**LAB # 1**

**Introduction to Eclipse IDE**

**OBJECTIVE**

Introduction to Eclipse IDE, creating projects, its classes and mapping into UML through plugins.

**THEORY:**

**Eclipse IDE:**

The Eclipse IDE (integrated development environment) provides strong support for Java developers. In 2020 Eclipse is one of the leading IDEs with approximately one million downloads per month. Eclipse can be extended with additional software components called plug-ins. Pre-packaged Eclipse distributions provide a consistent set of functionalities.

Eclipse IDE downloads provide Eclipse distributions for different use cases. The Eclipse IDE for Java Developers distribution is designed to support standard Java development. It includes support for the Maven and Gradle build system and support for the Git version control system.

**Plug-ins in eclipse:**

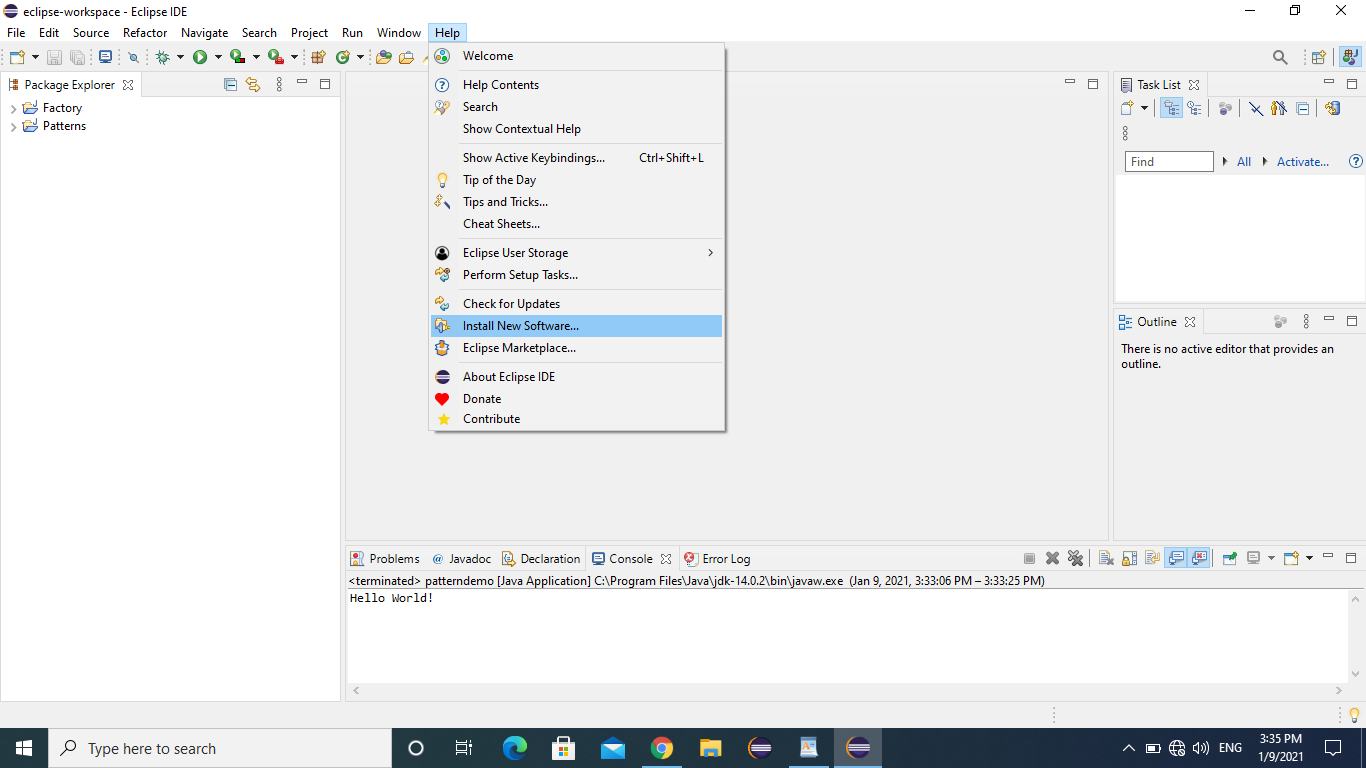
A plug-in in Eclipse is a component that provides a certain type of service within the context of the Eclipse workbench. By a component here I mean an object that may be configured into a system at system deployment time. The Eclipse runtime provides an infrastructure to support the activation and operation of a set of plug-ins working together to provide a seamless environment for development activities. Within a running Eclipse instance, a plug-in is embodied in an instance of some plug-in runtime class, or plug-in class, for short. The plug-in class provides configuration and management support for the plug-in instance. A plug-in class in Eclipse must extend org.eclipse.core.runtime.Plugin, which is an abstract class that provides generic facilities for managing plug-ins.

An Eclipse installation includes a plugins folder where individual plug-ins are deployed. Each plug-in is installed in its folder under the plugins folder. A plug-in is described in an XML manifest file, called plugin.xml, residing in the plug-in's folder. The manifest file tells the Eclipse runtime what it needs to know to activate the plug-in.

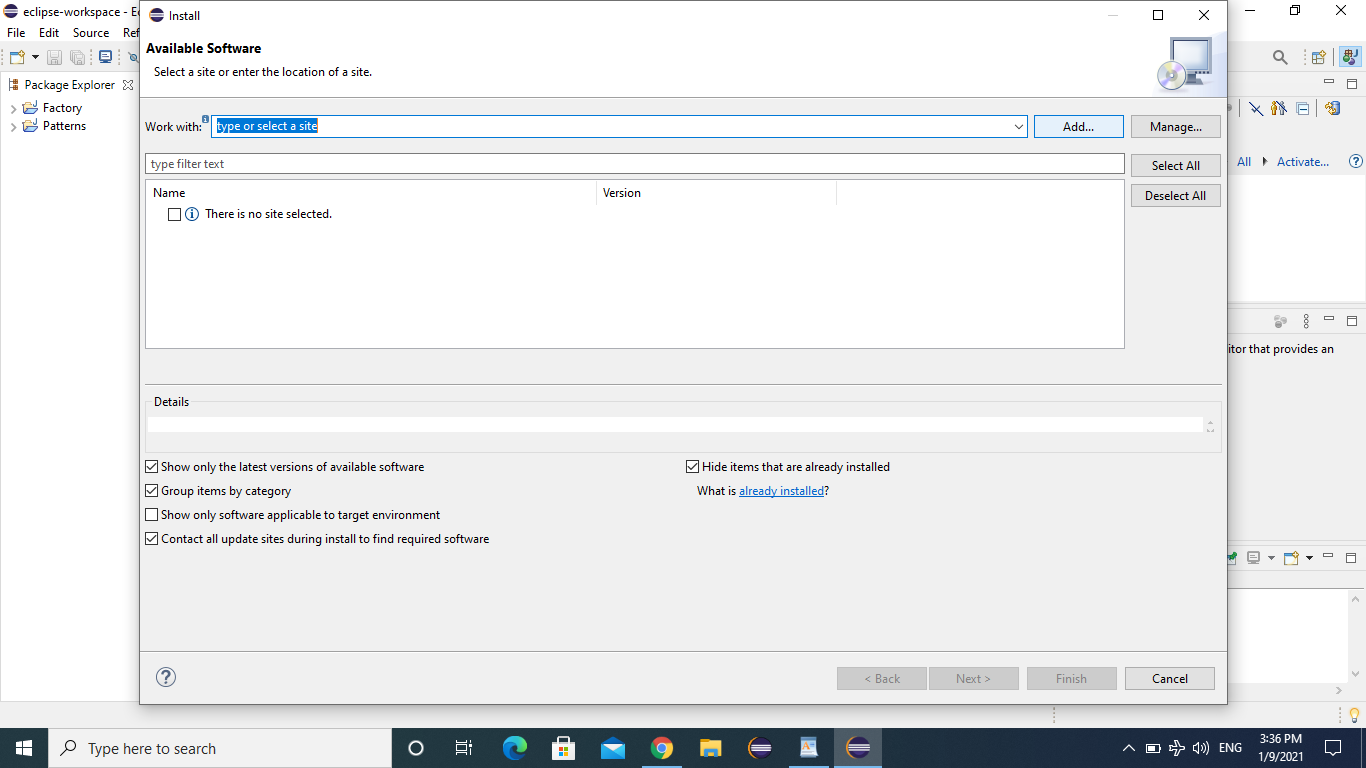
The parsed contents of plug-in manifest files are made available programmatically through a plug-in registry API. And parsed plug-in specifications are cached in an in-memory repository called the plug-in registry. The Eclipse runtime instantiates an instance of each plug-in by using the plug-in registry API. The plug-in registry API is also used by provider-supplied plug-in code to obtain information about plug-ins.

**STEPS FOR INSTALLING PLUG-INS ON ECLIPSE:**

**Step # 1: In the Eclipse main menu, go to Help > Install New Software.**



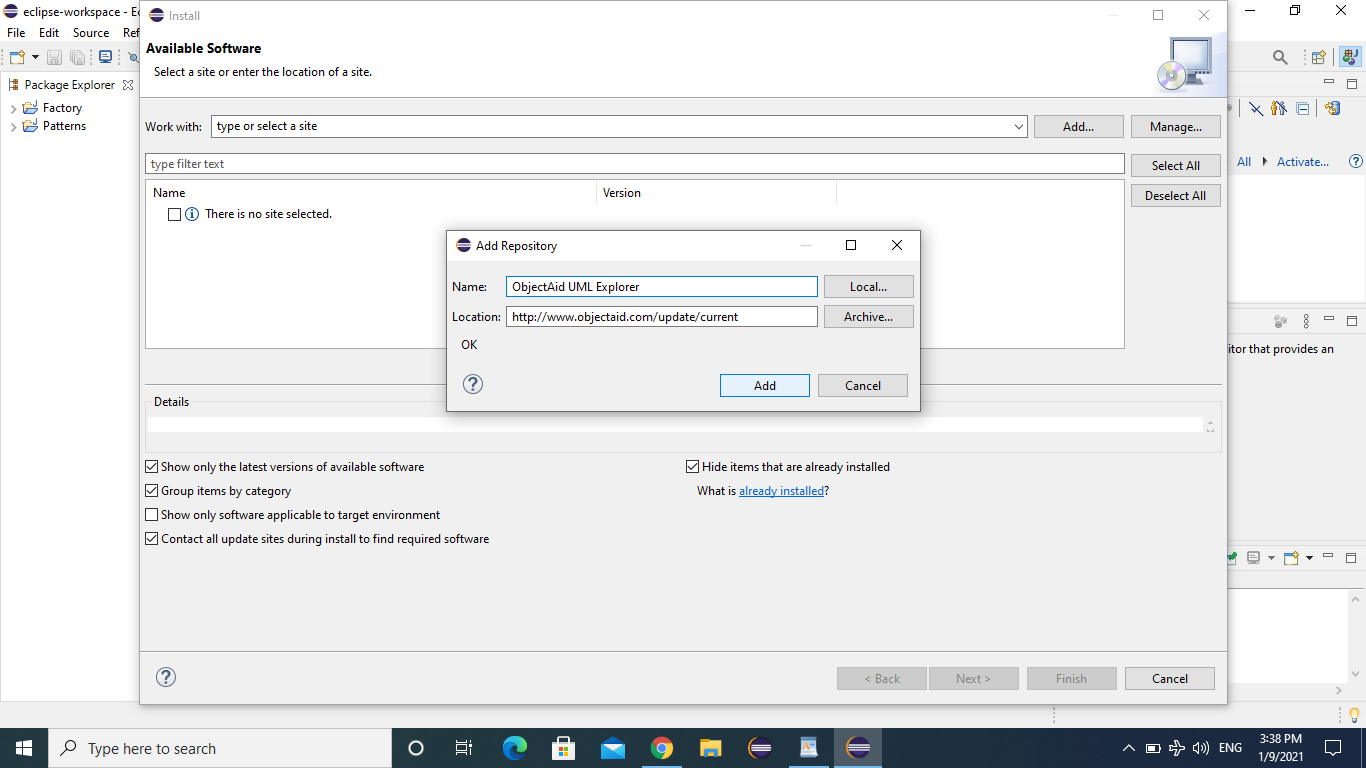
**Step # 2: In the 'Available Software' page of the 'Install' wizard, press the 'Add...' button.**



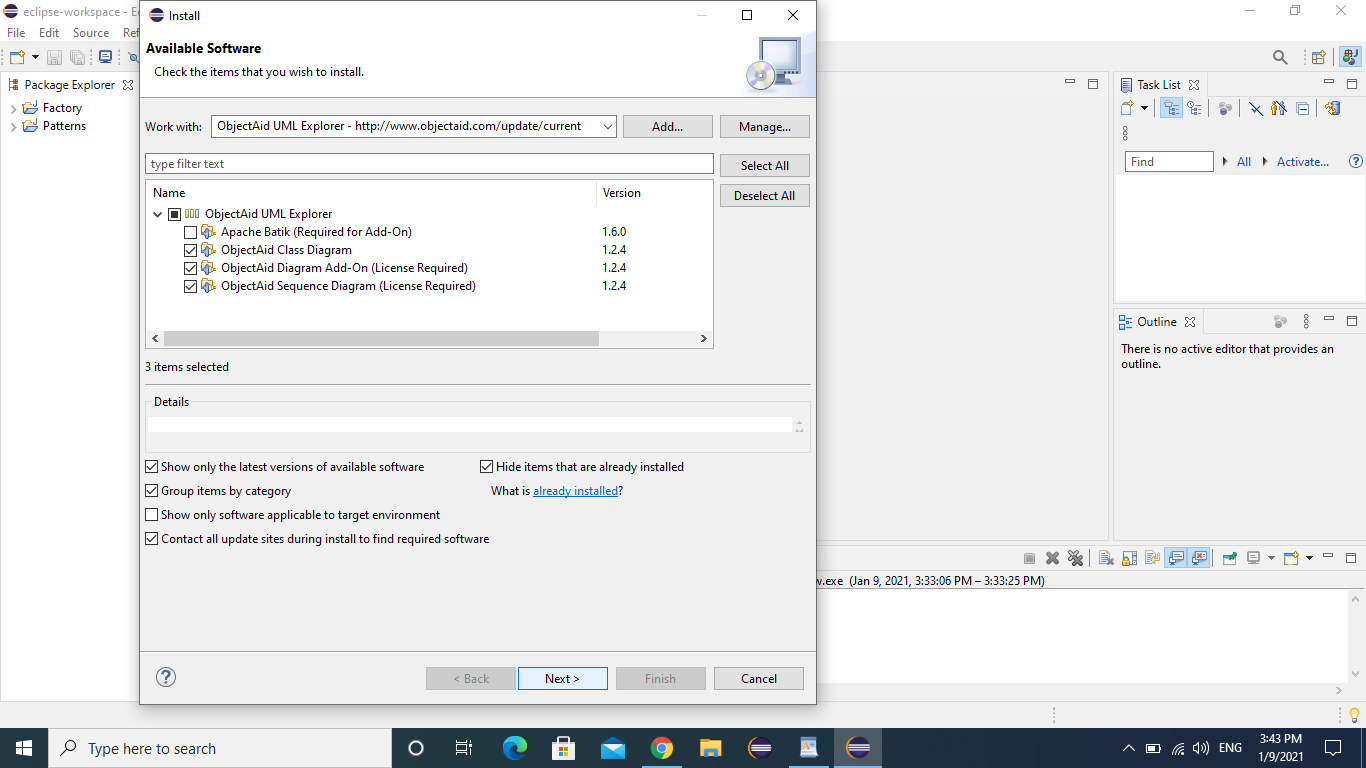
**Step # 3: In the 'Add Repository' dialog, enter this information and press 'OK':**

Name: ObjectAid UML Explorer

URL: <http://www.objectaid.com/update/current>

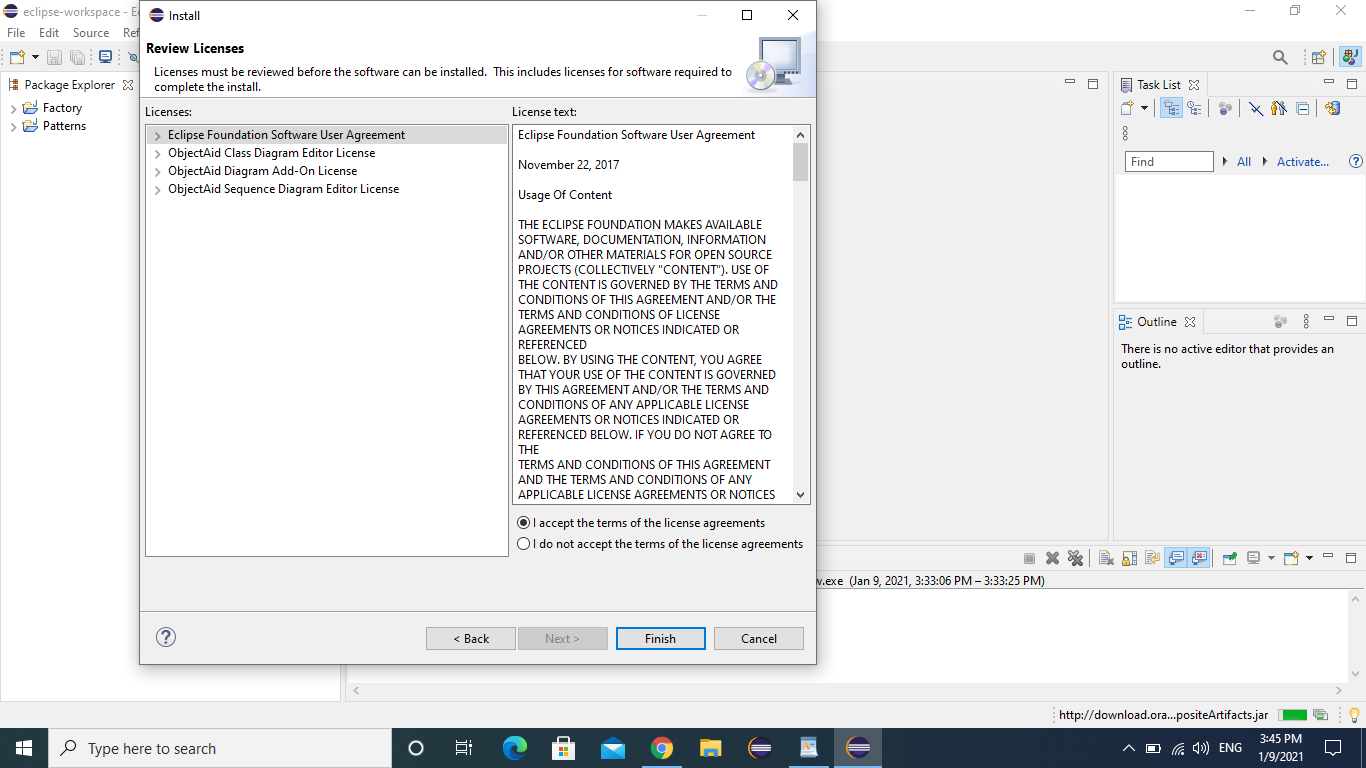


**Step # 4: The 'Install' dialog will now show the available ObjectAid plug-ins. Please select what you would like to install and press 'Next'. You may want to turn off the check box 'Contact all update sites during install to find required software' to speed up the download.**



**Step # 5: On the 'Install Details' page you should simply press 'Next'.**

On the 'Review Licenses' page, you need to accept the ObjectAid license and press 'Finish' to begin the installation.



**Step # 6: You will receive a 'Security Warning' message because the ObjectAid JARs are not signed. Please press 'OK' to continue.**

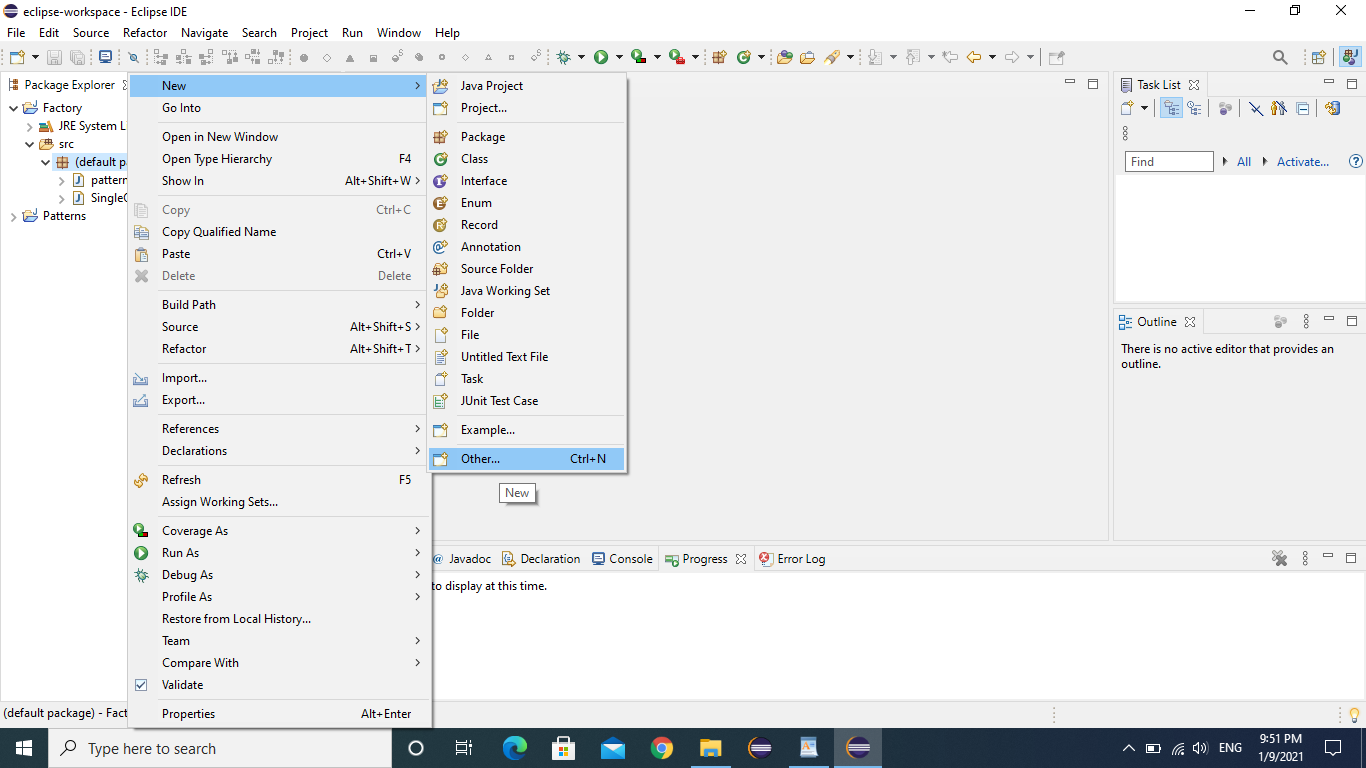
Once the installation is complete, you will be asked if you want to 'Restart Now', 'Not Now' or 'Apply Changes Now'. To be safe, please press the 'Restart Now' button and the ObjectAid UML Explorer will be available after the restart.

**STEPS FOR GENERATING UML ON ECLIPSE:**

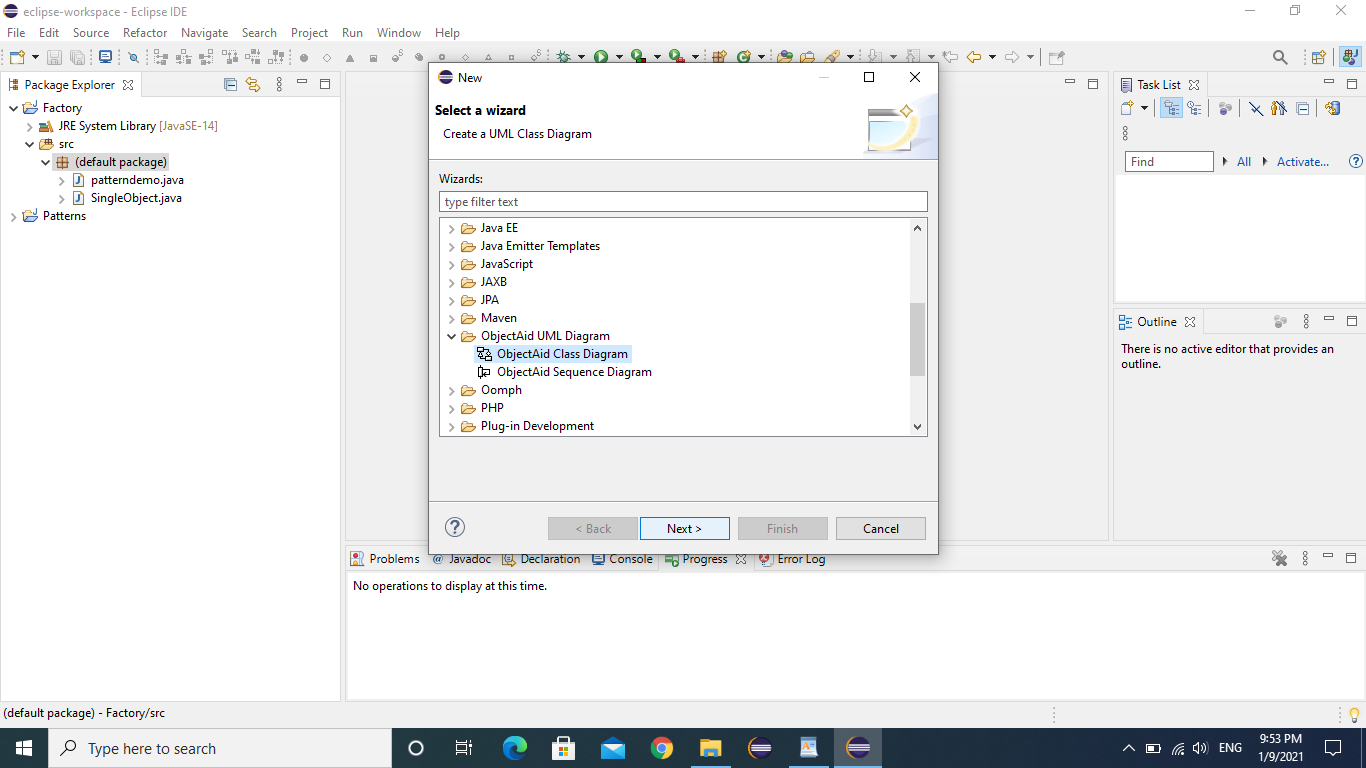
**Step # 1: Create a project in eclipse and create classes under default package. (as we have already created singleton design pattern learnt in software design architecture)**

|  |
| --- |
| **public** **class** SingleObject {  **private** **static** SingleObject *instance* = **new** SingleObject();  **private** SingleObject(){}  **public** **static** SingleObject getInstance(){  **return** *instance*;  }  **public** **void** Message(){  System.***out***.println("Hello World!");  }  }  **public** **class** patterndemo {  **public** **static** **void** main(String[] args) {  SingleObject object = SingleObject.*getInstance*();  object.Message();  }  } |

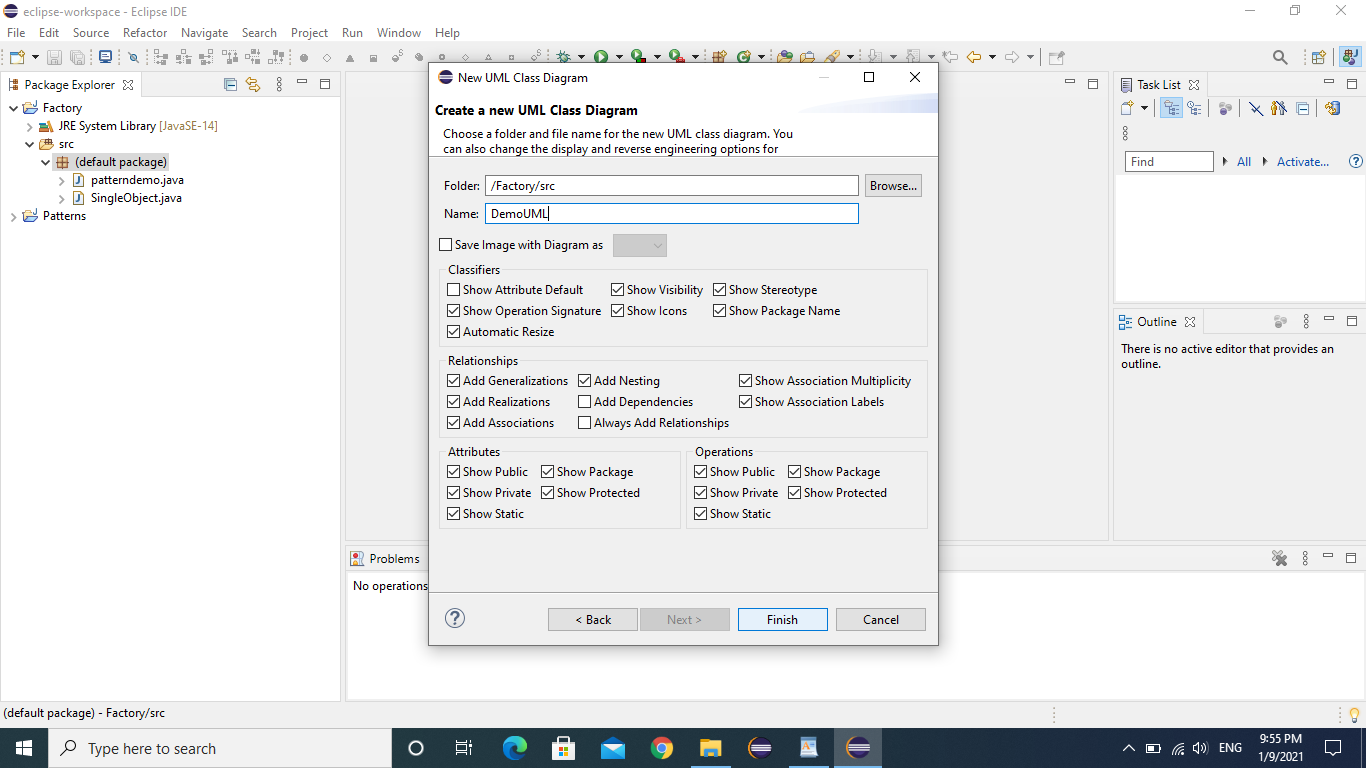
**Step # 2: Write click on default package -> New -> Other:**



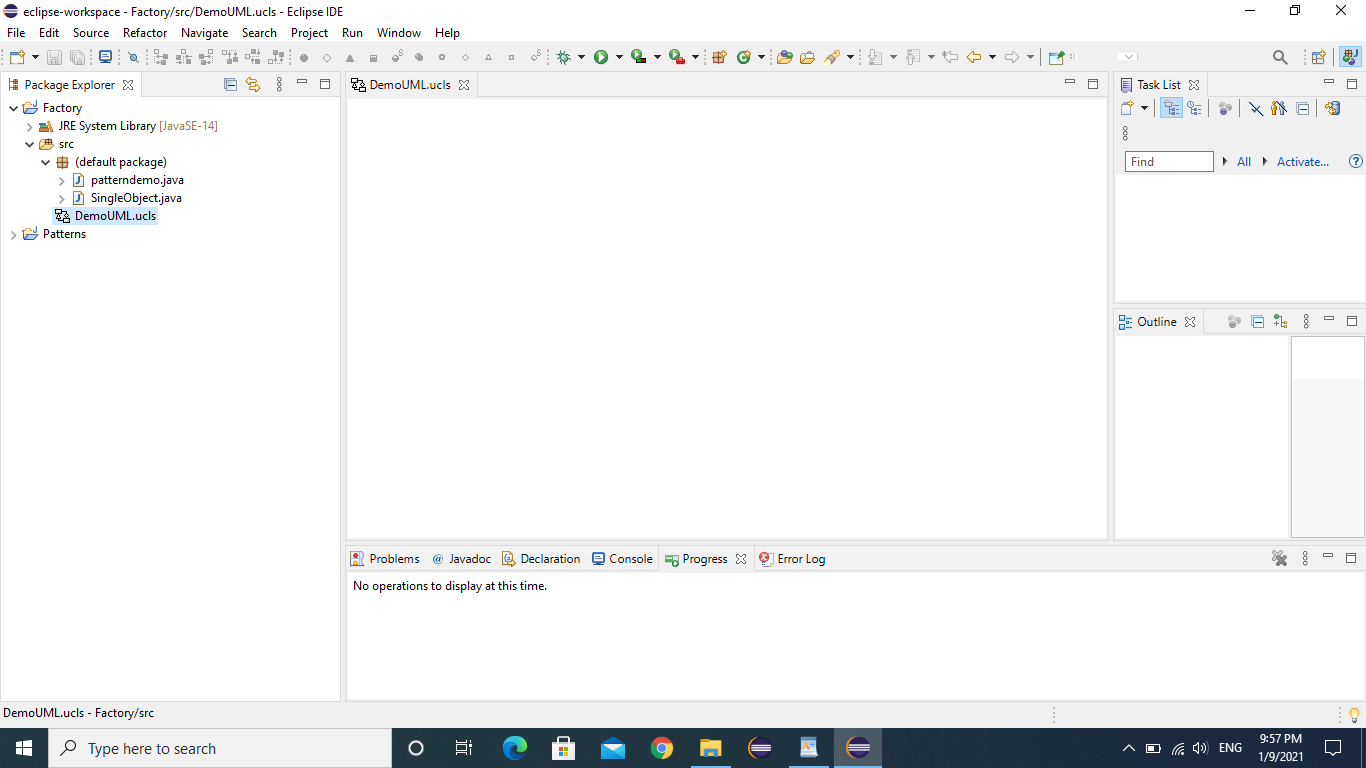
**Step # 3: Select ObjectAid Class Diagram plugin and click next:**



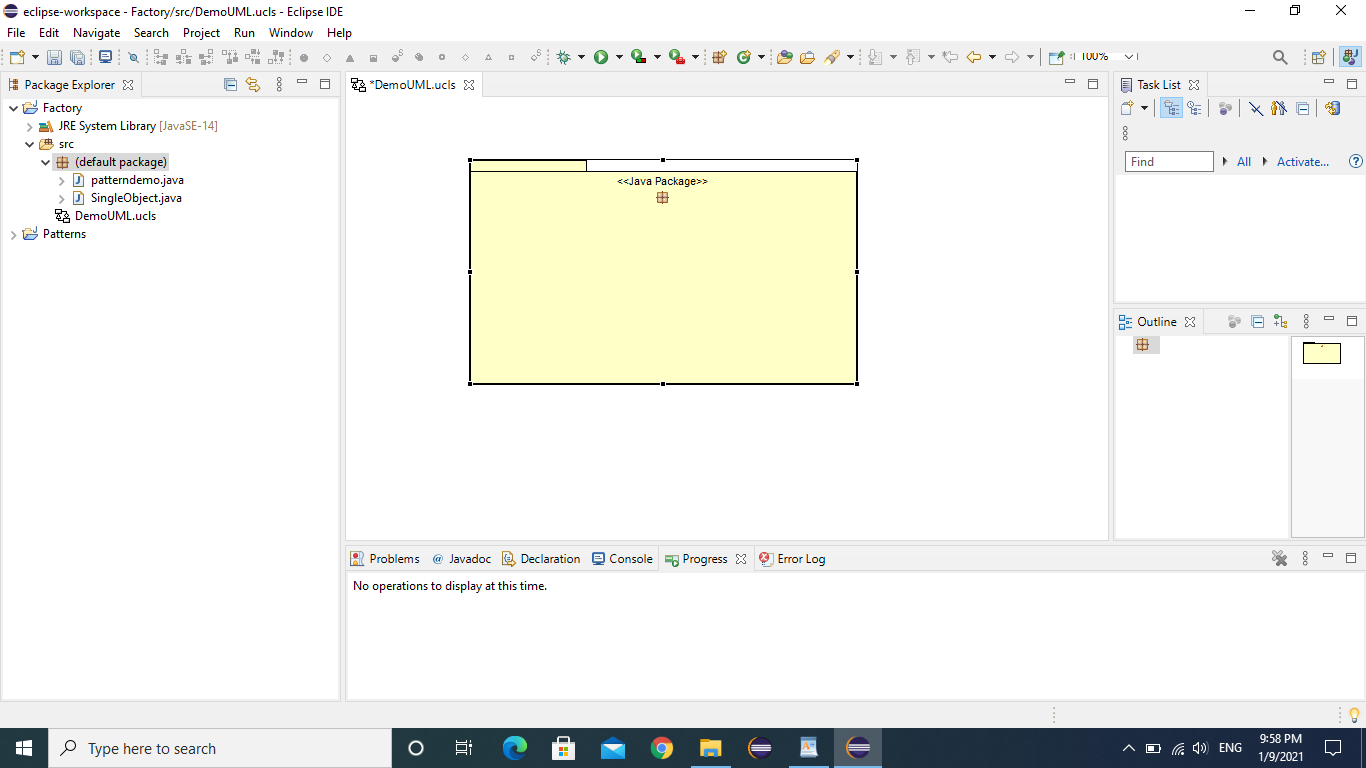
**Step # 4: Provide the name of a class diagram and click on 'Finish':**



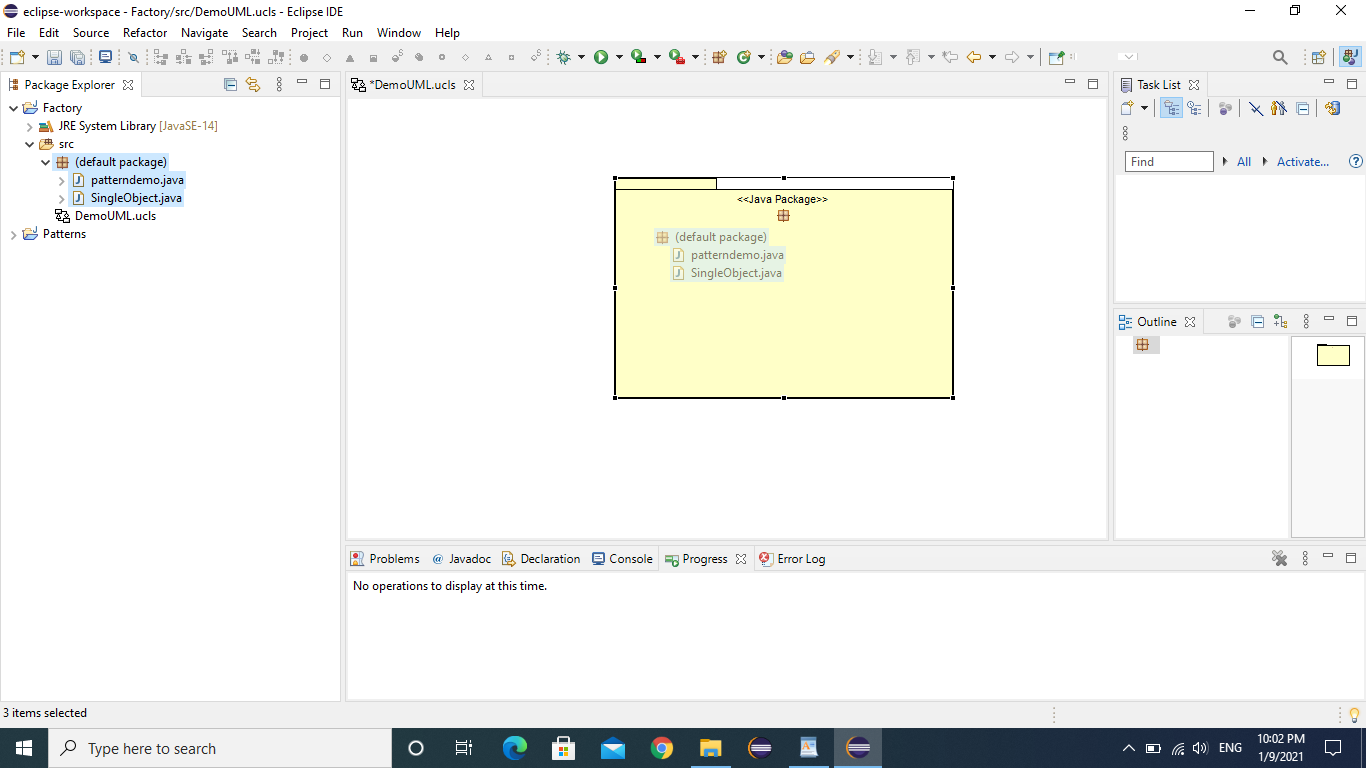
**Step # 5: Click on DemoUML.ucls for generating class diagram:**



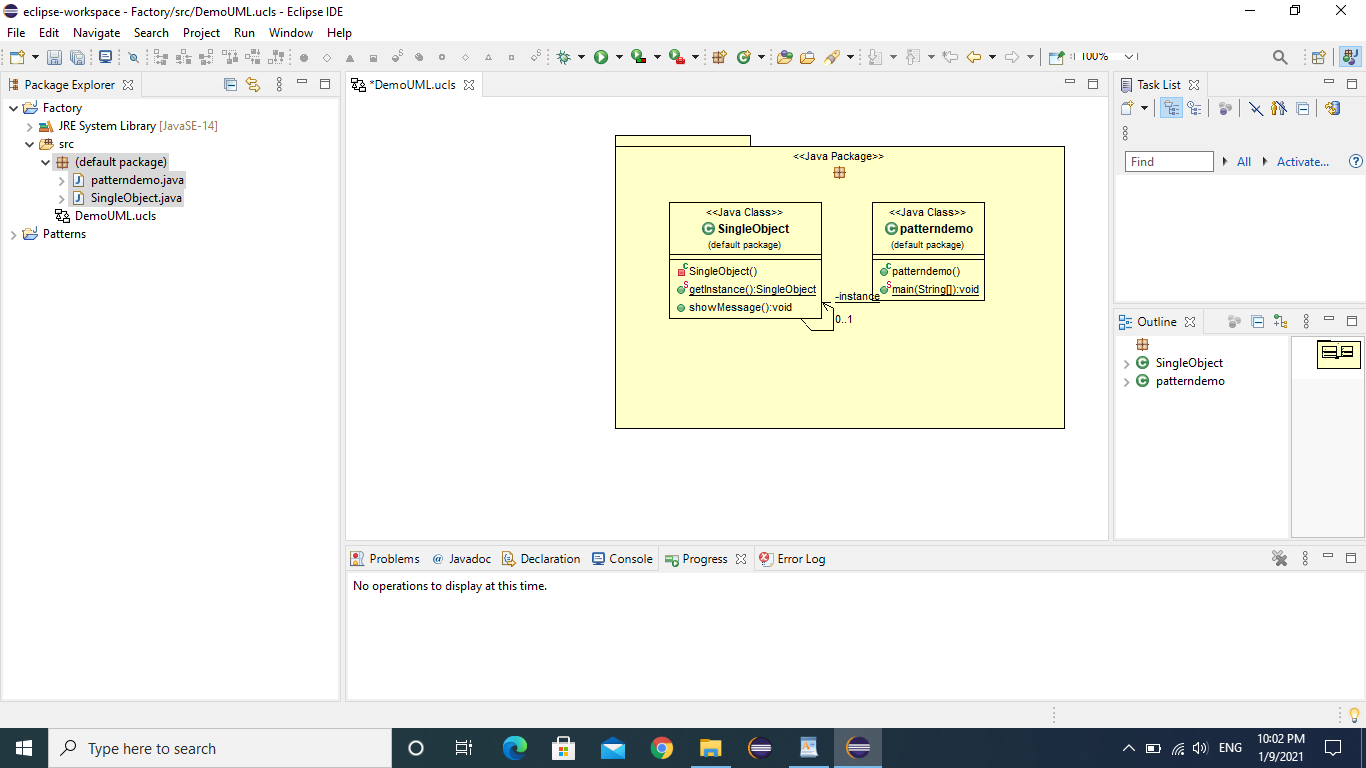
**Step # 6: Click on default package from the project explorer and drag it to UML editor:**



**Step # 7: Resize the package and drag all the classes in the package simultaneously:**



**Step # 8: Final class diagram would be generated as:**



Lab Task:

Design a university management system, an interface of ‘Employees’ that will have two/three method declarations. This interface will be implemented by two classes i.e Admin and Faculty. The faculty class will be extended by Professor, Assistant Professor, and Lecturer. On the other hand, admin class will be extended by examination and admission classes. All these inheritances will be called in the main class by instantiation. (The internal mechanism or method implementation is choice dependent; you can perform whatever you want). After completing the task, convert the program code into UML class diagram.