LAB MANUAL

Course: CSE 304- Object-Oriented Software Engineering

Department of Computer Science

COMSATS UNIVERSITY ISLAMABAD

Lahore Campus

Contents

Lab 1 & 2 – An Introduction to Object oriented Analysis and Object-Oriented Design

Objectives

The lab objective is to explore some basic concepts related to OOA and OOD

Lab Tasks

You are suppose to provide precise answers of following questions after reading related topics from recommended books and consulting material available on internet. Font size should be Times New Roman, 12 pt, and justified text with 1.15 line spaces.

1) What is Object-oriented design?
2) Why OOD and OOA is necessary?

Outcomes

In this task, students will learn about: Basic concepts of Object oriented Design & analysis for modelling the system.
Since, we are going to do a series of tutorials on UML diagrams and any tool for modelling those such as Visual Paradigm and MS Visio. In this manner, we need to have a familiarity with design approaches.

Recommended Readings

Text Book	1. Object Oriented Software Engineering, Practical Software development using UML and Java by Lethbridge and Laganiere 2. Object Oriented Software Engineering-A use-case driven approach by Jacobson, Christerson, Jonsson, Overgaard
Reference Books	1. The Unified Modeling Language User Guide by Booch, Rumbaugh, and Jacobson 2. Applying UML and Patterns 2nd edition by Craig Larman 3. Software Engineering 9th edition by Sommerville 4. Software Engineering A Practitioner's Approach by Roger S. Pressman, Edition: 7th

A Road map for OOA and OOD

In this subject, you have been hit with a bewildering array of new terminology. Words like "class", "instance", "method", "super class", "design", are all being thrown around with reckless abandon. It's easy to lose your place. The purpose of this road map is to help keep you grounded during the process of designing your object-oriented software systems.
Object-oriented program has become the dominant programming style in the software industry over the last 10 years or so. The reason for this has to do with the growing size and

scale of software projects. It becomes extremely difficult to understand a procedural program once it gets above a certain size. Object-oriented programs scale up better, meaning that they are easier to write, understand and maintain than procedural programs of the same size. There are basically three reasons for this:

1. Object-oriented programs tend to be written in terms of real-world objects, not internal data structures. This makes them somewhat easier to understand by maintainers and the people who have to read your code -- but it may make it harder for you as the initial designer. Identifying objects in a problem is a challenge

2. Object-oriented programs encourage encapsulation -- details of an objects implementation are hidden from other objects. This keeps a change in one part of the program from affecting other parts, making the program easier to debug and maintain. This does take some getting used to -- past students have called it "hypertext programming" because your program is spread out among a variety of objects.

Object-oriented programs encourage modularity. This means that pieces of the program do not depend on other pieces of the program. Those pieces can be reused in future projects, making the new projects easier to build.

Because of the overhead costs referred to above, OOP is probably not the best choice for very small programs. If all you want to do is sum up file sizes in a directory, you probably don't want to have to identify objects, create reusable structures and all that. Since, the programs that you write for this class are necessarily small (so you can do them in the quarter), so it may not always be obvious where the big benefit from using objects is.

A good object-oriented program, then, is one that takes advantage of the strengths of object oriented programs as listed above. Some top-level guidelines for doing this include:

- Objects in your system should usually represent real-world things. Talk about cash registers and vending machines and stop lights, not linked-lists, hash tables or binary trees.
- Objects in your system should have limited knowledge. They should only know about the other parts of the system that they absolutely have to in order to get their work done. This implies that you should always have more than one object in the system. It's not always clear, however, how to implement goals like this in practice. That's where the design comes into play.

Why Design?

In order to get the most out of using objects, OO programs require some kind of design work before programming starts. You, as the programmer, are much more in the position of an architect. Why do architects spend so much time on blueprints and models? One reason is that the blueprint allows the architect to solve the problem in a relatively low-cost environment before actually using brick and concrete. The final building is better: safer, more energy-efficient, and so on, because the worst mistakes are corrected on paper, cheaply and quickly, before a building falls on somebody's head. As a nice side effect, the precision

of the blueprint makes it a useful tool for communicating the design to the builders, saving time in the building process.

OO probably requires more up-front design work than procedural programs because the job of identifying the objects and figuring out how to divide the tasks among them is not really a programming job. You really do need to have that started before you sit down to hack -- otherwise, how will you know where to start?

Similarly, time spent getting the design right before you start programming will almost always save you time in the end. It's much, much easier to make major changes on a design, which is after all just squiggly lines on paper, and then it is to make changes in hundreds or thousands of lines of code.

Several Activities to a Brilliant Design

The design process is typically split into two distinct phases: Object-Oriented Analysis (OOA) and Object Oriented Design (OOD). Yes, I know that it is confusing to have one of the sub-steps of the design process also called "design". However, the term is so entrenched that it's hopeless to start creating new jargon.

We'll call these things "activities" rather than "steps" to emphasize that they don't have to be done in any particular order -- you can switch to another activity whenever it makes sense to do so. When you are actually doing this, you'll find that you want to go back and forth between OOA and OOD repeatedly. Maybe you'll start at one part of your program, and do OOA and OOD on that part before doing OOA and OOD on another part. Or maybe you'll do a preliminary OOA, and then decide while working on the OOD that you need more classes, so you jump back to OOA. That's great. Moving back and forth between OOA and OOD is the best way to create a good design -- if you only go through each step once; you'll be stuck with your first mistakes all the way through. I do think that it's important, though, to always be clear what activity you are currently doing -- keeping a sharp distinction between activities will make it easier for you to make design decisions without getting tied up in knots.

In the OOA phase, the overall questions is "What?". As in, "What will my program need to do?", "What will the classes in my program be?", and "What will each class be responsible for?" You do not worry about implementation details in the OOA phase -- there will be plenty of time to worry about them later, and at this point they only get in the way. The focus here is on the real world --.what are the objects, tasks and responsibilities of the real system? 10 In the OOD phase, the overall question is "How?" As in, "How will this class handle its responsibilities?", "How can I ensure that this class knows all the information it needs?", "How will classes in my design communicate?" At this point, you are worried about some

implementation details, but not all -- what the attributes and methods of a class will be, but not down to the level of whether things are integers or reals or ordered collections or dictionaries or whatnot -- those are programming decisions

Lab 2 and 3 - 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.

Outline

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

Outcome

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.

After this lab you will be able to define a project find its basic requirements , as well as you will be able to plan the project activities using CASE Tools.

Introduction to lab plan

- 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.

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.
  - MS Visio/ Visual Paradigm: for UML diagrams (Object Oriented Analysis and Design)
  - NetBeans : For Implementation of Object Oriented Project

Lab Task

1. Form groups of 4 students (with one of them as a leader).
2. Brainstorm and list suitable project titles 3. Present these to the class
3. Choose one of the projects from your list
4. Try to write (a hypothetical) project definition for it.
5. Present it to the instructor/ class
6. Deliverables
   - Form teams of 4 students for the term mini-project.

- Suggest/ research a project and write a project definition/ problem statement.

Lab 3– Software Processes

Objectives

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

Lab Task

You are suppose to provide precisely written notes after reading related topics from recommended books and consulting material available on internet. Font size should be Times New Roman, 12 pt, and justified text with 1.15 line spaces.

1) Review of the basic software process models and software processes.
2) Which software process is suitable for your selected project and why?

Recommended Readings

1. The Unified Modeling Language User Guide by Booch, Rumbaugh, and Jacobson
2. Applying UML and Patterns 2nd edition by Craig Larman
3. Software Engineering 9th edition by Sommerville
4. Software Engineering A Practitioner's Approach by Roger S. Pressman, Edition: 7th

Outcome

You will be able to choose a Process Model for your project by understanding of software process models and software processes.

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.

Lab 4 & 5- 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.

Outline

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

Lab Task :

You are supposed to provide:

1) Gantt chart for your assigned Project
2) Work Breakdown Structure for the activities of your Project

Outcome

In this lab, you will understand the process of planning and controlling the development of a system within a specified timeframe at a minimum cost with the right functionality.

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.

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.

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.

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.

CASE Tools

- Microsoft Project is the clear market leader among desktop project management applications.

Other tools which can be used:

- 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.

Lab 6 and 7 - Software Requirement Specification

Objectives

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

Lab Task:

Choose a hypothetical system of significant complexity and write an SRS for the same.

1 Outcome :

You will gain the understanding of Software Requirement Specification phase and the Software Requirement Specification (SRS). You will be able to write requirements and specifications.

2. Background

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 now 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)

Requirements Statement for Example ATM System

The software to be designed will control a simulated automated teller machine (ATM) having a magnetic stripe reader for reading an ATM card, a customer console (keyboard and display) for interaction with the customer, a slot for depositing envelopes, a dispenser for cash, a printer for printing customer receipts, and a key-operated switch to allow an operator to start or stop the machine. The ATM will communicate with the bank's computer over an appropriate communication link. (The software on the latter is not part of the requirements for this problem.)

The ATM will service one customer at a time. A customer will be required to insert an ATM card and enter a personal identification number (PIN) - both of which will be sent to the bank for validation as part of each transaction. The customer will then be able to perform one or more transactions. The card will be retained in the machine until the customer indicates that he/she desires no further transactions, at which point it will be returned - except as noted below.

The ATM must be able to provide the following services to the customer:

1. A customer must be able to make a cash withdrawal from any suitable account linked to the card. Approval must be obtained from the bank before cash is dispensed.
2.A customer must be able to make a deposit to any account linked to the card, consisting of cash and/or checks in an envelope. The customer will enter the amount of the deposit into the ATM, subject to manual verification when the envelope is removed from the machine by an operator. Approval must be obtained from the bank before physically accepting the envelope.
3. A customer must be able to make a transfer of money between any two accounts linked to the card.
4. A customer must be able to make a balance inquiry of any account linked to the card.

A customer must be able to abort a transaction in progress by pressing the Cancel key instead of responding to a request from the machine.

The ATM will communicate each transaction to the bank and obtain verification that it was allowed by the bank. Ordinarily, a transaction will be considered complete by the bank once it has been approved. In the case of a deposit, a second message will be sent to the bank indicating that the customer has deposited the envelope. (If the customer fails to deposit the envelope within the timeout period, or presses cancel instead, no second message will be sent to the bank and the deposit will not be credited to the customer.)

If the bank determines that the customer's PIN is invalid, the customer will be required to reenter the PIN before a transaction can proceed. If the customer is unable to successfully enter the PIN after three tries, the card will be permanently retained by the machine, and the customer will have to contact the bank to get it back.

If a transaction fails for any reason other than an invalid PIN, the ATM will display an explanation of the problem, and will then ask the customer whether he/she wants to do another transaction. The ATM will provide the customer with a printed receipt for each successful transaction, showing the date, time, machine location, type of transaction, account(s), amount, and ending and available balance(s) of the affected account ("to" account for transfers).

The ATM will have a key-operated switch that will allow an operator to start and stop the servicing of customers. After turning the switch to the "on" position, the operator will be required to verify and enter the total cash on hand. The machine can only be turned off when it is not servicing a customer. When the switch is moved to the "off" position, the machine will shut down, so that the operator may remove deposit envelopes and reload the machine with cash, blank receipts, etc. The ATM will also maintain an internal log of transactions to facilitate resolving ambiguities arising from a hardware failure in the middle of a transaction. Entries will be made in the log when the ATM is started up and shut down, for each message sent to the Bank (along with the response back, if one is expected), for the dispensing of cash, and for the receiving of an envelope. Log entries may contain card numbers and dollar amounts, but for security will never contain a PIN.

Lab 8 & 9: 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 Visual Paradigm.

Lab Task

- You should submit the solutions for the exercise given.
- You should use these techniques to draw use case diagrams for your term project using Visual Paradigm / MS Visio. The description of each Use Case diagram in MS Word using the proper format.

Outcomes

Now you will learn how to apply the learned methods to draw use case diagrams from the problem statement.

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

An actor is someone or something that must interact with the system under development

- 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.

2.5.1 Categories of actor

Principle: Who uses the main system functions.
Secondary: Who takes care of administration & maintenance.
External h/w: The h/w devices which are part of application domain and must be used.
Other system: The other system with which the system must interact.

2.6 Use Case

- A use case is a pattern of behavior, the system exhibits Each use case is a sequence of related transactions performed by an actor and the system in dialogue.
- USE CASE is dialogue between an actor and the system. Examples:
- 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.

2.6.1 Generic format for documenting the use case

- Pre condition:If any
  - Use case: Name of the case.

- Actors: List of actors(external agents), indicating who initiates the use case.
- Purpose: Intention of the use case.
- Overview: Description.
- Type:primary / secondary.
- Post condition:If any

2.7 Typical Course of Events

ACTOR ACTION: Numbered actions of the actor.
SYSTEM RESPONSE: Numbered description of system responses.

2.7.1 What is System Boundary?

- It is shown as a rectangle.
- It helps to identify what is an external verse internal, and what the responsibilities of the system are.
- The external environment is represented only by actors.

2.7.2 What is Relationship?

- Relationship between use case and actor: Communicates
- Relationship between two use cases: Extends,
- Uses Notation used to show the relationships: << >>
- Relationship between use case and actor is often referred as "communicates" Relationship between two use cases is refereed as either uses or extends

2.8 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 Visual Paradigm 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.

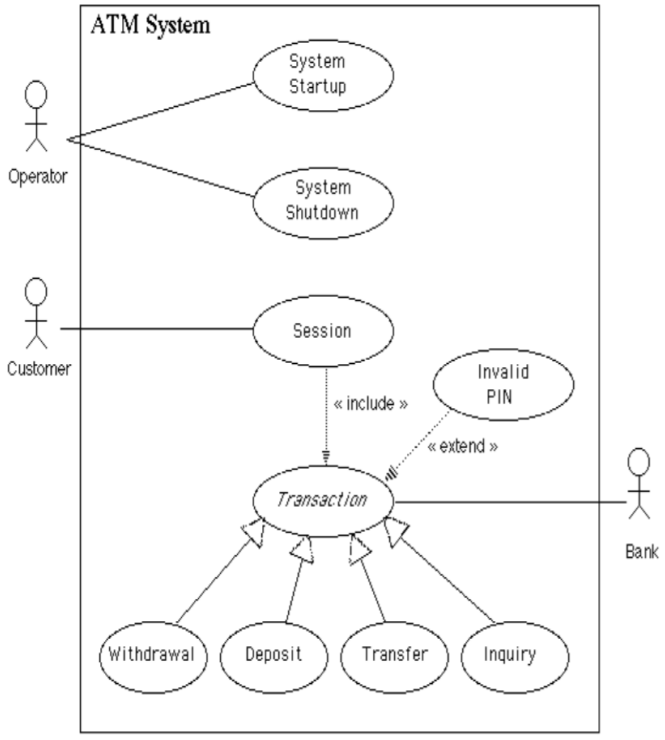
4. Use Cases for ATM System System

4.1 Startup Use Case:

The system is started up when the operator turns the operator switch to the "on" position. The operator will be asked to enter the amount of money currently in the cash dispenser, and a connection to the bank will be established. Then the servicing of customers can begin.

4.2 System Shutdown Use Case

The system is shut down when the operator makes sure that no customer is using the machine, and then turns the operator switch to the "off" position. The connection to the bank will be shut down. Then the operator is free to remove deposited envelopes, replenish cash and paper, etc.



4.3 Session Use Case

A session is started when a customer inserts an ATM card into the card reader slot of the machine. The ATM pulls the card into the machine and reads it. (If the reader cannot read the card due to improper insertion or a damaged stripe, the card is ejected, an error screen is displayed, and the session is aborted.) The customer is asked to enter his/her PIN, and is

then allowed to perform one or more transactions, choosing from a menu of possible types of transaction in each case. After each transaction, the customer is asked whether he/she would like to perform another. When the customer is through performing transactions, the card is ejected from the machine and the session ends. If a transaction is aborted due to too many invalid PIN entries, the session is also aborted, with the card being retained in the machine.

The customer may abort the session by pressing the Cancel key when entering a PIN or choosing a transaction type.

4.4 Transaction Use Case

A transaction use case is started within a session when the customer chooses a transaction type from a menu of options. The customer will be asked to furnish appropriate details (e.g. account(s) involved, amount). The transaction will then be sent to the bank, along with information from the customer's card and the PIN the customer entered. If the bank approves the transaction, any steps needed to complete the transaction (e.g. dispensing cash or accepting an envelope) will be performed, and then a receipt will be printed. Then the customer will be asked whether he/she wishes to do another transaction. If the bank reports that the customer's PIN is invalid, the Invalid PIN extension will be performed and then an attempt will be made to continue the transaction. If the customer's card is retained due to too many invalid PINs, the transaction will be aborted, and the customer will not be offered the option of doing another. If a transaction is cancelled by the customer, or fails for any reason other than repeated entries of an invalid PIN, a screen will be displayed informing the customer of the reason for the failure of the transaction, and then the customer will be offered the opportunity to do another. The customer may cancel a transaction by pressing the Cancel key as described for each individual type of transaction below. All messages to the bank and responses back are recorded in the ATM's log.

4.5 Withdrawal Transaction Use Case

A withdrawal transaction asks the customer to choose a type of account to withdraw from (e.g. checking) from a menu of possible accounts, and to choose a dollar amount from a menu of possible amounts. The system verifies that it has sufficient money on hand to satisfy the request before sending the transaction to the bank. (If not, the customer is informed and asked to enter a different amount.) If the transaction is approved by the bank, the appropriate amount of cash is dispensed by the machine before it issues a receipt. (The dispensing of cash is also recorded in the ATM's log.) A withdrawal transaction can be cancelled by the customer pressing the Cancel key any time prior to choosing the dollar amount.

4.6 Deposit Transaction Use Case

A deposit transaction asks the customer to choose a type of account to deposit to (e.g. checking) from a menu of possible accounts, and to type in a dollar amount on the keyboard.

The transaction is initially sent to the bank to verify that the ATM can accept a deposit from this customer to this account. If the transaction is approved, the machine accepts an envelope from the customer containing cash and/or checks before it issues a receipt. Once

the envelope has been received, a second message is sent to the bank, to confirm that the bank can credit the customer's account - contingent on manual verification of the deposit envelope contents by an operator later. (The receipt of an envelope is also recorded in the ATM's log.)
A deposit transaction can be cancelled by the customer pressing the Cancel key any time prior to inserting the envelope containing the deposit. The transaction is automatically cancelled if the customer fails to insert the envelope containing the deposit within a reasonable period of time after being asked to do so.

4.7 Transfer Transaction Use Case:

A transfer transaction asks the customer to choose a type of account to transfer from (e.g. checking) from a menu of possible accounts, to choose a different account to transfer to, and to type in a dollar amount on the keyboard. No further action is required once the transaction is approved by the bank before printing the receipt.

A transfer transaction can be cancelled by the customer pressing the Cancel key any time prior to entering a dollar amount.

4.8 Inquiry Transaction Use Case

An inquiry transaction asks the customer to choose a type of account to inquire about from a menu of possible accounts. No further action is required once the transaction is approved by the bank before printing the receipt.

An inquiry transaction can be cancelled by the customer pressing the Cancel key any time prior to choosing the account to inquire about.

4.9 Invalid PIN Extension

An invalid PIN extension is started from within a transaction when the bank reports that the customer's transaction is disapproved due to an invalid PIN. The customer is required to reenter the PIN and the original request is sent to the bank again. If the bank now approves the transaction, or disapproves it for some other reason, the original use case is continued; otherwise the process of re-entering the PIN is repeated. Once the PIN is successfully reentered, it is used for both the current transaction and all subsequent transactions in the session. If the customer fails three times to enter the correct PIN, the card is permanently retained, a screen is displayed informing the customer of this and suggesting he/she contact the bank, and the entire customer session is aborted.If the customer presses cancel instead of re-entering a PIN, the original transaction is cancelled.

5. Exercises

Read carefully the following problem statement

We are after a system that controls a recycling machine for returnable bottles and cans. The machine will allow a customer to return bottles or cans on the same occasion. When the customer returns an item, the system will check what type has been returned. The system will register how many items each customer returns and, when the customer asks for a receipt, the

system will print out what he deposited, the value of the returned items and the total return sum that will be paid to the customer. The system is also be used by an operator. The operator wants to know how many items of each type have been returned during the day. At the end of the day, the operator asks for a printout of the total number of items that have been deposited in the machine on that particular day. The operator should also be able to change information in the system, such as the deposit values of the items. If something is amiss, for example if a can gets stuck or if the receipt roll is finished, the operator will be called by a special alarm signal.

After reading the above problem statement, find:

1. Actors
2. Use cases with each actor
3. Find extend or include use cases (if applicable)
4. Finally: draw the main use case diagram

Lab 10 – 12 System Modeling

Objective:

Deeper understanding of System modeling:

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

Lab Tasks:

You are required to provide the following:

- Data model: entity-relationship diagram (ERD) of your assigned project.
- Functional model: data flow diagram (DFD) of your assigned project.

Outcomes

After this lab you will be able to produce data model and the functional model of any software system

System Modeling

2. 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:

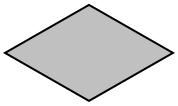
- An entity-relationship diagram (data model).
- A data flow diagram (functional model).
- A state transition diagram (behavioral model).
  NOTE: state transition diagram will be covered in lab 11.

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:

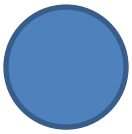
- Define “objects” by underlining all nouns in the written statement of scope: producers/consumers of data, places where data are stored, and “composite” data items.
- Define “operations” by double underlining all active verbs: processes relevant to the application and data transformations.
- Consider 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.

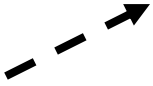
Data flow diagram notation:

External entity



Process

Data flow



Control flow

Data store

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 Visio or Visual Paradigm to create your ERD and DFD

Lab 13 & 14 - 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 visual paradigm

Outcome :

In this lab you will study how to document use cases in detail. Then you will know about scenarios (flow of events) and its importance with deeper understanding of UML activity diagrams.This will help you to Practice flow of events and activity diagrams using visual paradigm

Lab Task

You are required to produce the following

- Use cases of your system with its descriptions
- Activity diagrams of your system

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.

Use case ID: 001	**Use case Name**: *<<type name of use case >>*
Priority:	
Actors:	
Use Case Summary:	
Pre-condition:	
Normal Flow of Events	**Alternative Path**
1.	
2.	
Post Conditions	
Use Case Cross References	
Includes	
Extends	

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

MS Visio / Visual Paradigm

Lab 15 - 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.

Outcome :

In this lab you will study the object-oriented analysis phase, You will be able to understand the methods of class elicitation and find the classes in an object-oriented system.

Lab Task

You are required to find the appropriate classes for your system, with proper naming according to the Naming Conventions

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 Major Types of classes:

Concrete classes

- A concrete class is a class that is instantiable; that is it can have different instances.
- Only concrete classes may be leaf classes in the inheritance tree.

Abstract classes

- An abstract class is a class that has no direct instance but whose **descendant's** classes have direct instances.
- An abstract class can define the protocol for an operation without supplying a corresponding method we call this as an *abstract operation.*
- An abstract operation defines the form of operation, for which each concrete subclass should provide its own implementation.

2.4 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.

2.5 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.

2.6 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

Visual Paradigm

Lab 16 & 17 -Interaction Diagrams: Sequence & Collaboration Diagrams

Objectives

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

Outcome :

In this lab you will study how to document interaction diagrams in detail. Then you will know about scenarios (flow of events) and its importance with deeper understanding of UML sequence diagrams and collaboration diagrams. This will help you to Practice flow of interaction in these diagrams using visual paradigm

Lab Task

You are required to produce the following

- Sequence diagrams of your system with its descriptions
- Collaboration/Communication diagrams of your system by transforming the sequence diagram into collaboration diagrams

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 destructed. 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).

Lab 18- 19 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.

Outcome:

In this lab you will study how to document software design diagrams in detail. Then you will know about relationship of classes and its importance with deeper understanding of UML class diagrams.

Lab Task

You are required to produce the following

- Class diagram of your system with its descriptions
- Documenting the Class Diagram in SDD

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

Visual Paradigm.

Lab 20- State Chart Diagrams

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.

Outcome :

In this lab you will study how to document state chart diagrams in detail. Then you will know about states with transitions and its importance with deeper understanding of UML state machine diagrams. This will help you to Practice transitions of state in the diagram using visual paradigm

Lab Task

You are required to produce the state chart diagrams with proper transition of states of your assigned project

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

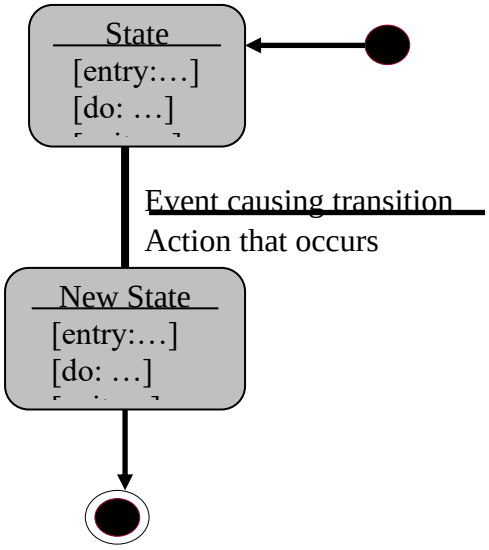
- States are represented by rectangles with rounded corners with an attribute name with a values associated with it.
- The name of the state is placed within the box.

- 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

Visual Paradigm and MS Visio

Lab 21 &22 Implementation Diagrams: Component & Deployment Diagrams

Objectives

- Become familiar with the implementation diagrams: component and deployment diagrams.
- Practice using Visual Paradigm.

Outcome :

In this lab you will study how to document implementation diagrams in detail. Then you will know about physical structure of software systems and its importance with deeper understanding of UML component and deployment diagrams. This will help you to practice physical implementation of software system through diagrams using visual paradigm

Lab Task

You are required to produce the following

- Component Diagram of your system with its descriptions
- Deployment Diagram of your system by transforming the sequence diagram into collaboration diagrams

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 buy 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

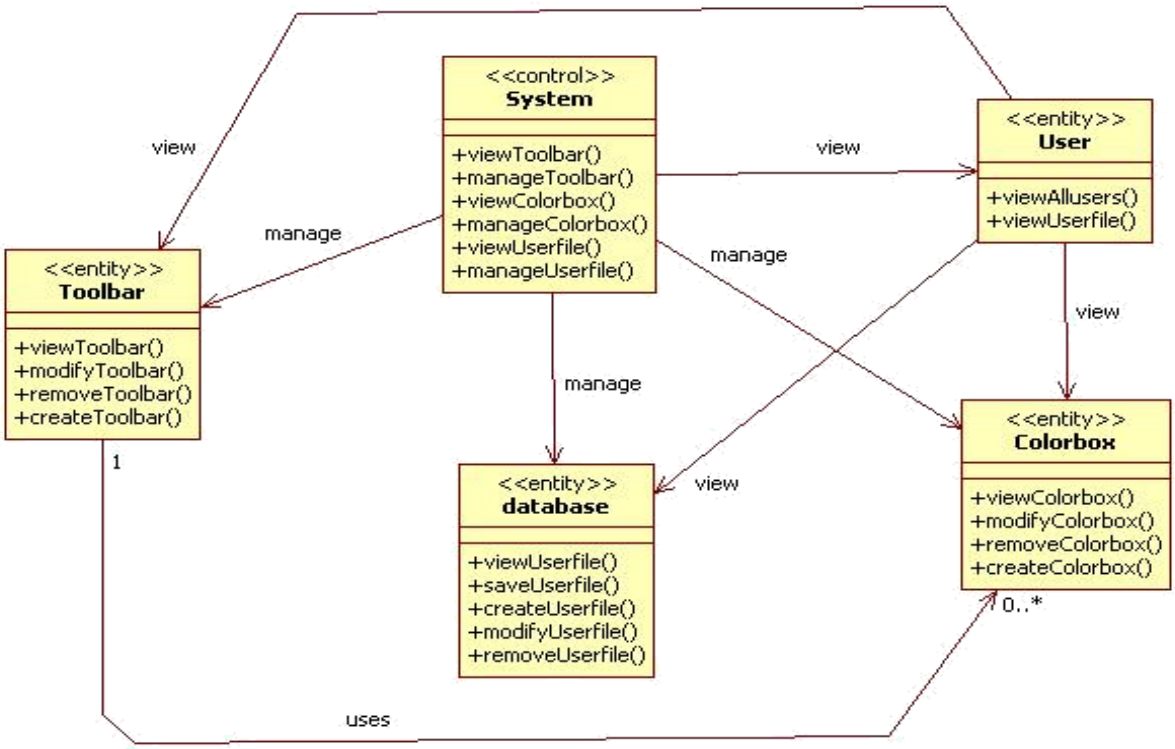
Visual Paradigm and MS Visio

Example and Case studies

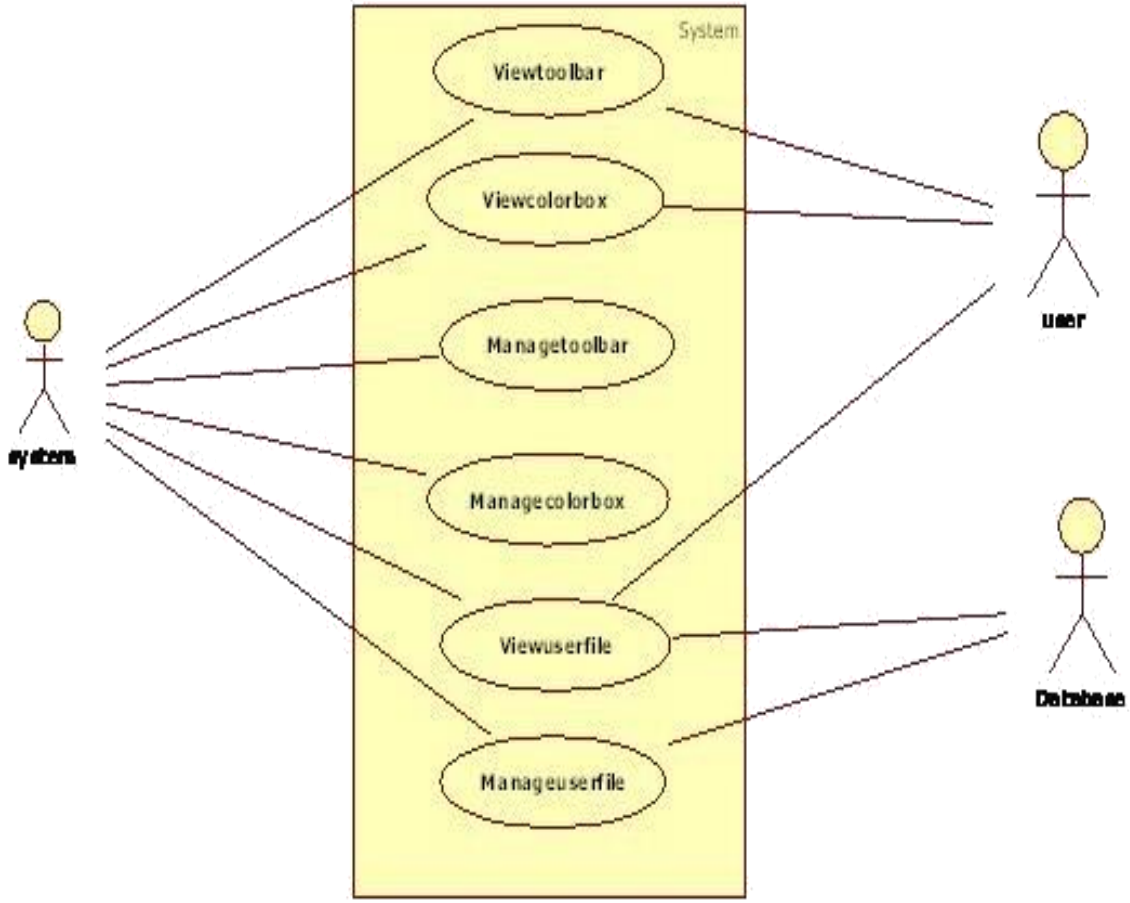
Example - Graphic Editor

Requirements Statement :
It is required to develop a graphics editor software package to create line drawings involving several types of graphics entities. It should support following functionalities:
• It contains the toolbox which contains tools like: Line, Circle, Rectangle, Arc, Polygon, Parallelogram, Text, Draw, Eraser
• Color box or palette
• Standard toolbar with options for New, Open, Save, toolbox and Text Toolbox.
• One integrated view to users for toolbar, color box, menu, and graphic screen. • Easy handling of tools for users.
• Ability to group several drawing into one i.e. complex drawing.
• Provision of zoom in and zoom out.
• Different shadings of line tool are provided.

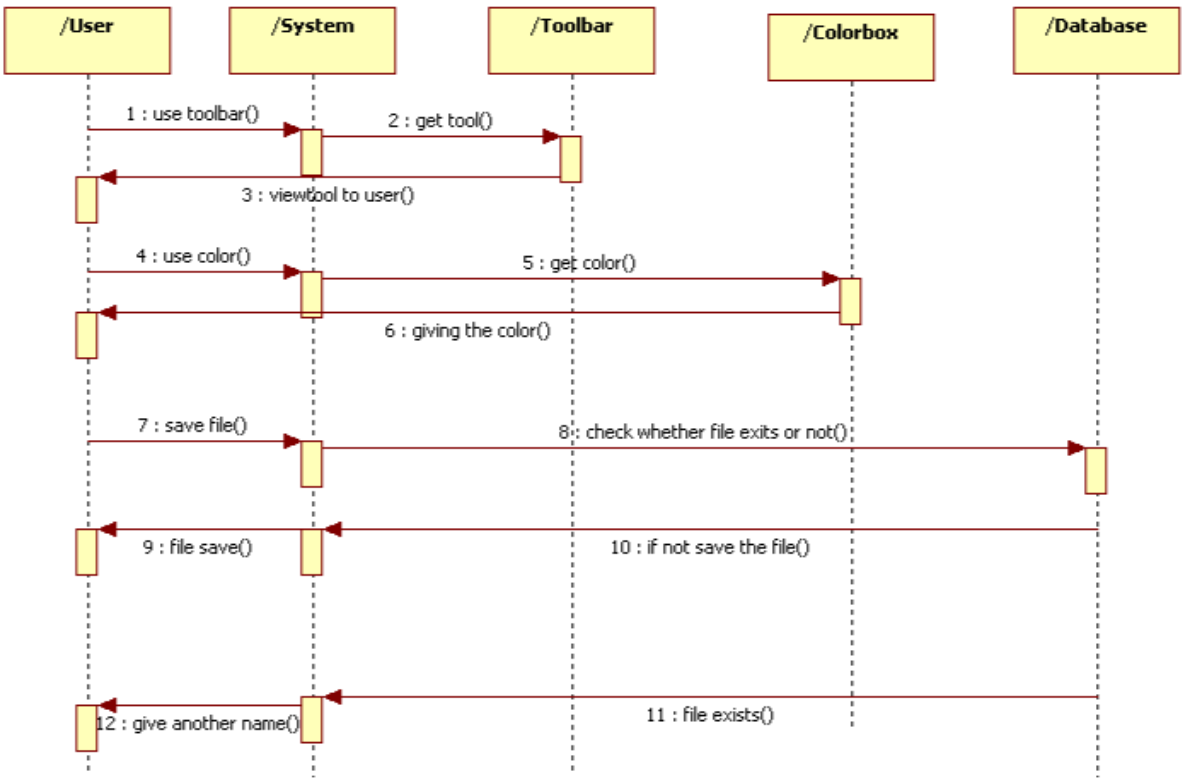
Class Diagram for Graphics Editor



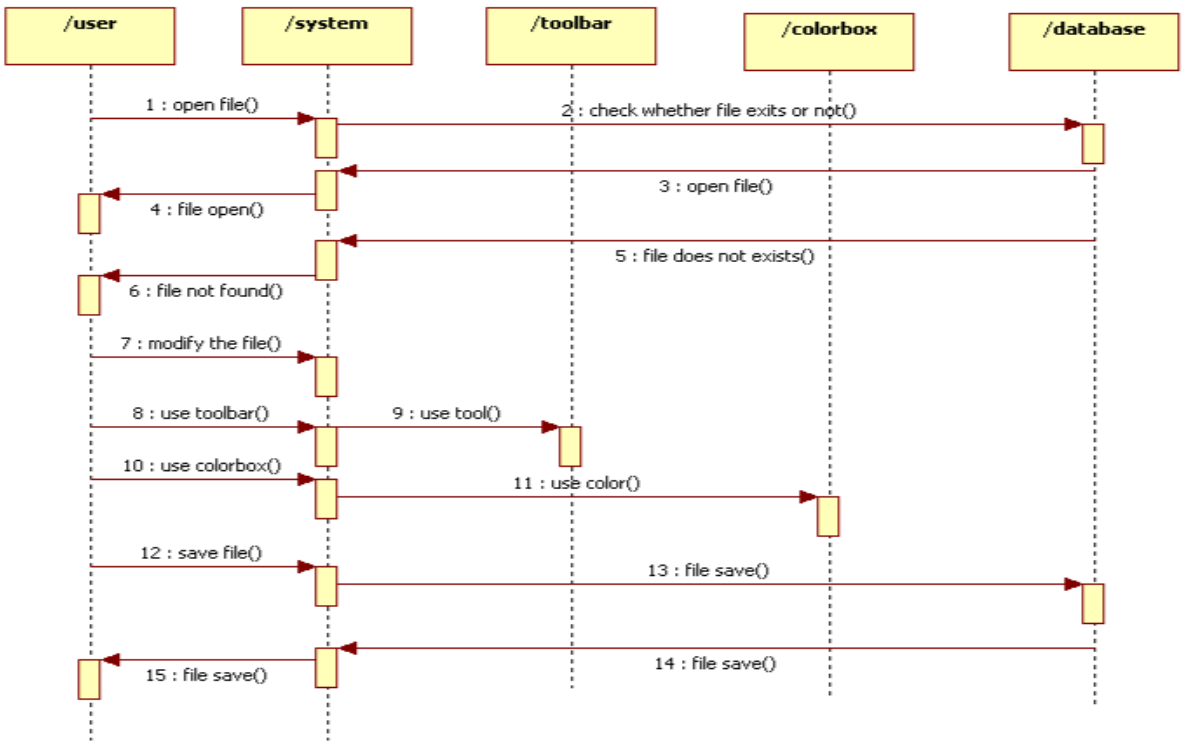
Use Case Diagram for Graphics Editor



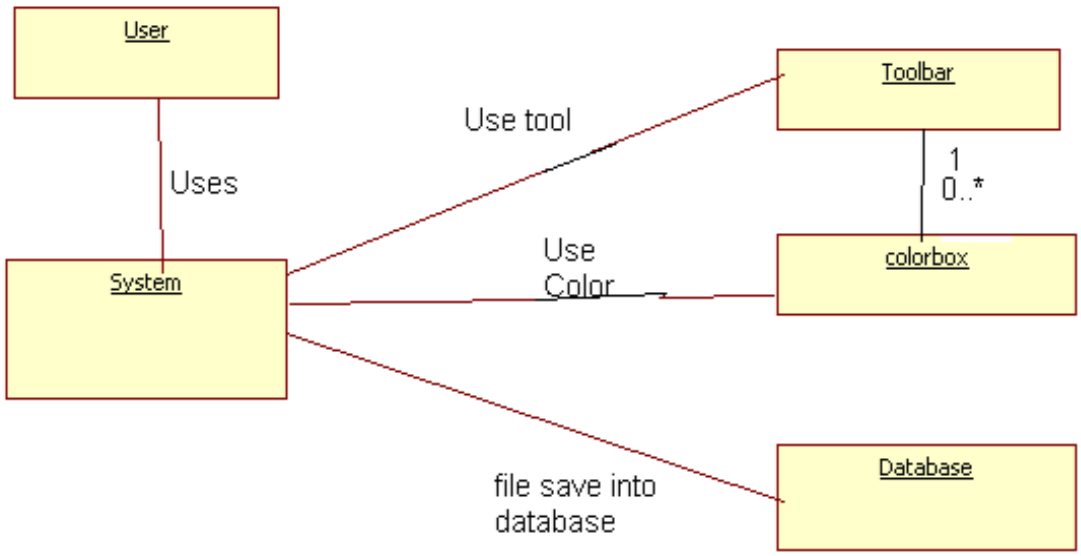
Sequence Diagram for Creating a file



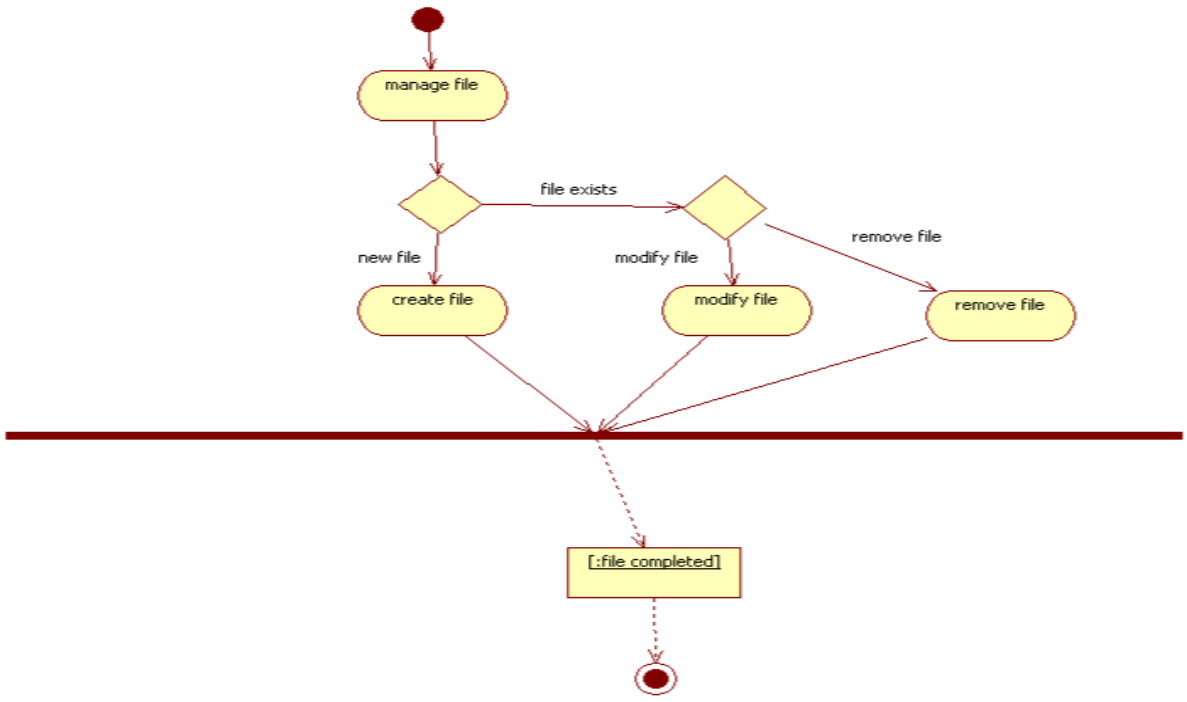
Sequence Diagram for Modifying a file



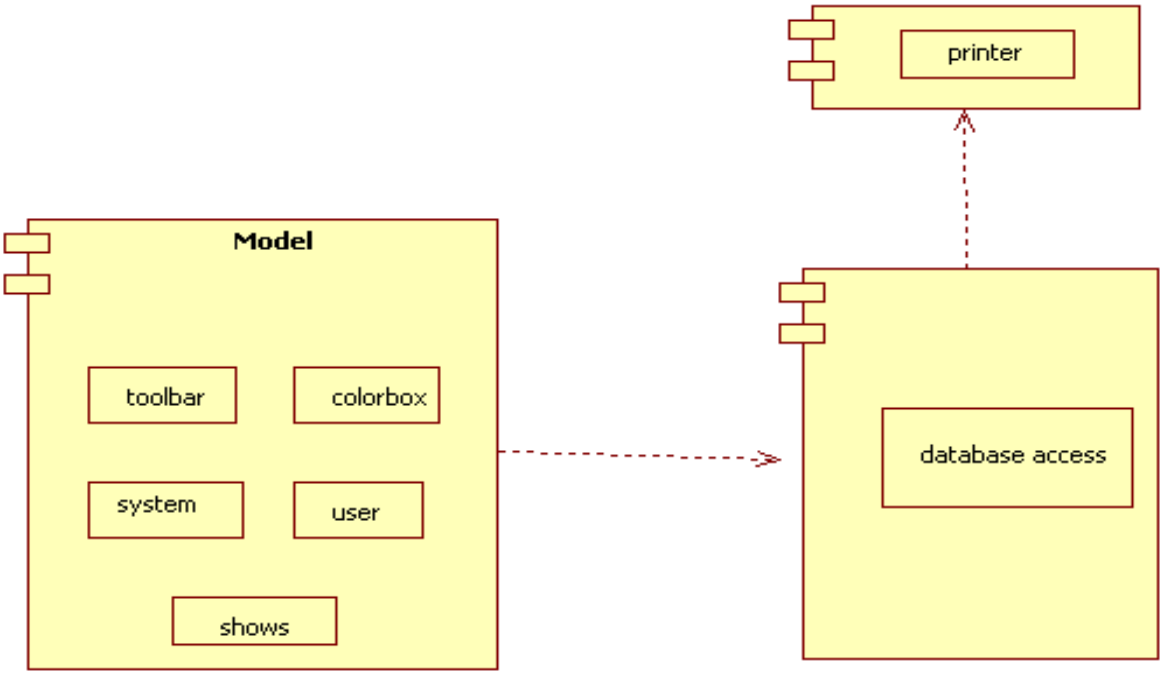
Collaboration diagram for graphics editor



Collaboration diagram for graphics editor



Component diagram for graphics editor



Case Study 1- A Point-of-Sale (POS) system

Project Analysis:

A POS system is a computerized application used to record sales and handle payments; it is typically used in retail store. It includes hardware components such as a computer and bar code scanner , and software to run to run the system. It interfaces to various applications, such as third party tax calculator and inventory control. These systems must be relatively fault tolerant; that is ,even if remote services are temporarily unavailable they must still be of capturing sales and handling at least cash payments . A POS system must support multiple and varied client-side terminals and interfaces such as browser, PDAs, touch-screens.

Creating Use case Diagrams:

Identifying actors:

1. Supermarket is a hardware thing where goods are brought
2. Customer is a person who buys items in a super market.
3. Staff is a person who interacts with super market.
4. Bank is a actor who gives loan to super market.
5. Items are part of super marke**t**

Identifying use cases:

1. **Sell items**
2. **Pay salaries**
3. **Maintenance**
4. **Issue bills**
5. **Take salary**
6. **Select items**
7. **Buy items**
8. **Give loans**
9. **Collect loans**
10. **Display**
11. **Sales**
12. **Profit**
13. **Loss**
14. **Pay money**

Identifying relationships:

Association:

1. Supermarket Sell items
2. Supermarket Pay salaries
3. Supermarket Maintain
4. Supermarket Issue bills
5. Staff works in Supermarket
6. Customer selects items
7. Star takes salaries
8. Customer selects items
9. Bank gives loans
10. Bank collects loans
11. Items are display in supermarket

Creating Class Diagrams:

Identifying classes:

1. Supermarket
2. Customer
3. Cash payment
4. Staff
5. Items
6. Bank
7. Sales
8. Cashier
9. Profit
10. Loss
11. Billing system
12. hardware
13. software
14. scanner
15. Barcode reader
16. Determine tax
17. Stationary

Identifying relationship between classes:

Aggregation:

1. Staff is a part of supermarket
2. Items are a part of supermarket
3. Billing system is a part of supermarket

Association:

1. Items are sold through sales
2. Bank provide loans to supermarket
3. Customer pays the cash payment
4. Cashier determine the tax
5. Sale are done by supermarket
6. Billing system determine the tax

Generalization:

1. Stationary is a kind of item
2. Vegetables is a kind of item
3. Chocolates is a kind of item
4. Fruits is a kind of item
5. Glossaries is a kind of item
6. Hardware is a kind of billing system
7. Software is a kind of billing system
8. Scanner is a kind of hardware
9. Barcode reader is a kind of

   hardware

Dependency:

1. Profit is dependent on sales
2. Loss is dependent on sales

Identifying attributes:

1. Supermarket: name, location, branches, code
2. Cash payment: cheque, credit card, debit card, cash
3. Billing system: sensors, serial no
4. Barcode scanner
5. Loss
6. Customer: name, credit card
7. Bank; name, location
8. Sales: no of items sold, cost of item sold
9. Staff: name, id, designation
10. Items: name, code, cost, quantity
11. Cashier: name, id
12. Hardware: name, serial no
13. Software
14. Scanner
15. Determine tax: amount, percentage
16. Glossaries
17. Stationary
18. Profit

Identifying operations

1.Supermaket:sell items
2.Customer:buy items
3.Cash patment:display
4.Staff:Worktimings:salary
5.Items:display
6.Sales:Display
7.Cashier:Collect cash
8.profit:display
9.loss:display
10.billing system:issuable,display
11.hardware
12.software
13.scanner:scam code
14.barcode reader:barcodescan
15.determine tax;display
16.staionary:display

Creating collaboration diagrams:

Identifying objects:

1.s.supermarket
2.cp.cash payment
3.st.staff
4.i.items5.b.bank
6.sl.sales
7.cs.cashier
8.bs.billing systems
9.br.barcode reader

Identifying messages:

1.Supermarket exhibit items to customer
2.Supermarket gives item details to customers
3.customer enters to super market
4.pos check password of customer
5.customer selects item
6.staff guide customer
7.Barcode scanner identify product of super market
8.Super market produces the billing from
9.Cashier verifies the account of customer
10.customer pay bill system
11.Cashier give product to customer

Creating activity diagrams:

Identifying activities:

1. Take loans from bank
2. Customer enters supermarket
3. Staff shows the items
4. Customer selects the items
5. Select
6. Decline
7. Selected items submit at cashier
8. Enter details in billing system
9. Issue the bill
10. Pay bill
11. Cashier collects the bill

Case Study 2 - ONLINE BOOKSHOP

Project Analysis:

Online book shop is an application used to maintain information about the items that are generally available in that book shop. Any customer can access the information about the item and he can also order item through online by creating his login in that site. Customer can change the already entered details. The items information in that site is entered by the bookshop staff and it can be change if there are any modifications.

Creating use case diagrams

The following needs must be addressed by the system for actors are:

1 Customers to order books, access the details of the books and to receive books.
2 Bookshop staff provides the details of books and to send the books.

Actors identified are:

1 Costumer
2 Bookshop staff

The following needs must be addressed by the system for use cases are:

1 Customer registers the details of the books that are required and his personal details.
2 Customer browses the site of bookshop and selects the required books and orders.
3 The books that ordered by the customers are delivered to the appropriate customers.

4 If customer wants to change his details that are registered then it is possible through customer update details.
5 If any new items enter the book shop or any items that removed are entered by the bookshop details.

The uses cases identified are:

1 Customer registers details

2 Customer browses and orders items

3 Bookshop staff ships to customer
4 Customer updates details
5 Bookshop staff updates items

Creating class diagrams

The following needs must be addressed by the system for classes are:

1 Home page of the bookshop is placed in the class book.
2 Details of customer who wants to order books.
3 Information about the items those are present the bookshop.

The main classes identified are:

1 Book
2 Item
3 Customer

From these classes sub classes can be identified. They are:

1 Bookshop staff

2 Music cd
3 Software
4 Book
5 Address

For each class attributes, relationships must be made.

For example the book attributes are book name, book id, author name etc.

Identifying relationships between classes:

Inheritance:

1 Book is an item
2 Music cd is an item
3 Software is an item
4 Billing address is an address
5 Shipping address is an address

Aggregation:

1 Item is part of book shop
2 Bookshop staff is a part of bookshop

Association:

1 address is given to customer

2 Billing address is given to customer
3 Shipping address is given to bookshop

Dependency:

1 Order is dependent on bookshop.
2 Customer is dependent on bookshop.
3 Item order is dependent on customer.

Identify attributes:

1 Bookshop: name, address
2 Bookshop staff: name, id
3 Item: title, publisher, year published, price
4 Book: author
5 Music cd: software
6 Software: version
7 Item order: item, quantity
8 Order: sales tax, shipping fee, total
9 Customer: name, id, password
10 Address: street number, street name, city, state, country, postcode
11 Shopping cart
12 Billing address
13 Shipping address

Identifying operations for classes:

Bookshop

Welcome message
Login and password
Error message
Browse and order
Show order and cost

Book:

Display

Music cd:

Display

Software:

Display

Shopping cart:

Additem

Display

Itemorder

Order:

Computeitemstotal

Printinvoice

Display

Itemorder

Customer:

Verifypassward

Additemtoshoppingcart

Creat order

Printbillinglable

Printshppinglable

Address:

Print

Billingaddress:

Print

Shippingadrress:

Print

Creating collaboration diagram:

Identifying objects:

1. B.Bookshop
2. Bs.Bookshop staff
3. O.Order
4. C.Customer
5. a.Address
6. sc.Shopping

Identifying messages:

1. Bookshop gives welcome message to customer
2. Bookshop checks password of customer
3. Customer gets verify of password from bookshop
4. Item are displayed for the customers
5. Staff explains to customer
6. Customer selects an item
7. Staff maintains customer details
8. Bookshop takes order from customer
9. Bookshop sends items to customers
10. Customer pay bill to bookshop
11. A bookshop collects travelling charges from customers
12. Bookshop sends item to the customers address

Creating activity diagrams:

The following needs must be the addresses by the systems for the activities are:

1. When the custom opens the site then welcome message must be displayed.

2. If the customer wants to order books then he creates the login name and password.
3. Authentication is done after the customer login.
4. Customer can login the desired books.
5. The ordered books are transferred to the appropriate customers.

Activities identified are:

1. System welcome message.

2. Customer login

3. System validates password.

4. Customer browses.

5. System displays item information.

6. Customer selects number

7. System creates order

8. Customer done

9. System creates order

10. System shows order and cost

11. Customer agree to pay

12. System sends invoice to customer

13. Bookshop staff ships to customer

Case Study 3 - An Automated Company

Project Analysis:

Automate a small manufacturing company. The resulting application will enable the user to take out a loan, purchase a machine, and over a series of monthly production runs, follow the performance of their company.

Creating use case diagram:

Identifying actors

1. Company is hardware thing
2. Employee is a person in company
3. Technical employee is a person in company
4. Non technical employee is person in company
5. Bank is hardware thing that gives loan to company
6. Customer is a person who buys goods

Identifying use cases

1. Recruit employee
2. Give salary
3. Pay for raw materials
4. Maintain machines
5. Dismiss employee
6. Product goods
7. Work
8. Maintain union
9. Take salary
10. Give loan
11. Collect interest
12. Collect loan
13. Bye goods
14. Pay money

Identifying relationships

Generalization

1. Technical staff is a type of employee
2. Non technical staff is a type of employee Association

1.Company recruits employee

2.Company gives salaries.

3.Company pays for raw materials

4.Company maintains machines

5.Company dismisses employee.
6.Company product goods

7.Employee works in company.

8.Employee takes salary.

9.Bank gives loan

10.Bank collect interest

11.Customer buy goods

12.Customer pays

Create class diagrams:

Identifying classes:

1.Company

2.Bank

3.Employees

4.Technical

5.Non technical

6.Customers

7.Shares

8.Machines

9.Goods

10.Customer

11.Market

12.Sales

13.Status

14.Profit

15.Loss

16.Rawmaterials

17.Department

Identifying relationships between classes:

Inheritance:

1.status contains profit

2.status contains loss

3.employee contais non technical

Aggregation:

1.employee is a part of company

2.machines are part of company

3.department is a part of company

Association:

1.shares are provided by the company

2.bank provide loan for company

3.machines produce the goods

4.raw materials are given to the company

5.sales tells the status

6.customer buys goods through sales in market

Dependency:

1.market is dependent on goods

Identifying attributes:

1.company:name,code,address

2.bank:name,address,branchname

3.employee:department,hrswork

4.technical:name,id,hourswork

5.nontechnical:name,id,hourswork

6.shares:code

7.machines:type,cost,size,capacity

8.goods:name,code

9.customer:name

10.market:name,address

11.sales:quantity,quantity sold, quantity left
12.status:mention status

13.profit

14.loss

15.rawmaterials:type,code, quality type

16.department:name of dept, type

Identifying operations for classes:

Company:

1.check attendance

2.give salary

3.pay for salary

4.maintained machines

5.sell goods to markets

6.recruit employee

7.dismiss employee

8.note raw materials

Customer:

1.buy goods

2.pay money

Bank:

1.give loan

2.collect interest

3.collect loan

Employee:

1.work

2.take salary

3.maintain union

Non technical:

1.work
2.take salary

3.maintain union

Shares

1.increase on profit

2.decrease on loss

machines

1.process raw materials

2.give finished goods

market

1.receive goods

2.sell goods

3.pay for company

4.collect form customer

sales

1.calculate status

Status

profit

1.compute profit or loss

2.calculate status

3.profit amount

loss

1.compute profit or loss

2.calculate status

3.loss amount

Creating collaboration diagram:

Identifying objects

1.c:company

2.s:shares

3.r:raw materials

4.m:machinery

5.sa:sales

6.c:customer

7. m:market

8.g:goods

9.e:employee

10.b:bank

Identifying messages

1.company plans for its development

2.company request loan from bank

3.bank check account of the company

4.company purchase machinery

5.company purchase raw materials

6.company receives order

7.company decides the quality of products

8.company manufactures the products

9.company analyze quality in market

10.customer buys from market

11.production analyses the sales depending on market

12.monthly production decides profit or loss depending on sales

Creating activity diagrams:

Identification of activities:

1.company takes loan from banks2.buy raw materials

Case Study 4- Multi-threaded airport automation

Automate the operations in an airport. your application should support multi aircrafts using several run ways and gates avoiding collisions/conflicts. landing: an aircraft uses the runway, lands and then taxes over to the terminal. Take-off: an air craft taxies to the run way and then takes off.

Creating use case diagram:

Identifying the cases:

1. check luggage
2. give ticket
3. ticket booking
4. take money
5. maintainance
6. give salary
7. take off
8. land off
9. visa check
10. shopping stalls
11. timings
12. pays the amount
13. management

Identifying relationships:

Genenaralization:

1.technical is a kind of staff

2.non technical is a kind of staff

3.local plane are part of aero planes

4.non local planes are part of aero planes

Association:

1. Staff checks luggage
2. Staff gives ticket
3. Staff does the ticket booking

4. Staff maintains the airport

5. Pilot makes the aero plane to take off

6. Pilot makes the aero plane to lane

7. Passenger books the ticket

8. Passenger pays the amount

Include

1. After ticket booking money should be taken

2. Management should give salaries

Extend

1 .after ticket booking the air plane may or may not come in time

Creating class diagram:

Identifying classes:

1. Airport

2. Location management

3. Staff

4. Security

5. Technical staff

6. Non technical staff

7. Customer

8. Visa check

9. Ticket booking

10. online booking

11. aeroplane

12.local planes

13.international planes

14. stopping stalls

15. bakery

16. stationary

17. electronic restaurant
18. Timings

19. Pilots

20. Helpdesk

Identifying relationships between classes:

Aggregation:

1.staff is a part of airport

2.aeroplanes is a part of airport

3.shopping stalls is a part of airport

4.online booking is a part of ticket booking

Generalization:

1. Staff contain visa check
2. Staff contains security
3. Staff contains technical staff
4. Staff contains customs
5. Staff contains nontechnical
6. Staff contains helpdesk
7. Aeroplanes contains local planes
8. Aeroplanes contains international planes
9. Shopping stalls contains bakery
10. Shopping stalls contains stationery
11. Shopping stalls contains electronic
12. Shopping stalls contains restaurant

Dependency:

1.timings are dependent on airport

2.ticket booking is dependent on technical staff

3.aeroplanes are dependent on pilots

Identifying attributes:

1. Airport: name, code

2 . location: name, city, country, state, pincode

3.management: name, designation

4.staff:name,id,type, salary, workinghrs

5.security: name,id,type

6.technical staff: name, id

7.nontechnical staff: name, id, type

8.customs:name,id

9.visacheck:name,id

10.ticketbooking:name of the lane, code, price, seatno, classtype, timings

11.onlinebooking:name,code,destination

12.shopping stalls: name, code, type

13.bakery:name, code, type of items

14.stationery:name, code, type of goods

15.electronic:name,code, type of items

16.restaurant:name,code, type of items

17.timings:name of the plane, code of the plane, time of plane

18.pilots:name, id

19.helpdesk:name, id

Identifying operations for classes:

airport

take off

landing

parking of planes

management

gives salary

maintenance

control staff

staff

work

take salary

customs

check luggage

check persons

visa check

visa check

ticket booking

book ticket

give ticket

Take money

Aeroplanes

Reach destination

Takeoff

Buy goods

Bakery

Pay to management

Stationary

Pay to management

Electronic

Pay to management

Restaurant

Pay to management

Pilots

Fly the plane

Helpdesk

Tell details

Creating collaboration diagrams:

Identifying objects:

1.P:Passenger

2.T:ticket booking

3.St:timings

4.Aa:aeroplanes

5.T:timings

6.A:airport

7.S:shoppingstalls

Identifying messages:

1.passenger uses online reservation of ticket reservation counter

2.passenger know details form ticket booking

3.passenger reserve ticket from ticket booking

4.ticket booking staff check allotments for passenger

5.ticket booking staff allots ticket to passengers

6.passenger know the arrival time of air plane

7.passenger enter to passenger

8.staff check passport

9.Custom staff check luggage of passenger

10.airplane starts check luggage of passenger

11.Passenger travels in plane

Creating Activity Diagram:

Identification of activities;

1.passenger enters

2. Check visa.

3. Grant permission

4.Send out

5.if brought good in shop

6.pay money

7.Don’t pay

8.Pilots land aeroplane in airport

9.inform arrival

10.Boarding of passenger

11.take off of plane

12.reach destination.

Design Patterns

INTRODUCTION

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.

Gang of Four (GOF)

In 1994, four authors Erich Gamma, Richard Helm; Ralph Johnson und John Vlissides published a book titled Design Patterns - Elements of Reusable Object-Oriented Software which initiated the concept of Design Pattern in Software development. These authors are collectively known as Gang of Four (GOF). According to these authors design patterns are primarily based on the following principles of object orientated design.

- Program to an interface not an implementation
- Favor object composition over inheritance

Usage of Design Pattern

Design Patterns have two main usages in software development.

Design patterns provide a standard terminology and are specific to particular scenario. For example, a singleton design pattern signifies use of single object so all developers familiar with single design pattern will make use of single object and they can tell each other that program is following a singleton pattern. Design patterns have been evolved over a long period of time and they provide best solutions to certain problems faced during software development. Learning these patterns helps inexperienced developers to learn software design in an easy and faster way.

Types of Design Pattern

As per the design pattern reference book Design Patterns - Elements of Reusable Object Oriented Software, there are 23 design patterns. These patterns can be classified in three categories: Creational, Structural and behavioral patterns. We'll also discuss another category of design patterns: J2EE design patterns.

Lab 23 & 24 ABSTRACT FACTORY Pattern

Objective:

Implement Abstract Factory Pattern using Java

Lab Task

Implement the Abstract Factory Pattern using Java according to the instructions given in the theory section. You are also supposed to draw class diagram using CASE tools.

Theory:

Abstract Factory patterns works around a super-factory which creates other factories. This factory is also called as Factory of factories. This type of design pattern comes under creational pattern as this pattern provides one of the best ways to create an object. In Abstract Factory pattern an interface is responsible for creating a factory of related objects,without explicitly specifying their classes. Each generated factory can give the objects as per the Factory pattern.

Implementation

We're going to create a *Shape* and *Color* interfaces and concrete classes implementing these interfaces. We creates an abstract factory class *AbstractFactory* as next step. Factory classes*ShapeFactory* and *ColorFactory* are defined where each factory extends *AbstractFactory*. A factory creator/generator class *FactoryProducer* is created.

AbstractFactoryPatternDemo, our demo class uses *FactoryProducer* to get a *AbstractFactory* object. It will pass information (*CIRCLE / RECTANGLE / SQUARE* for *Shape*)to *AbstractFactory* to get the type of object it needs. It also passes information (*RED / GREEN /BLUE* for *Color*) to *AbstractFactory* to get the type of object it needs.

Steps

Use the following steps to implement the above mentioned design pattern.

Step 1
Create an interface for Shapes.
Shape.java

Step 2
Create concrete classes implementing the same interface.
Rectangle.java
Square.java

Step 3
Create an interface for Colors.
Color.java

Step4
Green.java
Blue.java

Step 5
Create an Abstract class to get factories for Color and Shape Objects.
AbstractFactory.java

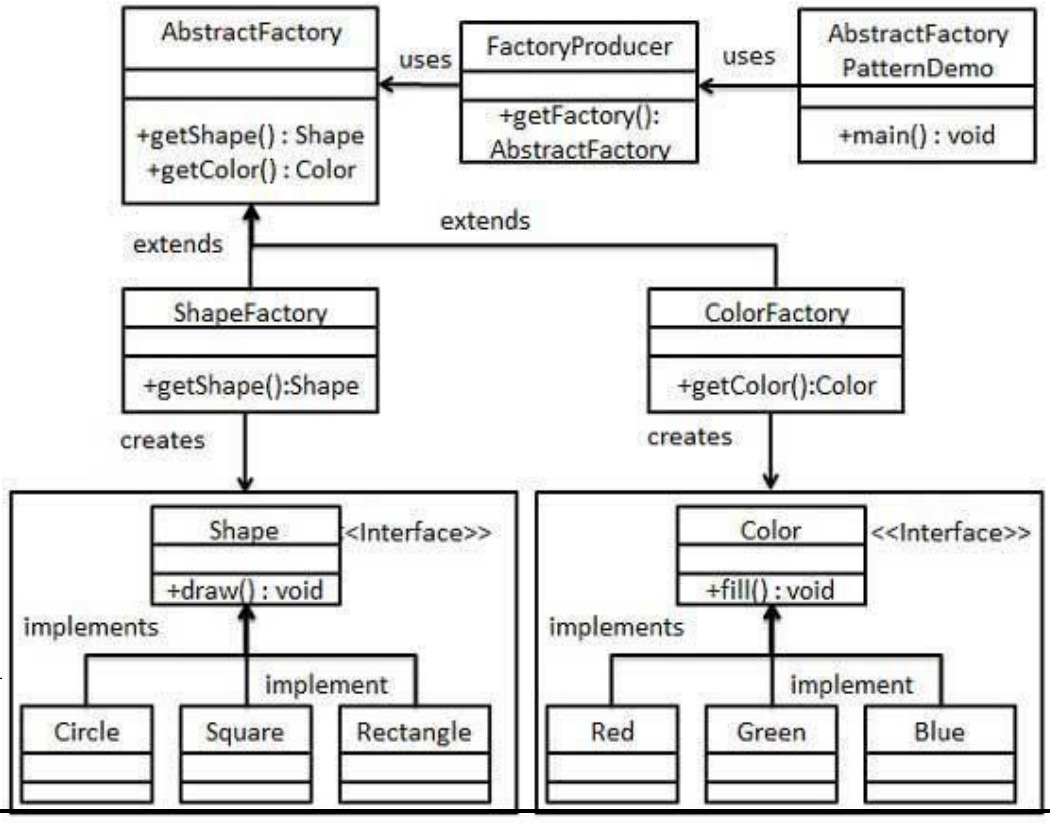
Step 6
Create Factory classes extending AbstractFactory to generate object of concrete class based on
given information.
ShapeFactory.java

ColorFactory.java
}

Step 7
Create a Factory generator/producer class to get factories by passing an information such as Shape or Color
FactoryProducer.java

Step 8
Use the FactoryProducer to get AbstractFactory in order to get factories of concrete classes by
passing information such as type.
AbstractFactoryPatternDemo.java

Class Diagram



Step 9

Verify the output.
Inside Circle::draw() method.
Inside Rectangle::draw() method.
Inside Square::draw() method.
Inside Red::fill() method.
Inside Green::fill() method.
Inside Blue::fill() method.

Tools

NetBeans
MS Visio/ Visual Paradigm

Lab 25 – Singleton Pattern

Objective:

Implement Singleton Pattern using Java

Lab Task

Implement the Singleton Pattern using Java according to the instructions given in the theory section. You are also supposed to draw class diagram using CASE tools.

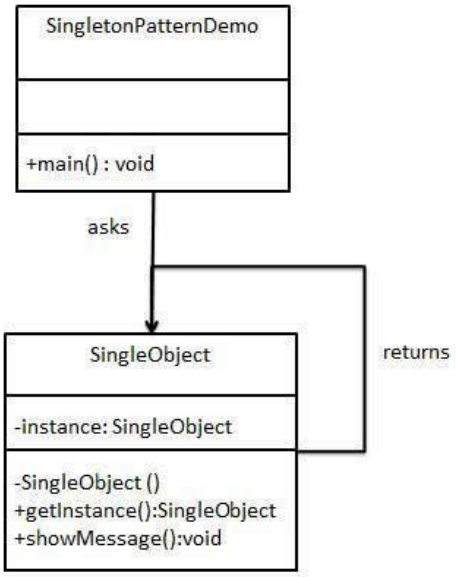
Theory

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 way to create an object. This pattern involves a single class which is responsible to creates own object while making sure that only single object get 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 geta *SingleObject* object.

Class Diagram



Steps

Use the following steps to implement the above mentioned design pattern.

Step 1

Create a Singleton Class.

SingleObject.java

Step 2
Get the only object from the singleton class.
SingletonPatternDemo.java

Step 3
Verify the output.
Hello World!

Tools

NetBeans
MS Visio/ Visual Paradigm

Lab 26-27 Decorator Pattern

Objective:

Implement Decorator Pattern using Java

Lab Task

Implement the Decorator Pattern using Java according to the instructions given in the theory section. You are also supposed to draw class diagram using CASE tools.

Theory

Decorator pattern allows adding new functionality an existing object without altering its structure. This type of design pattern comes under structural pattern as this pattern acts as a wrapper to existing class. This pattern creates a decorator class which wraps the original class and provides additionalfunctionality keeping class methods signature intact.

We are demonstrating use of Decorator pattern via following example in which we'll decorate a shape with some color without alter shape class.

Implementation

We're going to create a *Shape* interface and concrete classes implementing the *Shape* interface.We then create a abstract decorator class *ShapeDecorator* implementing the *Shape* interface and having*Shape* object as its instance variable.

RedShapeDecorator is concrete class implementing *ShapeDecorator*.

DecoratorPatternDemo, our demo class will use *RedShapeDecorator* to decorate *Shape* objects.

Steps

Use the following steps to implement the above mentioned design pattern.

Step 1

Create an interface.

Shape.java

Step 2

Create concrete classes implementing the same interface.

Rectangle.java

Step 3

Create abstract decorator class implementing the *Shape* interface.

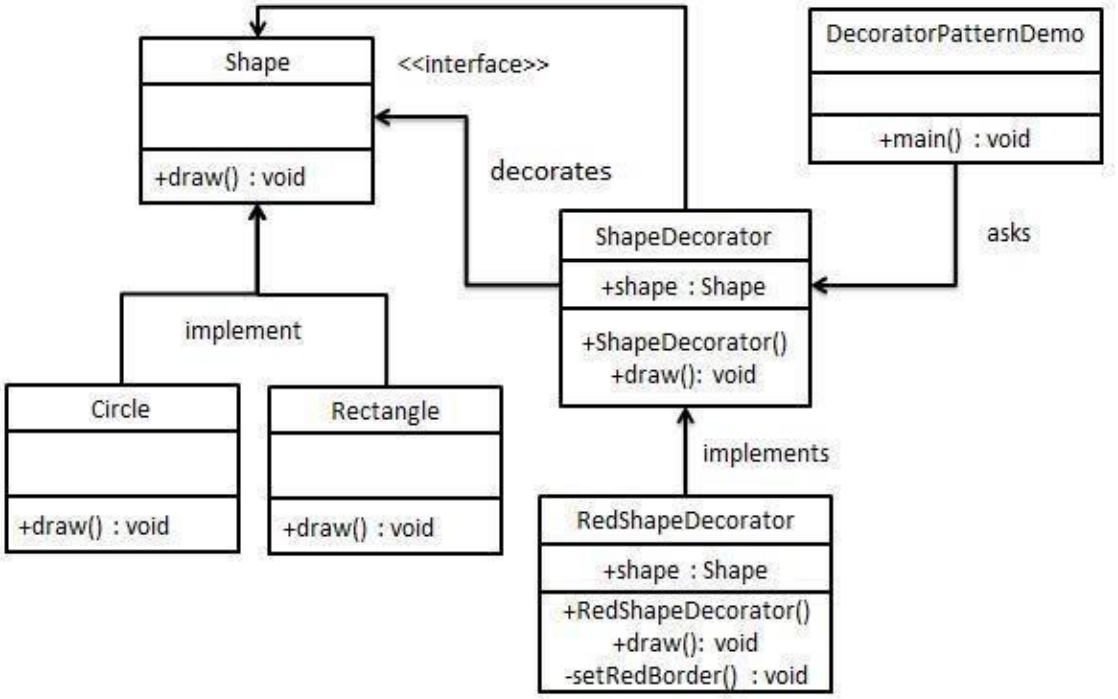
ShapeDecorator.java

Step 4

Create concrete decorator class extending the *ShapeDecorator* class.
RedShapeDecorator.java

Step 5
Use the *RedShapeDecorator* to decorate *Shape* objects.
DecoratorPatternDemo.java

Class Diagram



Step 6
Verify the output.
Circle with normal border
Shape: Circle

Circle of red border
Shape: Circle
Border Color: Red

Rectangle of red border
Shape: Rectangle
Border Color: Red

Tools

- NetBeans
- MS Visio/ Visual Paradigm

Lab 27-28 Adapter Pattern

Objective:

Implement Adapter Pattern using Java

Lab Task

Implement the Adapter Pattern using Java according to the instructions given in the theory section. You are also supposed to draw class diagram using CASE tools.

Theory

Adapter pattern works as a bridge between two incompatible interfaces. This type of design pattern comes under structural pattern as this pattern combines the capability of two independent interfaces. This pattern involves a single class which is responsible to join functionalities of independent or incompatible interfaces.

A real life example could be a case of card reader which acts as an adapter between memory card and a laptop. You plugins the memory card into card reader and card reader into the laptop so that memory card can be read via laptop. We are demonstrating use of Adapter pattern via following example in which an audio player device can play mp3 files only and wants to use an advanced audio player capable of playing vlc and mp4 files.

Implementation

We've an interface *MediaPlayer* interface and a concrete class *AudioPlayer* implementing the*MediaPlayer* interface. *AudioPlayer* can play mp3 format audio files by default.We're

having another interface *AdvancedMediaPlayer* and concrete classes implementingthe*AdvancedMediaPlayer* interface.These classes can play vlc and mp4 format files.

We want to make *AudioPlayer* to play other formats as well. To attain this, we've created an adapter class *MediaAdapter* which implements the *MediaPlayer* interface and uses *AdvancedMediaPlayer*objects to play the required format.*AudioPlayer* uses the adapter class *MediaAdapter* passing it the desired audio type without knowing the actual class which can play the desired format. *AdapterPatternDemo*, our demo class will use*AudioPlayer* class to play various formats.

Steps

Use the following steps to implement the above mentioned design pattern.

Step 1

Create interfaces for Media Player and Advanced Media Player.

MediaPlayer.java

Step 2

Create concrete classes implementing the *AdvancedMediaPlayer* interface.

VlcPlayer.java Mp4Player.java
Step 3

Create adapter class implementing the *MediaPlayer* interface.

MediaAdapter.java

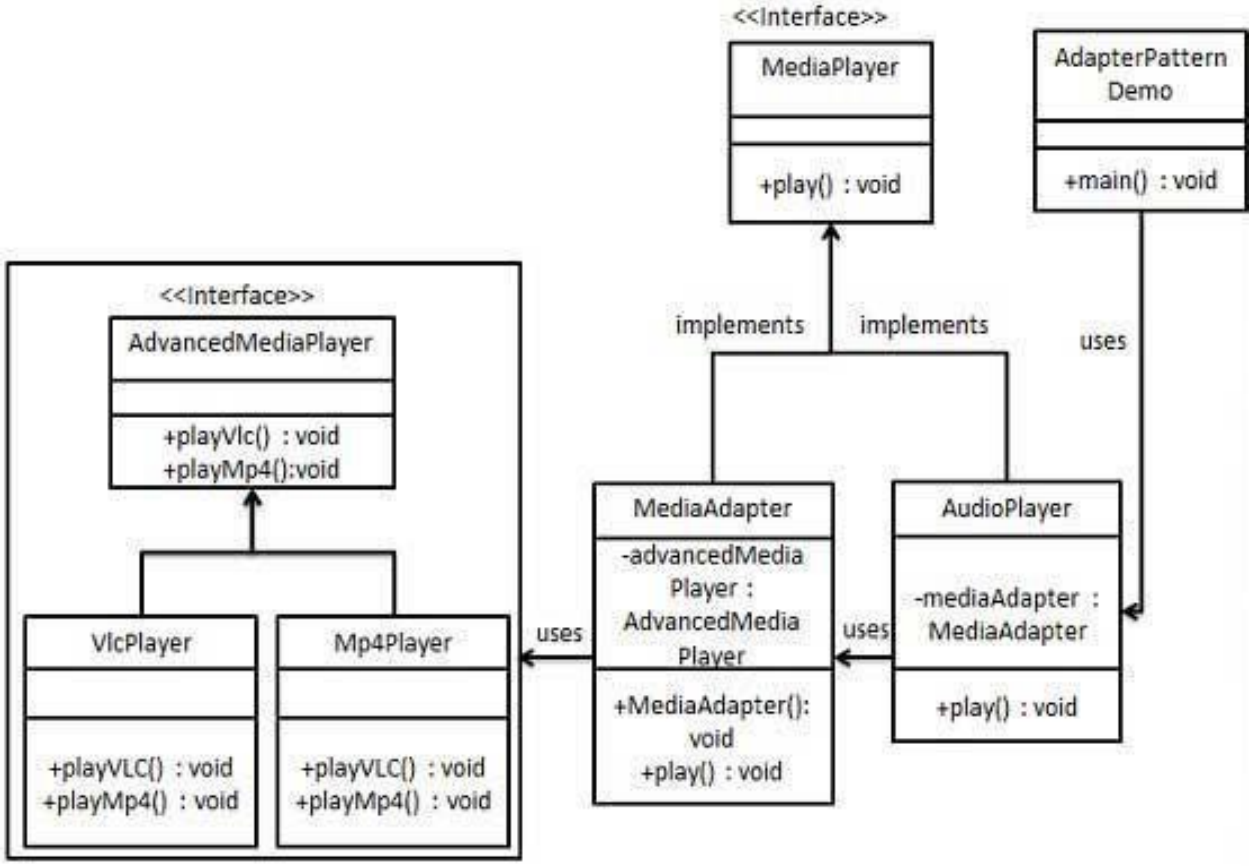
Step 4

Create concrete class implementing the *MediaPlayer* interface.

AudioPlayer.java

Step 5
Use the AudioPlayer to play different types of audio formats.
AdapterPatternDemo.java

Class Diagram



Step 6
Verify the output.
Playing mp3 file. Name: beyond the horizon.mp3

Playing mp4 file. Name: alone.mp4
Playing vlc file. Name: far far away.vlc
Invalid media. avi format not supported

Tools

- NetBeans
- MS Visio/ Visual Paradigm

Lab-29 & 30 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.