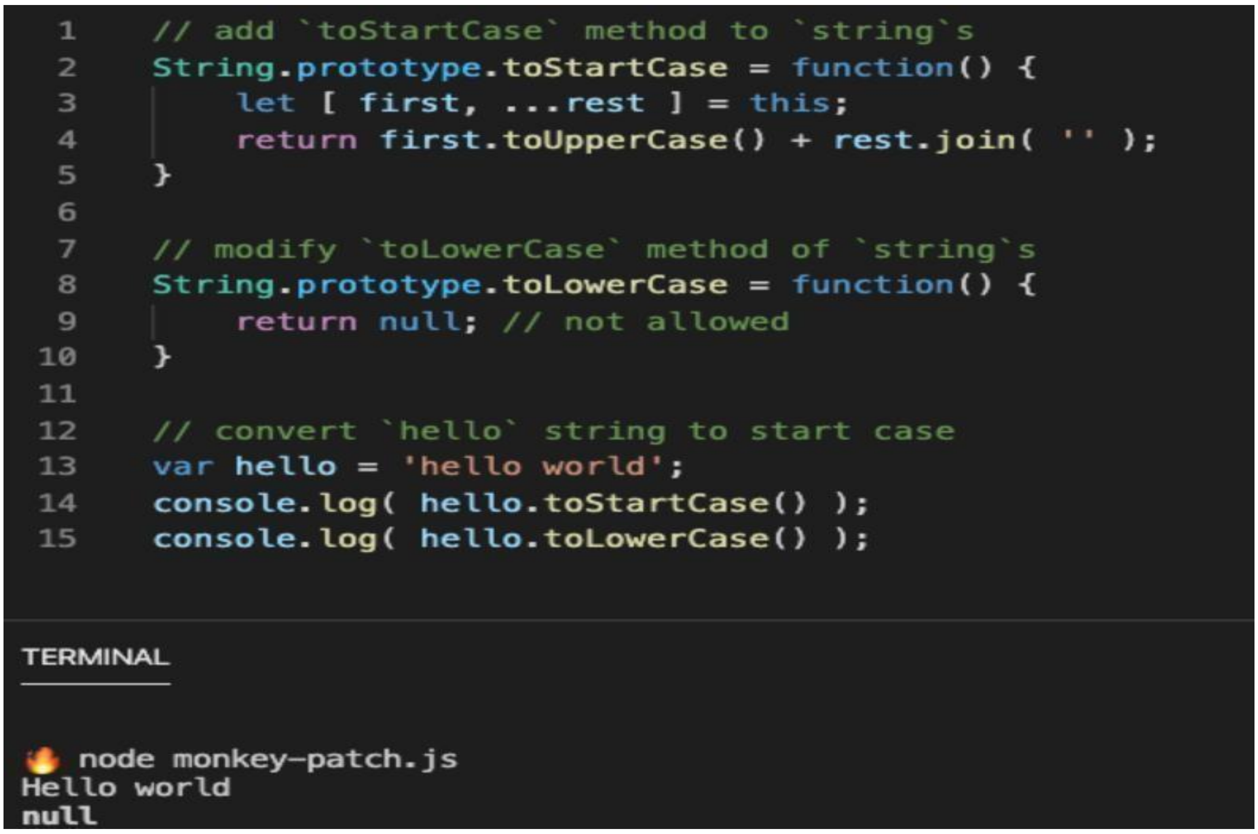
OBJECT: Metaprogramming (Monkey Patching)

THEORY

Metaprogramming is a programming technique in which computer programs have the ability to treat other programs as their data. It means that a program can be designed to read, generate, analyze, or transform other programs, and even modify itself while running.
A programming language that manipulates other programs that are written in completely different programming languages is called a metalanguage. A programming language that can manipulate itself is called a homoiconic language. So technically if a programming language has the capacity to change its behavior at runtime, then it can be said to support metaprogramming. If a program can manipulate itself at runtime, then it supports metaprogramming.

Monkey Patching

The action of the program to manipulate its behavior or the behavior of other programs at runtime is called monkey patching. Monkey patching is to tweak certain part(s) of the program at runtime to achieve desired results, without changing the original source code of the program.



```javascript
// add `toStartCase` method to `string`s
String.prototype.toStartCase = function() {
    let [ first, ...rest ] = this;
    return first.toUpperCase() + rest.join( '' );
}

// modify `toLowerCase` method of `string`s
String.prototype.toLowerCase = function() {
    return null; // not allowed
}

// convert `hello` string to start case
var hello = 'hello world';
console.log( hello.toStartCase() );
console.log( hello.toLowerCase() );
```

TERMINAL

```
node monkey-patch.js
Hello world
null
```

Fig. 9.1 Monkey Patching Example

In the above example, we have changed the behavior of the *String* class (which affected the *string* type) at runtime by using some monkey patching techniques as the source code of *String* class hasn't been changed. Here, the default behavior of the *toLowerCase()* has been altered at runtime. However, no change has been performed *toLowerCase()* class.

EXERCISE

1. Perform the alteration in *toUpperCase()* and display any string instead of converting input string to upper case.

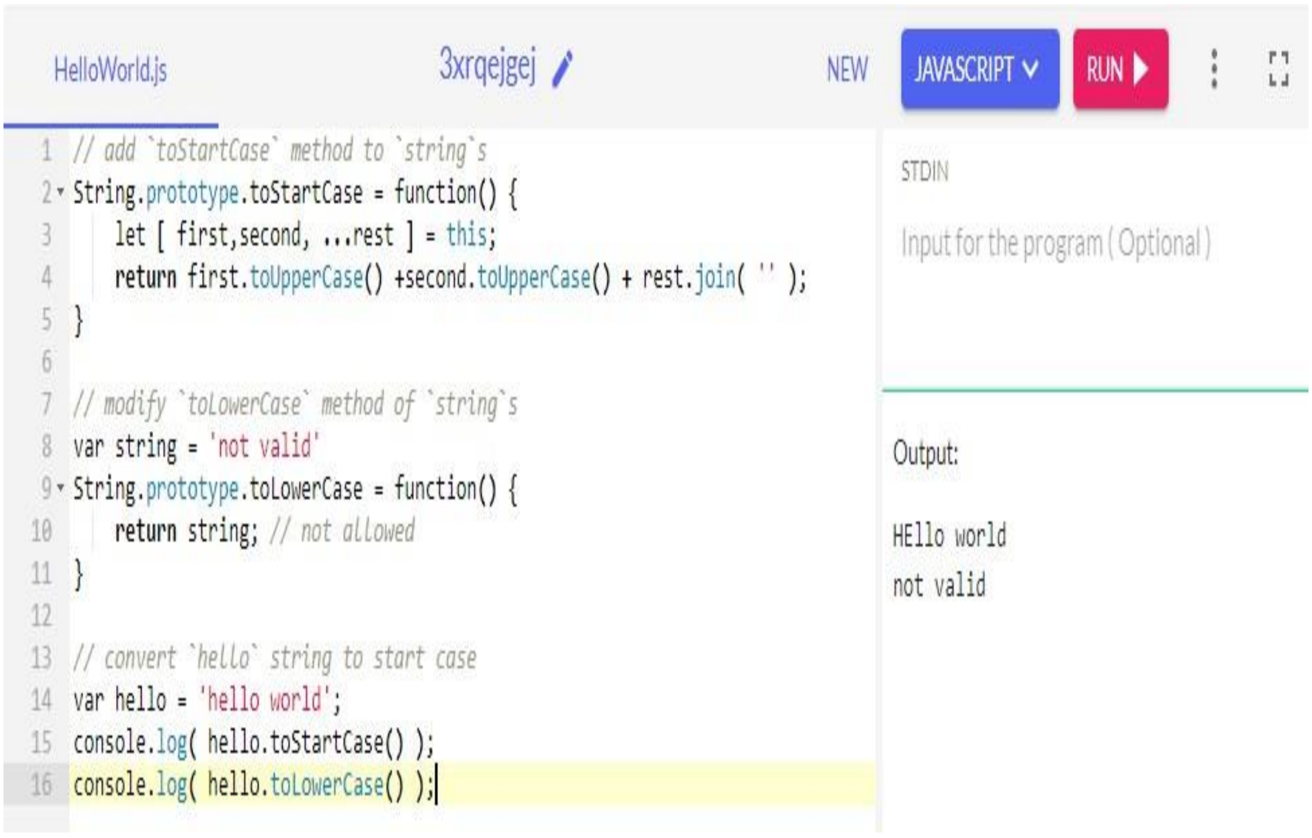
LAB # 09:

CODE:

```
// add `toStartCase` method to `string`s String.prototype.toStartCase
= function() {
   let [ first,second, ...rest ] = this;
   return first.toUpperCase() +second.toUpperCase() + rest.join( '' );
}

// modify `toLowerCase` method of `string`s var
string = 'not valid'
String.prototype.toLowerCase = function() {
   return string; // not allowed
}

// convert `hello` string to start case
var hello = 'hello world';
console.log( hello.toStartCase() );
console.log( hello.toLowerCase() );
```

Kabeer Ahmed SE-19028

OUTPUT:



```
// add `toStartCase` method to `string`s
String.prototype.toStartCase = function() {
    let [ first,second, ...rest ] = this;
    return first.toUpperCase() +second.toUpperCase() + rest.join( '' );
}

// modify `toLowerCase` method of `string`s
var string = 'not valid'
String.prototype.toLowerCase = function() {
    return string; // not allowed
}

// convert `hello` string to start case
var hello = 'hello world';
console.log( hello.toStartCase() );
console.log( hello.toLowerCase() );
```

Output:

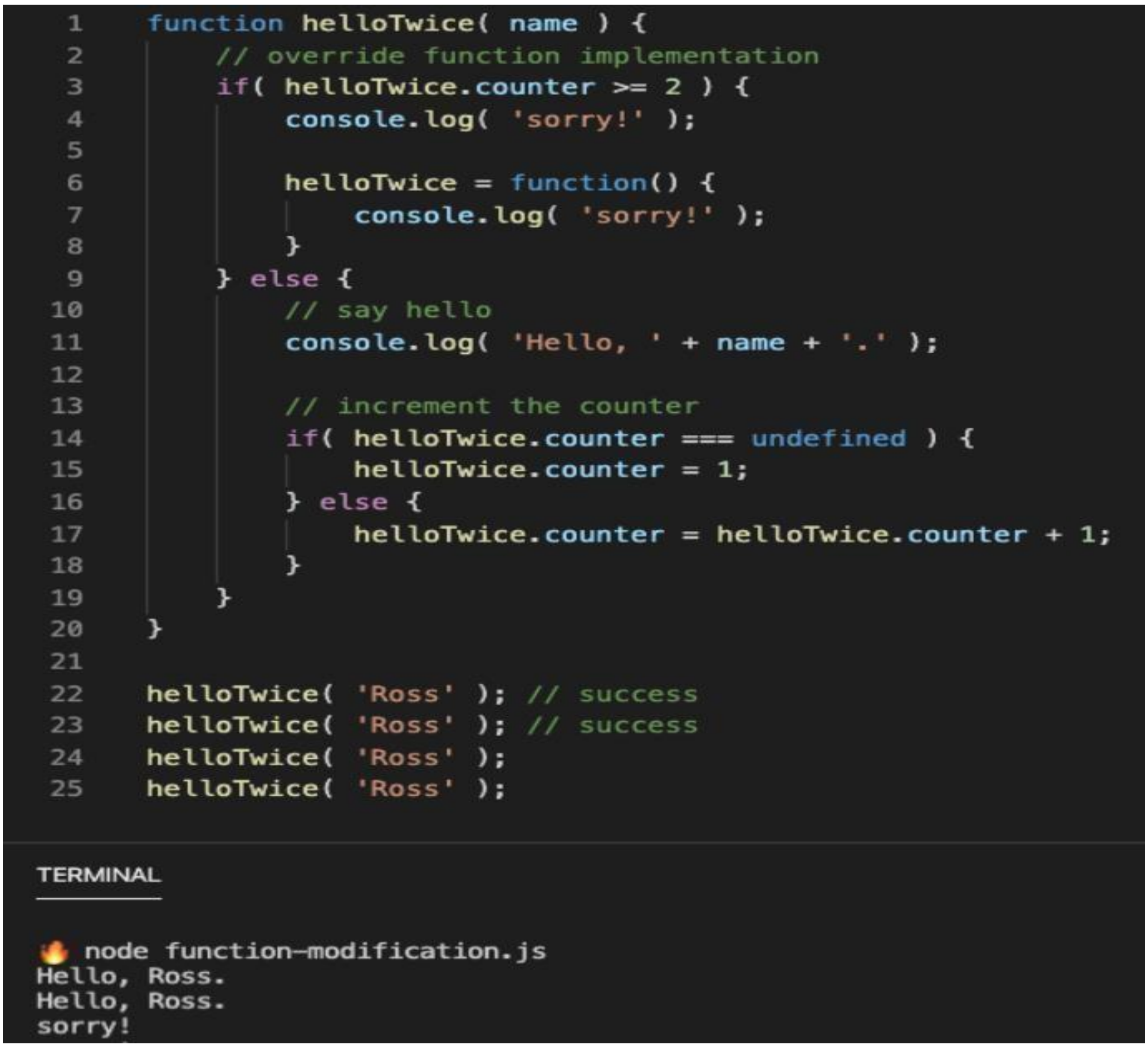
```
HEllo world
not valid
```

Lab Session 10

OBJECT: Metaprogramming (Modification)

THEORY

Modification is one type of metaprogramming. Modification refers to the modification of the program behavior through mutation. In case of modification, we are changing the behavior of the target itself so suit the receiver.
Overriding a function implementation would be a good example of modification. For example, if a function is designed to behave in a certain way, but we want to something else conditionally, we can do that by designing a self-overriding function.



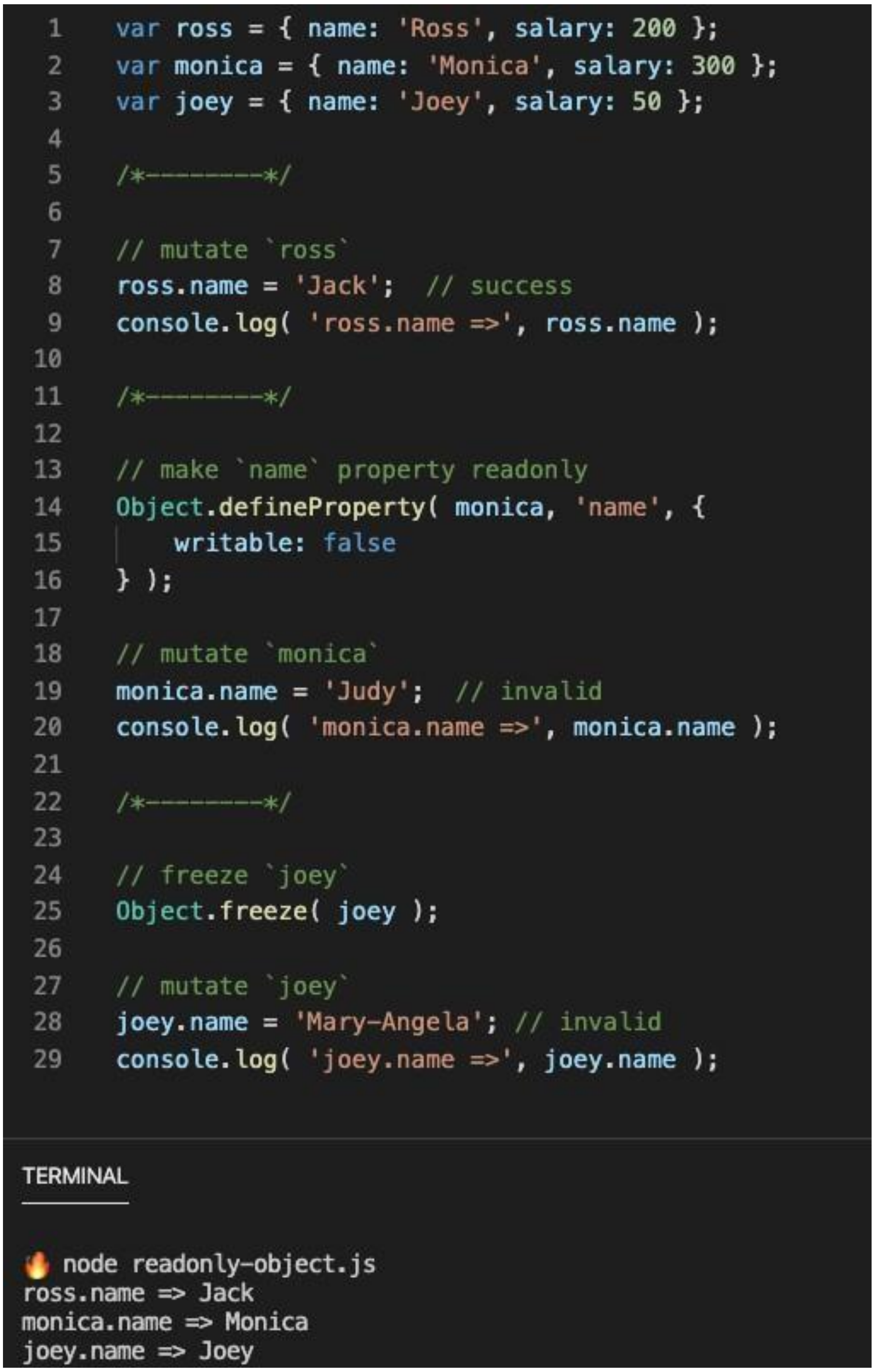
```
function helloTwice( name ) {
    // override function implementation
    if( helloTwice.counter >= 2 ) {
        console.log( 'sorry!' );

        helloTwice = function() {
            console.log( 'sorry!' );
        }
    } else {
        // say hello
        console.log( 'Hello, ' + name + '.' );

        // increment the counter
        if( helloTwice.counter === undefined ) {
            helloTwice.counter = 1;
        } else {
            helloTwice.counter = helloTwice.counter + 1;
        }
    }
}

helloTwice( 'Ross' ); // success
helloTwice( 'Ross' ); // success
helloTwice( 'Ross' );
helloTwice( 'Ross' );
```

TERMINAL

```
🔥 node function-modification.js
Hello, Ross.
Hello, Ross.
sorry!
```

Fig. 10.1 Modification Example 1

In the above example, we have created a function that overrides itself with a new function implementation. This would be the harshest example of modification.



```
var ross = { name: 'Ross', salary: 200 };
var monica = { name: 'Monica', salary: 300 };
var joey = { name: 'Joey', salary: 50 };

/*--------*/

// mutate `ross`
ross.name = 'Jack';  // success
console.log( 'ross.name =>', ross.name );

/*--------*/

// make `name` property readonly
Object.defineProperty( monica, 'name', {
    writable: false
} );

// mutate `monica`
monica.name = 'Judy';  // invalid
console.log( 'monica.name =>', monica.name );

/*--------*/

// freeze `joey`
Object.freeze( joey );

// mutate `joey`
joey.name = 'Mary-Angela'; // invalid
console.log( 'joey.name =>', joey.name );
```

```
🔥 node readonly-object.js
ross.name => Jack
monica.name => Monica
joey.name => Joey
```

Fig. 10.2 Modification Example 2

In the above example, we have used *Object.defineProperty()* method to change the default property descriptor of the i property in order to make it read-only. You can also use the *Object.freeze()* method to lock the entire object to avoid any mutations.
Some intercessions can happen through modifications. By setting *writable:false* in the property descriptor of the object, therefore mutating the object (internal implementation), we have incepted the value assignment operation.

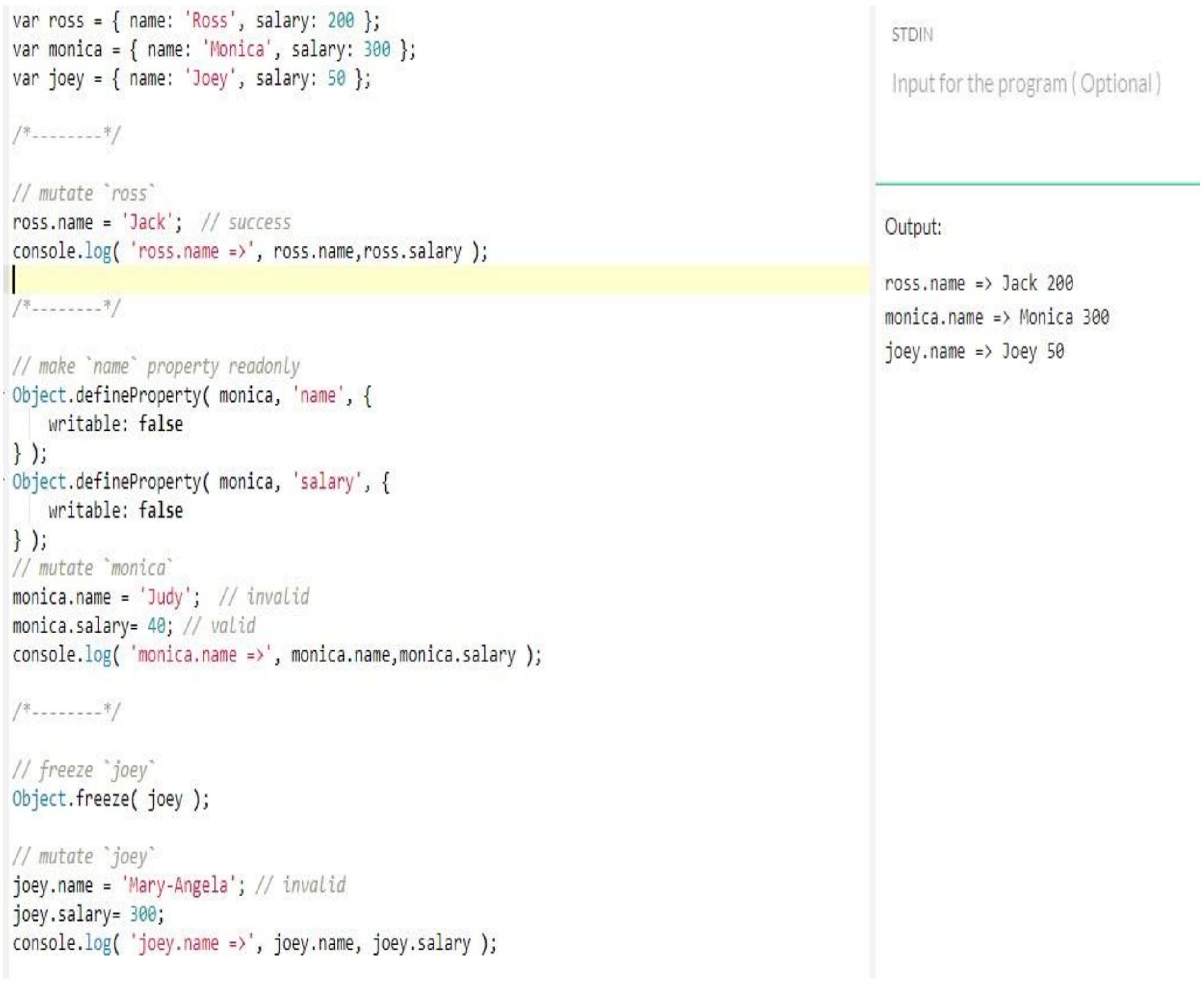
EXERCISE

1. Modify the *salary* property using *Object.defineProperty()* in example 2 and set writable:false the salary property of object *monica* and use *Object.freeze()* method to lock the *joey* object. Mutate name and salary both in all three objects and display the result of each.

LAB # 10:
CODE:

```
var ross = { name: 'Ross', salary: 200 }; var
monica = { name: 'Monica', salary: 300 };
var joey = { name: 'Joey', salary: 50 };

/*--------*/

// mutate `ross`
ross.name = 'Jack';  // success
console.log( 'ross.name =>', ross.name,ross.salary );

/*--------*/

// make `name` property readonly
Object.defineProperty( monica, 'name', {
   writable: false
} );
Object.defineProperty( monica, 'salary', {
   writable: false
} );
// mutate `monica`
monica.name = 'Judy';  // invalid monica.salary=
40; // valid
console.log( 'monica.name =>', monica.name,monica.salary );

/*--------*/
// freeze `joey`
Object.freeze( joey );

// mutate `joey`
joey.name = 'Mary-Angela'; // invalid
joey.salary= 300;
console.log( 'joey.name =>', joey.name, joey.salary );
```

OUTPUT:



```
var ross = { name: 'Ross', salary: 200 };
var monica = { name: 'Monica', salary: 300 };
var joey = { name: 'Joey', salary: 50 };

/*--------*/

// mutate `ross`
ross.name = 'Jack';  // success
console.log( 'ross.name =>', ross.name,ross.salary );

/*--------*/

// make `name` property readonly
Object.defineProperty( monica, 'name', {
    writable: false
} );
Object.defineProperty( monica, 'salary', {
    writable: false
} );
// mutate `monica`
monica.name = 'Judy';  // invalid
monica.salary= 40; // valid
console.log( 'monica.name =>', monica.name,monica.salary );

/*--------*/

// freeze `joey`
Object.freeze( joey );

// mutate `joey`
joey.name = 'Mary-Angela'; // invalid
joey.salary= 300;
console.log( 'joey.name =>', joey.name, joey.salary );
```

```
ross.name => Jack 200
monica.name => Monica 300
joey.name => Joey 50
```