|  |  |
| --- | --- |
| **Course Code: CT2316** | **Course Name: Lab: Software Engineering** |

|  |  |
| --- | --- |
| **Name: S Akshansh** | **Semester/ Section: 6/A** |
| **Roll No: 72** | **Enroll No: 19010927** |

**PRACTICAL NO: 1**

**AIM: *Introduction to Software Engineering fundamentals, UML and RATIONAL ROSE INTERFACE***THEORY:

1. Software:

A textbook description of software might take the following form: Software is (1) instructions (computer programs) that when executed provide desired function and performance, (2) data structures that enable the programs to adequately manipulate information, and (3) documents that describe the operation and use of the programs.

1. Software Characteristics:

To gain an understanding of software (and ultimately an understanding of software engineering), it is important to examine the characteristics of software that make it different from other things that human beings build. When hardware is built, the human creative process (analysis, design, construction, testing) is ultimately translated into a physical form. If we build a new computer, our initial sketches, formal design drawings, and bread boarded prototype evolve into a physical product (chips, circuit boards, power supplies, etc.). Software is a logical rather than a physical system element. Therefore, software has characteristics that are considerably different than those of hardware:

1. **Software is developed or engineered; it is not manufactured in the classical sense.**

Although some similarities exist between software development and hardware manufacture, the two activities are fundamentally different. In both activities, high quality is achieved through good design, but the manufacturing phase for hardware can introduce quality problems that are nonexistent (or easily corrected) for software. Both activities are dependent on people, but the relationship between people applied and work accomplished is entirely different. Both activities require the construction of a "product" but the approaches are different. Software costs are concentrated in engineering. This means that software projects cannot be managed as if they were manufacturing projects.

1. **Software doesn't "wear out."**

Figure 1 depicts failure rate as a function of time for hardware. The relationship, often called the "bathtub curve," indicates that hardware exhibits relatively high failure rates early in its life (these failures are often attributable to design or manufacturing defects); defects are corrected and the failure rate drops to a steady-state level (ideally, quite low) for some period of time. As time passes, however, the failure rate rises again as hardware components suffer from the cumulative effects of dust, vibration, abuse, temperature extremes, and many other environmental maladies. Stated simply, the hardware begins to wear out.

Software is not susceptible to the environmental maladies that cause hardware to wear out. In theory, therefore, the failure rate curve for software should take the form of the “idealized curve” shown in Figure 2. Undiscovered defects will cause high failure rates early in the life of a program. However, these are corrected (ideally, without introducing other errors) and the curve flattens as shown. The idealized curve is a gross oversimplification of actual failure models for software. However, the implication is clear—software doesn't wear out. But it does deteriorate! This seeming contradiction can best be explained by considering the “actual curve” shown in Figure 2. During its life, software will undergo change (maintenance). As changes are made, it is likely that some new defects will be introduced, causing the failure rate curve to spike as shown in Figure 2. Before the curve can return to the original steady-state failure rate, another change is requested, causing the curve to spike again. Slowly, the minimum failure rate level begins to rise—the software is deteriorating due to change. Another aspect of wear illustrates the difference between hardware and software. When a hardware component wears out, it is replaced by a spare part. There are no software spare parts. Every software failure indicates an error in design or in the process through which design was translated into machine executable code. Therefore, software maintenance involves considerably more complexity than hardware maintenance.

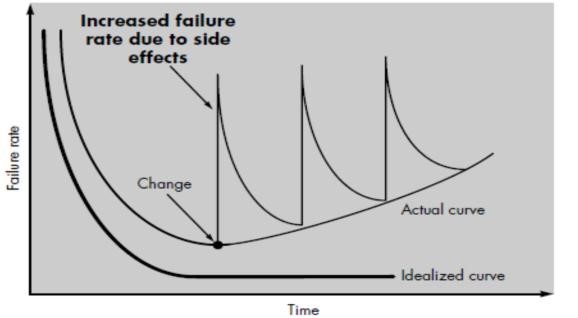
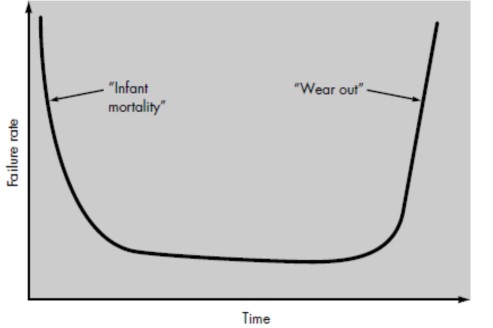


FIGURE 1. Failure curve for hardware FIGURE 2. Idealized and actual failure curves for

Software

III. **Although the industry is moving toward component-based assembly, most software continues to be custom built.**

Consider the manner in which the control hardware for a computer-based product is designed and built. The design engineer draws a simple schematic of the digital circuitry, does some fundamental analysis to assure that proper function will be achieved, and then goes to the shelf where catalogs of digital components exist. Each integrated circuit (called an *IC* or a *chip*) has a part number, a defined and validated function, a well-defined interface, and a standard set of integration guidelines. After each component is selected, it can be ordered off the shelf. As an engineering discipline evolves, a collection of standard design components is created. Standard screws and off-the-shelf integrated circuits are only two of thousands of standard components that are used by mechanical and electrical engineers as they design new systems. The reusable components have been created so that the engineer can concentrate on the truly innovative elements of a design, that is, the parts of the design that represent something new. In the hardware world, component reuse is a natural part of the engineering process. In the software world, it is something that has only begun to be achieved on a broad scale. A software component should be designed and implemented so that it can be reused in many different programs. In the 1960s, we built scientific subroutine libraries that were reusable in a broad array of engineering and scientific applications. These subroutine libraries reused well defined algorithms in an effective manner but had a limited domain of application. Today, we have extended our view of reuse to encompass not only algorithms but also data structure. Modern reusable components encapsulate both data and the processing applied to the data, enabling the software engineer to create new applications from reusable parts. For example, today's graphical user interfaces are built using reusable components that enable the creation of graphics windows, pull-down menus, and a wide variety of interaction mechanisms. The data structure and processing detail required to build the interface are contained with a library of reusable components for interface construction. Most software continues to be custom built.

**Software Applications:**

Software may be applied in any situation for which a pre-specified set of procedural steps (i.e., an algorithm) has been defined (notable exceptions to this rule are expert system software and neural network software). Information content and determinacy are important factors in determining the nature of a software application. Content refers to the meaning and form of incoming and outgoing information. For example, many business applications use highly structured input data (a database) and produce formatted “reports.” Software that controls an automated machine (e.g., a numerical control) accepts discrete data items with limited structure and produces individual machine commands in rapid succession.

*Information determinacy* refers to the predictability of the order and timing of information. An engineering analysis program accepts data that have a predefined order, executes the analysis algorithm(s) without interruption, and produces resultant data in report or graphical format. Such applications are determinate. A multiuser operating system, on the other hand, accepts inputs that have varied content and arbitrary timing, executes algorithms that can be interrupted by external conditions, and produces output that varies as a function of environment and time. Applications with these characteristics are indeterminate. It is somewhat difficult to develop meaningful generic categories for software applications. As software complexity grows, neat compartmentalization disappears. The following software areas indicate the breadth of potential applications:

1. **System software.** System software is a collection of programs written to service other programs. Some system software (e.g., compilers, editors, and file management utilities) process complex, but determinate, information structures. Other systems applications (e.g., operating system components, drivers, telecommunications processors) process largely indeterminate data. In either case, the system software area is characterized by heavy interaction with computer hardware; heavy usage by multiple users; concurrent operation that requires scheduling, resource sharing, and sophisticated process management; complex data structures; and multiple external interfaces.
2. **Real-time software.** Software that monitors/analyzes/controls real-world events as they occur is called *real time.* Elements of real-time software include a data gathering component that collects and formats information from an external environment, an analysis component that transforms information as required by the application, a control/output component that responds to the external environment, and a monitoring component that coordinates all other components so that real-time response (typically ranging from 1 millisecond to 1 second) can be maintained.
3. **Business software.** Business information processing is the largest single software application area. Discrete "systems" (e.g., payroll, accounts receivable/payable, inventory) have evolved into management information system (MIS) software that accesses one or more large databases containing business information. Applications in this area restructure existing data in a way that facilitates business operations or management decision making. In addition to conventional data processing application, business software applications also encompass interactive computing (e.g., point of- sale transaction processing).
4. **Engineering and scientific software.** Engineering and scientific software have been characterized by "number crunching" algorithms. Applications range from astronomy to volcanology, from automotive stress analysis to space shuttle orbital dynamics, and from molecular biology to automated manufacturing. However, modern applications within the engineering/scientific area are moving away from conventional numerical algorithms. Computer-aided design, system simulation, and other interactive applications have begun to take on real-time and even system software characteristics.
5. **Embedded software.** Intelligent products have become commonplace in nearly every consumer and industrial market. Embedded software resides in read-only memory and is used to control products and systems for the consumer and industrial markets. Embedded software can perform very limited and esoteric functions (e.g., keypad control for a microwave oven) or provide significant function and control capability (e.g., digital functions in an automobile such as fuel control, dashboard displays, and braking systems).
6. **Personal computer software.** The personal computer software market has burgeoned over the past two decades. Word processing, spreadsheets, computer graphics, multimedia, entertainment, database management, personal and business financial applications, external network, and database access are only a few of hundreds of applications.
7. **Web-based software.** The Web pages retrieved by a browser are software that incorporates executable instructions (e.g., CGI, HTML, Perl, or Java), and data (e.g., hypertext and a variety of visual and audio formats). In essence, the network becomes a massive computer providing an almost unlimited software resource that can be accessed by anyone with a modem.
8. **Artificial intelligence software.** Artificial intelligence (AI) software makes use of nonnumeric algorithms to solve complex problems that are not amenable to computation or straightforward analysis. Expert systems, also called knowledge based systems, pattern recognition (image and voice), artificial neural networks, theorem proving, and game playing are representative of applications within this category.

**How do we define *software engineering*?**

A definition proposed by Fritz Bauer [Software engineering is] the establishment and use of sound engineering principles in order to obtain economically software that is reliable and works efficiently on real machines

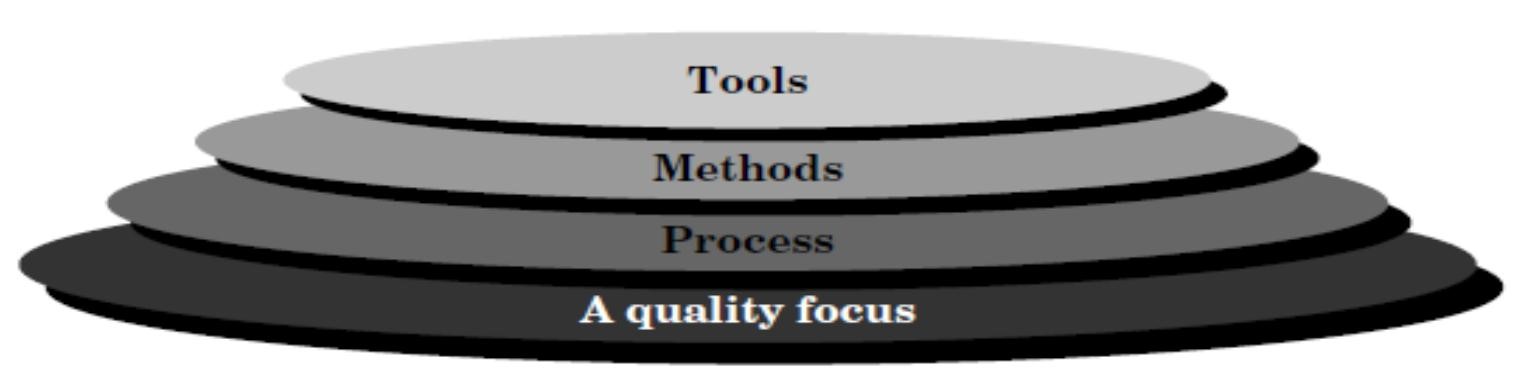
The IEEE [IEE93] has developed a more comprehensive definition when it states: Software Engineering: (1) the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software. (2) The study of approaches as in (1).

**SOFTWARE ENGINEERING: A LAYERED TECHNOLOG**Y:

Software engineering is a layered technology, any engineering approach (including software engineering) must rest on an organizational commitment to quality. Total quality management and similar philosophies foster a continuous process improvement culture, and this culture ultimately leads to the development of increasingly more mature approaches to software engineering. The bedrock that supports software engineering is a quality focus. The foundation for software engineering is the *process* layer. Software engineering process is the glue that holds the technology layers together and enables rational and timely development of computer software. Process defines a framework for a set of *key process areas* (KPAs) that must be established for effective delivery of software engineering technology. The key process areas form the basis for management control of software projects and establish the context in which technical methods are applied, work products (models, documents, data, reports, forms, etc.) are produced, milestones are established, quality is ensured, and change is properly managed.

Software engineering *methods* provide the technical how-to's for building software. Methods encompass a broad array of tasks that include requirements analysis, design, program construction, testing, and support. Software engineering methods rely on a set of basic principles that govern each area of the technology and include modeling activities and other descriptive techniques.

Software engineering *tools* provide automated or semi-automated support for the process and the methods. When tools are integrated so that information created by one tool can be used by another, a system for the support of software development, called *computer-aided software engineering,* is established. CASE combines software, hardware, and a software engineering database (a repository containing important information about analysis, design, program construction, and testing) to create a software engineering environment analogous to CAD/CAE (computer-aided design/engineering) for hardware.



Software engineering layers

**What Is Visual Modeling?**

If you were building a new addition to your house, you probably wouldn't start by just buying a bunch of wood and nailing it together until it looks about right. Similarly, you'd be more than a little concerned if the contractor doing the job decided to "wing it" and work without plans. You'd want some blueprints to follow so you can plan and structure the addition before you start working. Odds are, the addition will last longer this way. You wouldn't want the whole thing to come crashing down with the slightest rain.

Models do the same thing for us in the software world. They are the blueprints for systems. A blueprint helps you plan an addition before you build it; a model helps you plan a system before you build it. It can help you be sure the design is sound, the requirements have been met, and the system can withstand even a hurricane of requirement changes.

As you gather requirements for your system, you'll take the business needs of the users and map them into requirements that your team can use and understand. Eventually, you'll want to take these requirements and generate code from them. By formally mapping the requirements to the code, you can ensure that the requirements are actually met by the code, and that the code can easily be traced back to the requirements.

This process is called *modeling*. The result of the modeling process is the ability to trace the business needs to the requirements to the model to the code, and back again, without getting lost along the way.

*Visual modeling* is the process of taking the information from the model and displaying it graphically using a standard set of graphical elements. A standard is vital to realizing one of the benefits of visual modeling: communication.Communication between users, developers, analysts, testers, managers, and anyone elseinvolved with a project is the primary purpose of visual modeling. You could accomplish this communication using nonvisual (textual) information, but on the whole, humans are visual creatures. We seem to be able to understand complexity better when it is displayed to us visually as opposed to written textually. By producing visual models of a system, we can show how the system works on several levels. We can model the interactions between the users and a system. We can model the interactions of objects within a system. We can even model the interactions between systems, if we so desire.

After creating these models, we can show them to all interested parties, and those parties can glean theinformation they find valuable from the model. For example, users can visualize the interactions they will make with the system from looking at a model. Analysts can visualize the interactions between objects from the models. Developers can visualize the objects that need to be developed and what each one needs to accomplish. Testers can visualize the interactions between objects and prepare test cases based on these interactions. Project managers can see the whole system and how the parts interact. And chief information Officers can look at high−level models and see how systems in their organization interact with one another. All in all, visual models provide a powerful tool for showing the proposed system to all of the interested parties.

**Systems of Graphical Notation**

One important consideration in visual modeling is what graphical notation to use to represent various aspects of a system. Many people have proposed notations for visual modeling. Some of the popular notations that have strong support are *Booch, Object Modeling Technology (OMT), and UML*.Rational Rose supports these three notations; however, UML is a standard that has been adopted by themajority of the industry as well as the standards' governing boards such as ANSI and the Object Management Group (OMG).

**Booch Notation:**

The *Booch*method is named for its inventor, Grady Booch, at Rational Software Corporation. He has written several books discussing the needs and benefits of visual modeling, and has developed a notation of graphical symbols to represent various aspects of a model. For example, objects in this notation are represented by clouds, illustrating the fact that objects can be almost anything. Booch's notation also includes various arrows to represent the types of relationships between objects. Figure 1is a sampling of the objects and relationships represented in the Booch notation.

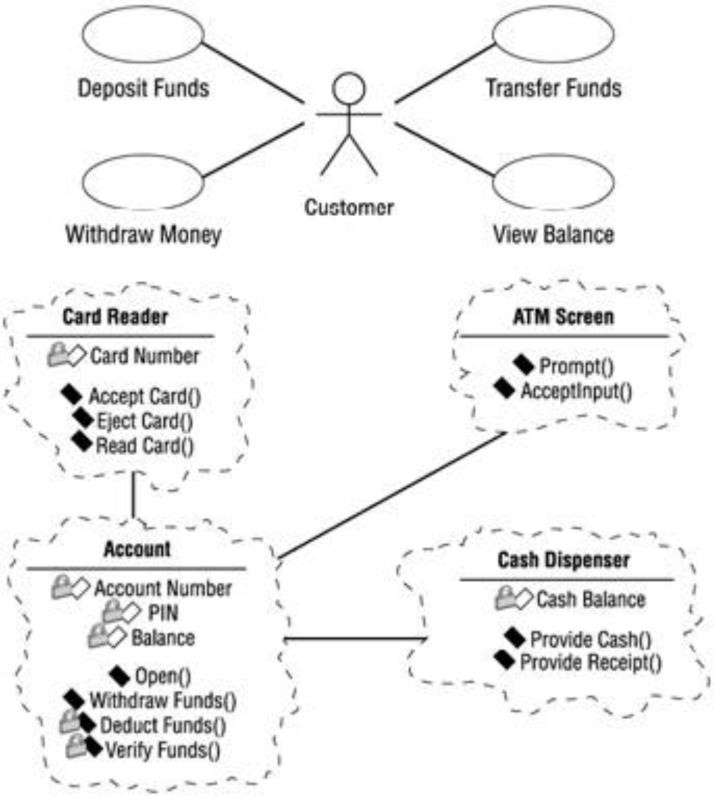


Figure 1.1: Examples of symbols in the Booch notation

**Object Management Technology (OMT):**

The *OMT* notation comes from Dr. James Rumbaugh, who has written several books about systems analysis and design. In an aptly titled book, *Object−Oriented Modeling and Design* (Prentice Hall, 1990), Rumbaugh discusses the importance of modeling systems in real−world components called objects. OMT uses simpler graphics than Booch to illustrate systems. A sampling of the objects and relationships represented in the OMT notation follows in Figure 1.1.

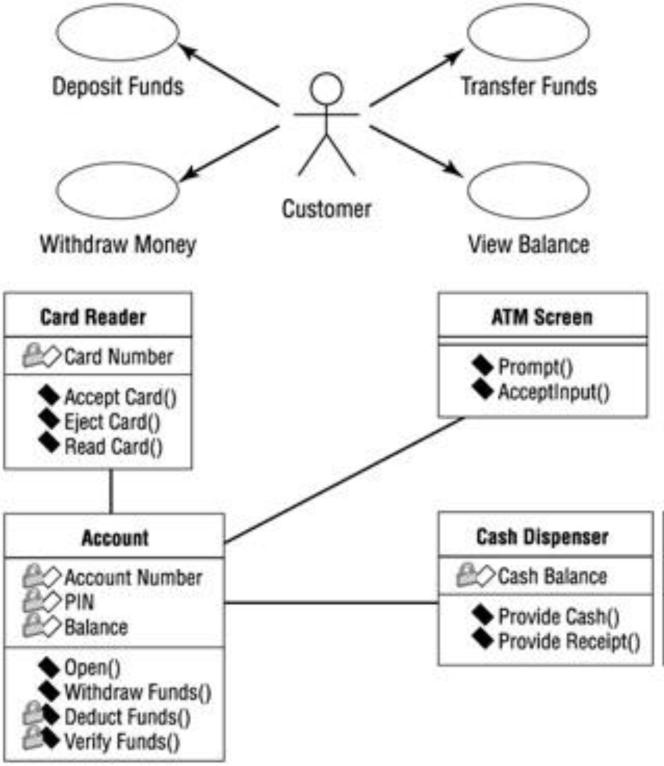


Figure 1.2: Examples of symbols in the OMT notation

**Unified Modeling Language (UML):**

UML notation comes from a collaborative effort of Grady Booch, Dr. James Rumbaugh, Ivar Jacobson, Rebecca Wirfs−Brock, Peter Yourdon, and many others. Jacobson is a scholar who has written about capturing system requirements in packages of transactions called *use cases*. We will discuss use cases in detail in Chapter 4. Jacobson also developed a method for system design called *Object−Oriented SoftwareEngineering (OOSE)* that focused on analysis. Booch, Rumbaugh, and Jacobson, commonly referred to as the "three amigos," all work at Rational Software Corporation and focus on the standardization and refinement of UML. UML symbols closely match those of the Booch and OMT notations, and also include elements from other notations. Figure 1.3.Shows a sample of UML notation.

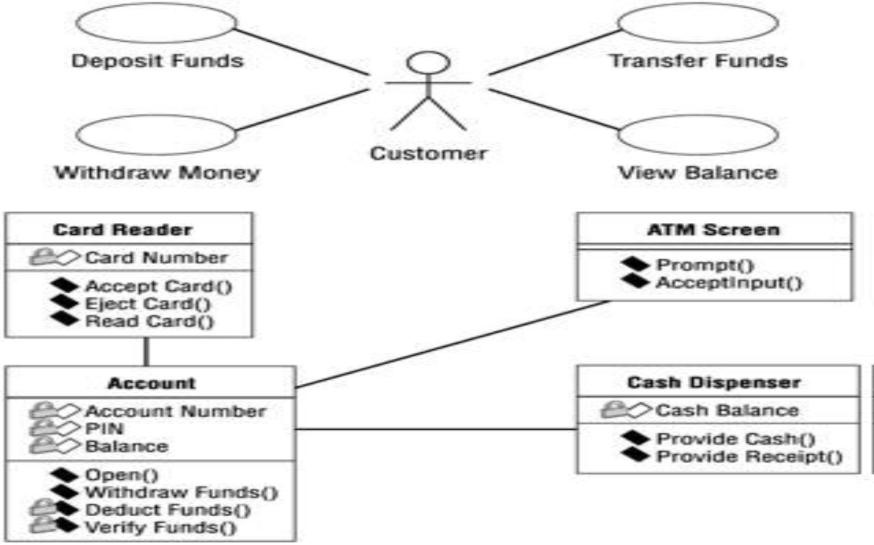


Figure 1.3: Examples of symbols in UML notation

Each of the three amigos of UML began to incorporate ideas from the other methodologies. Official unification of the methodologies continued until late 1995, when version 0.8 of the Unified Method was introduced. The Unified Method was refined and changed to the Unified Modeling Language in 1996. UML 1.0 was ratified and given to the Object Technology Group in 1997, and many major software development companies began adopting it. In 1997, OMG released UML 1.1 as an industry standard.Over the past years, UML has evolved to incorporate new ideas such as web−based systems and datamodeling. The latest release is UML 1.3, which was ratified in 2000. The specification for UML 1.3 can be found at the Object Management Group's website, http://www.omg.org/.

**Types of UML Diagrams**

Each UML diagram is designed to let developers and customers view a software system from a different perspective and in varying degrees of abstraction. UML diagrams commonly created in visual modeling tools include:

[**Use Case Diagram**](http://atlas.kennesaw.edu/~dbraun/csis4650/A%26D/UML_tutorial/use_case.htm) displays the relationship among actors and use cases.

[**Class Diagram**](http://atlas.kennesaw.edu/~dbraun/csis4650/A%26D/UML_tutorial/class.htm) models class structure and contents using design elements such as classes, packages and objects. It also displays relationships such as containment, inheritance, associations and others. [**Interaction Diagrams**](http://atlas.kennesaw.edu/~dbraun/csis4650/A%26D/UML_tutorial/interaction.htm)

* **Sequence Diagram**displays the time sequence of the objects participating in the interaction. This consists of the vertical dimension (time) and horizontal dimension (different objects)
* **Collaboration Diagram**displays an interaction organized around the objects and their links to one another. Numbers are used to show the sequence of messages.

[**State Diagram**](http://atlas.kennesaw.edu/~dbraun/csis4650/A%26D/UML_tutorial/state.htm) displays the sequences of states that an object of an interaction goes through during its life in response to received stimuli, together with its responses and actions.

[**Activity Diagram**d](http://atlas.kennesaw.edu/~dbraun/csis4650/A%26D/UML_tutorial/activity.htm)isplays a special state diagram where most of the states are action states and most of the transitions are triggered by completion of the actions in the source states. This diagram focuses on flows driven by internal processing [**Physical Diagrams**](http://atlas.kennesaw.edu/~dbraun/csis4650/A%26D/UML_tutorial/physical.htm)

* **Component Diagram** displays the high level packaged structure of the code itself. Dependencies among components are shown, including source code components, binary code components, and executable components. Some components exist at compile time, at link time, at run times well as at more than one time.
* **Deployment Diagram**displays the configuration of run-time processing elements and the software components, processes, and objects that live on them. Software component instances represent run-time manifestations of code units

**UML Diagram Classification—Static, Dynamic, and Implementation**

A software system can be said to have two distinct characteristics: a structural, "static" part and a behavioral, "dynamic" part. In addition to these two characteristics, an additional characteristic that a software system possesses is related to implementation. Before we categorize UML diagrams into each of these three characteristics, let us take a quick look at exactly what these characteristics are.

* **Static:** The static characteristic of a system is essentially the structural aspect of the system. The static characteristics define what parts the system is made up of.
* **Dynamic:** The behavioral features of a system; for example, the ways a system behaves in response to certain events or actions are the dynamic characteristics of a system.
* **Implementation:** The implementation characteristic of a system is an entirely new feature that describes the different elements required for deploying a system. The UML diagrams that fall under each of these categories are:
* Static o Use case diagram o Class diagram
* Dynamic o State diagram o Activity diagram o Sequence diagram o Collaboration diagram
* Implementation o Component diagram o Deployment diagram

Finally, let us take a look at the 4+1 view of UML diagrams.

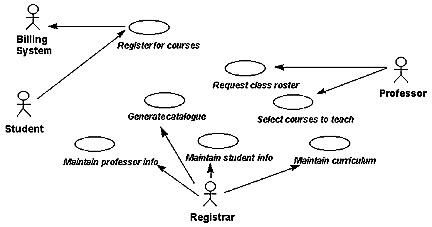
**4+1 View of UML Diagrams**

Considering that the UML diagrams can be used in different stages in the life cycle of a system. The 4+1 view offers a different perspective to classify and apply UML diagrams. The 4+1 view is essentially how a system can be viewed from a software life cycle perspective. Each of these views represents how a system can be modeled. This will enable us to understand where exactly the UML diagrams fit in and their applicability. These different views are:

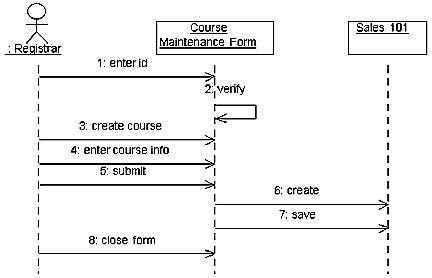
* **Design View**: The design view of a system is the structural view of the system. This gives an idea of what a given system is made up of. Class diagrams and object diagrams form the design view of the system.
* **Process View**: The dynamic behavior of a system can be seen using the process view. The different diagrams such as the state diagram, activity diagram, sequence diagram, and collaboration diagram are used in this view.
* **Component View**: Next, you have the component view that shows the grouped modules of a given system modeled using the component diagram.
* **Deployment View**: The deployment diagram of UML is used to identify the deployment modules for a given system. This is the deployment view of the system.
* **Use case View**: Finally, we have the use case view. Use case diagrams of UML are used to view a system from this perspective as a set of discrete activities or transactions.

**UML diagrams for case study:**

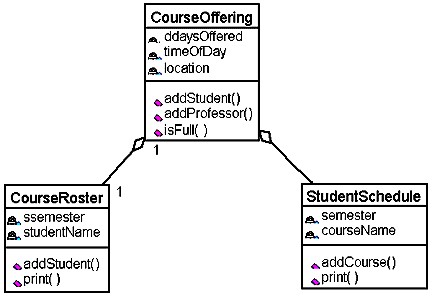
# 1. Use case diagram



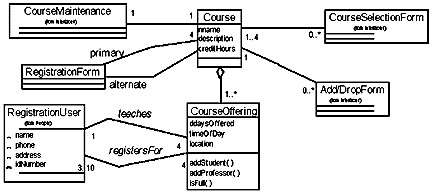
# 2. Sequence Diagram for the Add a Course Scenario



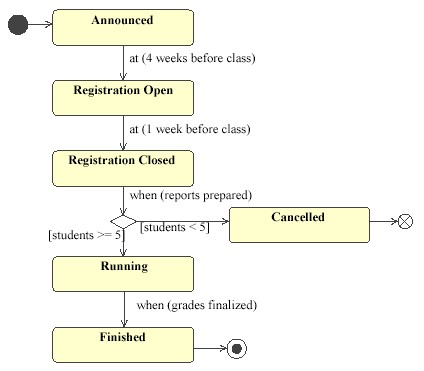
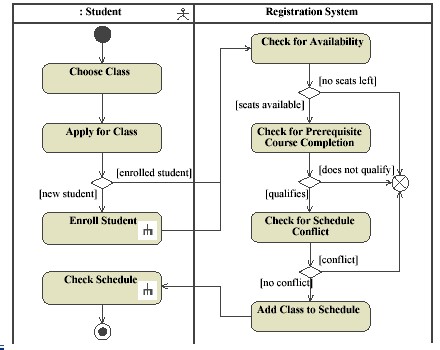
# 2. Main Class Diagram for the University Artifacts Package



# 3. Course Reporting Class Diagram in the University Artifacts Package



# 4. Activity diagram



**Rational Rose:**

Rational Rose is an object-oriented Unified Modeling Language (UML) software design tool intended for visual modeling and component construction of enterprise-level software applications. In much the same way a theatrical director blocks out a play, a software designer uses Rational Rose to visually create (model) the framework for an application by blocking out classes with actors (stick figures), use case elements (ovals), objects (rectangles) and messages/relationships (arrows) in a sequence diagram using drag-and-drop symbols. Rational Rose documents the diagram as it is being constructed and then generates code in the designer's choice of C++, Visual Basic, Java, Oracle8, CORBA or Data Definition Language.

Rational Rose is commercial case-tool software. It supports two essential elements of modern software engineering: component based development and controlled iterative development. Models created with Rose can be visualized with several UML diagrams. Rose also supports Round-Trip engineering with several lan4guages.

Two popular features of Rational Rose are its ability to provide iterative development and round-trip engineering. Rational Rose allows designers to take advantage of iterative development (sometimes called evolutionary development) because the new application can be created in stages with the output of one iteration becoming the input to the next. (This is in contrast to waterfall development where the whole project is completed from start to finish before a user gets to try it out.) Then, as the developer begins to understand how the components interact and makes modifications in the design, Rational Rose can perform what is called "round-trip engineering" by going back and updating the rest of the model to ensure the code remains consistent.

**Summary of the Rational Rose Positive factors**

* The tool itself was quite easy to install.
* The creation of the different diagrams can be learned quite fast.
* Code generation is simple.
* C++ Analyzer was also easy to use (though it’s functionality could be included in theRose itself)

**Negative factors**

* At first the tool seems to be quite complex.
* Some minor bugs were found.
* Separate tool had to be used (and learned) to reverse-engineer files.
* Layout manager could have been a bit more effective. • Generated code was a bit obfuscated.

ROSE is Rational Object Oriented Software Engineering. Rational Rose is a set of visual modeling tools for development of object oriented software. Rose uses the UML to provide graphical methods for non-programmers wanting to model business processes as well as programmers modeling application logic.Rational Rose includes tools for reverse engineering as well as forward engineering of classes and component architectures. You can gain valuable insights to your actual constructed architecture and pinpoint deviations from the original design.Rose offers a fast way for clients and new employees to become familiar with system internals.

**RATIONAL ROSE INTERFACE:**

**Parts of the Screen:**

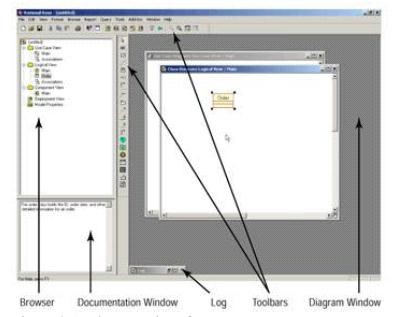
The five primary pieces of the Rose interface are the browser, the documentation window, the toolbars, the diagram window, and the log. In this section, we'll look at each of these. Briefly, their purposes are:

**Browser** Used to quickly navigate through the model

**Documentation window** Used to view or update documentation of model elements

**Toolbars** Used for quick access to commonly used commands

**Diagram window** Used to display and edit one or more UML diagrams **Log** Used to view errors and report the results of various commands



**Conclusion:** In this Practical, we have studied software engineering fundamentals, UML and RATIONAL ROSE INTERFACE

.