



**SOFTWARE ENGINEER LAB FILE
(KCA-352)
(SESSION 2021-2022)**

Submitted To :-

**Mr. Sandeep Singh
Assistant Professor**

Submitted By :-

**Shubham Patkar
Roll No : 2002250140017
3rd Sem /2nd year**

ACCURATE GROUP OF INSTITUTES

GREATER NOIDA UP

**Departments of
Master of Computer Application**

Project based Software Engineering LAB

LABORATORY MANUAL

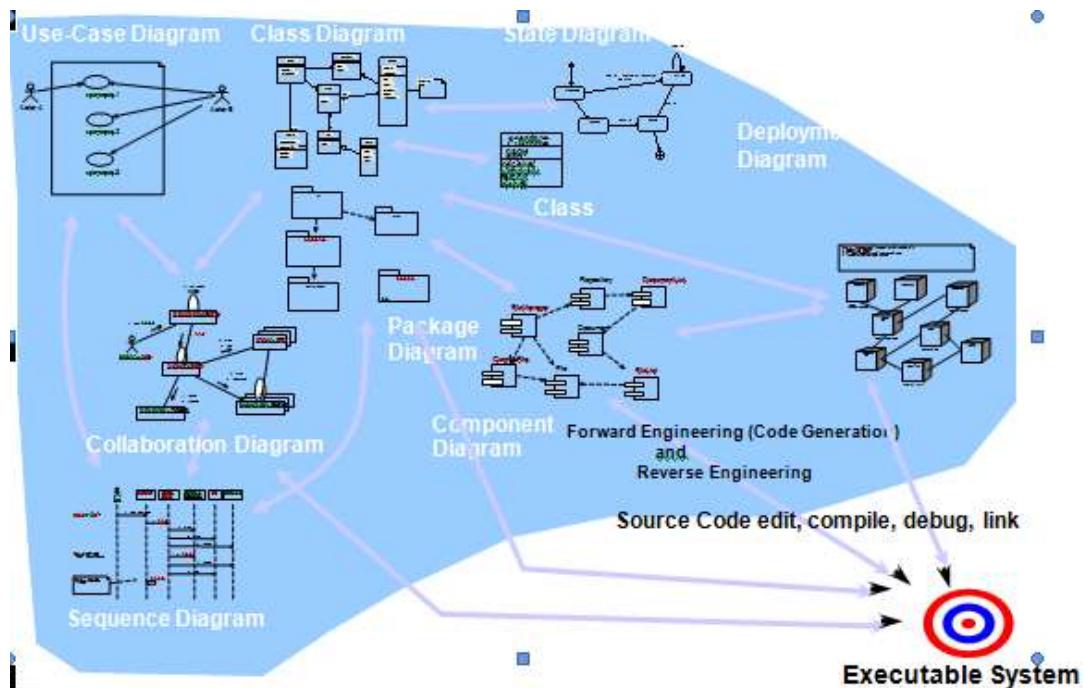
Prepared by

Mr. Sandeep Kr Singh

ACADEMIC YEAR 2021 - 2022 (ODD SEMESTER)

1

Introduction and Project Definition



Introduction and Project Definition

Objectives

- Introduce the lab environment and tools used in the software engineering lab.
- Discuss the Project & learn how to write project definition.

1. Outline

- Introduction to the lab plan and objectives.
- Project definition.

2. Background

The software engineer is a key person analyzing the business, identifying opportunities for improvement, and designing information systems to implement these ideas. It is important to understand and develop through practice the skills needed to successfully design and implement new software systems.

2.1 Introduction

- In this lab you will practice the software development life cycle (project management, requirements engineering, systems modeling, software design, prototyping, and testing) using CASE tools within a team work environment.
- UML notation is covered in this lab as the modeling language for analysis and design.

2.2 Tools Used in the Lab

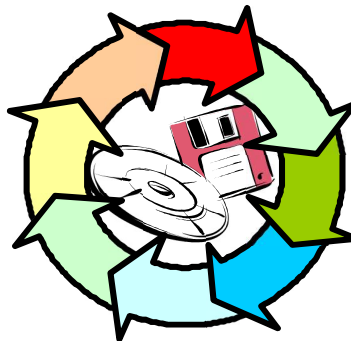
- SWE lab is one of the most challenging of all labs. Developing a complete software application requires from each of you a good level of know-how of various tools.
- There are some tools which will be taught, but there are some which are assumed you already know, if you don't, then you learn should it individually.
 - MS Project: for Project Planning / Management
 - Rational Rose: for UML diagrams (Object Oriented Analysis and Design)

2.3 Software Engineering Lab Objectives

- Learn the software life cycle phases (project management, requirements engineering, software design, prototyping and testing).
- Practice the software phases using a project.
- Learn a number of CASE tools and use them in a project within a team work environment.
- Get familiar with UML (modeling language for analysis and design).

2

Software Processes



Software Processes

Objectives

- Obtain a deeper understanding of software process models and software processes.

1. Outline

- Review of the basic software process models and software processes.

2. Background

IEEE defined a software process as: “a set of activities, practices and transformations that people use to develop and maintain software and the associated products, e.g., project plans, design documents, code, test cases and user manual”.

Following the software process will stabilize the development life cycle and make the software more manageable and predictable. It will also reduce software development risk and help to organize the work products that need to be produced during a software development project. A well-managed process will produce high quality products on time and under budget. So following a mature software process is a key determinant to the success of a software project.

A number of software processes are available. However, no single software process works well for every project. Each process works best in certain environments. Examples of the available software process models include: the Waterfall model (the Linear model), the evolutionary development, the formal systems development, and reuse-based (component-based) development. Other process models support iteration; these include: the Incremental model, the Spiral model, and Extreme Programming.

3

Project Planning and Management



Project Planning and Management

Objectives

- Gain understanding of project management.
- Learn how to prepare project plans.
- Learn about project risks.
- Learn MS Project case tool.

1. Outline

- Project work planning.
- Risk management.
- MS project.
- Examples.

2. Background

Project management is the process of planning and controlling the development of a system within a specified timeframe at a minimum cost with the right functionality.

2.1 Project Work Plan

Prepare a list of all tasks in the work breakdown structure, plus:

- Duration of task.
- Current task status.
- Task dependencies.
- Key milestone dates.

2.2 Tracking Progress

- Gantt Chart:
 - Bar chart format.
 - Useful to monitor project status at any point in time.
- PERT Chart:
 - Flowchart format.
 - Illustrate task dependencies and critical path.

2.3 Risk Management

- Risk management is concerned with identifying risks and drawing up plans to minimize their effect on a project.
- A risk is a probability that some adverse circumstance will occur.
- **Project** risks which affect schedule or resources.
- **Product** risks which affect the quality or performance of the software being developed.
- **Business** risks which affect the organization developing the software.

2.4 Risk Management Process

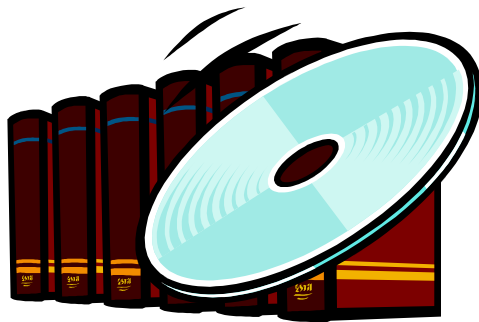
- Risk identification: identify project, product and business risks.
- Risk analysis: assess the likelihood and consequences of these risks.
- Risk planning: draw up plans to avoid or minimize the effects of the risk.
- Risk monitoring: monitor the risks throughout the project.

3. CASE Tools

- Microsoft Project is the clear market leader among desktop project management applications.
- Primavera Project Planner: multi-project, multi-user project, and resource planning and scheduling.
- SureTrak Project Manager: resource planning and control for small-to-medium sized projects.
- According to the Gartner Group, it accounts for about two-thirds of all project management software sales.

4

Software Requirement Specification (SRS)



Software Requirement Specification (SRS)

Objectives

- Gain a deeper understanding of the Software Requirement Specification phase and the Software Requirement Specification (SRS).
- Learn how to write requirements and specifications.
- Gain exposure to requirements management using RequisitePro.

1. Outline

- Review of the requirements engineering process.
- Write requirements and specifications.
- RequisitePro tutorial.
- Software Requirement Specification (SRS).

2. Background

A requirement is a statement of a behavior or attribute that a system must possess for the system to be acceptable to a stakeholder.

Software Requirement Specification (SRS) is a document that describes the requirements of a computer system from the user's point of view. An SRS document specifies:

- The required behavior of a system in terms of: input data, required processing, output data, operational scenarios and interfaces.
- The attributes of a system including: performance, security, maintainability, reliability, availability, safety requirements and design constraints.

Requirements management is a systematic approach to eliciting, organizing and documenting the requirements of a system. It is a process that establishes and maintains agreement between the customer and the project team on the changing requirements of a system.

Requirements management is important because, by organizing and tracking the requirements and managing the requirement changes, you improve the chances of completing the project on time and under budget. Poor change management is a key cause of project failure.

2.1 Requirements Engineering Process

Requirements engineering process consists of four phases:

- Requirements elicitation: getting the customers to state exactly what the requirements are.
- Requirements analysis: making qualitative judgments and checking for consistency and feasibility of requirements.
- Requirements validation: demonstrating that the requirements define the system that the customer really wants.
- Requirements management: the process of managing changing requirements during the requirements engineering process and system development, and identifying missing and extra requirements.

2.2 Writing Requirements

Requirements always need to be correct, unambiguous, complete, consistent, and testable.

2.2.1 Recommendations When Writing Requirements

- Never assume: others do not know what you have in mind.
- Use meaningful words; avoid words like: process, manage, perform, handle, and support.

- State requirements not features:
 - Feature: general, tested only for existence.
 - Requirement: specific, testable, measurable.
- Avoid:
 - Conjunctions: ask yourself whether the requirement should it be split into two requirements.
 - Conditionals: if, else, but, except, although.
 - Possibilities: may, might, probably, usually.

2.3 Writing Specifications

Specification is a description of operations and attributes of a system. It can be a document, set of documents, a database of design information, a prototype, diagrams or any combination of these things.

Specifications are different from requirements: specifications are sufficiently complete — not only what stakeholders say they want; usually, they have no conflicts; they describe the system as it will be built and resolve any conflicting requirements.

Creating specifications is important. However, you may not create specifications if:

- You are using a very incremental development process (small changes).
- You are building research or proof of concept projects.
- You rebuilding very small projects.
- It is not cheaper or faster than building the product.

2.4 Software Requirement Specification (SRS)

Remember that there is no “Perfect SRS”. However, SRS should be:

- Correct: each requirement represents something required by the target system.
- Unambiguous: every requirement in SRS has only one interpretation
- Complete: everything the target system should do is included in SRS (no sections are marked TBD-to be determined).
- Verifiable: there exists some finite process with which a person/machine can check that the actual as-built software product meets the requirements.
- Consistent in behavior and terms.
- Understandable by customers.
- Modifiable: changes can be made easily, completely and consistently.
- Design independent: doesn't imply specific software architecture or algorithm.
- Concise: shorter is better.
- Organized: requirements in SRS are easy to locate; related requirements are together.
- Traceable: each requirement is able to be referenced for later use (by the using paragraph numbers, one requirement in each paragraph, or by using convention for indication requirements)

3. CASE Tools

RequisitePro is a powerful, easy-to-use requirements management tool that helps teams manage project requirements comprehensively, promotes communication and collaboration among team members, and reduces project risk. It thereby increases the chances of delivering a product that the client wants and does so in a timely manner.

RequisitePro offers the power of a database and Microsoft Word and is integrated with other Rational Suite products.

5

Introduction to UML and Use Case Diagram



Introduction to UML and Use Case Diagram

Objectives

- Study the benefits of visual modeling.
- Learn use case diagrams: discovering actors and discovering use cases.
- Practice use cases diagrams using Rational Rose.

1. Outline

- Visual modeling.
- Introduction to UML.
- Introduction to visual modeling with UML.
- Use case diagrams: discovering actors and use cases.

2. Background

Visual Modeling is a way of thinking about problems using models organized around real-world ideas. Models are useful for understanding problems, communicating with everyone involved with the project (customers, domain experts, analysts, designers, etc.), modeling enterprises, preparing documentation, and designing programs and databases

2.1 Visual Modeling

- Capture the structure and behavior of architectures and components.
- Show how the elements of the system fit together.
- Hide or expose details appropriate for the task.
- Maintain consistency between a design and its implementation.
- Promote unambiguous communication.

2.2 What is UML?

The UML is the standard language for visualizing, specifying, constructing and documenting the artifacts of a software-intensive system. UML can be used with all processes throughout the development life cycle and across different implementation technologies.

2.3 History of UML

The UML is an attempt to standardize the artifacts of analysis and design: semantic models, syntactic notation and diagrams. The first public draft (version 0.8) was introduced in October 1995. Feedback from the public and Ivar Jacobson's input were included in the next two versions (0.9 in July 1996 and 0.91 in October 1996). Version 1.0 was presented to the Object Management Group (OMG) for standardization in July 1997. Additional enhancements were incorporated into the 1.1 version of UML, which was presented to the OMG in September 1997. In November 1997, the UML was adopted as the standard modeling language by the OMG.

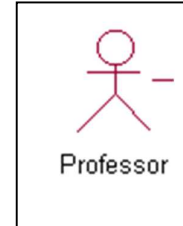
2.4 Putting UML into Work: Use Case Diagram

The behavior of the system under development (i.e. what functionality must be provided by the system) is documented in a use case model that illustrates the system's

intended functions (use cases), its surroundings (actors), and relationships between the use cases and actors (use case diagrams).

2.5 Actors

- Are NOT part of the system – they represent anyone or anything that must interact with the system.
- Only input information to the system.
- Only receive information from the system.
- Both input to and receive information from the system.
- Represented in UML as a stickman.



2.6 Use Case

- A sequence of transactions performed by a system that yields a measurable result of values for a particular actor
- A use case typically represents a major piece of functionality that is complete from beginning to end. A use case must deliver something of value to an actor.



2.7 Use Case Relationships

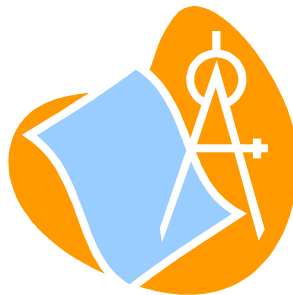
- Between actor and use case.
- Association / Communication.
- Arrow can be in either or both directions; arrow indicates who initiates communication.
- Between use cases (generalization):
 - Uses
 - Where multiple use cases share pieces of same functionality.
 - Extends
 - Optional behavior.
 - Behavior only runs under certain conditions (such as alarm).
 - Several different flows run based on the user's selection.

3. CASE Tools

The Rational Rose product family is designed to provide the software developer with a complete set of visual modeling tools for development of robust, efficient solutions to real business needs in the client/server, distributed enterprise and real-time systems environments. Rational Rose products share a common universal standard, making modeling accessible to nonprogrammers wanting to model business processes as well as to programmers modeling applications logic.

6

System Modeling



System Modeling

Objective:

Deeper understanding of System modeling:

- Data model: entity-relationship diagram (ERD).
- Functional model: data flow diagram (DFD).

- **Outline**

System analysis model elements:

- Data model: entity-relationship diagram (ERD)
- Functional model: data flow diagram (DFD)

- **Background**

Modeling consists of building an abstraction of reality. These abstractions are simplifications because they ignore irrelevant details and they only represent the relevant details (what is relevant or irrelevant depends on the purpose of the model).

2.1 Why Model Software?

Software is getting larger, not smaller; for example, Windows XP has more than 40 million lines of code. A single programmer cannot manage this amount of code in its entirety. Code is often not directly understandable by developers who did not participate in the development; thus, we need simpler representations for complex systems (modeling is a mean for dealing with complexity).

A wide variety of models have been in use within various engineering disciplines for a long time. In software engineering a number of modeling methods are also available.

2.2 Analysis Model Objectives

- To describe what the customer requires.
- To establish a basis for the creation of a software design.
- To define a set of requirements that can be validated once the software is built.

2.3 The Elements of the Analysis Model

The generic analysis model consists of:

- An entity-relationship diagram (data model).
- A data flow diagram (functional model).
- A state transition diagram (behavioral model).

NOTE: state transition diagram will be covered in lab 11.

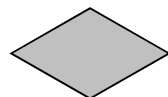
2.3.1 Entity Relationship Diagram

An entity relationship diagram (ERD) is one means of representing the objects and their relationships in the data model for a software product.

Entity Relationship diagram notation:



Entity



Relationship

To create an ERD you need to:

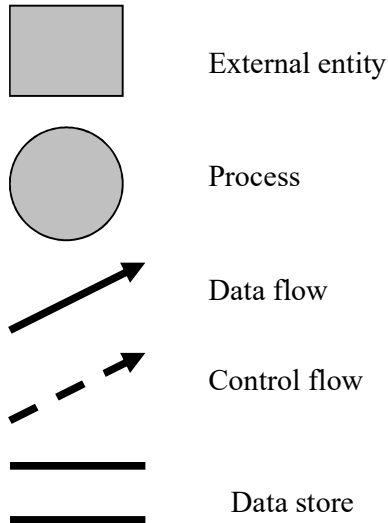
- Define “objects” by underlining all nouns in the written statement of scope: producers/consumers of data, places where data are stored, and “composite” data items.

- Define “operations” by double underlining all active verbs: processes relevant to the application and data transformations.
- Consider other “services” that will be required by the objects.
- Then you need to define the relationship which indicates “connectedness”: a "fact" that must be "remembered" by the system and cannot be or is not computed or derived mechanically.

2.3.2 Data Flow Diagram

A data flow data diagram is one means of representing the functional model of a software product. DFDs do not represent program logic like flowcharts do.

Data flow diagram notation:



To create a DFD you need to:

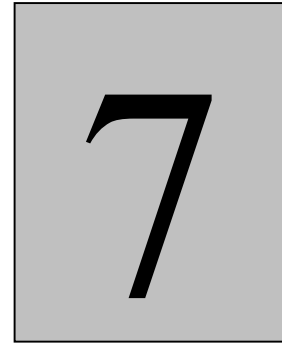
- Review ERD to isolate data objects and grammatical parse to determine operations.
- Determine external entities (producers and consumers of data).
- Create a level 0 DFD “Context Diagram” (one single process).
- Balance the flow to maintain data flow continuity.
- Develop a level 1 DFD; use a 1:5 (approx.) expansion ratio.

Data Flow Diagram Guidelines:

- All icons must be labeled with meaningful names.
- Always show external entities at level 0.
- Always label data flow arrows.
- Do not represent procedural logic.
- Each bubble is refined until it does just one thing.

- **CASE Tools**

You can use MS word to create your ERD and DFD since we do not have a license for a case tool that supports these diagrams.



Documenting Use Cases and Activity Diagrams



Documenting Use Cases and Activity Diagrams

Objectives

- Study how to document use cases in detail.
- Know about scenarios (flow of events) and its importance.
- Deeper understanding of UML activity diagrams.
- Practicing flow of events and activity diagrams using Rational Rose.

1. Outline

- Writing flow of events.
- Flow of events template and example.
- Activity diagrams.
- Examples.

2. Background

Each use case is documented with a flow of events. The flow of events for a use case is a description of the events needed to accomplish the required behavior of the use case.

Activity diagrams may also be created at this stage in the life cycle. These diagrams represent the dynamics of the system. They are flow charts that are used to show the workflow of a system; that is, they show the flow of control from one activity to another in the system,

2.1 Flow of Events

A description of events required to accomplish the behavior of the use case, that:

- Show WHAT the system should do, not HOW the system does it.
- Written in the language of the domain, not in terms of implementation.
- Written from an actor point of view.

A **flow of events** document is created for each use case:

- Actors are examined to determine how they interact with the system.
- Break down into the most atomic actions possible.

2.2 Contents of Flow of Events

- When and how the use case starts and ends.
- What interaction the use case has with the actors.
- What data is needed by the use case.
- The normal sequence of events for the use case.
- The description of any alternate or exceptional flows.

2.3 Template for the flow of events document

Each project should use a standard template for the creation of the flow of events document. The following template seems to be useful.

X Flow of events for the <name> use case

X.1 Preconditions

X.2 Main flow

X.3 Sub-flows (if applicable)

X.4 Alternative flows

where X is a number from 1 to the number of use cases.

A sample completed flow of events document for the *Select Courses to Teach* use case follows.

1. Flow of Events for the Select Courses to Teach Use Case

1.1 Preconditions

Create course offerings sub-flow of the maintain course information use case must execute before this use case begins.

1.2 Main Flow

This use case begins when the professor logs onto the registration system and enters his/her password. The system verifies that the password is valid (E-1) and prompts the professor to select the current semester or a future semester (E-2). The professor enters the desired semester. The system prompts the Professor to select the desired activity: ADD, DELETE, REVIEW, PRINT, or QUIT.

If the activity selected is ADD, the S-1: add a course offering sub-flow is performed.

If the activity selected is DELETE, the S-2: delete a course offering sub-flow is performed.

If the activity selected is REVIEW, the S-3: review schedule sub-flow is performed.

If the activity selected is PRINT, the S-4: print a schedule sub-flow is performed.

If the activity selected is QUIT, the use case ends.

1.3 Sub-flows

S-1: Add a Course Offering:

The system displays the course screen containing a field for a course name and number. The professor enters the name and number of a course (E-3). The system displays the course offerings for the entered course (E-4). The professor selects a course offering. The system links the professor to the selected course offering (E-5). The use case then begins again.

S-2: Delete a Course Offering:

The system displays the course offering screen containing a field for a course offering name and number. The professor enters the name and number of a course offering (E-6). The system removes the link to the professor (E-7). The use case then begins again.

S-3: Review a Schedule:

The system retrieves (E-8) and displays the following information for all course offerings for which the professor is assigned: course name, course number, course offering number, days of the week, time, and location. When the professor indicates that he or she is through reviewing, the use case begins again.

S-4: Print a Schedule

The system prints the professor schedule (E-9). The use case begins again.

1.4 Alternative Flows

- E-1: An invalid professor ID number is entered. The user can re-enter a professor ID number or terminate the use case.
- E-2: An invalid semester is entered. The user can re-enter the semester or terminate the use case.
- E-3: An invalid course name/number is entered. The user can re-enter a valid name/number combination or terminate the use case.
- E-4: Course offerings cannot be displayed. The user is informed that this option is not available at the current time. The use case begins again.
- E-5: A link between the professor and the course offering cannot be created. The information is saved and the system will create the link at a later time. The use case continues.
- E-6: An invalid course offering name/number is entered. The user can re-enter a valid course offering name/number combination or terminate the use case.
- E-7: A link between the professor and the course offering cannot be removed. The information is saved and the system will remove the link at a later time. The use case continues.
- E-8: The system cannot retrieve schedule information. The use case then begins again.
- E-9: The schedule cannot be printed. The user is informed that this option is not available at the current time. The use case begins again.

Use case flow of events documents are entered and maintained in documents external to Rational Rose. The documents are linked to the use case.

2.4 Activity Diagrams

Activity diagrams are flow charts that are used to show the workflow of a system. They also:

- Represent the dynamics of the system.
- Show the flow of control from activity to activity in the system.
- Show what activities can be done in parallel, and any alternate paths through the flow.

Activity diagrams may be created to represent the flow across use cases or they may be created to represent the flow within a particular use case. Later in the life cycle, activity diagrams may be created to show the workflow for an operation.

2.5 Activity Diagram Notation


- **Activities**- performance of some behavior in the workflow.
- **Transition**- passing the flow of control from activity to activity.
- **Decision**- show where the flow of control branches based on a decision point:
 - Guard condition is used to determine which path from the decision point is taken.
- **Synchronization**-what activities are done concurrently? What activities must be completed before processing may continue (join).

3. CASE Tools

Rational Rose (introduced in lab 5).

8

Object Oriented Analysis: Discovering Classes



Student
name : String
id : int
major : String

Object-Oriented Analysis: Discovering Classes

Objective

- Learn the object-oriented analysis phase by understanding the methods of class elicitation and finding the classes in an object-oriented system.

1. Outline

- Object-Oriented concepts
- Discovering classes' approaches: noun phrase approach, common class patterns, use case driven method, CRC (Class-Responsibility-Collaboration) and mixed approach.
- Examples.

2. Background

Classes: a description of a group of objects with common properties (attributes), common behavior (operations), common relationships to other objects and common semantics.

2.1 Object-Oriented Concepts

- Attribute: the basic data of the class.
- Method (operation): an executable procedure that is encapsulated in a class and is designed to operate on one or more data attributes that are defined as part of the class.
- Object: when specific values are assigned to all the resources defined in a class, the result is an instance of that class. Any instance of any class is called an object.

2.2 Discovering Classes

Discovering and defining classes to describe the structure of a computerized system is not an easy task. When the problem domain is new or unfamiliar to the software developers it can be difficult to discover classes; a cookbook for finding classes does not exist.

2.3 Classes Categories

Classes are divided into three categories:

- Entity: models information and associated behavior that is long-lived, independent of the surrounding, application independent, and accomplishes some responsibility
- Boundary: handles the communication between the system surroundings and the inside of the system, provides interface, and facilitates communication with other systems
- Control: model sequencing behavior specific to one or more use cases. Control classes coordinate the events needed to realize the behavior specified in the use case, and they are responsible for the flow of events in the use case.

2.4 Discovering Classes Approaches

Methods of discovering classes:

2.4.1 Noun Phrase Approach: Examine the requirements and underline each noun.

Each noun is a candidate class; divide the list of candidate classes into:

- Relevant classes: part of the application domain; occur frequently in requirements.
- Irrelevant classes: outside of application domain
- Fuzzy classes: unable to be declared relevant with confidence; require additional analysis

2.4.2 Common Class Patterns: Derives candidate classes from the classification theory of objects; candidate classes and objects come from one of the following sources:

- Tangible things: e.g. buildings, cars.
- Roles: e.g. teachers, students.
- Events: things that happen at a given date and time, or as steps in an ordered sequence: e.g. landing, request, interrupt.
- Interactions: e.g. meeting, discussion.
- Sources, facilities: e.g. departments.
- Other systems: external systems with which the application interacts.
- Concept class: a notion shared by a large community.
- Organization class: a collection or group within the domain.
- People class: roles people can play.
- Places class: a physical location relevant to the system.

2.4.3 Use Case Driven Method: The scenarios - use cases that are fundamental to the system operation are enumerated. Going over each scenario leads to the identification of the objects, the responsibilities of each object, and how these objects collaborate with other objects.

2.4.4 CRC (Class-Responsibility-Collaboration): Used primarily as a brainstorming tool for analysis and design. CRC identifies classes by analyzing how objects collaborate to perform business functions (use cases).

A CRC card contains: name of the class, responsibilities of the class and collaborators of the class. Record name of class at the top; record responsibilities down the left-hand side; record other classes (collaborators) that may be required to fulfill each responsibility on the right-hand side.

CRC cards are effective at analyzing scenarios; they force you to be concise and clear; they are cheap, portable and readily available.

2.4.5 Mixed Approach: A mix of these approaches can be used, one possible scenario is:

- Use CRC for brainstorming.
- Identify the initial classes by domain knowledge.
- Use common class patterns approach to guide the identification of the classes.
- Use noun phrase approach to add more classes.
- Use the use case approach to verify the identified classes.

2.5 Class Elicitation Guidelines

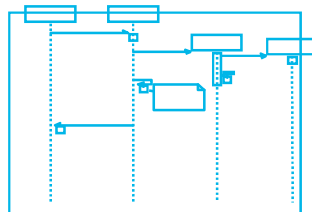
- A class should have a single major role.
- A class should have defined responsibilities (use CRC cards if needed).
- Classes should be of a manageable size: if a class has too many attributes or operations, consider splitting it.
- A class should have a well-defined behavior, preferably by implementing a given requirement or an interface.

3. CASE Tools

Rational Rose (introduced in lab 7).

9

Interaction Diagrams: Sequence & Collaboration Diagrams



Interaction Diagrams: Sequence & Collaboration Diagrams

Objectives

- Better understanding of the interaction diagrams.
- Get familiar with sequence & collaboration diagrams.
- Practice drawing the interaction diagrams using Rational Rose.

1. Outline

- Interaction diagrams:
 - Sequence diagrams
 - Collaboration diagrams

2. Background

Interaction diagrams describe how groups of objects collaborate in some behavior. An interaction diagram typically captures the behavior of a single use case. Interaction diagrams do not capture the complete behavior, only typical scenarios.

2.1 Analyzing a System's Behavior

UML offers two diagrams to model the dynamics of the system: sequence and collaboration diagrams. These diagrams show the interactions between objects.

2.2 Sequence Diagrams

Sequence diagrams are a graphical way to illustrate a scenario:

- They are called sequence diagrams because they show the sequence of message passing between objects.
- Another big advantage of these diagrams is that they show when the objects are created and when they are destroyed. They also show whether messages are synchronous or asynchronous

2.3 Creating Sequence Diagrams

- You must know the scenario you want to model before diagramming sequence diagrams.
- After that specify the classes involved in that scenario.
- List the involved objects in the scenario horizontally on the top of the page.
- Drop a dotted line beneath every object. They are called lifelines.
- The scenario should start by a message pass from the first object.
- You must know how to place the objects so that the sequence is clear.
- You may start the scenario by an actor.
- Timing is represented vertically downward.
- Arrows between life lines represents message passing.
- Horizontal arrows may pass through the lifeline of another object, but must stop at some other object.
- You may add constraints to these horizontal arrows.
- Objects may send messages to themselves.
- Long, narrow rectangles can be placed over the lifeline of objects to show when the object is active. These rectangles are called activation lines.

2.4 Collaboration Diagrams

They are the same as sequence diagrams but without a time axis:

- Their message arrows are numbered to show the sequence of message sending.
- They are less complex and less descriptive than sequence diagrams.
- These diagrams are very useful during design because you can figure out how objects communicate with each other.

2.5 Notes

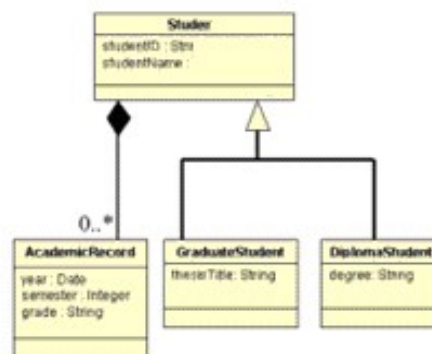
- Always keep your diagrams simple.
- For “IF... then ...” else scenarios, you may draw separate sequence diagrams for the different branches of the “if statement”. You may even hide them, (at least during the analysis phase) and document them by the text description accompanying the sequence diagrams.

3. CASE Tools

Rational Rose (introduced in lab 5).

10

Software Design: Software Architecture and Object- Oriented Design



Software Design: Software Architecture and Object-Oriented Design

Objectives

- Deeper understanding of software design and the software design document (SDD).
- Learn how to find the relationships between classes to create UML class diagram.

1. Outline

- Software design concepts and principals.
- Software architecture.
- Specifying the attributes and the operations and finding the relationships between classes.
- Creating UML class diagram.
- Software design document.

2. Background

The purpose of software design is “to produce a workable (implementable) solution to a given problem.” David Budgen in Software Design: An Introduction.

2.1 The Design Process

Software design is an iterative process that is traceable to the software requirements analysis process. Many software projects iterate through the analysis and design phases several times. Pure separation of analysis and design may not always be possible.

2.2 Design Concepts

- The design should be based on requirements specification.
- The design should be documented (so that it supports implementation, verification, and maintenance).
- The design should use abstraction (to reduce complexity and to hide unnecessary detail).
- The design should be modular (to support abstraction, verification, maintenance, and division of labor).
- The design should be assessed for quality as it is being created, not after the fact.
- Design should produce modules that exhibit independent functional characteristics.
- Design should support verification and maintenance.

2.3 Software Architecture

Software architecture is a description of the subsystems and components of a software system and the relationships between them.

You need to develop an architectural model to enable everyone to better understand the system, to allow people to work on individual pieces of the system in isolation, to prepare for extension of the system and to facilitate reuse and reusability.

2.4 Describing an Architecture Using UML

All UML diagrams can be useful to describe aspects of the architectural model. Four UML diagrams are particularly suitable for architecture modeling:

- Package diagrams
- Subsystem diagrams
- Component diagrams
- Deployment diagrams

2.5 Specifying Classes

Each class is given a name, and then you need to specify:

- Attributes: initially those that capture interesting object states. Attributes can be public, protected, private or friendly/package.
- Operations: can be delayed till later analysis stages or even till design. Operations also can be public, protected, private or friendly/package.
- Object-Relationships:
 - Associations: denote relationships between classes.
 - An aggregation: a special case of association denoting a “consists of” hierarchy.
 - Composition: a strong form of aggregation where components cannot exist without the aggregate.
 - Generalization relationships: denote inheritance between classes.

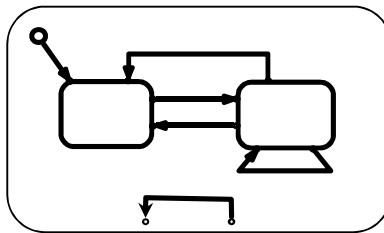
This will build the class diagram, which is a graphical representation of the classes (including their attributes and operations) and their relationship with other classes.

3. CASE Tools

Rational Rose (introduced in lab 7).

11

State Transition Diagram



State Transition Diagram

State Transition Diagrams

Objectives

- Deeper understanding of UML state transition diagrams (STD).
- Practicing using Rational Rose.

1. Outline

- UML state diagrams.
- UML state diagram notation
- UML state details
- Examples

2. Background

Mainly, we use interaction diagrams to study and model the behavior of objects in our system. Sometimes, we need to study the behavior of a specific object that shows complex behavior to better understand its dynamics. For that sake, UML provides state transition diagrams used to model the behavior of objects of complex behavior. In this Lab, UML state transition diagrams will be introduced. We will study their notation and how can we model them using Rational Rose.

2.1 UML State Diagrams

State diagrams show how one specific object changes state as it receives and processes messages:

- Since they are very specific, they are used for analyzing very specific situations if we compare them with other diagrams.
- A state refers to the set of values that describe an object at a specific moment in time.
- As messages are received, the operations associated with the object's parent class are invoked to deal with the messages.
- These messages change the values of these attributes.
- There is no need to prepare a state diagram for every class you have in the system.

2.2 Creating State Transition Diagrams

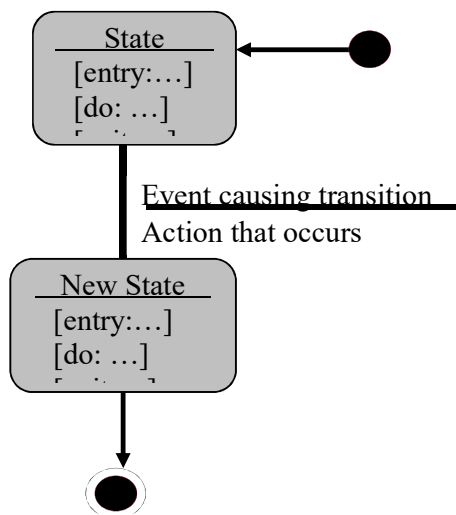
- States are represented by rectangles with rounded corners with an attribute name with a values associated with it.
- The name of the state is placed within the box.
- Events are shown by arrows.
- An event occurs when at an instant in time when a value is changed.
- A message is data passed from one object to another.
- The name of a state usually refers to the name of the attribute and the values associated to it.
- Example, a student object may receive a message to change its name. The state of that object changes from the first name state to the new state name.
- The name of the state is placed in the top compartment.
- State variables are placed in the next compartment.
- The operations associated with the state are listed in the lowest compartment of the state box.

- In the operations part, we usually use one of the following reserved words:
 - **Entry:** a specific action performed on the entry to the state.
 - **Do:** an ongoing action performed while in the state.
 - **On:** a specific action performed while in the state.
 - **Exit:** a specific action performed on exiting the state.
- There are two special states added to the state transition diagram- start state and end state.
- Notation of start state is a solid black circle and for the end state a bull's eye is used.

2.3 State Transition Details

- A state transition may have an action and/or guard condition associated with it and it may also trigger an event.
- An action is the behavior that occurs when the state transition occurs.
- An event is a message that is sent to another object in the system.
- A guard condition is a Boolean expression of attribute values that allows a state transition only if the condition is true.
- Both actions and guards are behaviors of the object and typically become operations. Also they are usually private operations (used by the object itself)
- Actions that accompany all state transitions into a state may be placed as an entry action within the state.
- Actions that accompany all state transitions out of a state may be placed as exit actions within the state
- A behavior that occurs within the state is called an activity.
- An activity starts when the state is entered and either completes or is interrupted by an outgoing state transition.
- A behavior may be an action, or it may be an event sent to another object.
- This behavior is mapped to operations on the object.

State transition diagram notation:

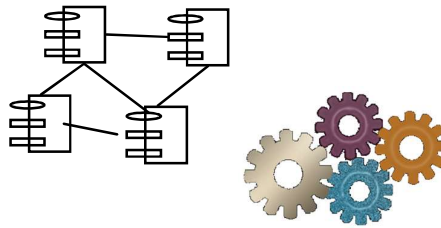


3. CASE Tools

Rational Rose (introduced in Lab 5).

12

Implementation Diagrams: Component & Deployment Diagrams



Implementation Diagrams: Component & Deployment Diagrams

Objectives

- Become familiar with the implementation diagrams: component and deployment diagrams.
- Practice using Rational Rose.

1. Outline

- Implementation diagrams: component and deployment diagrams.
- Examples.

2. Background

Implementation diagrams capture design information. The main implementation diagrams in UML are: component and deployment diagrams. In this Lab we will study these diagrams and their notation.

2.1 UML Implementation Diagrams

The main implementation diagrams we have in UML are: component diagrams and deployment diagrams. These diagrams are high level diagrams in comparison with old diagrams you have already learned.

2.2 UML Component Diagram

Component diagrams capture the physical structure of the implementation.

- Remember always that when you talk about components, you are talking about the physical models of code.
- You can name them and show dependency between different components using arrows.
- A component diagram shows relationships between component packages and components.
- Each component diagram provides a physical view of the current model.
- Component diagrams contain icons representing:
 - Component packages.
 - Components.
 - Main programs.
 - Packages.
 - Subprograms.
 - Tasks.
 - Dependencies.

2.3 Deployment Diagrams

A deployment diagram shows processors, devices and connections. Each model contains a single deployment diagram which shows the connections between its processors and devices, and the allocation of its processes to processors.

2.4.1 Deployment Diagrams: Processor

A processor is a hardware component capable of executing programs.

- A processor is given a name and you should specify the processes that will run on that processor.
- You can also specify the scheduling of these processes on that processor.
- Types of scheduling are:
 - Pre-emptive: a higher priority process may take the process from lower priority one.
 - Non-preemptive: a process will own the processor until it finishes
 - Cyclic: control passes from one process to another.
 - Executive: an algorithm controls the scheduling of the processes
 - Manual: scheduling by the user.

2.4.2 Deployment Diagrams: Device

A device is a hardware component with no computing power. Each device must have a name. Device names can be generic, such as “modem” or “terminal.”

2.4.3 Deployment diagrams: Connection

A connection represents some type of hardware coupling between two entities. An entity is either a processor or a device. The hardware coupling can be direct, such as an RS232 cable, or indirect, such as satellite-to-ground communication. Connections are usually bi-directional.

3. CASE Tools

Rational Rose (introduced in Lab 5).

13

Software Testing



Software Testing

Objectives

- Gain a deeper understanding of software testing and the software testing document.
- Become familiar with a software testing tool (JUnit).

1. Outline

- Overview of software testing.
- Unit testing.
- JUnit tutorial.
- Software test specification.

2. Background

Testing is the process of executing a program with the intent of finding errors. A good test case is one with a high probability of finding an as-yet undiscovered error. A successful test is one that discovers an as-yet-undiscovered error.

The causes of the software defects are: specification may be wrong; specification may be a physical impossibility; faulty program design; or the program may be incorrect.

2.1 Basic Definitions

- A **failure** is an unacceptable behavior exhibited by a system.
- A **defect** is a flaw in any aspect of the system that contributes, or may potentially contribute, to the occurrence of one or more failures. It might take several defects to cause a particular failure.
- An **error** is a slip-up or inappropriate decision by a software developer that leads to the introduction of a defect.

2.2 Good Test Attributes

A good test has a high probability of finding an error, not redundant, and should not be too simple or too complex.

2.3 Unit Testing

Unit testing is testing each unit separately. In unit testing interfaces tested for proper information flow and local data are examined to ensure that integrity is maintained. Boundary conditions and all error handling paths should also be tested.

3. CASE Tools

JUnit is an open-source project to provide a better solution for unit testing in Java. It can be integrated with many Java IDEs.

Central idea: create a separate Java testing class for each class you're creating, and then provide a means to easily integrate the testing class with the tested class for unit testing.

3.1 JUnit Terminology

- **A unit test:** is a test of a single class.
- **A test case:** tests the response of a single method to a particular set of inputs.
- **A test suite:** is a collection of test cases.
- **A test runner:** is software that runs tests and reports results.
- **A test fixture:** sets up the data (both objects and primitives) that are needed to run tests. For example if you are testing code that updates an employee record, you need an employee record to test it on.

3.2 How JUnit Works

- Define a subclass of **TestCase**.
- Override the **setUp()** & **tearDown()** methods.
- Define one or more public **testXXX()** methods
 - Exercise the object(s) under test.
 - Asserts the expected results.
- Define a static **suite()** factory method
 - Create a TestSuite containing all the tests.
- Optionally define **main()** to run the TestCase in batch mode.