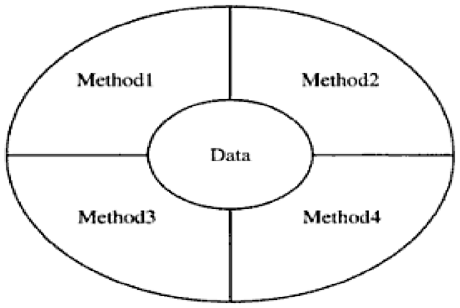
CS2353 OBJECT ORIENTED ANALYSIS AND DESIGN

UNIT 1

INTRODUCTION TO OBJECT ORIENTATION

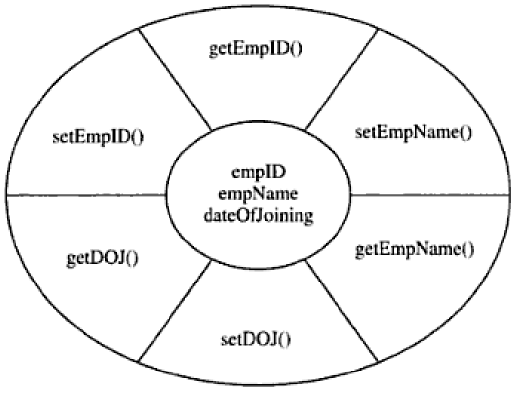
Object Orientation is a term used to describe the object – oriented(OO) method of building software. In an OO approach, the data is treated as the most important element and it cannot flow freely around the system. Restrictions are placed on the number of units that can manipulate the data. This approach binds the data and the methods that will manipulate the data closely and prevents the data from being inadvertently modified. The following figure shows the method1, method2, method3, and method4.

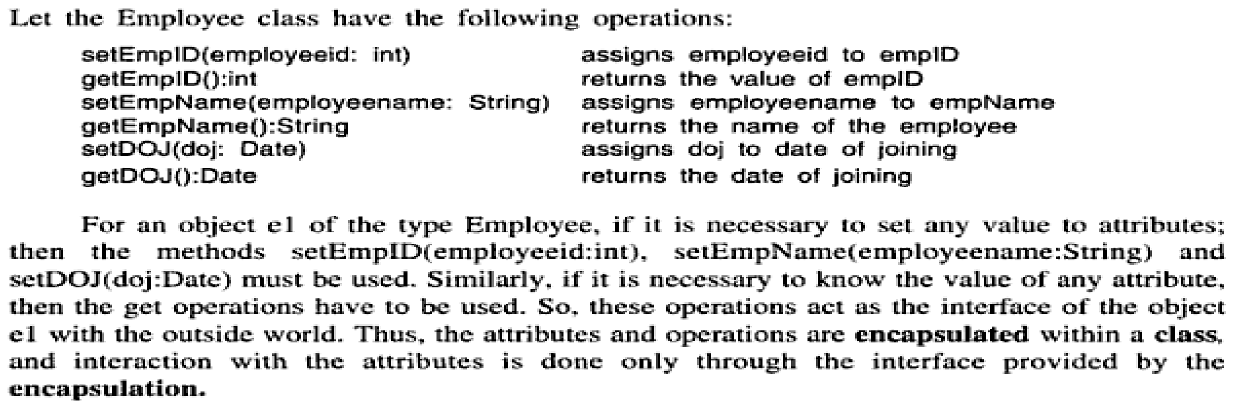


The 'object' forms the basis around which the following properties revolve:
1. Encapsulation
2. Abstraction, Implementation Hiding
3. Inheritance, dynamic binding, polymorphism
4. Overriding and overloading

Encapsulation:

In Object Orientation, a class is used as a unit to group related attributes and operations together. The outside world can interact with the data stored in the variables that represent the attributes of the class only through the operations of that class. Thus, the operations act as interfaces of the object of the class with the outside world.
For example, consider the class Employee with attributes empID, empName and dateOfJoining with is given below





Let the Employee class have the following operations:

setEmpID(employeeid: int)	assigns employeeid to empID
getEmpID():int	returns the value of empID
setEmpName(employeename: String)	assigns employeename to empName
getEmpName():String	returns the name of the employee
setDOJ(doj: Date)	assigns doj to date of joining
getDOJ():Date	returns the date of joining

For an object e1 of the type Employee, if it is necessary to set any value to attributes; then the methods setEmpID(employeeid:int), setEmpName(employeename:String) and setDOJ(doj:Date) must be used. Similarly, if it is necessary to know the value of any attribute, then the get operations have to be used. So, these operations act as the interface of the object e1 with the outside world. Thus, the attributes and operations are **encapsulated** within a **class**, and interaction with the attributes is done only through the interface provided by the **encapsulation.**

What is OOAD?

Object-oriented analysis and design (OOAD) is a software engineering approach that models a system as a group of interacting objects. Each object represents some entity of interest in the system being modeled, and is characterised by its class, its state (data elements), and its behavior. Various models can be created to show the static structure, dynamic behavior, and run-time deployment of these collaborating objects. There are a number of different notations for representing these models, such as the Unified Modeling Language (UML).
Object-oriented analysis (OOA) applies object-modelling techniques to analyze the functional requirements for a system. Object-oriented design (OOD) elaborates the analysis models to produce implementation specifications. OOA focuses on *what* the system does, OOD on *how* the system does it.

Object-oriented systems

An object-oriented system is composed of objects. The behavior of the system results from the collaboration of those objects. Collaboration between objects involves them sending messages to each other. Sending a message differs from calling a function in that when a target object receives a message, it itself decides what function to carry out to service that message. The same message may be implemented by many different functions, the one selected depending on the state of the target object.
The implementation of "message sending" varies depending on the architecture of the system being modeled, and the location of the objects being communicated with.

Object-oriented analysis

Object-oriented analysis (OOA) looks at the problem domain, with the aim of producing a conceptual model of the information that exists in the area being analyzed. Analysis models do not consider any implementation constraints that might exist, such as concurrency, distribution, persistence, or how the system is to be built. Implementation constraints are dealt during object-oriented design (OOD). Analysis is done before the Design[*citation needed*].
The sources for the analysis can be a written requirements statement, a formal vision document, interviews with stakeholders or other interested parties. A system may be divided into multiple domains, representing different business, technological, or other areas of interest, each of which are analyzed separately.
The result of object-oriented analysis is a description of *what* the system is functionally required to do, in the form of a conceptual model. That will typically be presented as a set of use cases, one or more UML class diagrams, and a number of interaction diagrams. It may also include some kind of user interface mock-up. The purpose of object oriented analysis is to develop a

model that describes computer software as it works to satisfy a set of customer defined requirements.

Object-oriented design
Object-oriented design (OOD) transforms the conceptual model produced in object-oriented analysis to take account of the constraints imposed by the chosen architecture and any non-functional – technological or environmental – constraints, such as transaction throughput, response time, run-time platform, development environment, or programming language.
The concepts in the analysis model are mapped onto implementation classes and interfaces. The result is a model of the solution domain, a detailed description of *how* the system is to be built.

What is UML?

Unified Modelling Language (UML) is the set of notations,models and diagrams used when developing object-oriented (OO) systems.

UML is the industry standard OO visual modelling language. The latest version is UML 1.4 and was formed from the coming together of three leading software methodologists; Booch, Jacobson and Rumbaugh.

UML allows the analyst ways of describing structure, behaviour of significant parts of system and their relationships.

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.

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 are often filled with bugs, and it can take programmers weeks to find and fix them. This is time that has been wasted, since an approach could have been used which would have reduced the number of bugs before the application was completed.

However, it should be emphasized that UML is not limited simply modeling software. It can also be used to build models for system engineering, business processes, and organization structures. A special language called Systems Modeling Language was designed to handle systems which were defined within UML 2.0. The Unified Modeling Language is important for a number of reasons. First, it has been used as a catalyst for the advancement of technologies which are model driven, and some of these include Model Driven Development and Model Driven Architecture.

Because an emphasis has been placed on the importance of graphics notation, UML is proficient in meeting this demand, and it can be used to represent behaviors, classes, and aggregation. While software developers were forced to deal with more rudimentary issues in the past, languages like UML have now allowed them to focus on the structure and design of their software programs. It should also be noted that UML models can be transformed into various other representations, often without a great deal of effort. One example of this is the ability to transform UML models into Java representations.

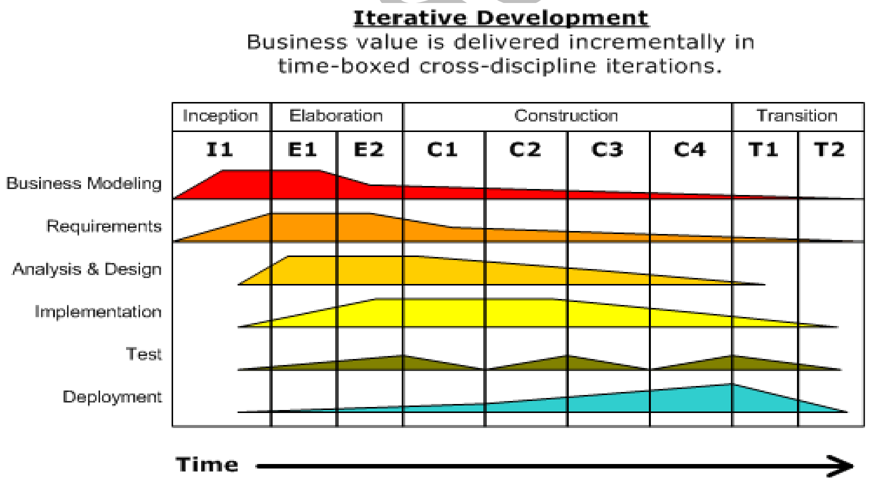
This transformation can be accomplished through a transformation language that is similar to QVT. Many of these languages may be supported by OMG. The Unified Modeling Language has a number of features and characteristics which separate it from other languages within the same category. Many of these attributes have allowed it to be useful for developers. In this article, I intend to show you many of these attributes, and you will then understand why the Unified Modeling Language is one of the most powerful languages in existence today.

Unified 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).
Profile of a typical project showing the relative sizes of the four phases of the Unified Process.

Overview

The Unified Process is not simply a process, but rather an extensible framework which should be customized for specific organizations or projects. The *Rational Unified Process* is, similarly, a customizable framework. As a result it is often impossible to say whether a refinement of the process was derived from UP or from RUP, and so the names tend to be used interchangeably.
The name *Unified Process* as opposed to *Rational Unified Process* is generally used to describe the generic process, including those elements which are common to most refinements. The *Unified Process* name is also used to avoid potential issues of trademark infringement since *Rational Unified Process* and *RUP* are trademarks of IBM. The first book to describe the process was titled *The Unified Software Development Process* and published in 1999 by Ivar Jacobson, Grady Booch and James Rumbaugh. Since then various authors unaffiliated with Rational Software have published books and articles using the name *Unified Process*, whereas authors affiliated with Rational Software have favored the name *Rational Unified Process*.



Unified Process Characteristics

Iterative and Incremental

The Unified Process is an iterative and incremental development process. The Elaboration, Construction and Transition phases are divided into a series of timeboxed iterations. (The Inception phase may also be divided into iterations for a large project.) Each iteration results in

an *increment*, which is a release of the system that contains added or improved functionality compared with the previous release.
Although most iterations will include work in most of the process disciplines (*e.g.* Requirements, Design, Implementation, Testing) the relative effort and emphasis will change over the course of the project.

Use Case Driven

In the Unified Process, use cases are used to capture the functional requirements and to define the contents of the iterations. Each iteration takes a set of use cases or scenarios from requirements all the way through implementation, test and deployment.

Architecture Centric

The Unified Process insists that architecture sit at the heart of the project team's efforts to shape the system. Since no single model is sufficient to cover all aspects of a system, the Unified Process supports multiple architectural models and views.
One of the most important deliverables of the process is the executable architecture baseline which is created during the Elaboration phase. This partial implementation of the system serves and validate the architecture and act as a foundation for remaining development.

Risk Focused

The Unified Process requires the project team to focus on addressing the most critical risks early in the project life cycle. The deliverables of each iteration, especially in the Elaboration phase, must be selected in order to ensure that the greatest risks are addressed first.

Project Lifecycle

The Unified Process divides the project into four phases:

- Inception
- Elaboration
- Construction
- Transition

Inception Phase

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.

Elaboration Phase

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).
The architecture is validated primarily through the implementation of an Executable Architecture Baseline. This is a partial implementation of the system which includes the core, most architecturally significant, components. It is built in a series of small, timeboxed iterations. By the end of the Elaboration phase the system architecture must have stabilized and the executable

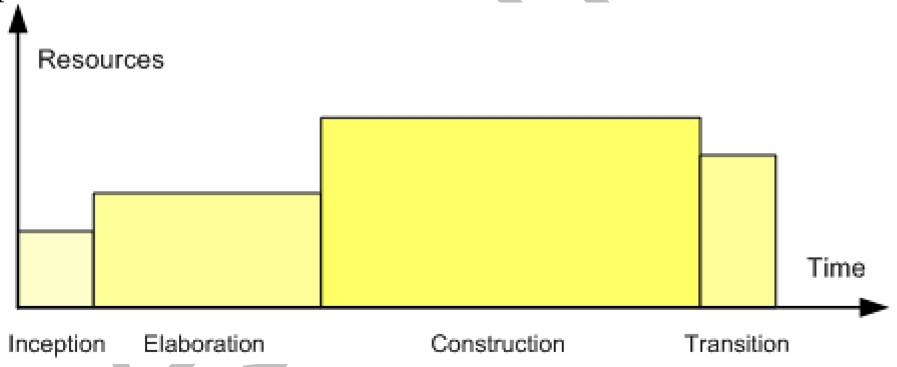
architecture baseline must demonstrate that the architecture will support the key system functionality and exhibit the right behavior in terms of performance, scalability and cost.
The final Elaboration phase deliverable is a plan (including cost and schedule estimates) for the Construction phase. At this point the plan should be accurate and credible, since it should be based on the Elaboration phase experience and since significant risk factors should have been addressed during the Elaboration phase.
The Lifecycle Architecture Milestone marks the end of the Elaboration phase.

Construction Phase

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.

Transition Phase

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.



<u>Case Study: NextPOS System</u>

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).

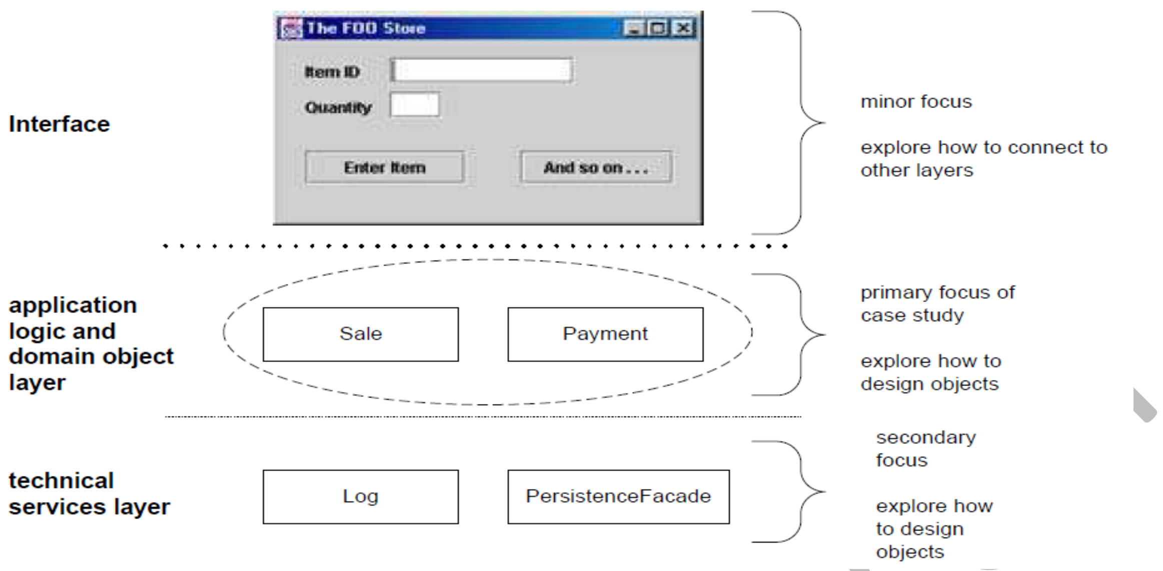
A POS system increasingly must support multiple and varied client-side terminals and interfaces. These include a thin-client Web browser terminal, a regular personal computer with something like a Java Swing graphical user interface, touch screen input, wireless PDAs, and so forth.

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 (see Figure 3.1). 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.


Interface
The FOO Store
Item ID
Quantity
Enter Item
And so on . . .
minor focus
explore how to connect to other layers
application logic and domain object layer
Sale
Payment
primary focus of case study
explore how to design objects
technical services layer
Log
PersistenceFacade
secondary focus
explore how to design objects

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 proposed system.
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 what are considered to be the areas of high risk. The prototyping effort during inception should be limited to gaining confidence that a solution is possible - the solution is realized during elaboration and construction.

Preparing the environment for the project, assessing the project and the organization, selecting tools, deciding which parts of the process to improve.

Milestone

The Lifecycle Objectives Milestone evaluates the basic viability of the project.

Tailoring Decisions

The example iteration workflow shown at the top of this page represents a typical Inception iteration in medium sized projects. The Sample Iteration Plan for Inception represents a different perspective of the breakdown of activities to undertake in an Inception iteration. This iteration plan is more complete in terms of workflow details and activities, and as such, more suitable for large projects. Small projects might decide to do only a subset of these workflow details, deviations should be challenged and documented as part of the project-specific process.

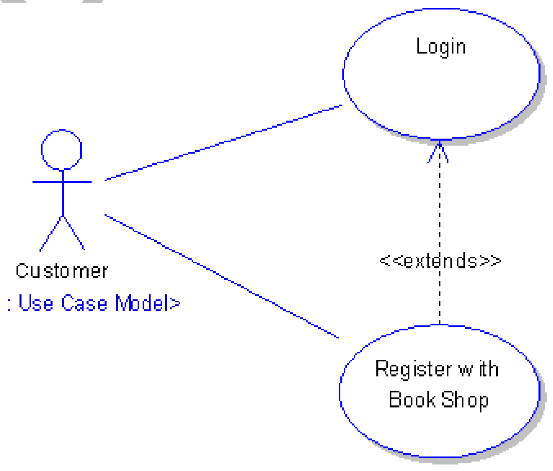
Inception Phase includes:

- Refining the scope of the project
- Project planning
- Risk identification and analysis
- Preparing the project environment
- Estimating the Budget

The Use Case Model

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 behaviour.

Use Cases are typically related to 'actors'. An actor is a human or machine entity that interacts with the system to perform meaningful work.

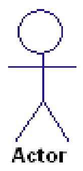


A Use Case description will generally include:

1. General comments and notes describing the use case;
2. Requirements - Things that the use case must allow the user to do, such as <ability to update order>, <ability to modify order> & etc.
3. Constraints- Rules about what can and can't be done. Includes i) pre-conditions that must be true before the use case is run -e.g. <create order> must precede <modify order>; ii) post-conditions that must be true once the use case is run e.g. <order is modified and consistent>; iii) invariants: these are always true - e.g. an order
4. Scenarios - Sequential descriptions of the steps taken to carry out the use case. May include multiple scenarios, to cater for exceptional circumstances and alternate processing paths;
5. Scenario diagrams -Sequence diagrams to depict the workflow - similar to (4) but graphically
6. Additional attributes such as implementation phase, version number, complexity rating,

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:

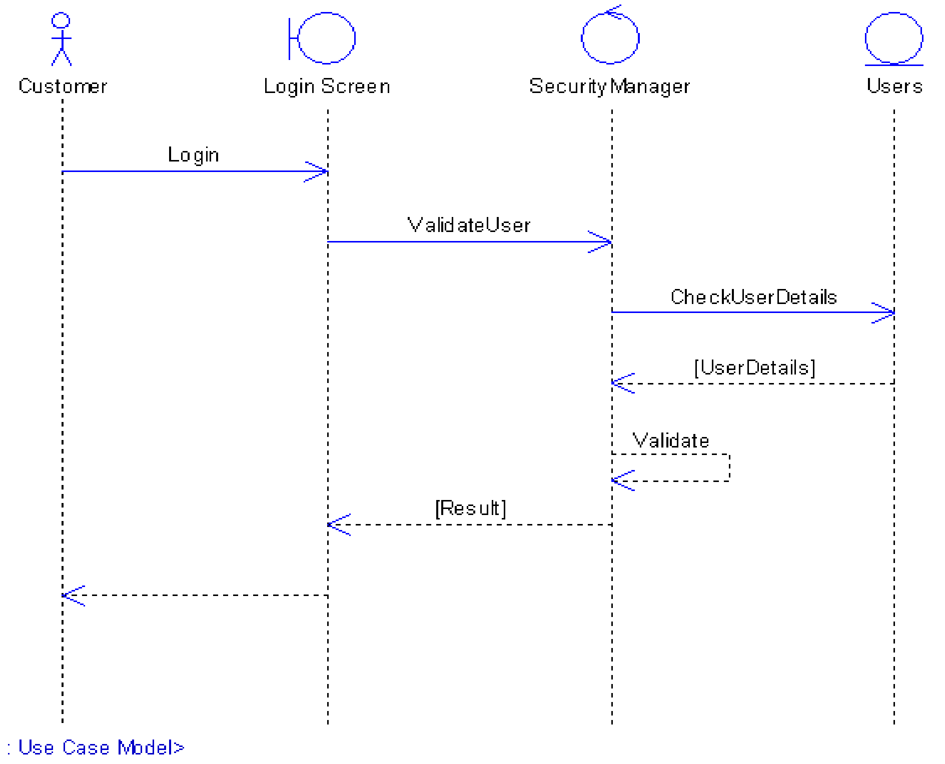
1. Requirements. These are the formal functional requirements that a Use Case must provide to the end user. They correspond to the functional specifications found in structured methodologies. A requirement is a contract that the Use Case will perform some action or provide some value to the system.
2. 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.

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 included every time the <modify order> Use Case is run.

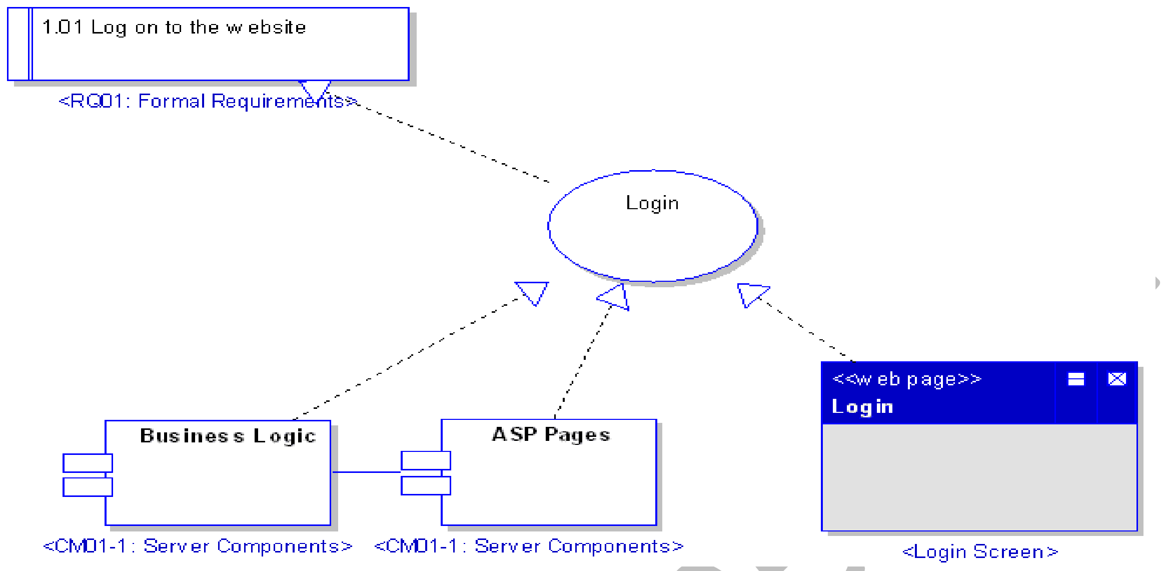
A Use Case may be included by one or more Use Cases, so it helps to reduce duplication of functionality by factoring out common behaviour into Use Cases that are re-used many times.
One Use Case may extend the behaviour of another - typically when exceptional circumstances are encountered. For example, if before modifying a particular type of customer order, a user must get approval from some higher authority, then the <get approval> Use Case may optionally extend the regular <modify order> Use Case.

Sequence Diagrams
UML provides a graphical means of depicting object interactions over time in Sequence Diagrams. These typically show a user or actor, and the objects and components they interact with in the execution of a use case. One sequence diagram typically represents a single Use Case 'scenario' or flow of events.
Sequence diagrams are an excellent way to document usage scenarios and to both capture required objects early in analysis and to verify object usage later in design. Sequence diagrams show the flow of messages from one object to another, and as such correspond to the methods and events supported by a class/object.
The diagram illustrated below shows an example of a sequence diagram, with the user or actor on the left initiating a flow of events and messages that correspond to the Use Case scenario. The messages that pass between objects will become class operations in the final model.



Implementation Diagram
A Use Case is a formal description of functionality the system will have when constructed. An implementation diagram is typically associated with a Use Case to document what design elements (eg. components and classes) will implement the Use Case functionality in the new system. This provides a high level of traceability for the system designer, the

customer and the team that will actually build the system. The list of Use Cases that a component or class is linked to documents the minimum functionality that must be implemented by the component.

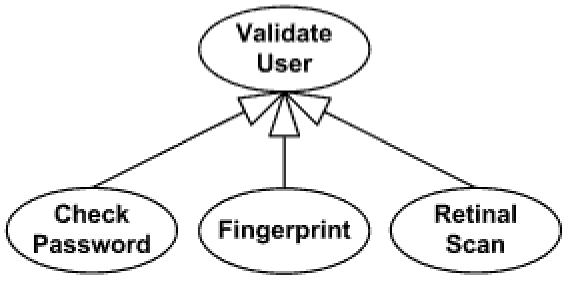

1.01 Log on to the website
<RQ01: Formal Requirements>
Login
<<web page>>
Login
Business Logic
ASP Pages
<CM01-1: Server Components>
<CM01-1: Server Components>
<Login Screen>

The example above shows that the Use Case "Login" implements the formal requirement "1.01 Log on to the website". It also states that the Business Logic component and ASP Pages component implement some or all of the Login functionality. A further refinement is to show the Login screen (a web page) as implementing the Login interface. These implementation or realisation links define the traceability from the formal requirements, through Use Cases on to Components and Screens.

Relationships Between Use Cases
Use cases could be organized using following relationships:

- Generalization
- Association
- Extend
- Include

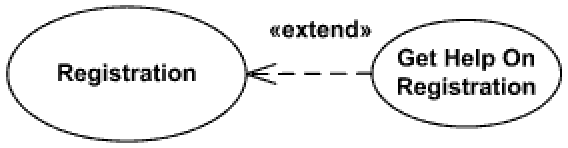
Generalization Between Use Cases
Generalization between use cases is similar to generalization between classes – child use case inherits properties and behavior of the parent use case and may override the behavior of the parent.
Notation: Generalization is rendered as a solid directed line with a large open arrowhead (same as generalization between classes).



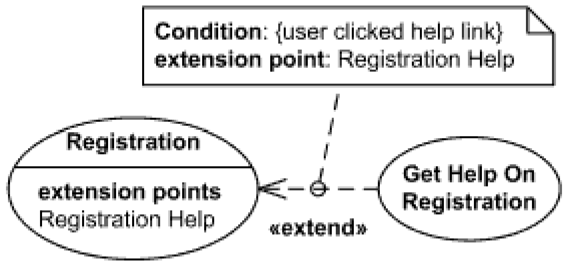
Generalization between use cases

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.
Note: **Extended use case is meaningful** on its own, independently of the extending use case, while the **extending use case** typically defines behavior that is **not necessarily meaningful by itself**.
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 **«extend»**.



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

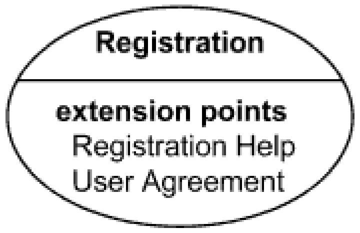
Extension Point

An **extension point** is a **feature** of a use case which identifies (references) a point in the behavior of the use case where that behavior can be extended by some other (extending) use case, as specified by an extend relationship.

Extension points may be shown in a compartment of the use case oval symbol under the heading **extension points**. Each extension point must have a **name**, unique within a use case. Extension points are shown as text string according to the syntax:

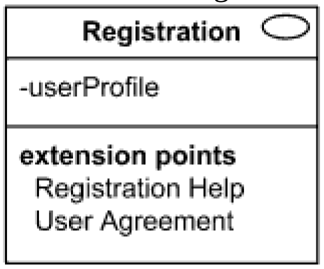
<extension point> ::= <name> [: <explanation>]

The optional description is given usually as informal text, but can also be given in other forms, such as the name of a state in a state machine, an activity in an activity diagram, a precondition, or a postcondition.



Registration **use case** with **extension points** Registration Help and User Agreement

Extension points may be shown in a compartment of the use case rectangle with **ellipse icon** under the heading **extension points**.



Extension points of the Registration **use case** shown using the rectangle notation

Include Relationship

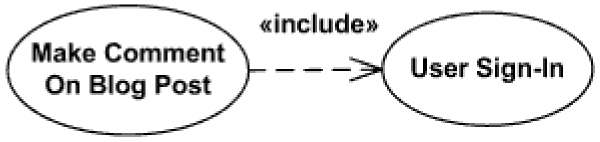
An **include** relationship is a **directed relationship** between two use cases, implying that the behavior of the required (not optional) **included** use case is inserted into the behavior of the **including** (base) use case. Including use case **depends on** the addition of the included use case.

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.

Execution of the included use case is analogous to a subroutine call or macro command in programming. All of the behavior of the included use case is executed at a single location in the including use case before execution of the including use case is resumed.

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»**.



Four Phases of Unified Process

The Unified Process consists of cycles that may repeat over the long-term life of a system. A cycle consists of four phases: Inception, Elaboration, Construction and Transition. Each cycle is concluded with a release, there are also releases within a cycle. Let's briefly review the four phases in a cycle:

* **Inception Phase** - During the inception phase the core idea is developed into a product vision. In this phase, we review and confirm our understanding of the core business drivers. We want to understand the business case for why the project should be attempted. The inception phase establishes the product feasibility and delimits the project scope.

* **Elaboration Phase** - During the elaboration phase the majority of the Use Cases are specified in detail and the system architecture is designed. This phase focuses on the "Do-Ability" of the project. We identify significant risks and prepare a schedule, staff and cost profile for the entire project.

* **Construction Phase** - During the construction phase the product is moved from the architectural baseline to a system complete enough to transition to the user community. The architectural baseline grows to become the completed system as the design is refined into code.

* **Transition Phase** - In the transition phase the goal is to ensure that the requirements have been met to the satisfaction of the stakeholders. This phase is often initiated with a beta release of the application. Other activities include site preparation, manual completion, and defect identification and correction. The transition phase ends with a postmortem devoted to learning and recording lessons for future cycles.