CS6502 OBJECT ORIENTED ANALYSIS AND DESIGN L T P C
3 0 0 3

UNIT I UML DIAGRAMS 9

Introduction to OOAD – Unified Process – UML diagrams – Use Case – Class Diagrams– Interaction Diagrams – State Diagrams – Activity Diagrams – Package, component and Deployment Diagrams.

UNIT II DESIGN PATTERNS 9

GRASP: Designing objects with responsibilities – Creator – Information expert – Low Coupling – High Cohesion – Controller – Design Patterns – creational – factory method – structural – Bridge – Adapter – behavioral – Strategy – observer.

UNIT III CASE STUDY 9

Case study – the Next Gen POS system, Inception -Use case Modeling – Relating Use cases – include, extend and generalization – Elaboration – Domain Models – Finding conceptual classes and description classes – Associations – Attributes – Domain model refinement – Finding conceptual class Hierarchies – Aggregation and Composition.

UNIT IV APPLYING DESIGN PATTERNS 9

System sequence diagrams – Relationship between sequence diagrams and use cases Logical architecture and UML package diagram – Logical architecture refinement – UML class diagrams – UML interaction diagrams – Applying GoF design patterns.

UNIT V CODING AND TESTING 9

Mapping design to code – Testing: Issues in OO Testing – Class Testing – OO Integration Testing – GUI Testing – OO System Testing.

TOTAL: 45 PERIODS

TEXT BOOK:

1. Craig Larman, “Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development”, Third Edition, Pearson Education, 2005.

REFERENCES:

1. Simon Bennett, Steve Mc Robb and Ray Farmer, “Object Oriented Systems Analysis and Design Using UML”, Fourth Edition, Mc-Graw Hill Education, 2010.
2. Erich Gamma, a n d Richard Helm, Ralph Johnson, John Vlissides, “Design patterns: Elements of Reusable Object-Oriented Software”, Addison-Wesley, 1995.
3. Martin Fowler, “UML Distilled: A Brief Guide to the Standard Object Modeling Language”, Third edition, Addison Wesley, 2003.
4. Paul C. Jorgensen, “Software Testing:- A Craftsman’s Approach”, Third Edition, Auerbach Publications, Taylor and Francis Group, 2008.

TABLE OF CONTENTS

UNIT I

UML DIAGRAMS

PRE-REQUSITE DISCUSSION:

The Unified Modeling Language is a visual language for specifying, constructing and documenting the artifacts of systems.

Unified Modeling Language (UML) is a standardized general-purpose modeling language in the field of software engineering. The standard is managed, and was created by, the Object Management Group. UML includes a set of graphic notation techniques to create visual models of software-intensive systems.

CONCEPTS:

The Unified Modeling Language is commonly used to visualize and construct systems which are software intensive.

Because software has become much more complex in recent years, developers are finding it more challenging to build complex applications within short time periods.

Even when they do, these software applications re often filled with bugs, and it can take programmers weeks to find and fix them.

This s time that has been wasted, since an approach could have been used which would have educed the number of bugs before the application was completed.

Three ways to apply UML:

1. UML as sketch:
   Informal and incomplete diagrams Created to explore difficult parts of the problem
2. UML as blueprint:
   Detailed design diagram Used for better understanding of code
3. UML as programming language:
   Complete executable specification of a software system in UML

Three perspectives to apply UML:

1. Conceptual perspective: Diagrams describe the things of real world. UML diagrams are used to describe things in situations of real world. Raw UML object diagram notation used to visualize.

2. Specification perspective: Diagrams describe software abstractions or components with specifications and interfaces.

It describes the real world things, software abstraction and component with specification and interfaces. Raw UML class diagram notation used to visualize software components.

3. Implementation perspective: Diagrams describe software implementation in a particular technology

PHASES OF UP:

The Unified Process has emerged as a popular iterative software development process for building object oriented systems.

Unified process is a iterative process, risk driven process and architecture centric approach to software development. It comes under software development process.

The Unified Software Development Process or Unified Process is a popular iterative and incremental software development process framework. The best-known and extensively documented refinement of the Unified Process is the Rational Unified Process (RUP).

I.Inception:

Inception is the initial stage of project. It deals with approximate vision, business case, scope of project and vague estimation.

Initial stage of project
Approximate vision
Business case and scope
Vague estimate

Inception is the smallest phase in the project, and ideally it should be quite short. If the Inception Phase is long then it may be an indication of excessive up-front specification, which is contrary to the spirit of the Unified Process.

The following are typical goals for the Inception phase.
Establish a justification or business case for the project
Establish the project scope and boundary conditions
Outline the use cases and key requirements that will drive the design tradeoffs
Outline one or more candidate architectures
Identify risks
Prepare a preliminary project schedule and cost estimate
The Lifecycle Objective Milestone marks the end of the Inception phase.

Advantages of inception.

Estimation or plans are expected to be reliable.
After inception, design architecture can be made easily becauseall the use cases are written in detail.

II. Elaboration:

During the Elaboration phase the project team is expected to capture a healthy majority of the system requirements. However, the primary goals of Elaboration are to address known risk factors and to establish and validate the system architecture. Common processes undertaken in this phase include the creation of use case diagrams, conceptual diagrams (class diagrams with only

basic notation) and package diagrams (architectural diagrams).

Refined vision
Core architecture
Resolution of high riskIdentification of most requirement and scope
Realistic estimate

III. Construction:

Construction is the largest phase in the project. In this phase the remainder of the system is built on the foundation laid in Elaboration.

System features are implemented in a series of short, timeboxed iterations. Each iteration results in an executable release of the software.

It is customary to write full text use cases during the construction phase and each one becomes the start of a new iteration.

Common UML (Unified Modelling Language) diagrams used during this phase include Activity, Sequence, Collaboration, State (Transition) and Interaction Overview diagrams. The Initial Operational Capability Milestone marks the end of the Construction phase.

Design the elements
Preparation for deployment

IV. Transition:

The final project phase is Transition. In this phase the system is deployed to the target users. Feedback received from an initial release (or initial releases) may result in further refinements to be incorporated over the course of several Transition phase iterations. The Transition phase also includes system conversions and user training. The Product Release Milestone marks the end of the Transition phase. Beta tests Deployments

NEXTPOS SYSTEM:

The case study is the NextGen point-of-sale (POS) system. In this apparently straightforward problem domain, we shall see that there are very interesting requirement and design problems to solve. In addition, it is a realistic problem; organizations really do write POS systems using object technologies.

A POS system is a computerized application used (in part) to record sales and handle payments; it is typically used in a retail store. It includes hardware components such as a computer and bar code scanner, and software to run the system.

It interfaces to various service applications, such as a third-party tax calculator and inventory control. These systems must be relatively fault-tolerant; that is, even if remote services are temporarily unavailable (such as the inventory system), they must still be capable of capturing sales and handling at least cash payments (so that the business is not crippled).

A POS system increasingly must support multiple and varied client-side terminals and interfaces. These include a thin-client Web browser terminal, a regular personal computer with

something like a Java Swing graphical user interface, touch screen input, wireless PDAs, and so forth.

Furthermore, we are creating a commercial POS system that we will sell to different clients with disparate needs in terms of business rule processing.

Each client will desire a unique set of logic to execute at certain predictable points in scenarios of using the system, such as when a new sale is initiated or when a new line item is added.

Therefore, we will need a mechanism to provide this flexibility and customization.

Using an iterative development strategy, we are going to proceed through requirements, object-oriented analysis, design, and Implementation.

ARCHITECTURAL LAYERS AND CASE STUDY EMPHASIS

A typical object-oriented information system is designed in terms of several architectural layers or subsystems.

The following is not a complete list, but provides an example:

• User Interface: Graphical interface; windows.

• Application Logic and Domain Objects: Software objects representing domain concepts (for example, a software class named Sale) that fulfill application requirements.

• Technical Services: General purpose objects and subsystems that provide supporting technical services, such as interfacing with a database or error logging. These services are usually application-independent and reusable across several systems.

OOA/D is generally most relevant for modeling the application logic and technical service layers. The NextGen case study primarily emphasizes the problem domain objects, allocating responsibilities to them to fulfill the requirements of the application. Object-oriented design is also applied to create a technical service subsystem for interfacing with a database. In this design approach, the UI layer has very little responsibility; it is said to be thin. Windows do not contain code that performs application logic or processing. Rather, task requests are forwarded on to other layers.

Inception Phase

This is the part of the project where the original idea is developed. The amount of work done here is dependent on how formal project planning is done in your organization and the size of the project. During this part of the project some technical risk may be partially evaluated and/or eliminated. This may be done by using a few throw away prototypes to test for technical feasability of specific system functions. Normally this phase would take between two to six weeks for large projects and may be only a few days for smaller projects.

The following should be done during this phase:

1. Project idea is developed.
2. Assess the capablilities of any current system that provides similar functionality to the new project even if the current system is a manual system. This will help determine cost savings that the new system can provide.
3. Utilize as many users and potential users as possible along with technical staff, customers, and management to determine desired system features, functional capabilities, and performance requirements. Analyze the scope of the proposedsystem.
4. Identify feature and functional priorities along with preliminary risk assessment of each

system feature or function.

5. Identify systems and people the system will interact with.

6. For large systems, break the system down into subsystems if possible.

7. Identify all major use cases and describe significant use cases. No need to make expanded use cases at this time. This is just to help identify and present system functionality.

8. Develop a throw away prototype of the system with breadth and not depth. This prototype will address some of the greatest technical risks. The time to develop this prototype should be specifically limited. For a project that will take about one year, the prototype should take one month.

9. Present a business case for the project (white paper) identifying rough cost and value of the project. The white paper is optional for smaller projects. Define goals, estimate risks, and resources required to complete the project.

10. Set up some major project milestones (mainly for the elaboration phase). A rough estimate of the overall project size is made.

11. Preliminary determination of iterations and requirements for each iteration. This outlines system functions and features to be included in each iteration. Keep in mind that this plan will likely be changes as risks are further assessed and more requirements are determined.

12. Management Approval for a more serious evaluation of the project. This phase is done once the business case is presented with major milestones determined (not cast in stone yet) and management approves the plan.

At this point the following should be complete:

Business case (if required) with risk assessment.

Preliminary project plan with preliminary iterations planned.

Core project requirements are defined on paper.

Major use cases are defined.

The inception phase has only one iteration. All other phases may have multiple iterations. The overriding goal of the inception phase is to achieve concurrence among all stakeholders on the lifecycle objectives for the project.

The inception phase is of significance primarily for new development efforts, in which there are significant business and requirements risks which must be addressed before the project can proceed.

For projects focused on enhancements to an existing system, the inception phase is more brief, but is still focused on ensuring that the project is both worth doing and possible to do.

Objectives

The primary objectives of the Inception phase include:

Establishing the project's software scope and boundary conditions, including an operational vision, acceptance criteria and what is intended to be in the product and what is not.

Discriminating the critical use cases of the system, the primary scenarios of operation that will drive the major design tradeoffs.Exhibiting, and maybe demonstrating, at least one candidate architecture against some of the primary scenarios

Estimating the overall cost and schedule for the entire project (and more detailed estimates for the elaboration phase that will immediately follow) Estimating potential risks (the sources of unpredictability)

Preparing the supporting environment for the project.

Essential Activities

The essential activities of the Inception include:

Formulating the scope of the project. This involves capturing the context and the most important requirements and constraints to such an extent that you can derive acceptance criteria for the end product.

Planning and preparing a business case. Evaluating alternatives for risk management, staffing, project plan, and cost/schedule/profitability tradeoffs.

Synthesizing a candidate architecture, evaluating tradeoffs in design, and in make/buy/reuse, so that cost, schedule and resources can be estimated. The aim here is to demonstrate feasibility through some kind of proof of concept. This may take the form of a model which simulates what is required, or an initial prototype which explores that are considered to be the areas of high risk. Preparing the environment for the project, assessing the project and the organization, selecting tools, deciding which parts of the process to improve.

USECASE MODELING:

The Use Case Model describes the proposed functionality of the new system. A Use Case represents a discrete unit of interaction between a user (human or machine) and the system. A Use Case is a single unit of meaningful work; for example login to system, register with system and create order are all Use Cases. Each Use Case has a description which describes the functionality that will be built in the proposed system. A Use Case may 'include' another Use Case's functionality or 'extend' another Use Case with its own behavior. Use Cases are typically related to 'actors'. An actor is a human or machine entity that interacts with the system to perform meaningful work.

Actors

An Actor is a user of the system. This includes both human users and other computer systems. An Actor uses a Use Case to perform some piece of work which is of value to the business.

The set of Use Cases an actor has access to defines their overall role in the system and the scope of their action.

Constraints, Requirements and Scenarios

The formal specification of a Use Case includes:

1. Requirements:

These are the formal functional requirements that a Use Case must provide to the end user. They correspond to the functional specifications found in structured methodologies. A requirement is a contract that the Use Case will perform some action or provide some value to the system.

2. Constraints:

These are the formal rules and limitations that a Use Case operates under, and includes pre- post- and invariant conditions. A pre-condition specifies what must have already occurred or be in place before the Use Case may start. A post-condition documents what will be true once the Use Case is complete. An invariant specifies what will be true throughout the time the Use Case operates.

3.Scenarios:

Scenarios are formal descriptions of the flow of events that occurs during a Use Case instance. These are usually described in text and correspond to a textual representation of the Sequence Diagram.

USE CASE RELATIONSHIPS.

Use case relationships is divided into three types

1. Include relationship
2. Extend relationship
3. Generalization

1. Include relationship:

It is common to have some practical behavior that is common across several use cases. It is simply to underline it or highlight it in some fashion

Example:

Paying by credit: Include Handle Credit Payment

2. Extend relationship:

Extending the use case or adding new use case to the process Extending use case is triggered by some conditions called extension point.

3. Generalization:

Complicated work and unproductive time is spending in this use case relationship. Use case experts are successfully doing use case work without this relationship.

Includes and Extends relationships between Use Cases

One Use Case may include the functionality of another as part of its normal processing. Generally, it is assumed that the included Use Case will be called every time the basic path is run. An example may be to list a set of customer orders to choose from before modifying a selected order in this case the <list orders> Use Case may be ncluded every time the <modify order> Use Case is run. A Use Case may be included by one or more Use Cases, so it helps to reduce duplication of functionality by factoring out common behavior into Use Cases that are re-used many times. One Use Case may extend the behavior of another - typically when exceptional circumstances are encountered.

Relationships Between Use Cases

Use cases could be organized using following relationships:

Generalization
Association
Extend
Include

Generalization Between Use Cases

Generalization between use cases is similar to generalization between classes; child use case inherits properties and behavior of the parent use case and may override the behavior of the parent.

NOTATION:

Generalization is rendered as a solid directed line with a large open arrowhead (same as generalization between classes).

Generalization between use cases

Association Between Use Cases

Use cases can only be involved in binary Associations. Two use cases specifying the same subject cannot be associated since each of them individually describes a complete usage of the system.

Extend Relationship

Extend is a directed relationship from an extending use case to an extended use case that specifies how and when the behavior defined in usually supplementary (optional) extending use case can be inserted into the behavior defined in the use case to be extended.

The extension takes place at one or more extension points defined in the extended use case.

The extend relationship is owned by the extending use case. The same extending use case can extend more than one use case, and extending use case may itself be extended.

Extend relationship between use cases is shown by a dashed arrow with an open arrowhead from the extending use case to the extended (base) use case. The arrow is labeled with the keyword Registration use case is meaningful on its own, and it could be extended with optional Get Help On Registration use case. The condition of the extend relationship as well as the references to the extension points are optionally shown in a Note attached to the corresponding extend relationship.

Registration use case is conditionally extended by Get Help On Registration use case in extension point Registration Help

Include Relationship

An include relationship is a directed relationship between two use cases, implying that the behavior of the required (not optional) included use case is inserted into the behavior of the including (base) use case. Including use case depends on the addition of the included use case. The include relationship is intended to be used when there are common parts of the behavior of two or more use cases. This common part is extracted into a separate use case to be included by all the base use cases having this part in common. As the primary use of the include relationship is to reuse common parts, including use cases are usually not complete by themselves but dependent on the included use cases.

Include relationship between use cases is shown by a dashed arrow with an open arrowhead from the including (base) use case to the included (common part) use case. The arrow is labeled with the keyword «include».

CLASS DIAGRAM:

The class diagram is the main building block of object oriented modelling. It is used both for general conceptual modelling of the systematics of the application, and for detailed modelling translating the models into programming code. Class diagrams can also be used for data modeling.

The classes in a class diagram represent both the main objects, interactions in the application and the classes to be programmed.

The top part contains the name of the class. It is printed in bold and centered, and the first letter is capitalized.

The middle part contains the attributes of the class. They are left-aligned and the first letter is lowercase.

The bottom part contains the methods the class can execute. They are also left-aligned and the first letter is lowercase.

In the design of a system, a number of classes are identified and grouped together in a class diagram which helps to determine the static relations between those objects. With detailed modelling, the classes of the conceptual design are often split into a number of subclasses.

Links

A Link is the basic relationship among objects.

Association

An association represents a family of links. A binary association (with two ends) is normally represented as a line. An association can link any number of classes. An association with three links is called a ternary association. An association can be named, and the ends of an association can be adorned with role names, ownership indicators, multiplicity, visibility, and other properties.

There are four different types of association: bi-directional, uni-directional, Aggregation (includes Composition aggregation) and Reflexive. Bi-directional and uni-directional associations are the most common ones. For instance, a flight class is associated with a plane class bi-directionally. Association represents the static relationship shared among the objects of two classes.

Aggregation

Aggregation is a variant of the "has a" association relationship; aggregation is more specific than association. It is an association that represents a part-whole or part-of relationship. As shown in the image, a Professor 'has a' class to teach. As a type of association, an aggregation can be named and have the same adornments that an association can. However, an aggregation may not involve more than two classes; it must be a binary association.

Aggregation can occur when a class is a collection or container of other classes, but the contained classes do not have a strong lifecycle dependency on the container. The contents of the container are not automatically destroyed when the container is.

In UML, it is graphically represented as a hollow diamond shape on the containing class with a single line that connects it to the contained class. The aggregate is semantically an extended object that is treated as a unit in many operations, although physically it is made of several lesser objects.

INTERACTION DIAGRAMS:

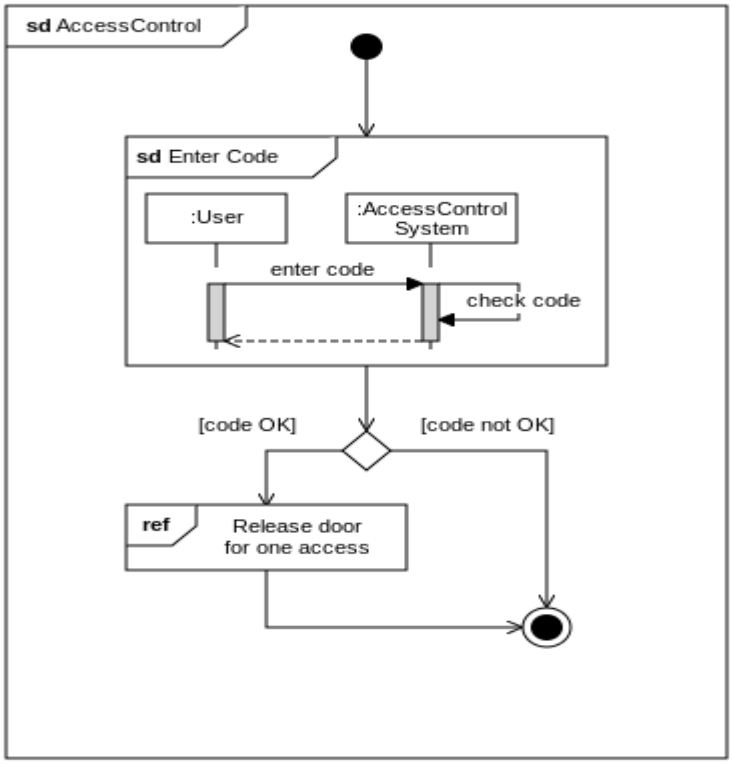
Interaction Overview Diagram is one of the thirteen types of diagrams of the Unified Modeling Language (UML), which can picture a control flow with nodes that can contain interaction diagrams.

The interaction overview diagram is similar to the activity diagram, in that both visualize a sequence of activities.

The difference is that, for an interaction overview, each individual activity is pictured as a frame which can contain a nested interaction diagrams. This makes the interaction overview diagram useful to "deconstruct a complex scenario that would otherwise require multiple if-then-else paths to be illustrated as a single sequence diagram".

The other notation elements for interaction overview diagrams are the same as for activity diagrams.

These include initial, final, decision, merge, fork and join nodes. The two new elements in the interaction overview diagrams are the "interaction occurrences" and "interaction elements."



STATE DIAGRAMS:

The state diagram in the Unified Modeling Language is essentially a Harel statechart with standardized notation which can describe many systems, from computer programs to business processes.

In UML 2 the name has been changed to State Machine Diagram. The following are the basic notational elements that can be used to make up a diagram:

Filled circle, representing to the initial state

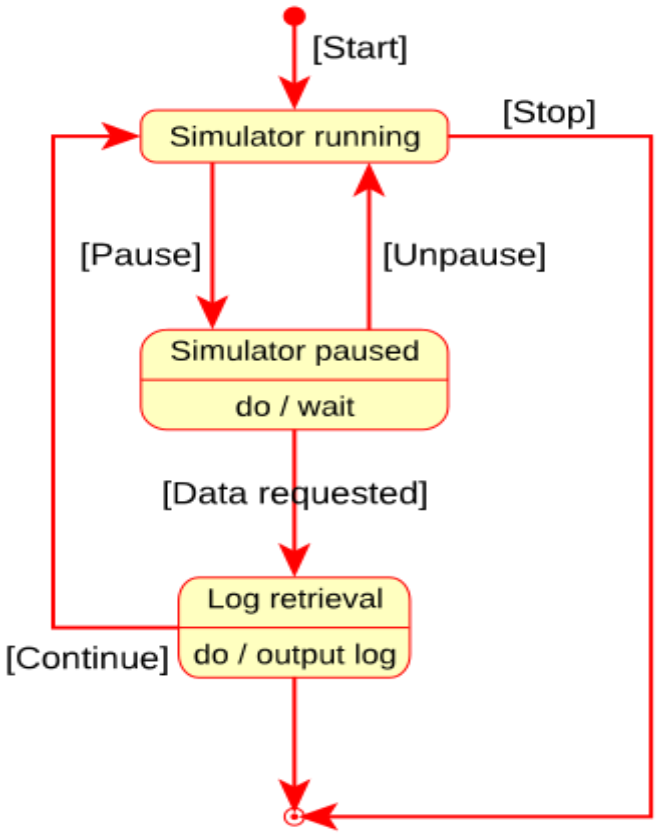
Hollow circle containing a smaller filled circle, indicating the final state (if any)

Rounded rectangle, denoting a state. Top of the rectangle contains a name of the state. Can contain a horizontal line in the middle, below which the activities that are done in that state are indicated

Arrow, denoting transition. The name of the event (if any) causing this transition labels the arrow body.

A guard expression may be added before a "/" and enclosed in square-brackets (eventName[guardExpression]), denoting that this expression must be true for the transition to take place. If an action is performed during this transition, it is added to the label following a "/" (eventName[guardExpression]/action).

Thick horizontal line with either x>1 lines entering and 1 line leaving or 1 line entering and x>1 lines leaving. These denote join/fork, respectively.



ACTIVITY DIAGRAMS:

Class diagram
Component diagram
Composite structure diagram
Deployment diagram
Object diagram
Package diagram
Profile diagram

Behavioral UML diagrams

Activity diagram
Communication diagram
Interaction overview diagram
Sequence diagram
State diagram
Timing diagram
Use case diagram

Activity diagrams are graphical representations of workflows of stepwise activities and actions with support for choice, iteration and concurrency. In the Unified Modeling Language, activity diagrams are intended to model both computational and organisational processes (i.e. workflows). Activity diagrams show the overall flow of control.

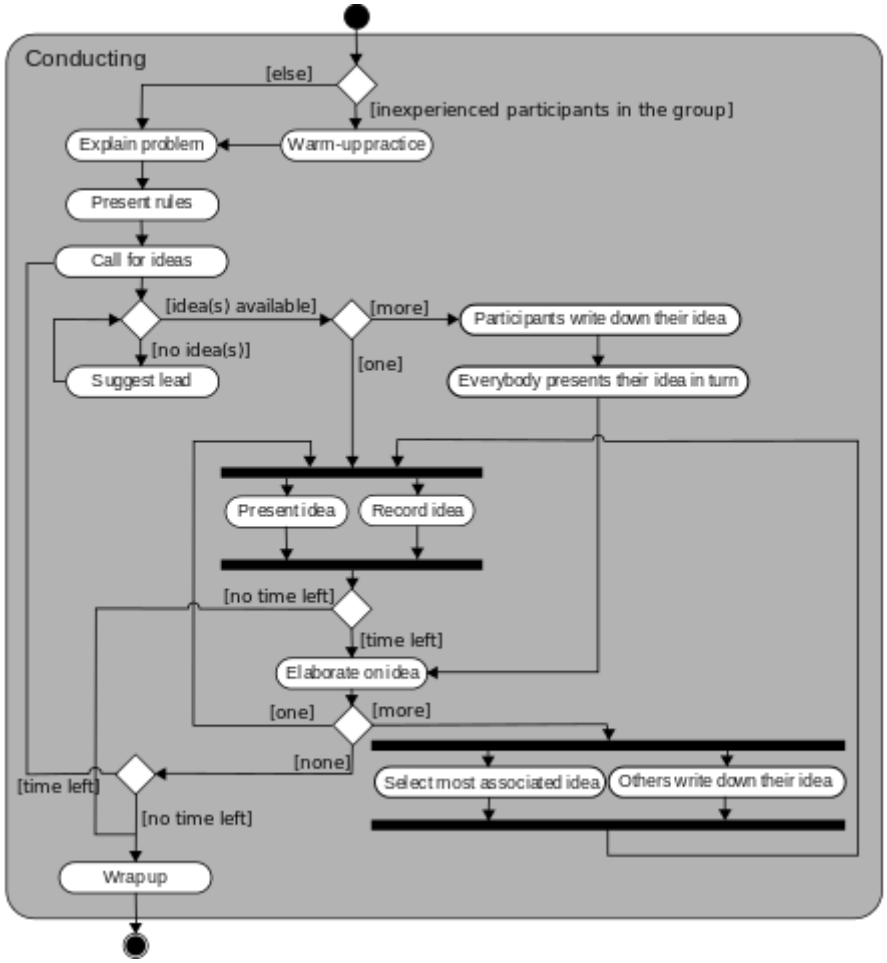
Activity diagrams are constructed from a limited number of shapes, connected with arrows. The most important shape types:

rounded rectangles represent actions;
diamonds represent decisions;
bars represent the start (split) or end (join) of concurrent activities;
a black circle represents the start (initial state) of the workflow;
an encircled black circle represents the end (final state).

Arrows run from the start towards the end and represent the order in which activities happen.

Activity diagrams may be regarded as a form of flowchart. Typical flowchart techniques lack constructs for expressing concurrency[citation needed. However, the join and split symbols in activity diagrams only resolve this for simple cases; the meaning of the model is not clear when they are arbitrarily combined with decisions or loops.

UML activity diagrams in version 2.x can be used in various domains, i.e. in design of embedded systems. It is possible to verify such a specification using model checking technique.



PACKAGE, COMPONENT AND DEPLOYMENT DIAGRAMS:

A deployment diagram in the Unified Modeling Language models the physical deployment of artifacts on nodes.

To describe a web site, for example, a deployment diagram would show what hardware components ("nodes") exist (e.g., a web server, an application server, and a database server), what software components ("artifacts") run on each node (e.g., web application, database), and how the different pieces are connected (e.g. JDBC, REST, RMI).

The nodes appear as boxes, and the artifacts allocated to each node appear as rectangles within the boxes. Nodes may have subnodes, which appear as nested boxes. A single node in a deployment diagram may conceptually represent multiple physical nodes, such as a cluster of database servers.

There are two types of Nodes:

Device Node
Execution Environment Node

Device nodes are physical computing resources with processing memory and services to execute software, such as typical computers or mobile phones. An execution environment node (EEN) is a software computing resource that runs within an outer node and which itself provides a service to host and execute other executable software elements.

SIG**NIFICANCE:**

These are the important for the design of any project done by the software designer

APPLICATION AREA:

used in all IT companies for Designing

REFERENCE:

1. Craig Larman, “Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development”, Third Edition, Pearson Education, 2005.

2. Martin Fowler, “UML Distilled: A Brief Guide to the Standard Object Modeling Language”, Third edition, Addison Wesley, 2003.

UNIT II

DESIGN PATTERNS

PRE-REQUSITE DISCUSSION:

One way to describe object design “After identifying your requirements and creating a domain model, then add methods to the software classes, and define the messaging between the objects to fulfill the requirements”.

•Not really answer the following questions:
•What methods belong to where?
•How the objects should interact?
•GARSP as s Methodical Approach to Learning Basic Object Design

CONCEPT:

GRASP: Designing objects with responsibilities:

General Responsibility Assignment Software Patterns (or Principles), abbreviated GRASP, consists of guidelines for assigning responsibility to classes and objects in object-oriented design.

The different patterns and principles used in GRASP are: Controller, Creator, Indirection, Information Expert, High Cohesion, Low Coupling, Polymorphism, Protected Variations, and Pure Fabrication.

All these patterns answer some software problem, and in almost every case these problems are common to almost every software development project. These techniques have not been invented to create new ways of working, but to better document and standardize old, tried-and-tested programming principles in object-oriented design.

Computer scientist Craig Larman states that "the critical design tool for software development is a mind well educated in design principles. It is not the UML or any other technology."[1] Thus, GRASP is really a mental toolset, a learning aid to help in the design of object-oriented software.

CONTROLLER:

The Controller pattern assigns the responsibility of dealing with system events to a non-UI class that represents the overall system or a use case scenario. A Controller object is a non-user interface object responsible for receiving or handling a system event.

A use case controller should be used to deal with all system events of a use case, and may be used for more than one use case (for instance, for use cases Create User and Delete User, one can have a single UserController, instead of two separate use case controllers).

It is defined as the first object beyond the UI layer that receives and coordinates ("controls") a system operation.

The controller should delegate the work that needs to be done to other objects; it coordinates or controls the activity. It should not do much work itself.

The GRASP Controller can be thought of as being a part of the Application/Service layer (assuming that the application has made an explicit distinction between the application/service layer and the domain layer) in an object-oriented system with Common layers in an information system logical architecture.

CREATOR:creational – factory method

Creation of objects is one of the most common activities in an object-oriented system. Which class is responsible for creating objects is a fundamental property of the relationship between objects of particular classes.

In general, a class B should be responsible for creating instances of class A if one, or preferably more, of the following apply:

Instances of B contain or compositely aggregate instances of A
Instances of B record instances of A
Instances of B closely use instances of A
Instances of B have the initializing information for instances of A and pass it on creation.

"Factory pattern" redirects here. For the GoF design patterns using factories, see factory method pattern and abstract factory pattern.

In object-oriented programming, a factory is an object for creating other objects – formally a factory is simply an object that returns an object from some method call, which is assumed to be "new".

More broadly, a subroutine that returns a "new" object may be referred to as a "factory", as in factory method or factory function. This is a basic concept in OOP, and forms the basis for a number of related software design patterns.

Object creation:

Factory objects are used in situations where getting hold of an object of a particular kind is a more complex process than simply creating a new object, notably if complex allocation or initialization is desired.

Some of the processes required in the creation of an object include determining which object to create, managing the lifetime of the object, and managing specialized build-up and tear-down concerns of the object.

The factory object might decide to create the object's class (if applicable) dynamically, return it from an object pool, do complex configuration on the object, or other things. Similarly, using this definition, a singleton implemented by the singleton pattern is a formal factory – it returns an object, but does not create new objects beyond the single instance.

The simplest example of a factory is a simple factory function, which just invokes a constructor and returns the result. In Python, a factory function f that instantiates a class A can be implemented as:

```
def f():
```

```
        return A()
```

A simple factory function implementing the singleton pattern is:

```
def f():
  if f.obj is None:
    f.obj = A()
  return f.obj

f.obj = None
```

Factories are used in various design patterns, specifically in creational patterns. Specific recipes have been developed to implement them in many languages.

For example, several "GoF patterns", like the "Factory method pattern", the "Builder" or even the "Singleton" are implementations of this concept. The "Abstract factory pattern" instead is a method to build collections of factories.

In some design patterns, a factory object has a method for every kind of object it is capable of creating. These methods optionally accept parameters defining how the object is created, and then return the created object.

Applications

Factory objects are common in toolkits and frameworks where library code needs to create objects of types which may be subclassed by applications using the framework. They are also used in test-driven development to allow classes to be put under test.

Factories determine the actual concrete public class Complex

```
{
    public double real;
    public double imaginary;

    public static Complex FromCartesianFactory(double real, double imaginary)
    {
        return new Complex(real, imaginary);
    }

    public static Complex FromPolarFactory(double modulus, double angle)
    {
        return new Complex(modulus * Math.Cos(angle), modulus * Math.Sin(angle));
    }

    private Complex(double real, double imaginary)
    {
        this.real = real;
        this.imaginary = imaginary;
    }
}
```

Complex product = Complex.FromPolarFactory(1, Math.PI); type of object to be created, and it is here that the object is actually created. As the factory only returns an abstract pointer, the client code does not know – and is not burdened by – the actual concrete type of the object which was just created. However, the type of a concrete object is known by the abstract factory. In particular, this means:

The client code has no knowledge whatsoever of the concrete type, not needing to include any header files or class declarations relating to the concrete type. The client code deals only with the abstract type. Objects of a concrete type are indeed created by the factory, but the client code accesses such objects only through their abstract interface.

Adding new concrete types is done by modifying the client code to use a different factory, a modification which is typically one line in one file. This is significantly easier than modifying the client code to instantiate a new type, which would require changing every location in the code where a new object is created.

```
public class Complex
  {
     public double real;
     public double imaginary;

     public static Complex FromCartesianFactory(double real, double imaginary)
     {
        return new Complex(real, imaginary);
     }

     public static Complex FromPolarFactory(double modulus, double angle)
     {
        return new Complex(modulus * Math.Cos(angle), modulus * Math.Sin(angle));
     }

     private Complex(double real, double imaginary)
     {
        this.real = real;
        this.imaginary = imaginary;
     }
  }
```

Complex product = Complex.FromPolarFactory(1, Math.PI);

High Cohesion

High Cohesion is an evaluative pattern that attempts to keep objects appropriately focused, manageable and understandable.

High cohesion is generally used in support of Low Coupling. High cohesion means that the responsibilities of a given element are strongly related and highly focused. Breaking programs into classes and subsystems is an example of activities that increase the cohesive properties of a system.

Alternatively, low cohesion is a situation in which a given element has too many unrelated responsibilities. Elements with low cohesion often suffer from being hard to comprehend, hard to reuse, hard to maintain and averse to change.

The Indirection pattern supports low coupling (and reuse potential) between two elements by assigning the responsibility of mediation between them to an intermediate object. An example of this is the introduction of a controller component for mediation between data (model) and its representation (view) in the Model-view-controller pattern.
Information Expert

Information Expert (also Expert or the Expert Principle) is a principle used to determine where to delegate responsibilities. These responsibilities include methods, computed fields, and so on.

Using the principle of Information Expert, a general approach to assigning responsibilities is to look at a given responsibility, determine the information needed to fulfill it, and then determine where that information is stored.

Information Expert will lead to placing the responsibility on the class with the most information required to fulfill it.Low Coupling

Low Coupling is an evaluative pattern, which dictates how to assign responsibilities to support:

lower dependency between the classes,
change in one class having lower impact on other classes,
higher reuse potential.

Polymorphism

According to Polymorphism, responsibility of defining the variation of behaviors based on type is assigned to the types for which this variation happens. This is achieved using polymorphic operations.

The Protected Variations pattern protects elements from the variations on other elements (objects, systems, subsystems) by wrapping the focus of instability with an interface and using polymorphism to create various implementations of this interface.

A Pure Fabrication is a class that does not represent a concept in the problem domain, specially made up to achieve low coupling, high cohesion, and the reuse potential thereof derived (when a solution presented by the Information Expert pattern does not). This kind of class is called "Service" in Domain-driven design.

Information expert:

GRASP Patterns

–Information Expert
Name:Information Expert
Problem:What is a general principle of assigning responsibility to objects?
Solution: Assign a responsibility to the class that has the information needed to fulfill i**t**

Example

In the NextGen application, some class needs to know the grand total of a sale

Discussion
•Information Expert is frequently used in the assignment of responsibilities; it is a basic guiding principles used continuously in object design.

•The fulfillment of a responsibility often requires information spread across different classes of objects.

• Expert usually leads to designs where a software object does those operations that are normally done to the inanimate real
-world thing it represents.

•The Information Expert pattern
–like many things in OO
– has a real
- world analogy.

Contraindications
• There are situations where a solution suggested by Expert is undesirable, usually because of problems in coupling and cohesion
•Ex. who should be responsible for saving a Sale in a database?

LOW COUPLING:

Name : Low Coupling
Problem : How to support low dependency, low change impact, and increased reuse?
Solution : Assign a responsibility so that coupling remains low

Example
Assume a need to create a Payment instance and associate it with the Sale, who should be responsible for this?

Discussion

• Low Coupling is a principle to keep in mind during all design decisions; it is an underlying goal to continually consider.

• Although, in general, a low coupling is preferred, it shouldn't be taken to excess. Some moderate degree of coupling between classes is normal and necessary to create an OO system in which tasks are fulfilled by a collaboration between connected objects Contradictions

•High coupling to stable elements and to pervasive elements is seldom a problem.

Ex. a Java application can safely couple itself to the Java libraries
(java.util, and so on) because they are stable and widespread

Benefits
• not affected by changes in other components
• simple to understand in isolation
• convenient to reuse

choices:
– Represents the overall “system”, a device, or a subsystem (façade controller)
– Represents a use case scenario (use case controller)

• Controller is a non
- user interface object responsible for receiving or handling a system event.
• A controller defines the method for the system operations System

Discussion
• Systems receive externally input events, typically involving a GUI operated by a person.
• In all cases, some handler for these events must be chosen
• The Controller pattern provides guidance for generally accepted, suitable choices.
• Normally, a controller should delegate to other objects the work that needs to be done; it coordinates or controls the activity. It doesn’t do much work itself.

SIGNIFICANCE:

• Increased potential for reuse and pluggable interfaces
• Opportunity to reason about the state of the use case Implementation
• Implementation Java Swing: Rich Client UI
• Implementation with Java Struts: Client Browser and Web UI Issues and Solutions: Bloated Controllers
• Poorly designed, a controller will have low cohesion
– unfocused and handling too many areas of responsibility
• Signs of bloating

GRASP – HIGH COHESION

• Concept – Cohesion:
  – Cohesion is a measure of “relatedness”.
  – High Cohesion says elements are strongly related to one another.
  – Low Cohesion says elements are not strongly related to one another. Ex:

• System level: ATM with a use case (function) called “Teller Reports”.
• Class level: A Student class with a method called “getDrivingRecord().
• Method level: Methods with the word “And” or “Or” in them.
• Also applies to subsystem (package) level, component level, etc.Designs with low cohesion are

difficult to maintain and reuse.
– One of the fundamental goals of an effective design is to achieve higH cohesion with low coupling (which we will see later)

Problem: How do you keep complexity manageable?
•Solution: Assign responsibility so that cohesion remains high.
• Mechanics: Look for classes with too-few or disconnected methods. Look for methods that do to much (hint: method name)Rework your design as needed

STRUCTURAL BRIDGE:

The bridge pattern is a design pattern used in software engineering which is meant to "decouple an abstraction from its implementation so that the two can vary independently".[1] The bridge uses encapsulation, aggregation, and can use inheritance to separate responsibilities into different classes.

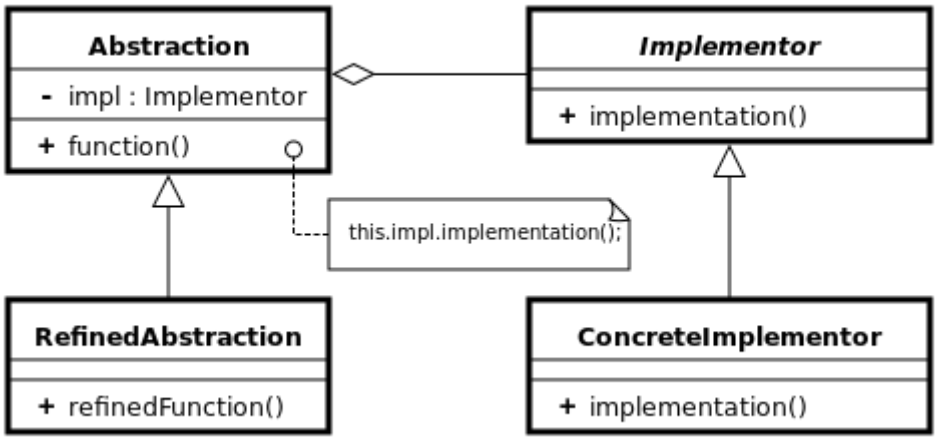
When a class varies often, the features of object-oriented programming become very useful because changes to a program's code can be made easily with minimal prior knowledge about the program.

The bridge pattern is useful when both the class and what it does vary often. The class itself can be thought of as the implementation and what the class can do as the abstraction. The bridge pattern can also be thought of as two layers of abstraction.

When there is only one fixed implementation, this pattern is known as the Pimpl idiom in the C++ world.

The bridge pattern is often confused with the adapter pattern. In fact, the bridge pattern is often implemented using the class adapter pattern, e.g. in the Java code below.

Variant: The implementation can be decoupled even more by deferring the presence of the implementation to the point where the abstraction is utilized.



Abstraction (abstract class) defines the abstract interface maintains the Implementor reference.
RefinedAbstraction (normal class) extends the interface defined by Abstraction
Implementor (interface)defines the interface for implementation classes
ConcreteImplementor (normal class)implements the Implementor interface

ADAPTER and STRUCTURE:

Intent

Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces. [GoF, p139]
Wrap an existing class with a new interface.
Impedance match an old component to a new system

Problem

An "off the shelf" component offers compelling functionality that you would like to reuse, but its "view of the world" is not compatible with the philosophy and architecture of the system currently being developed.

Structure category: wrapper
Similar patterns: Facade Proxy

Discussion

Reuse has always been painful and elusive. One reason has been the tribulation of designing something new, while reusing something old. There is always something not quite right between the old and the new. It may be physical dimensions or misalignment. It may be timing or synchronization. It may be unfortunate assumptions or competing standards.

It is very similar to the electrical engineering activity of "impedance matching" – adapting the input resistance, inductance, and capacitance of a load to match the output impedance of a source.

STRUCTURE:

Below, a legacy Rectangle component's display() method expects to receive "x, y, w, h" parameters. But the client wants to pass "upper left x and y" and "lower right x and y". This incongruity can be reconciled by adding an additional level of indirection – i.e. an Adapter object.

BEHAVIORAL – STRATEGY:

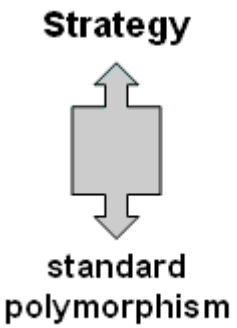
A generic value of the software community for years has been, "maximize cohesion and minimize coupling". The object-oriented design approach shown in Figure 21-1 is all about minimizing coupling. Since the client is coupled only to an abstraction (i.e. a useful fiction), and not a particular realization of that abstraction, the client could be said to be practicing "abstract coupling" . an object-oriented variant of the more generic exhortation "minimize coupling".

A more popular characterization of this "abstract coupling" principle is ...

Program to an interface, not an implementation.

Clients should prefer the "additional level of indirection" that an interface (or an abstract base class) affords. The interface captures the abstraction (i.e. the "useful fiction") the client wants to exercise, and the implementations of that interface are effectively hidden.

The Strategy design pattern discusses and documents the leverage represented in Figure 21-1. Other terms that are part and parcel of this same concept are: polymorphism, and dynamic binding.

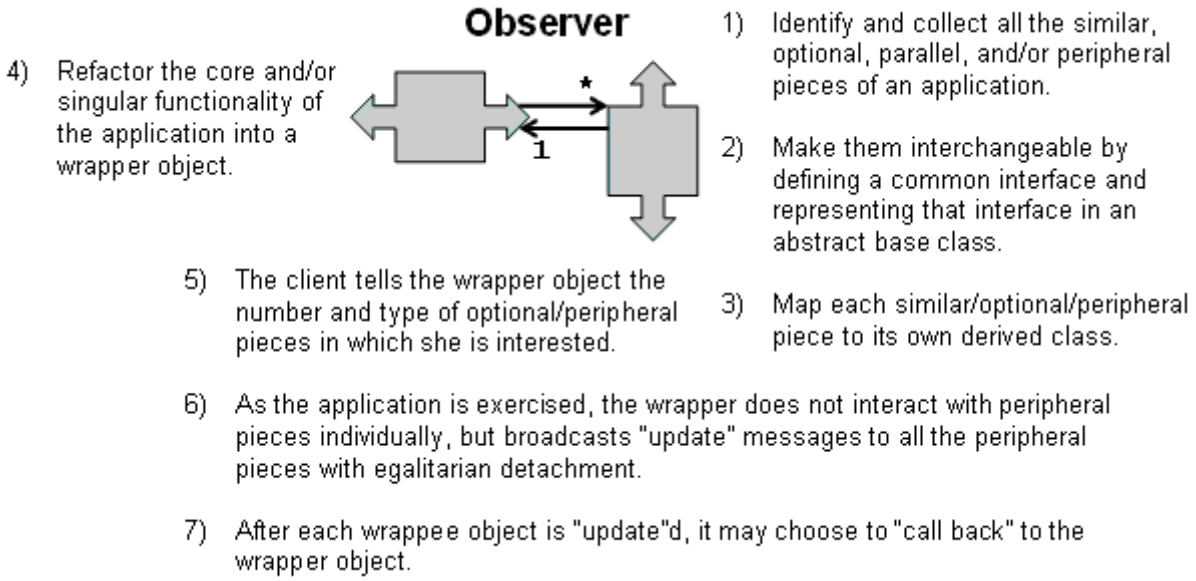


1) Elevate the interface of an algorithm to an abstract base class.
2) Bury each possible implementation in its own derived class.
3) The client couples herself to the interface, **not** to any particular implementation.

Example:

"A Strategy defines a set of algorithms that can be used interchangeably. Modes of transportation to an airport is an example of a Strategy. Several options exist such as driving one's own car, taking a taxi, an airport shuttle, a city bus, or a limousine service. For some airports, subways and helicopters are also available as a mode of transportation to the airport. Any of these modes of transportation will get a traveler to the airport, and they can be used interchangeably. The traveler must chose the Strategy based on tradeoffs between cost, convenience, and time."

OBSERVER:



Discussion:

Define an object that is the "keeper" of the data model and/or business logic (the Subject). Delegate all "view" functionality to decoupled and distinct Observer objects. Observers register themselves with the Subject as they are created.

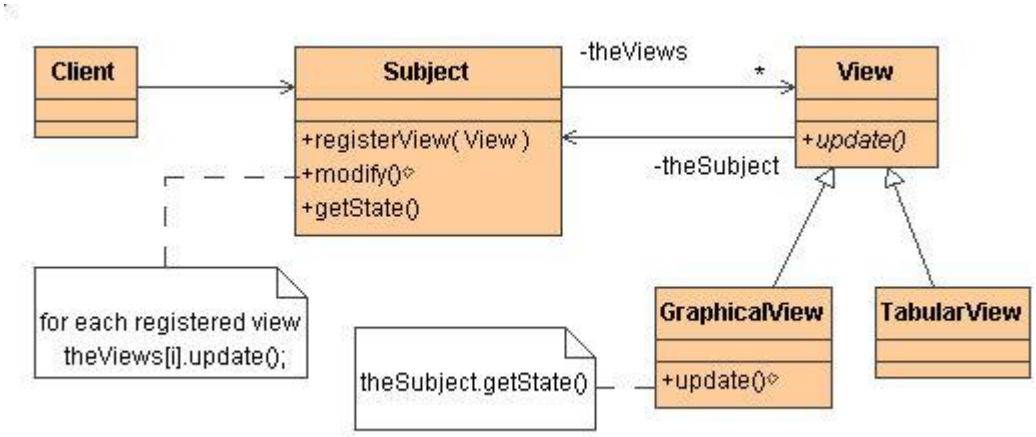
Whenever the Subject changes, it broadcasts to all registered Observers that it has changed, and each Observer queries the Subject for that subset of the Subject's state that it is responsible for monitoring.

This allows the number and "type" of "view" objects to be configured dynamically, instead of being statically specified at compile-time.

The protocol described above specifies a "pull" interaction model. Instead of the Subject "pushing" what has changed to all Observers, each Observer is responsible for "pulling" its particular "window of interest" from the Subject. The "push" model compromises reuse, while the "pull" model is less efficient.

Issues that are discussed, but left to the discretion of the designer, include: implementing event compression (only sending a single change broadcast after a series of consecutive changes has occurred), having a single Observer monitoring multiple Subjects, and ensuring that a Subject notify its Observers when it is about to go away.

The Observer pattern captures the lion's share of the Model-View-Controller architecture that has been a part of the Smalltalk community for years.



SIGNIFICANCE:

What Are the Outputs?
•Modeling for the difficult part of the design that we wished to explore before coding
• Specifically for object design, UML
– Interaction diagrams
–class diagrams
–package diagrams
•UI sketches and prototypes
•Database models Report sketches and prototypes

APPLICATION AREA:

Dynamic and static modeling (draw both interaction and complementary class diagrams)
– Applying various OOD principles
• GRASP
• GoF design pattersn
•Responsibility
-Driven Design

GLOSSARY:

OOD-Object Oriented Design
OOA-Object Oriented Design
UP-Unified Process

REFERENCE:

1. Craig Larman, “Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development”, Third Edition, Pearson Education, 2005.

UNIT III

CASE STUDY

PRE-REQUSITE DISCUSSION:

Case Study Focus
• Applications generally can be divided into 3 layers
– User interface
– Application logic
– Other components/layers

CONCEPT:

Case study – the Next Gen POS system, Inception :

USECASE MODELING:

The Use Case Model describes the proposed functionality of the new system. A Use Case represents a discrete unit of interaction between a user (human or machine) and the system. A Use Case is a single unit of meaningful work; for example login to system, register with system and create order are all Use Cases. Each Use Case has a description which describes the functionality that will be built in the proposed system. A Use Case may 'include' another Use Case's functionality or 'extend' another Use Case with its own behavior. Use Cases are typically related to 'actors'. An actor is a human or machine entity that interacts with the system to perform meaningful work.
Actors

An Actor is a user of the system. This includes both human users and other computer systems. An Actor uses a Use Case to perform some piece of work which is of value to the business.
The set of Use Cases an actor has access to defines their overall role in the system and the scope of their action.

Constraints, Requirements and Scenarios

The formal specification of a Use Case includes:

1. Requirements:

These are the formal functional requirements that a Use Case must provide to the end user. They correspond to the functional specifications found in structured methodologies. A requirement is a contract that the Use Case will perform some action or provide some value to the system.

2. Constraints:

These are the formal rules and limitations that a Use Case operates under, and includes pre- post- and invariant conditions. A pre-condition specifies what must have already occurred or be in place before the Use Case may start. A post-condition documents what will be true once the Use Case is complete. An invariant specifies what will be true throughout the time the Use Case operates.

3.Scenarios:

Scenarios are formal descriptions of the flow of events that occurs during a Use Case

instance. These are usually described in text and correspond to a textual representation of the Sequence Diagram.

USE CASE RELATIONSHIPS.

Use case relationships is divided into three types

1. Include relationship
2. Extend relationship
3. Generalization

1. Include relationship:

It is common to have some practical behavior that is common across several use cases. It is simply to underline it or highlight it in some fashion

Example:

Paying by credit: Include Handle Credit Payment

2. Extend relationship:

Extending the use case or adding new use case to the process Extending use case is triggered by some conditions called extension point.

3. Generalization:

Complicated work and unproductive time is spending in this use case relationship. Use case experts are successfully doing use case work without this relationship.

Includes and Extends relationships between Use Cases

One Use Case may include the functionality of another as part of its normal processing. Generally, it is assumed that the included Use Case will be called every time the basic path is run. An example may be to list a set of customer orders to choose from before modifying a selected order in this case the <list orders> Use Case may be ncluded every time the <modify order> Use Case is run. A Use Case may be included by one or more Use Cases, so it helps to reduce duplication of functionality by factoring out common behavior into Use Cases that are re-used many times. One Use Case may extend the behavior of another - typically when exceptional circumstances are encountered.

Relationships Between Use Cases

Use cases could be organized using following relationships:

Generalization
Association
Extend
Include

Generalization Between Use Cases

Generalization between use cases is similar to generalization between classes; child use case inherits properties and behavior of the parent use case and may override the behavior of the parent.

NOTATION:

Generalization is rendered as a solid directed line with a large open arrowhead (same as generalization between classes).

Generalization between use cases

Association Between Use Cases

Use cases can only be involved in binary Associations. Two use cases specifying the same subject cannot be associated since each of them individually describes a complete usage of the system.

Extend Relationship

Extend is a directed relationship from an extending use case to an extended use case that specifies how and when the behavior defined in usually supplementary (optional) extending use case can be inserted into the behavior defined in the use case to be extended.

The extension takes place at one or more extension points defined in the extended use case.

The extend relationship is owned by the extending use case. The same extending use case can extend more than one use case, and extending use case may itself be extended.

Extend relationship between use cases is shown by a dashed arrow with an open arrowhead from the extending use case to the extended (base) use case. The arrow is labeled with the keyword Registration use case is meaningful on its own, and it could be extended with optional Get Help On Registration use case. The condition of the extend relationship as well as the references to the extension points are optionally shown in a Note attached to the corresponding extend relationship.

Registration use case is conditionally extended by Get Help On Registration use case in extension point Registration Help

Include Relationship

An include relationship is a directed relationship between two use cases, implying that the behavior of the required (not optional) included use case is inserted into the behavior of the including (base) use case. Including use case depends on the addition of the included use case. The include relationship is intended to be used when there are common parts of the behavior of two or more use cases. This common part is extracted into a separate use case to be included by all the base use cases having this part in common. As the primary use of the include relationship is to reuse common parts, including use cases are usually not complete by themselves but dependent on the included use cases.

Include relationship between use cases is shown by a dashed arrow with an open arrowhead from the including (base) use case to the included (common part) use case. The arrow is labeled with the keyword «include».

ELABORATION-DOMAIN MODELS, DOMAIN MODEL HIERARCHY

A domain model captures the most important types of objects in the context of the business. The domain model represents the ‘things’ that exist or events that transpire in the business environment.”

Gives a conceptual framework of the things in the problem space

Helps you think – focus on semantics

Provides a glossary of terms – noun based

It is a static view - meaning it allows us convey time invariant business rules

Foundation for use case/workflow modelling

Based on the defined structure, we can describe the state of the problem domain at any time.

Features of a domain model

The following features enable us to express time invariant static business rules for a domain:-

Domain classes – each domain class denotes a type of object.

Attributes – an attribute is the description of a named slot of a specified type in a domain class; each instance of the class separately holds a value.

Associations – an association is a relationship between two (or more) domain classes that describes links between their object instances. Associations can have roles, describing the multiplicity and participation of a class in the relationship.

Additional rules – complex rules that cannot be shown with symbology can be shown with attached notes

FINDING CONCEPTUAL CLASSES AND DESCRIPTION CLASSES:

A class diagram in the Unified Modeling Language (UML) is a type of static structure diagram that describes the structure of a system by showing the system's classes, their attributes, operations (or methods), and the relationships among objects.

Contents

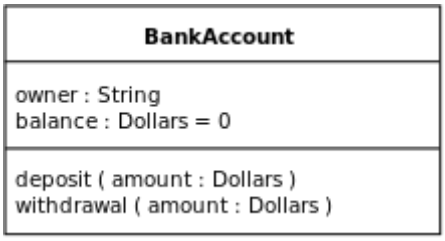
The class diagram is the main building block of object oriented modelling. It is used both for general conceptual modelling of the systematics of the application, and for detailed modelling translating the models into programming code. Class diagrams can also be used for data modeling.

The classes in a class diagram represent both the main objects, interactions in the application and the classes to be programmed.
A class with three sections.

In the diagram, classes are represented with boxes which contain three parts:

The top part contains the name of the class. It is printed in bold and centered, and the first letter is capitalized.
The middle part contains the attributes of the class. They are left-aligned and the first letter is lowercase.
The bottom part contains the methods the class can execute. They are also left-aligned and the first letter is lowercase.

In the design of a system, a number of classes are identified and grouped together in a class diagram which helps to determine the static relations between those objects. With detailed modelling, the classes of the conceptual design are often split into a number of subclasses.



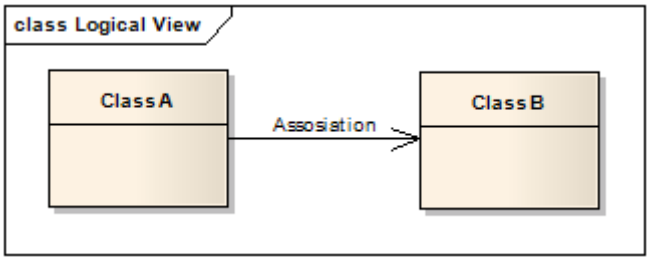
ASSOCIATIONS,ATTRIBUTES,AGGREGATION AND COMPOSITION:

UML CLASS DIAGRAM: ASSOCIATION, AGGREGATION AND COMPOSITION

The UML Class diagram is used to visually describe the problem domain in terms of types of object (classes) related to each other in different ways. There are three primary inter-object relationships: association, aggregation, and composition. Using the right relationship line is important for placing implicit restrictions on the visibility and propagation of changes to the related classes, matter which play major role in reducing system complexity.

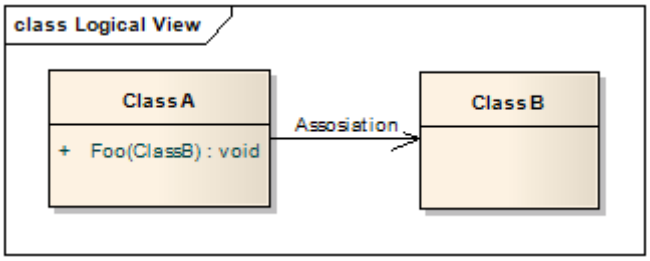
Association

The most abstract way to describe static relationship between classes is using the 'Association' link, which simply states that there is some kind of a link or a dependency between two classes or more.



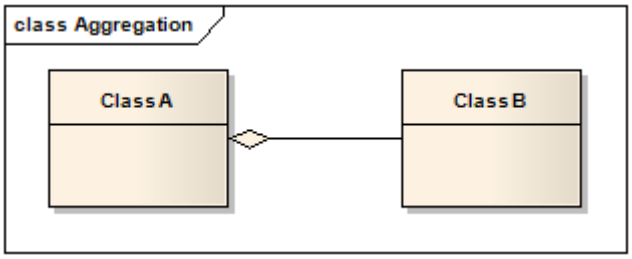
Weak Association

ClassA may be linked to ClassB in order to show that one of its methods includes parameter of ClassB instance, or returns instance of ClassB.

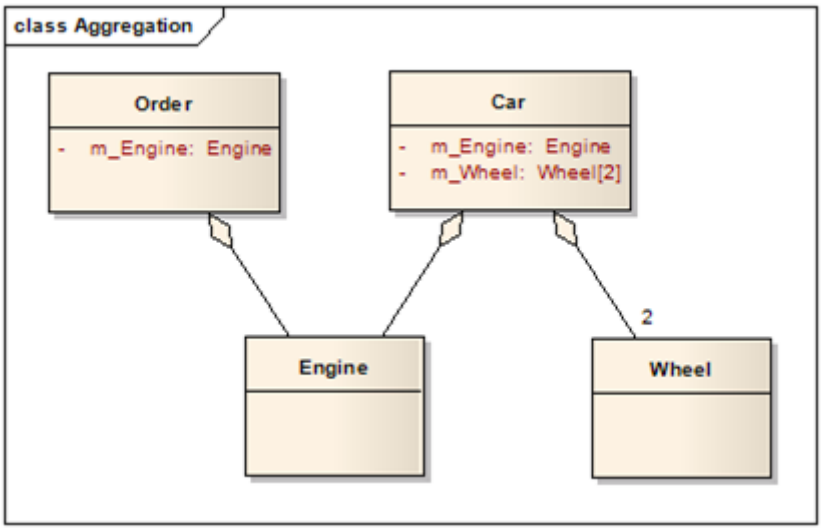


AGGREGATION (SHARED ASSOCIATION)

In cases where there's a part-of relationship between ClassA (whole) and ClassB (part), we can be more specific and use the aggregation link instead of the association link, taking special notice that ClassB can also be aggregated by other classes in the application (therefore aggregation is also known as shared association).

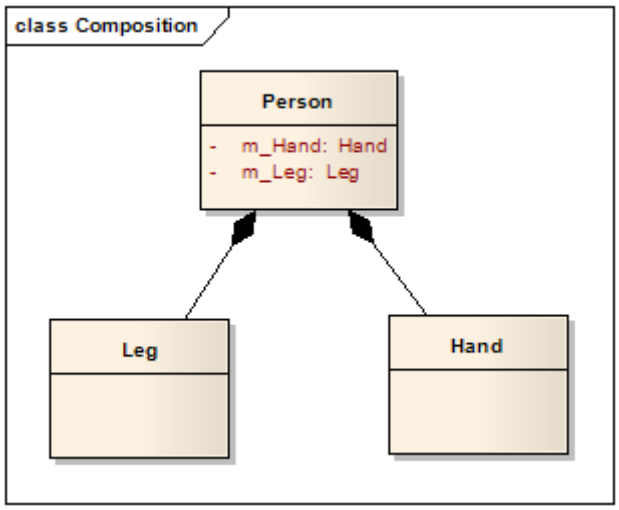


So basically, the aggregation link doesn't state in any way that ClassA owns ClassB nor that there is a parent-child relationship (when parent deleted all its child's are being deleted as a result) between the two. Actually, quite the opposite! The aggregation link usually used to stress the point that ClassA is not the exclusive container of ClassB, as in fact ClassB has another container.



COMPOSITION (NOT-SHARED ASSOCIATION)

In cases where in addition to the part-of relationship between ClassA and ClassB - there's a strong life cycle dependency between the two, meaning that when ClassA is deleted then ClassB is also deleted as a result, we should be more specific and use the composition link instead of the aggregation link or the association link.



The composition link shows that a class (container, whole) has exclusive ownership over other class/s (parts), meaning that the container object and its parts constitute a parent-child/s relationship.

Unlike association and aggregation, in the composition relationship, the composed class cannot appear as a return type or parameter type of the composite class, thus changes in the composed class cannot be propagated to the rest of the system. Consequently, usage of composition limits complexity growth as the system grows.

SIGNIFICANCE:

-Provides standard for software development.
- Reducing of costs to develop diagrams of UML using supporting tools.
- Development time is reduced.
- The past faced issues by the developers are no longer exists.
- Has large visual elements to construct and easy to follow.

APPLICATION AREA:

Web applications of UML can be used to model user interfaces of web applications and make the purpose of the website clear. Web applications are software-intensive systems and UML is among the efficient choice of languages for modeling them. Web software complexity of an application can be minimized using various UML tools.

REFERENCE:

1. Craig Larman, “Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development”, Third Edition, Pearson Education, 2005.
2. Simon Bennett, Steve Mc Robb and Ray Farmer, “Object Oriented Systems Analysis and Design Using UML”, Fourth Edition, Mc-Graw Hill Education, 2010.

UNIT IV

APPLYING DESIGN PATTERNS

PRE-REQUSITE DISCUSSION:

A Sequence diagram is an interaction diagram that shows how processes operate with one another and what is their order.

It is a construct of a Message Sequence Chart.

A sequence diagram shows object interactions arranged in time sequence. It depicts the objects and classes involved in the scenario and the sequence of messages exchanged between the objects needed to carry out the functionality of the scenario.

Sequence diagrams are typically associated with use case realizations in the Logical View of the system under development. Sequence diagrams are sometimes called event diagrams or event scenarios.

SYSTEM SEQUENCE DIAGRAMS:

If the lifeline is that of an object, it demonstrates a role. Leaving the instance name blank can represent anonymous and unnamed instances.

Messages, written with horizontal arrows with the message name written above them, display interaction. Solid arrow heads represent synchronous calls, open arrow heads represent asynchronous messages, and dashed lines represent reply messages.

If a caller sends a synchronous message, it must wait until the message is done, such as invoking a subroutine. If a caller sends an asynchronous message, it can continue processing and doesn’t have to wait for a response.

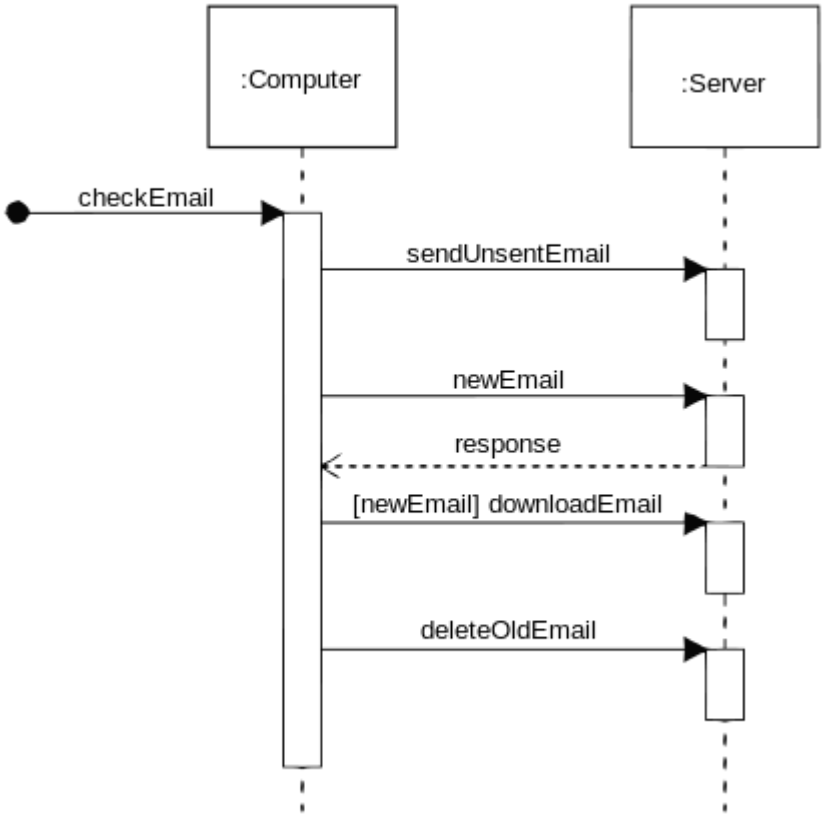
Asynchronous calls are present in multithreaded applications and in message-oriented middleware. Activation boxes, or method-call boxes, are opaque rectangles drawn on top of lifelines to represent that processes are being performed in response to the message (ExecutionSpecifications in UML).

Objects calling methods on themselves use messages and add new activation boxes on top of any others to indicate a further level of processing.

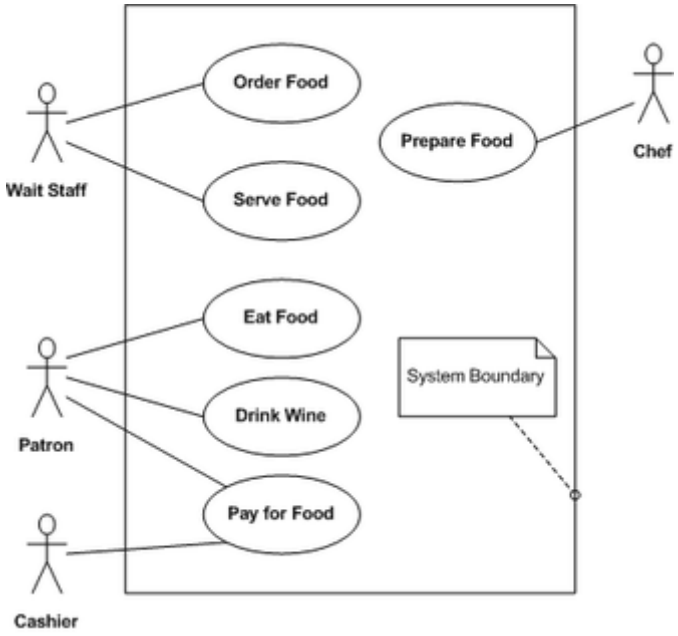
When an object is destroyed (removed from memory), an X is drawn on top of the lifeline, and the dashed line ceases to be drawn below it (this is not the case in the first example though). It should be the result of a message, either from the object itself, or another.

A message sent from outside the diagram can be represented by a message originating from a filled-in circle (found message in UML) or from a border of the sequence diagram (gate in UML).

UML has introduced significant improvements to the capabilities of sequence diagrams. Most of these improvements are based on the idea of interaction fragments[2] which represent smaller pieces of an enclosing interaction. Multiple interaction fragments are combined to create a variety of combined fragments,[3] which are then used to model interactions that include parallelism, conditional branches, optional interactions.



RELATIONSHIP BETWEEN SEQUENCE DIAGRAMS AND USE CASES:



use case diagram:

Realizing use cases by means of sequence diagrams is an important part of our analysis. It ensures that we have an accurate and complete class diagram.

The sequence diagrams increase the completeness and understandability of our analysis model. Often, analysts use the sequence diagram to assign responsibilities to classes. The behavior is associated with the class the first time it is required, and then the behavior is reused for every other use case that requires the behavior.

When assigning behaviors or responsibilities to a class while mapping a use case to the analysis model, you must take special care to assign the respon- sibility to the correct class. The responsibility or behavior belongs to the class if it is something you would do to the thing the class represents.

For example, if our class represented a table and our application must keep track of the physical location of the table, we would expect to find the method MOVE in the class. We also expect the object to maintain all the information associated with an object of a given type.

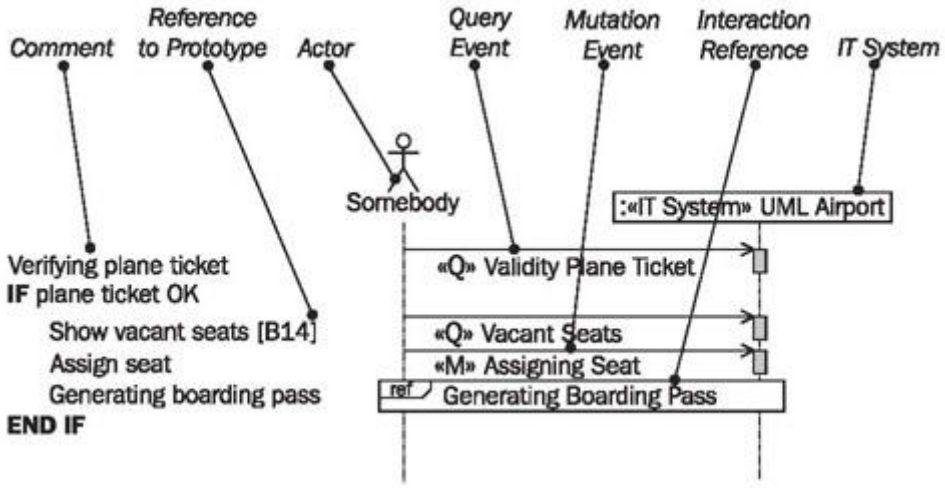
Therefore, the TABLE class should include the method WHERE LOCATED.

This simple rule of thumb states that a class will include any methods needed to provide information about an object of the given class, wilprovide necessary methods to manipulate the objects of the given class, and will be responsible for creating and deleting instances of the class.

Another guideline helpful in building an analysis model is to examine the model from the whole-part perspective.

The start of the sequence diagram is normally the most difficult aspect. It is important to have access to the object on which you will call a method to start the sequence. It is often the case that you will have to return to the whole to navigate through the whole-part relationship to arrive at the class you will be working with.

A simple example illustrates the whole-part relationship navigation. Suppose you were asked to read the first paragraph of three chapters of a book. First, you would need to know where to go to get the book. We might state that all books we are referring to are available at the Fourth St. library.



LOGICAL ARCHITECTURE AND UML PACKAGE DIAGRAM:

Logical Architecture and UML Package Diagrams...

• High level, large scale Architecture Model.

• At this level, the design of a typical OO system is based on several architectural layers, such as a UI layer, an application logic (or "domain") layer, and so forth.

• Goal is to design a logical architecture with layers and partitions using UML package diagrams

Logical Architecture And Layers

• Logical architecture is the large-scale organization of the software classes into packages (or namespaces), subsystems, and layers.

• Logical - because there's no decision about how these elements are deployed across different operating system processes or across physical computers in a network (deployment architecture)

Logical architecture is the large-scale organization of the software classes into packages (or namespaces), subsystems, and layers.

It's called the logical architecture because there's no decision about how these elements are deployed across different operating system processes or across physical computers in a network (these latter decisions are part of the deployment architecture)

LOGICAL ARCHITECTURE REFINEMENT :

UML CLASS DIAGRAM:

The class diagram is the main building block of object oriented modelling. It is used both for general conceptual modelling of the systematics of the application, and for detailed modelling translating the models into programming code. Class diagrams can also be used for data modeling.

The classes in a class diagram represent both the main objects, interactions in the application and the classes to be programmed.

The top part contains the name of the class. It is printed in bold and centered, and the first letter is capitalized.

The middle part contains the attributes of the class. They are left-aligned and the first letter is lowercase.

The bottom part contains the methods the class can execute. They are also left-aligned and the first letter is lowercase.

In the design of a system, a number of classes are identified and grouped together in a class diagram which helps to determine the static relations between those objects. With detailed modelling, the classes of the conceptual design are often split into a number of subclasses.

Links

A Link is the basic relationship among objects.

Association

An association represents a family of links. A binary association (with two ends) is normally represented as a line. An association can link any number of classes. An association with three links is called a ternary association. An association can be named, and the ends of an association can be adorned with role names, ownership indicators, multiplicity, visibility, and other properties.

There are four different types of association: bi-directional, uni-directional, Aggregation (includes Composition aggregation) and Reflexive. Bi-directional and uni-directional associations are the most common ones. For instance, a flight class is associated with a plane class bi-directionally. Association represents the static relationship shared among the objects of two classes.

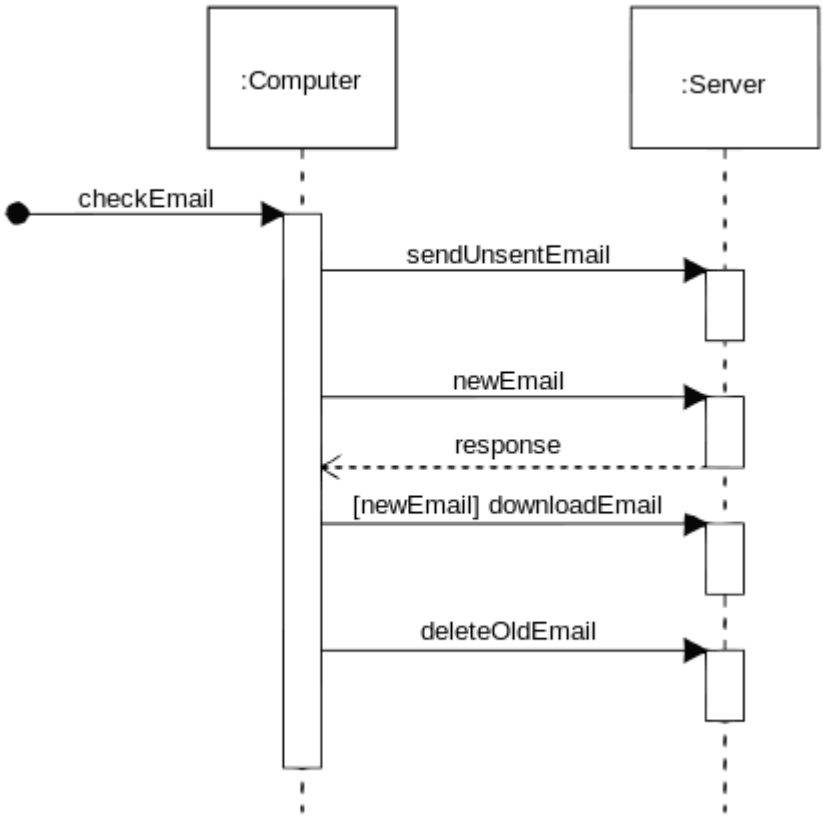
Aggregation

Aggregation is a variant of the "has a" association relationship; aggregation is more specific than association. It is an association that represents a part-whole or part-of relationship. As shown in the image, a Professor 'has a' class to teach. As a type of association, an aggregation can be named and have the same adornments that an association can. However, an aggregation may not involve more than two classes; it must be a binary association.

Aggregation can occur when a class is a collection or container of other classes, but the contained classes do not have a strong lifecycle dependency on the container. The contents of the container are not automatically destroyed when the container is.

In UML, it is graphically represented as a hollow diamond shape on the containing class with a single line that connects it to the contained class. The aggregate is semantically an extended object that is treated as a unit in many operations, although physically it is made of several lesser objects.

UML INTERACTION DIAGRAMS:



Interaction Overview Diagram is one of the thirteen types of diagrams of the Unified Modeling Language (UML), which can picture a control flow with nodes that can contain interaction diagrams.

The interaction overview diagram is similar to the activity diagram, in that both visualize a

sequence of activities.

The difference is that, for an interaction overview, each individual activity is pictured as a frame which can contain a nested interaction diagrams. This makes the interaction overview diagram useful to "deconstruct a complex scenario that would otherwise require multiple if-then-else paths to be illustrated as a single sequence diagram".

The other notation elements for interaction overview diagrams are the same as for activity diagrams.

These include initial, final, decision, merge, fork and join nodes. The two new elements in the interaction overview diagrams are the "interaction occurrences" and "interaction elements."

APPLYING GOF DESIGN PATTERNS:

Why Design Patterns?
● Apply well known and proven solutions
● many problems are not new
→ no need to invent wheels
●code structure easier to understand
→ easier maintainance
● great help for beginners to learn good practice
● patterns are not static, guide to individual solutions
● Analogies
● song styles, theatre pieces, novels, (architecure), engineering.

Design patterns help to translate “OOD rules”
●dependency management
● components
● code reuse
● ease of planned (and unplanned) changes
● maintainance
● code quality

Structured pattern description
●Pattern name
● one or twoword descriptive title
●Intent
●what happens? Why? Design issue or problem?
●Motivation
●example pattern application scenario
●Applicability
●when to use? What problems solved?

Structured pattern description

Participants and Collaborations
classes, objects, their roles and collaborations
Consequences and Implementation
results and tradeoffs, implementation tricks

Examples

code, projects
Related patterns relation to other patterns, combined uses

SIGNIFICANCE:

UML shows the future modeling where the entire applications are generated from high-level UML models and highlights the best practices for adopting UML in an enterprise.

APPLICATION AREA:

used in all designing feilds of systems

REFERENCE:

1. Craig Larman, “Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development”, Third Edition, Pearson Education, 2005.

UNIT V

CODING AND TESTING

PRE-REQUSITE DISCUSSION:

Object-oriented analysis and design (OOAD) is a popular technical approach to analyzing, designing an application, system, or business by applying the object-oriented paradigm and visual modeling throughout the development life cycles to foster better stakeholder communication and product quality.

According to the popular guide Unified Process, OOAD in modern software engineering is best conducted in an iterative and incremental way.

Iteration by iteration, the outputs of OOAD activities, analysis models for OOA and design models for OOD respectively, will be refined and evolve continuously driven by key factors like risks and business value.

CONCEPT:

MAPPING DESIGN TO CODE:

Write source code for:

Class and interface definitions

Method definitions

Work from OOA/D artifacts

Create class definitions for Domain Class Diagrams (DCDs)

Create methods from Interaction diagrams

Why is it wise to consider large-scale exception handling strategies during design modeling?

In UML, exceptions can be inserted as property strings of messages

OOD produced artifacts
• Interaction diagrams
• Design Class Diagrams
• UP

Implementation Model
• Source code
• Database definitions
• JSP / XML / HTML pages

Programming and Iterative, Evolutionary Development
•The creation of code in an OO programming language is not a part of OOAD, is an end goal
• The artifacts created in the UP Design Model provide some of the information necessary to generate the code
•Roadmap to software development
•Use case

- requirements
• OOAD

- logical solution, blueprints
• OOP

-running applications Creativity and Change During Implementation
• OOAD produces a base for the application
• Scales up with elegances
• Robustness
•Code Changes are inevitable
•CASE Tools, and Reverse

- Engineering
• Rational ROSE, or Borland Together

Testing: Issues in OO Testing:

Testing in an OO context must address the basics of testing a base class and the code that uses the base class. Factors that affect this testing are inheritance and dynamic binding.

Therefore, some systems are harder to test (e.g., systems with inheritance of

implementations harder than inheritance of interfaces) and OO constructs such as inheritance and dynamic binding may serve to hide faults

Use the OO Metrics techniques to assess the design and identify areas that need special attention in testing for adequate coverage. For example, depth of tree complicates testing and number of methods that have to be tested in a class and a lack of cohesion in methods means more tests to ensure there are no side effects among the disjoint methods.

OO Testing Strategies

While there are efforts underway to develop more automated testing processes from test models of the object model characteristics (for example states, data flows, or associations), testing is still based on the creation of test cases and test data by team members using a structural (White Box Testing) and/or a functional (SeeBlack Box Testing) strategy.

Overview of Object Orientated Unit Testing

Implications of Object Oriented Testing

Once a class is testing thoroughly it can be reused without being unit tested again

UML class state charts can help with selection of test cases for classes

Classes easily mirror units in traditional software testing

Objective of OO is to facilitate easy code reuse in the form of classes

To allow this each class has to be rigiriously unit tested

Due to classes potentially used in unforeseeable ways when composed in new systems

Example: A XML parser for a web browser

Classes must be created in a way promoting loose coupling and strong cohesion

CLASS TESTING:

Class testing is testing that ensures a class and its instances (objects) perform as defined. (Source: Object Testing Patterns).

When designing a society of classes, an excellent goal is to structure classes so they can be tested with a minimum of fuss. After all, a class that requires compiling the application,
starting the application, logging into the application, navigating to a particular point in the application ...will likely not get rigorously tested.

In computer programming, unit testing is a software testing method by which individual

units of source code, sets of one or more computer program modules together with associated control data, usage procedures, and operating procedures, are tested to determine whether they are fit for use.

Intuitively, one can view a unit as the smallest testable part of an application. In procedural programming, a unit could be an entire module, but it is more commonly an individual function or procedure.

In object-oriented programming, a unit is often an entire interface, such as a class, but could be an individual method.

Unit tests are short code fragments created by programmers or occasionally by white box testers during the development process. It forms the basis for component testing.

Ideally, each test case is independent from the others. Substitutes such as method stubs, mock objects,[5] fakes, and test harnesses can be used to assist testing a module in isolation. Unit tests are typically written and run by software developers to ensure that code meets its design and behaves as intended**.**

OO INTEGRATION TESTING:

Research confirms that testing methods proposed for procedural approach are not adequate for OO approach
Ex. Statement coverage

OO software testing poses additional problems due to the distinguishing characteristics of OO
Ex. Inheritance

Testing time for OO software found to be increased compared to testing procedural software

Typical OO softwarecharacteristics that impact testing ...

State dependent behavior
Encapsulation
Inheritance
Polymorphism and dynamic
binding
Abstract and generic classes
Exception handling
Procedural software
unit = single program, function, or procedure Object oriented software
unit = class
unit testing =intra-class testing
integration testing = inter-class testing

cluster of classes dealing with single methods separately is usually too expensive (complex scaffolding), so methods are usually tested in the context of the class they belong to.

GUI Testing – OO System Testing.

Overview

- This chapter discusses the testing of object-oriented systems. The process of testing object-oriented systems begins with a review of the object-oriented analysis and design models. Once the code is written object-oriented testing (OOT) begins by testing "in the small" with class testing (class operations and collaborations). As classes are integrated to become subsystems class collaboration problems are investigated. Finally, use-cases from the OOA model are used to uncover software validation errors.

- OOT similar to testing conventional software in that test cases are developed to exercise the classes, their collaborations, and behavior.

- OOT differs from conventional software testing in that more emphasis is placed assessing the completeness and consistency of the OOA and OOD models as they are built.

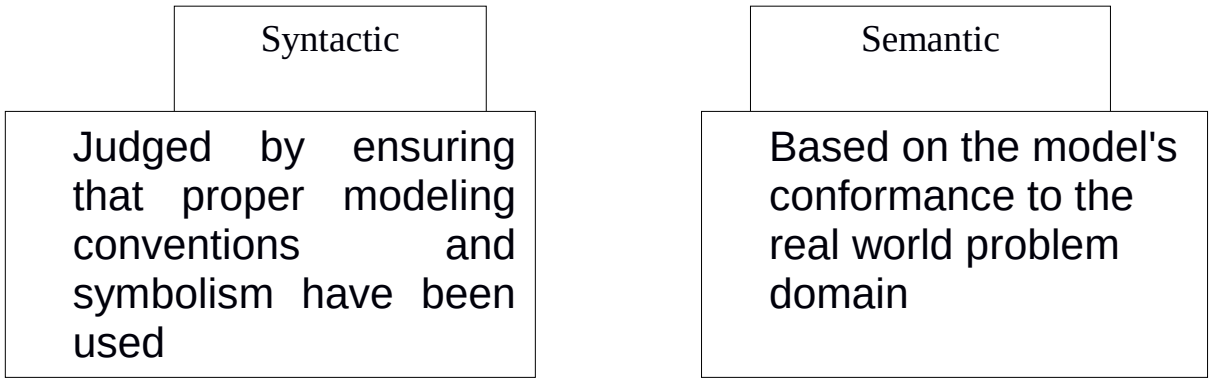
- OOT tends to focus more on integration problems than on unit testing.

Object-Oriented Testing Activities

- Review OOA and OOD models
- Class testing after code is written
- Integration testing within subsystems
- Integration testing as subsystems are added to the system
- Validation testing based on OOA use-cases

Testing OOA and OOD Models

- OOA and OOD cannot be tested but can review the correctness and consistency.
- Correctness of OOA and OOD models



- **Consistency of OOA and OOD Models**

  - Assess the class-responsibility-collaborator (CRC) model and object-relationship diagram

- Review system design (examine the object-behavior model to check mapping of system behavior to subsystems, review concurrency and task allocation, use use-case scenarios to exercise user interface design)
- Test object model against the object relationship network to ensure that all design object contain necessary attributes and operations needed to implement the collaborations defined for each CRC card
- Review detailed specifications of algorithms used to implement operations using conventional inspection techniques

- **Object-Oriented Testing Strategies**

  - **Unit testing** in the OO context

- Smallest testable unit is the encapsulated class or object
- Similar to system testing of conventional software
- Do not test operations in isolation from one another
- Driven by class operations and state behavior, not algorithmic detail and data flow across module interface
- Complete test coverage of a class involves

  - Testing all operations associated with an object
  - Setting and interrogating all object attributes
  - Exercising the object in all possible states

- Design of test for a class uses a verity of methods:

  - fault-based testing
  - random testing
  - partition testing

- each of these methods exercises the operations encapsulated by the class
- test sequences are designed to ensure that relevant operations are exercised
- state of the class (the values of its attributes) is examined to determine if errors exist

  - **Integration testing** in the OO context

    focuses on groups of classes that collaborate or communicate in some manner
    integration of operations one at a time into classes is often meaningless
    thread-based testing (testing all classes required to respond to one system input or event)
    use-based testing (begins by testing independent classes first and the dependent classes that make use of them)
    cluster testing (groups of collaborating classes are tested for interaction errors)
    regression testing is important as each thread, cluster, or subsystem is added to the system
    Levels of integration are less distinct in object-oriented systems

  - **Validation testing** in the OO context

- focuses on visible user actions and user recognizable outputs from the system

- validation tests are based on the use-case scenarios, the object-behavior model, and the event flow diagram created in the OOA model
- conventional black-box testing methods can be used to drive the validation tests

- **Test Case Design for OO Software**

  - Each test case should be uniquely identified and be explicitly associated with a class to be tested
  - State the purpose of each test
  - List the testing steps for each test including:

    - list of states to test for each object involved in the test
    - list of messages and operations to exercised as a consequence of the test
    - list of exceptions that may occur as the object is tested
    - list of external conditions needed to be changed for the test
    - supplementary information required to understand or implement the test

  - Testing Surface Structure and Deep Structure

    - Testing surface structure (exercising the structure observable by end-user, this often involves observing and interviewing users as they manipulate system objects)
    - Testing deep structure (exercising internal program structure - the dependencies, behaviors, and communications mechanisms established as part of the system and object design)

- **Testing Methods Applicable at The Class Level**

  Random testing - requires large numbers data permutations and combinations, and can be inefficient

    - Identify operations applicable to a class
    - Define constraints on their use
    - Identify a minimum test sequence
    - Generate a variety of random test sequences.

  Partition testing - reduces the number of test cases required to test a class

    - state-based partitioning - tests designed in way so that operations that cause state changes are tested separately from those that do not.
    - attribute-based partitioning - for each class attribute, operations are classified according to those that use the attribute, those that modify the attribute, and those that do not use or modify the attribute
    - category-based partitioning - operations are categorized according to the function they perform: initialization, computation, query, termination

- Fault-based testing

    - best reserved for operations and the class level

- uses the inheritance structure
- tester examines the OOA model and hypothesizes a set of plausible defects that may be encountered in operation calls and message connections and builds appropriate test cases
- misses incorrect specification and errors in subsystem interactions

❑ **Inter-Class Test Case Design**

- Test case design becomes more complicated as integration of the OO system begins – testing of collaboration between classes
- Multiple class testing
    - for each client class use the list of class operators to generate random test sequences that send messages to other server classes
    - for each message generated determine the collaborator class and the corresponding server object operator
    - for each server class operator (invoked by a client object message) determine the message it transmits
    - for each message, determine the next level of operators that are invoked and incorporate them into the test sequence
- Tests derived from behavior models
    - Use the state transition diagram (STD) as a model that represent the dynamic behavior of a class.
    - test cases must cover all states in the STD
    - breadth first traversal of the state model can be used (test one transition at a time and only make use of previously tested transitions when testing a new transition)
    - test cases can also be derived to ensure that all behaviors for the class have been adequately exercised

❑ **Testing Methods Applicable at Inter-Class Level**

<u>Cluster Testing</u>

Is concerned with integrating and testing clusters of cooperating objects
Identify clusters using knowledge of the operation of objects and the system features that are implemented by these clusters

- Approaches to Cluster Testing
    - Use-case or scenario testing
        - Testing is based on a user interactions with the system
        - Has the advantage that it tests system features as experienced by users
    - Thread testing – tests the systems response to events as processing threads through the system
    - Object interaction testing – tests sequences of object interactions that stop when an object operation does not call on services from another object

Use Case

Scenario-based Testing

- Based on
  - use cases
  - corresponding sequence diagrams
- Identify scenarios from use-cases and supplement these with interaction diagrams that show the objects involved in the scenario
- Concentrates on (functional) requirements
  - Every use case
  - Every fully expanded extension (<<extend>>) combination
  - Every fully expanded uses (<<uses>>) combination
  - Tests normal as well as exceptional behavior

- A scenario is a path through sequence diagram
- Many different scenarios may be associated with a sequence diagram
- using the user tasks described in the use-cases and building the test cases from the tasks and their variants
- uncovers errors that occur when any actor interacts with the OO software
- concentrates on what the use does, not what the product does
- you can get a higher return on your effort by spending more time on reviewing the use-cases as they are created, than spending more time on use-case testing

- **OO Test Design Issues**

- White-box testing methods can be applied to testing the code used to implement class operations, but not much else
- Black-box testing methods are appropriate for testing OO systems

- Object-oriented programming brings additional testing concerns

  - classes may contain operations that are inherited from super classes
  - subclasses may contain operations that were redefined rather than inherited
  - all classes derived from an previously tested base class need to be thoroughly tested

SIGNIFICANCE:

Extensions required for new variations are easy to add
• New implementations can be introduced without affecting clients
• Coupling is lowered
• The impact or cost of changes can be lowered

APPLICATION AREA:

- meets the requirements that guided its design and development,
- responds correctly to all kinds of inputs,
- performs its functions within an acceptable time,
- is sufficiently usable,

- can be installed and run in its intended environments, and
- achieves the general result its stakeholders desire.

REFERENCE:

1. Craig Larman, “Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development”, Third Edition, Pearson Education, 2005.

GLOSSARY:

CRC:class-responsibility-collaborator
DCD:Domain Class Diagrams
GUI: Graphical User Interface
OO:Object Oriented
OOA: Object Oriented analysis
OOD: Object Oriented Design
OOT:Object Oriented Testing
OOAD: Object Oriented Analysis and Design
STD:state transition diagram
UP: Unified Process