Object Orientation – UML Applied
- PART 1

- A Path to Successful Software Development

By
Piyal Perera

Email : piyalpy@sltnet.lk

Contents at a Glance

- Graphical User Interface(GUI) Design
  - User Centered Design
  - Usability Testing
  - Guidelines: User Interface (General)
- Database Design using the UML
  - The Database Life Cycle
  - System Architecture and Design
  - Modeling requirements with Use cases
  - Building Entity-Relationship (ER) Model
  - Building Class Models in UML
  - Designing an Object-Oriented Database Schema
- Code Generation and Reverse Engineering
  - Using Rational Rose 2002
- Software Testing
  - Levels of test
    - Developer Testing, Independent Testing, Unit Testing, Integration testing, System Testing, Acceptance Testing
    - Testing Documentations
    - Steps needed to develop and Run software tests
    - Test Plan, Test case
    - Problems and Answers
- The Transition Phase
  - Objectives, Essential Activities
  - The UML Package Diagram
  - The Facade Pattern

Appendix – 1 : e-Business Development

- Introduction
- Entering world of e-business
- Characteristics of e-business Development
- e-business Technologies
- e-business Development
  - Inception Phase Activities
  - Elaboration Phase Activities
  - Construction Phase Activities
  - Transaction Phase Activities
- Web Application Architecture
- Modeling Web Pages

Appendix – 2 : Roles and Activities

- System Analyst
- Software Architect
- Project Manager
- User-Interface Designer
- Database Designer
- Designer
- Technical Writer
- Test Manager
- Test Analyst
- Test Designer
- Tester
- Implementer

Appendix – 3 : Documents

- Vision
- Business Case
- Use Case Model Survey
- Software Requirement Specification
- Software Development Plan
- Class Report
- Design Model Survey
- Test Design Specification
- End-User Support Materials
- Status Assessment
- Change Request
- Risk Management Plan
- Risk List
- Examples

Appendix – 4: Case Study: Inventory Control System

- Requirement Gathering
- Identifying the Use case scenarios
- Applying the Class Diagram to the case study
- Applying the Activity Diagram to the case study
- Applying the Sequence Diagram to the case study
- Applying the Collaboration Diagram to the case study
- Applying the basic state chart to the case study
- Applying the extended state chart features to the case study

Appendix – 5: Software Project Management

- Introduction, purpose, RUP guidelines
- Organizational Context, SEPA, PRA, SEEA
- Risk Management
- Project Measurement
  - Metric Activities
  - Calculating Software Effort
  - Evaluating Software Quality

Table of Contents

1. Introduction

In the early days of computing, programmers typically did not rely on in-depth analyses of the problem at hand. If they did any analysis at all, it was typically on the back of a napkin. They often wrote programs from the ground up, creating code as they went along. While this added an aura of romance and daring to the process, it's inappropriate in today's high stakes business world.

Today, a well thought out plan is crucial. A client has to understand what a development team is going to do, and be able to indicate changes if the team has not fully grasped the client's needs (or if the client changes his or her mind along the way). Also, development is typically a team-oriented effort, so that each member of the team has to know where his or her work fits into the big picture (and what that big picture is).

As the world becomes more complex, the computer based systems that inhabit the world also must increase in complexity. They often involve multiple pieces of hardware and software, networked across great distances, linked to databases that contain mountains of information. If we want to create the systems that will take us into the next technology era, how do we get our hands around the complexity? From this moment, our aim is to go through this complexity with easy and simple mind.

2. The Software Development Process

In this section we will discuss software development models and issues related with these models.

2.1. The Waterfall Model

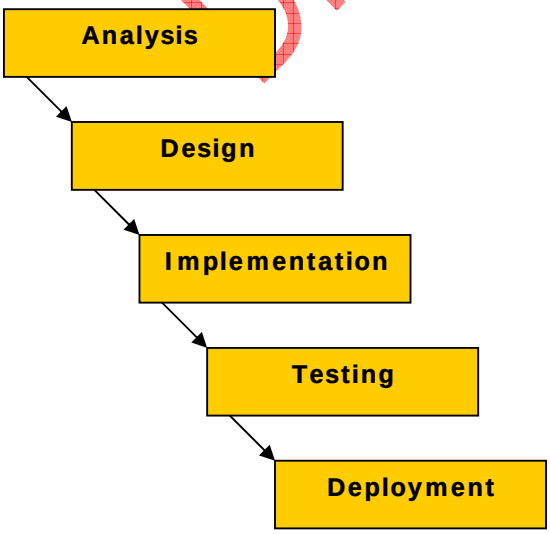


Figure 1: The traditional Waterfall Model

The waterfall method prescribes that each stage must be complete before the next stage can commence. This simplistic process begins to break down as the complexity and the size of the project increase. The main problems are:

- Even large systems must be fully understood and analyzed before progress can be made to the design stage. The complexity increases, and becomes overwhelming for the developers.
- Risk is push forward. Major problems often emerge at the later stages of the process – especially during the system integration. The cost to rectify errors increase exponentially as time progresses.
- On large projects, each stage will run for extremely long periods.

Also, as the analysis phase is performed in a short period at the outset of the project, we run a serious risk of failing to understand the customer requirements. Even if we follow a rigid requirements management procedure and sign off requirements with the customer, the chances are that by the end of Design, Coding, Integration and Testing, the final product will not necessarily be what the customer wanted.

In summary, the waterfall model is easy to understand and simple to manage. But the advantages of the model begin to break down once the complexity of the project increases. The waterfall method is a valuable process, if the project life cycle is short.

2.2. The Spiral Model

An alternative approach is the Spiral Model. In this approach, we attack the project in a series of short lifecycles, each one ending with a release of executable software.

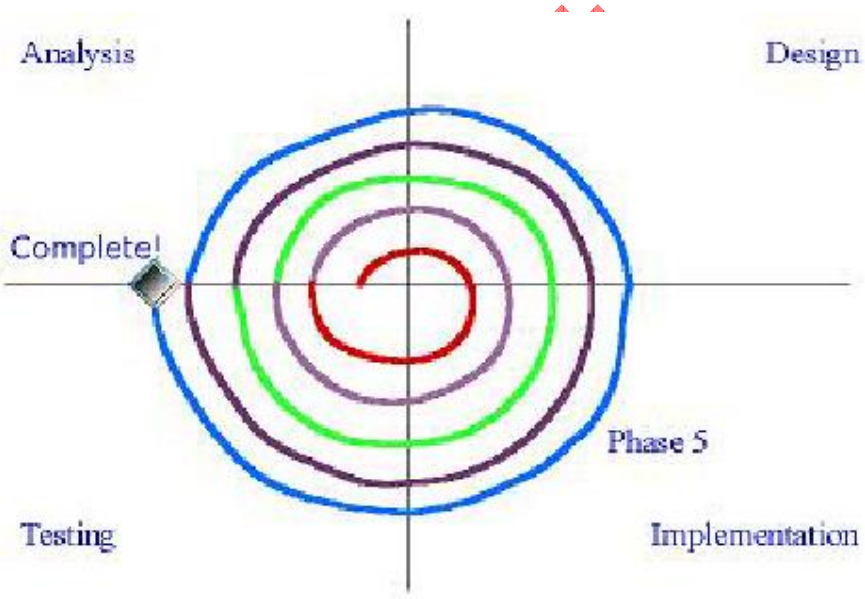


Figure 2: The Spiral Model

In this example the project has been divided into five phases, each phase building on the previous one and with the running release of software produced at the end of each phase.

With this approach:

- The teams are able to work on the entire lifecycle (Analysis, Design, Code and Test) rather than spending years on a single activity
- We can receive early and regular feedback from the customer, and spot potential problems before going too far with development

- We can attack risk up-front. Particularly risky iterations (for example, an iteration requiring the implementation of new and untested technology) can be developed first
- The scale and complexity of work can be discovered earlier
- Changes in technology can be incorporated more easily
- A regular release of software improves morale
- The status of the project can be assessed more accurately

The drawbacks of a spiral process are:

- The process is commonly associated with Rapid Application Development, which is considered by many to be a hacker's charter
- The process much more difficult to manage. The Waterfall modal fits in closely with classic project management techniques such as Gantt charts, but spiral process requires different approach.
- Customer has to fulfill the required infrastructure at the beginning of the project. (Hardware, staff and other resources)

2.3. Iterative, Incremental Frameworks

The Iterative, Incremental Framework is a logical extension to the spiral model. The framework is divided into four major phases.

Inception; Elaboration; Construction and Transition

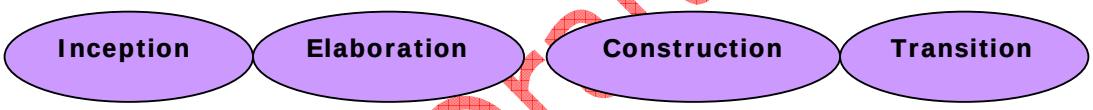


Figure 3: The four phases of an Iterative, Incremental Framework

2.3.1.2.3.1 Inception:

The inception phase is concerned with establishing the scope of the project and generally defining a vision for the project. Possible deliverables from this phase are:

- A Vision Document
- An initial exploration of the customer's requirements
- A first-cut project glossary
- A Business Case (including success criteria and a financial forecast, estimates of the Return of Investment, etc)
- An initial risk assessment
- A project plan

Please refer Appendix – 3 (Documents) for more details.

2.3.2. 2.3.2 Elaboration

The purpose of elaboration is to analyze the problem, develop the project plan further, and eliminate the riskier areas of the project. By the end of the elaboration phase, we aim to have a general understanding of the entire project, even if it is not necessarily a deep understanding.

Two of the UML models are often invaluable at this stage. The ***Use Case Model*** helps us to understand the customer's requirements, and we can

also use the ***Analysis Class diagram*** to explore the major concepts our customer understands. ***Prototyping*** is also done at this stage.

2.3.3.2.3.3 Construction

At the construction phase, we build the product. This phase of the product is not carried out in a linear fashion. It is followed by a series of **iterations**. Iteration is a simple waterfall model. By keeping each iteration as short as possible, we aim to avoid the nasty problems associated with waterfalls.

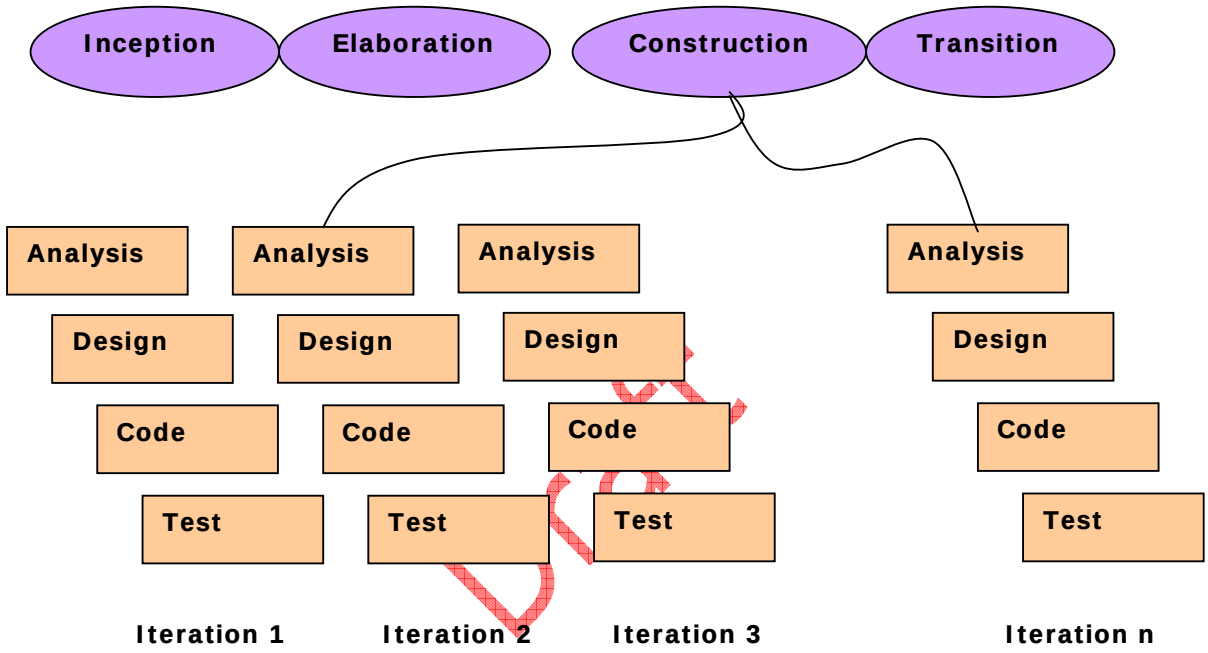


Figure 4: The Construction Phase consists of series of 'mini waterfalls'

At the end of as much iteration as possible, we will have a running system. These iterations are called ***Increments***.
A single iteration should typically last between 2 weeks and 2 months. Anymore than two months leads to an increase in complexity.
Some factors that should influence the iteration length include:

- Early development cycles may need to be longer. This gives developers chance to perform exploratory work on untested or new technology, or to define the infrastructure for the project.
- Novice Staff
- Parallel development teams
- Distributed teams
- A high ceremony project – one which might have to deliver a lot of project documentation to the customer, or perhaps a project which must meet lot of legal requirements.

2.3.4.2.3.4 Transition:

The final phase is concerned with moving the final product across to the customers. Typical activities in this phase include:

- Beta-releases for testing by the user community
- Factory testing, or running the product in parallel with the legacy system that the product is replacing
- Data take on (i.e. converting existing databases across to new formats, importing data etc.)
- Training the new users
- Marketing, Distribution and Sales

The transition phase should not be confused with the traditional test phase at the end of the waterfall model. At the start of the Transition, a full, tested and running product should be available for the users.

2.4. Timing and Assessments

2.4.1. Time Boxing

A radical approach to managing an iterative, incremental process is ***Time Boxing***. This is a rigid approach which sets a fixed time period in which a particular iteration must be completed by.

If iteration is not complete by the end of the time box, the iteration ends any way. The crucial activity associated with time boxing is the review at the end of iteration. The review must explore the reasons for any delays, and must reschedule any unfinished work into future iterations.

The better way to approach is that, the developers should be held responsible for (or at least, have a large say in) setting which requirements are covered in each iteration, as they are the ones who will have to meet the dead lines.

Implementing time boxing is difficult. It requires a culture of extreme discipline through the entire project. It is extremely tempting to forgo the review and overstep the time box if the iteration is “99%” complete when the deadline arrives. Once a project succumbs to temptation and one review is missed, the whole concept begins to fall apart. Many reviews are missed, future iterations planning become sloppy and chaos begins to set in.

Some managers assume that time boxing prevents slippage. It does not. If an iteration is not complete once the time box has expired, then the unfinished work must be reallocated to later iterations, and the iterations plans are reworked, this could include slipping the delivery date or adding more iterations. However, the benefits of time boxing are:

- The rigid structure enforces planning and re-planning. Plans are not discarded once the project begins to slip
- If time boxes are enforced, there is less of a tendency for the project to descend into chaos once problems emerge, as there is always a formal time box review not far away
- If panic sets in and developers start to furiously hack, the hacking is stemmed once the review is held.

Essentially, time boxing allows the entire project to regularly “stand back” and take stock. It does not prevent slippage, and requires strong project management to work.

2.5. 3.2 Typical Project Timing

This is entirely up to individual projects, but a loose guideline is 10% inception, 30% elaboration, 50% construction and 10% transition.

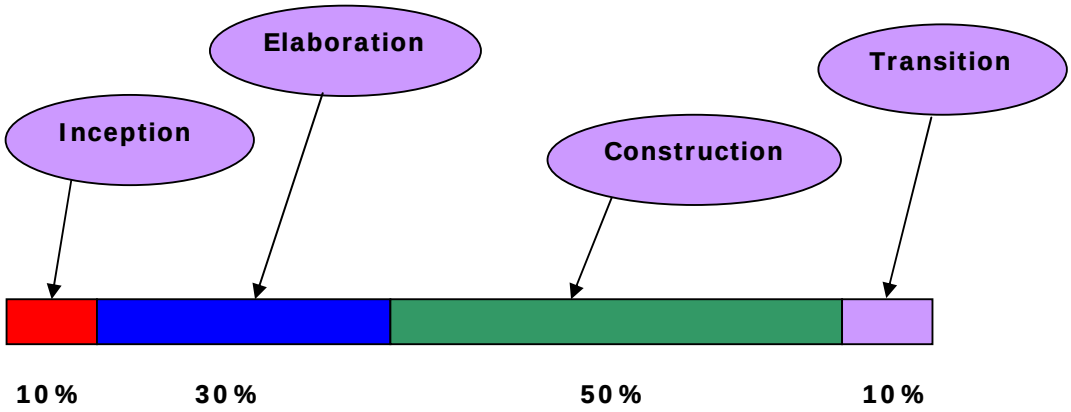


Figure 5: Possible timing for each phase

2.6. 3.3 Milestones Assessment

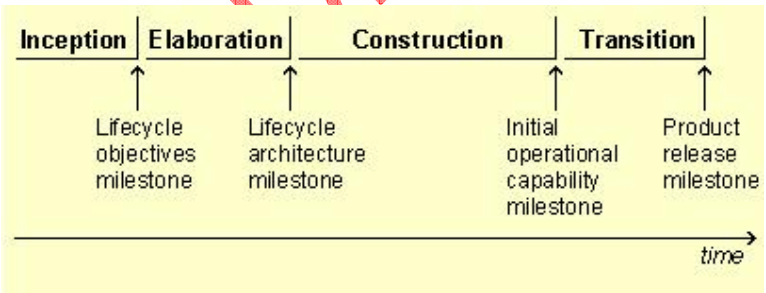


Figure 6: Milestones Assessment

Each sequential phase concluded by a major milestone, each phase is essentially a span of time between two major milestones. At each phase-end an assessment is performed to determine whether the objectives of the phase have been met. A satisfactory assessment allows the project to move to the next phase.

Development Methodologies

A methodology consists of a process, a vocabulary, and a set of rules and guidelines. The ***process*** defines a set of activities that together accomplish all the goals of the methodology. The ***vocabulary*** of the methodology is used to describe the process and the work products created during the application of the process. The ***rules and guide lines*** define the quality of the process and the work products.
The part of all methodologies that can be standardized is the vocabulary, often expressed in a ***notation***. The UML standard is a common notation that may be applied to many different types of software projects using very different methodologies. The variations appear in the use of the UML extensions, like stereotypes, and the emphasis placed on different diagrams for different types of projects.
We can identify four different development methodologies: RUP, Shlaer-Mellor, CRC, and Extreme Programming. They represent the diversity of the current set of methodologies. In brief, the Rational Unified Process (RUP) works well in large projects where quality artifacts and communication are critical. The Shlaer-Mellor method was developed primarily to address the unique needs of real-time systems long before the UML existed, but has since adopted the UML notation. CRC (Class, Responsibilities, and Collaborators) is almost more of a technique than a methodology. It was designed as a tool to help people gain an understanding of objects and how they work. Extreme Programming tries to reduce the many of the existing development practices to the bare essentials, attempting to optimize the relationship between developers and clients.

2.7. 4.1 The Rational Unified Process

The Rational Unified Process (The RUP) is the most famous example of an iterative, Incremental Lifecycle in use at the moment. The RUP is tailor able, and enables each phase of the process to be customized. Actually, RUP is about successful software development.
The RUP was developed by the same "Three Amigos" that developed the UML, so the RUP is very complementary to the UML. Essentially, Rational appreciate that every project is different, with different needs. For example, for some projects, a tiny Inception phase is appropriate, whereas for defense projects, the Inception project could last years.

2.7.1. 4.1.2 Strengths of the RUP

- The emphasis on iterative development and incremental deliveries is a time-tested and valuable approach that prevents many common project problems. However, it must be noted that this approach is common to most of the current methodologies.
- The process is well defined and supported by the Rational Modeling tool.
- The artifacts and the roles of the project participants are also very well defined.
- The process combines many of the best practices from many successful methodologies.
- The process is comprehensive.

2.7.2.4.1.3 Weaknesses of the RUP

- In trying to be comprehensive, the RUP becomes very large and difficult, both to learn and to manage.
- It is easy to get so caught up in the rules for using the RUP that you forget why you are using it (to deliver software).
- A substantial amount of time is spent trying to customize the RUP for each project. Here, too, you run the risk of becoming slave to the process and losing sight of the reason for the process.
- Tool support for the process is limited to Rational's own products, which are at the high end of the cost range. A few other vendors are now starting to provide limited support.

The core of the RUP, the Iterative, Incremental Life Cycle will be followed through out this book to illustrate the key aspects of the UML model. Sorry, about my decision, but if you need to have more information about current methodologies, visit www.cetus.links.org where you can find related articles and books.

Object Orientation

It is worth introducing OO and looking at the advantages that OO can offer to software development.

2.8. 5.1 Structured Programming

In Structured Programming, the general method was to look at the problem, and then design collection of functions that can carry out the required tasks. If these functions are too large, then the functions are broken down until they are small enough to handle and understand. This is a process known as functional decomposition.
As a simple example, consider a college Management System. This system holds the details of every student and tutor in the college. In addition, the system also stores information about the courses available at the college, and tracks which student is following which courses.
A possible functional design would be to write the following functions:
Add_student
Enter_for_exam
Check_exam_marks
Issue_certificate
Expel_student

A possible database scheme would be:

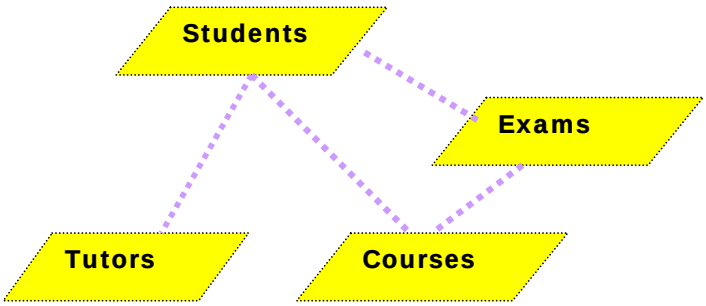


Figure 7: Simple Database schema

The following diagram is a sketch of all the functions, together with the data, and lines have been drawn where a dependency exists:

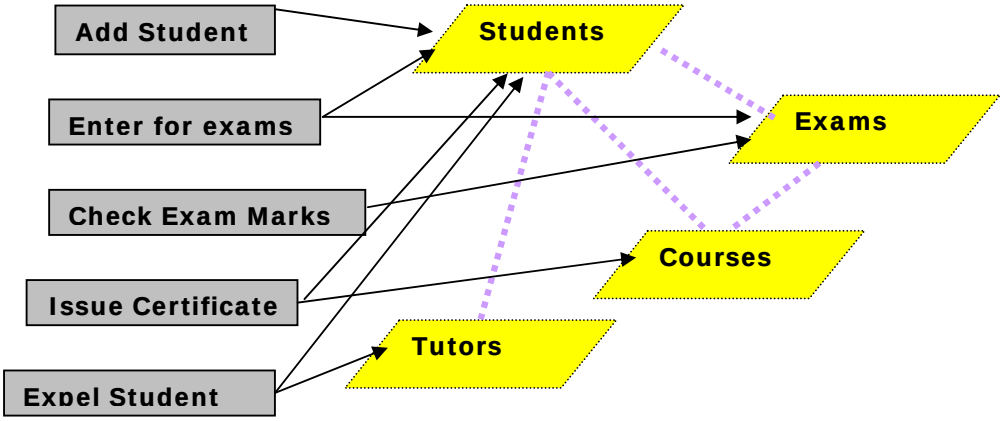


Figure 8: Plan of the functions, the data, and the dependencies

The problem with this approach is that if the problem we are tackling becomes too complex, the system becomes harder and harder to maintain. Taking the example above, what would happen if the requirement changes that leads to an alteration in the way in which Student data is handled?
As an example, imagine our system is running perfectly well, but we realize that storing the Student's date of birth with the two digit year was a bad idea. The obvious solution is to change the 'Date of Birth' field in the student table, from two digit year to four digit year.
The serious problem with this is that we might have caused unexpected side effects to occur. The Exam data, the Course data and the Tutors data all depend (in some way) on the Student data, so we might have broken some functionality with our simple change.
Specially, we have a large degree of potential knock-on problems. What is far, far worse is that in our program code, we can not easily see what these dependencies actually are.

2.9. 5.2 The Object Oriented Approach

OO tries to lessen the impact of this problem by combining related data and functions into the same module.
The following figure shows the full grouping of the related data and functions, in the form of modules:

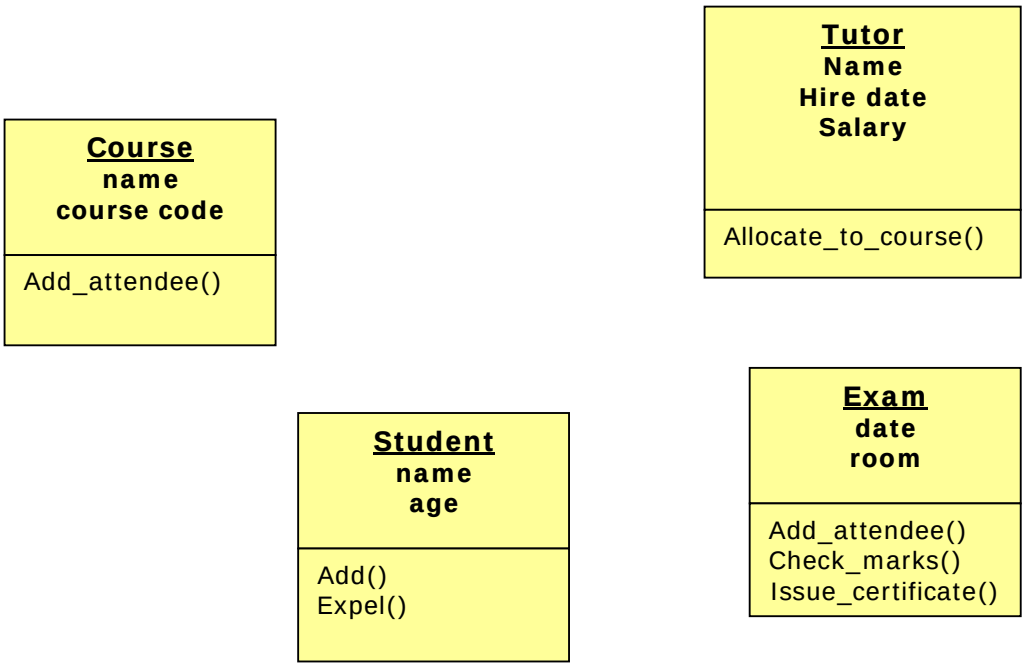


Figure 9: Related data and functions placed in modules

A couple of points to note about this new modular system of programming:

- More than one instance of a single module can exist when the program is running. In the college system, there would be an instance of 'Student' for every student that belongs to the college. Each instance would have its own values for the data
- Modules can 'talk' to other modules by calling each other's functions

Object Orientation is a mindset that depends on a few fundamental principles. An object is an instance of a class. A class is a general category of objects that have the same attributes and operations. When you create an object, the problem area you are working in determines how many of the attributes and operations to consider.

2.10. 5.3 The Object Oriented Strategy

One significant weakness of Object Orientation in the past has been that while OO is strong at working at the class/object level, OO is poor at expressing the behavior of an entire system. Looking at classes is all very well, but classes are very low level entities and do not really describe what the system as whole can do.
The modern approach, strongly supported by the modeling software is to forget all about object and classes at the early stages of a project, and instead concentrate on what the system must be able to do. Then, as the project progress, classes are gradually built to realize the required system functionality.

Object Orientation offers numerous advantages, like reusability and fast development time. The ones that aren't object oriented are often called 'legacy' systems.

2.11. 5.4 Objects

Objects in the real world can be characterized by two things: each real world object has data and behavior. For example, a television is an object and posses data in the sense that it is tuned to a particular channel, the scan rate is set to a certain value, the contrast and brightness is a particular value and so on. The television object can also 'do' things. The television can switch on and off, the channel can be changed and so on.
We can represent this information in the same way as our previous software 'modules':

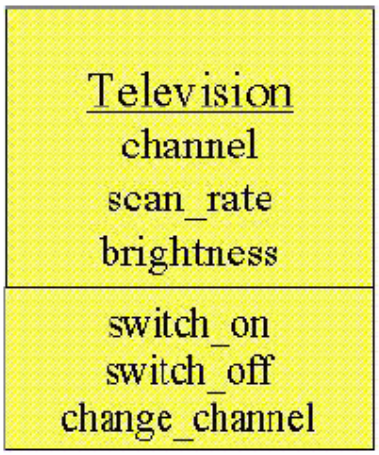


Figure 10: The data and behavior of a television

In some sense, then, real word 'objects' can be modeled in a similar way to the software modules, hence, we have the term Object Oriented Design and Programming.

2.12. 5.5 Terminology

The data for an object generally called the ***Attributes*** of the project. The different behaviors of an object are called the ***Methods*** of the object. Methods are directly analogous to functions or procedures in programming languages. The other big jargon term is ***Class***. A class is simply a template for an object. A class describes what attributes and methods will exist for all instances of the class. In the college system, we had a class called *Student*.

2.13. 5.6 Abstraction

Abstraction means, simply, to filter out an object's properties and operations until just the ones you need are left. Generally speaking, try to get rid of unnecessary properties and operations.

2.14. 5.7 Inheritance

An object is an instance of a class. As an instance of a class, an object has all the characteristics of its class. This is called ***inheritance***. Not only can an

object inherit from a class, a class can inherit from another class. Super classes can also be sub classes, and inherit from other super classes.

2.15. 5.8 Polymorphism

In ***polymorphism***, an operation can have the same name in different classes, and proceed differently in each class. For example, you can open a door, you can open a news paper, a present, or a bank account. In each case, you are performing a different operation. In Object Orientation, each class 'knows' how that operation is supposed to take place.

2.16. 5.9 Encapsulation

Objects ***encapsulate*** what they do. That is, they hide the inner workings of their operations from the outside world and from other objects. An object hides what it does from other objects and from the outside world. For this reason, encapsulation is also called *information-hiding*. But an object does have to present a 'face' to the outside world so we can initiate those operations.

2.17. 5.10 Message Sending

One object sends another a message to perform an operation, and the receiving object performs that operation.

2.18. 5.11 Associations

Objects are often associated with each other in some way. Sometimes, objects are associated with each other in more than one way. A class can associate with more than one other class.

2.19. 5.12 Aggregation

An object that is made up of a combination of a number of different types of objects is called aggregate object. This combination is called ***aggregation***. A typical computer system is an example of an aggregation, which consists of a CPU box, a keyboard, a mouse, a monitor and other devices.

2.20. 5.13 Composition

One form of aggregation involves strong relationship between an aggregate object and its component objects. It's called ***composition***. For example, a shirt is a composite of a body, a collar, sleeves, buttons, button holes, and cuffs. Do away with the shirt and the collar becomes useless. Sometimes, a component in a composite does not last as long as the composite itself. The leaves on a tree can die out before tree does. If you destroy the tree, the leaves also die. Aggregation and composition are important because they reflect extremely common occurrences, and thus help to create models that closely resemble reality.

2.21. 5.14 Why Model Software?

The purpose of diagrams is to present multiple views of a system, and this set of multiple views is called a ***model***. Modeling accelerates development of applications and improves the quality of the delivered project. Specifically, modeling enhances communication, leads to better planning, reduces risk and ultimately cost of a project. By using standard modeling languages, teams can make themselves more effective by compressing a tremendous amount of high-level and granular information into a visual form that can be easily communicated. Project components such as architecture, system dependencies,

complexity, business requirements, and source code can all be represented on one model.

What is the UML?

The Unified Modeling Language, or the **UML**, is a graphical modeling language that provides us with syntax for describing the major elements (called **artifacts** in UML) of software systems. Artifacts are either final or intermediate work products produced and used during a project. They capture and convey project information, and may take various shapes or forms.

In the mid 90's, from out of 50 software modeling languages following three languages emerged as the strongest.

- *Booch* was excellent for design and implementation
- *OMT* (Object Modeling Technique) was best for analysis and data-intensive systems.
- *OOSE* (Object Oriented Software Engineering) featured a model known as Use Cases.

In 1995, creators of the above three models joined hand and the UML consortium formed. The team of the UML creators is affectionately known as the *'Three Amigos'*. The creators of the UML prefer 'the UML' instead of 'UML', therefore I am following their wish.

The UML was adopted by OMG (Object Management Group – www.omg.org), and since then the OMG have owned and maintained the language. Therefore, the UML is an effectively a public, non proprietary language. The UML has become a de facto standard in the software industry, and it continues to evolve.

The UML is, as its name implies, a modeling language and not a method or process. The UML is made up of a very specific notation and the related grammatical rules for constructing software models. The UML in itself does not proscribe or advise on how to use that notation in a software development process or as part of an object oriented design methodology.
The UML supports a rich set of graphical notation elements. It describes the notation for classes, components, nodes, activities, work flow, use cases, objects, states and how to model relationship between these elements. The UML also supports the notion of custom extensions through stereotyped elements.
The UML provides significant benefits to software engineers and organizations by helping to build rigorous, traceable, and maintainable models, which support the full software development lifecycle.

2.22. An Overview of the UML

The UML model can be either Platform-Independent Module (PIM) or Platform-Specific Model (PSM). The PIM represents its business functionality and behavior very precisely but does not include technical aspects.
The PSM contains the same information as an implementation, but in the form of the UML model instead of running code. In the next step, the tool generates the running code from the PSM, along with other necessary files (including interface definition files if necessary, configuration files, makefiles, and other file types. After giving the developer and opportunity to hand-tune the generated code, the tool executes the makefiles to produce a deployable final application.

The UML is that there are a lot of different diagrams (models) to get used to. The reason for this is that it is possible to look at a system from many different viewpoints. A software development will have many stakeholders playing a part – for example:

- Analysts
- Designers
- Coders
- Testers
- QA
- The Customer
- Technical Authors

All of these people are interested in different aspects of the system, and each of them requires a different level of detail. Therefore, the UML attempts to provide a language so expressive that all stakeholders can benefit from at least one UML diagram.

The UML defines twelve types of diagrams, divided into three categories: Four diagram types represent static application structure, five represents different aspects of dynamic behavior, and three represent ways you can organize and manage your application modules.

Structure Diagrams include the Class diagram, Object Diagram, Component diagram, and Deployment diagram.

Behavior Diagrams include the Use Case Diagram, Sequence Diagram, Activity Diagram, Collaboration Diagram, and State chart Diagram.

Model Management Diagrams include Packages, Subsystems, and Models.

2.23. The Use Case Diagram

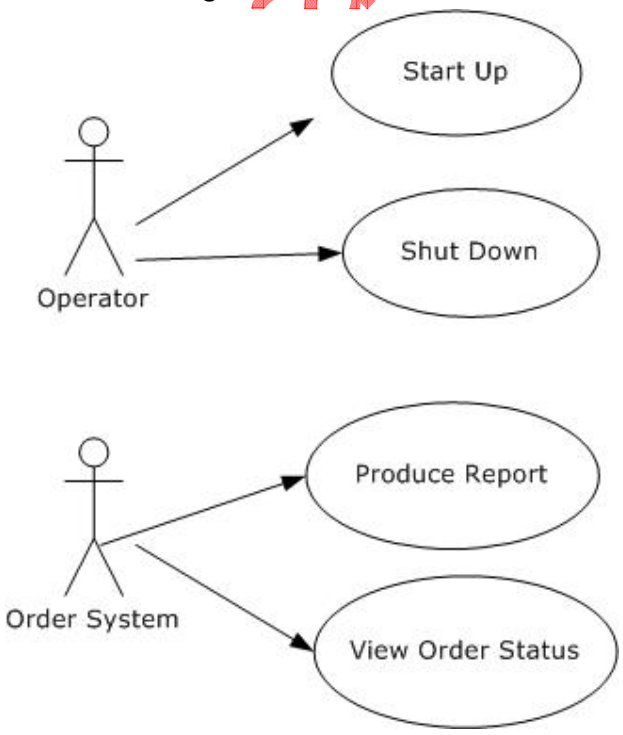


Figure 1: The Use Case Diagram

A Use Case is a description of the system's behavior from a user's viewpoint. This diagram is a valuable aid during analysis stage. For system developers, this is a valuable tool: it's a tried-and-true technique for gathering system requirements from a user's point of view. That's important if the goal is to build a system that real people (and not just computerphiles) can use.
The little stick figure that corresponds to the operator is called an *actor*. The ellipse represents the use case. An actor, the entity that initiates the use case, can be a person or another system.

2.24. The Class Diagram

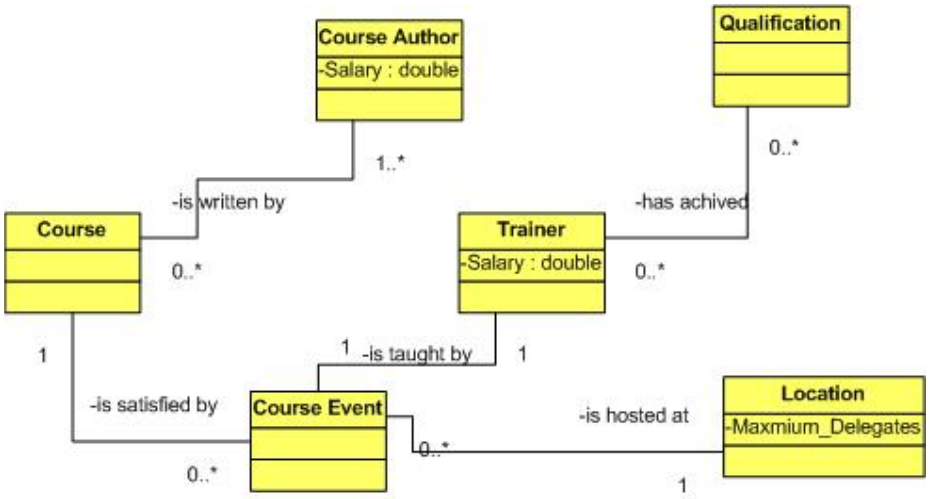
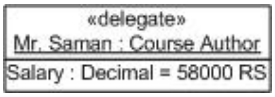


Figure 2: The UML Class Diagram

A rectangle is the icon that represents the class. It's divided into three areas. The uppermost area contains the name, the middle area holds the attributes, and the lowest area the operations. A class diagram consists of a number of these rectangles connected by lines that show how the classes relate to one another.
Class diagrams help on the analysis side and as well as development side. They provide the representations that developers work from and enable analyst to talk to clients in the client's terminology and thus stimulate the clients to reveal important details about the problem they want solved.

An *object* is an instance of a class, a specific thing that has specific values of the attributes and behavior.



The UML representation of an object icon is a rectangle, just like the class icon, but the name is underlined. The name of the specific instance is on the left side of a colon, while the name of the class is on the right side of the colon.

Drawing Class Diagrams (to draw conceptual model) is an essential aspect of any Object Oriented Design method. Together with the Use Cases, a Conceptual Model is a powerful technique in requirements analysis.

2.25. Collaboration Diagram

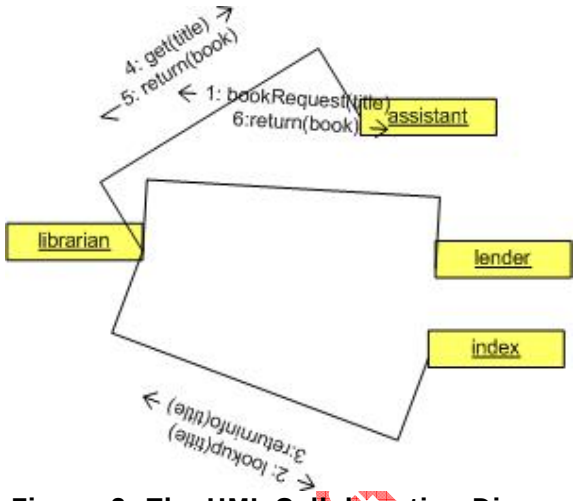


Figure 3: The UML Collaboration Diagram

The elements of a system work together to accomplish the system's objectives, and a modeling language must have a way of representing this. The UML collaboration diagram designed for this purpose.

As we are developing object-oriented software, anything our software needs to do is going to be achieved by objects collaborating. We can draw a collaboration diagram to describe how we want the objects we build to collaborate.

Though we discuss guidelines on good design principles, but much of the skill in design comes from experience.

2.26. Sequence Diagram

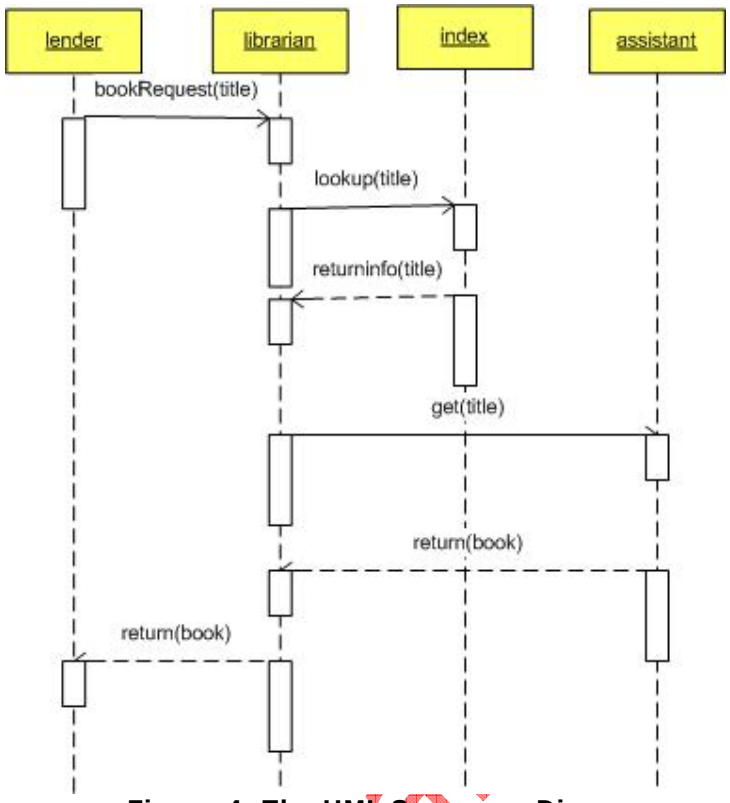


Figure 4: The UML Sequence Diagram

The sequence diagram is, in fact, directly related to the collaboration diagram and displays the same information, but slightly different form. The dotted lines down the diagram indicate time, so what we can see here is a description of how the objects in our system interact over time.
Some of the UML modeling tools, such as Rational Rose, can generate the sequence diagram automatically from the collaboration diagram.

2.27. State Diagrams

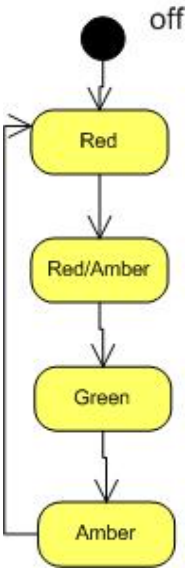


Figure 5: The UML State Diagram

Some objects can, at any particular time, be in certain state. For example, a traffic light can be any one of the following states:

Red, Amber, Green, Off

As state transitions can be quite complex, the UML provides a syntax to allow us model them.

2.28. Package Diagrams

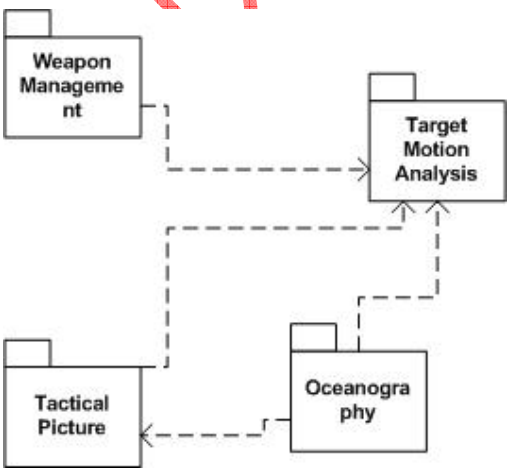


Figure 6: The UML Package Diagram

Any non-trivial system needs to be divided up in smaller, easier to understand 'Chunks', and the UML Package Diagram enables us to model this in a simple and effective way.

2.29. Component Diagrams

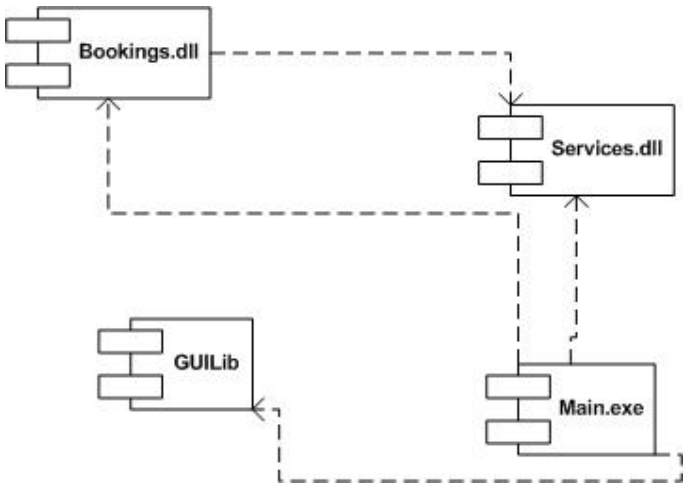


Figure 7: The UML Component Diagram

The Component Diagram is similar to the package diagram – it allows us to notate how our system is split up, and what the dependencies between each module are. However, the Component Diagram emphasizes the physical software components (files, link libraries, executables, headers, packages) rather than the logical partitioning of the Package Diagram.

2.30. Deployment Diagram

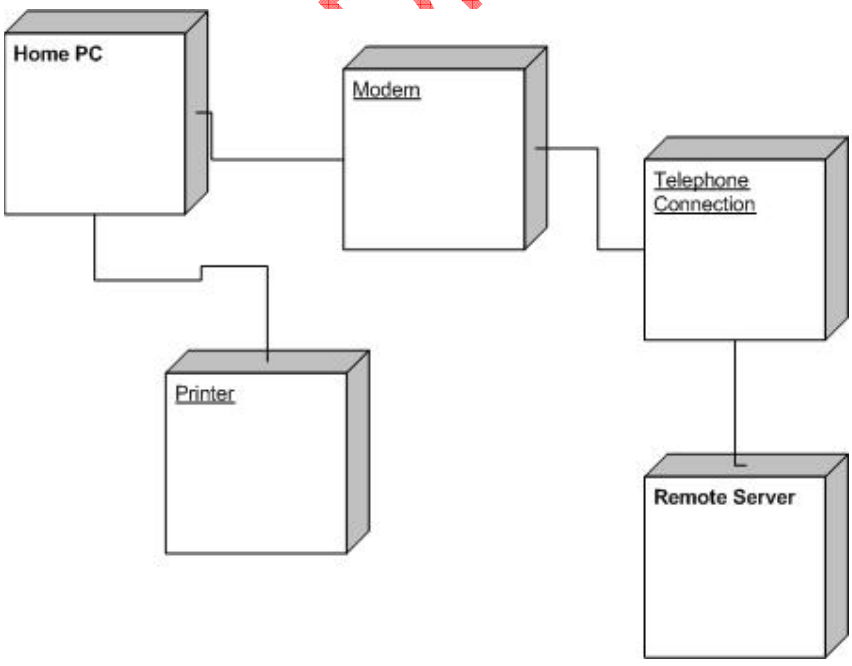
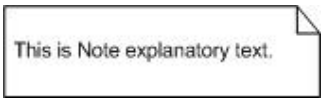


Figure 8: The UML Deployment Diagram

The UML deployment diagram shows the physical architecture of a computer based system. It can depict the computers and devices, show their connections with one another, and show the software that sits on each machine. Each computer is represented by a cube, with interconnections between computers drawn as lines connecting the cubes.

2.31. Notes

It often happens that one part of a diagram does not present a non ambiguous explanation of why it's there or how to work with it. When that's the case, the UML *note* is helpful. Its icon is a rectangle with a folded corner. Inside the rectangle is explanatory text. You can attach the not to a diagram element by connecting a dotted line from the element to the note.



2.32. Stereotypes

The UML provides a number of useful items. Every now and then, you will be designing a system that needs some tailor made items. Stereotypes enable you to take existing UML elements and turn them into new ones. You can represent it as a name enclosed in two pairs of angle brackets called *guillemets* and then apply that name appropriately.

2.33. Summary

The UML provides many different models of a system. The following is a list of them, with a one sentence summary of the purpose of the model:

- Use Cases – "How will our system interact with outside world?"
- Class Diagram – "What objects do we need? How will they be related?"
- Collaboration Diagram – "How will the object interact?"
- Sequence Diagram – "How will the object interact?"
- State Diagram – "What states should our objects be in?"
- Package Diagram – "How are we going to modularize our development?"
- Component Diagram – "How will our software components be related?"
- Deployment Diagram – "How will the software be deployed?"

2.34. What's New in the UML 2.0?

Now, let's briefly discuss current OMG's development process on UML model. Despite its enhanced focus on architecture-oriented diagrams, gaping holes in UI development and data modeling remain unchanged even in UML new version.

The Inception Phase

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.

2.35. 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 may be demonstrating, at least one candidate architecture against some of the primary scenarios.
- Estimating the overall cost and schedule for the entire project (and more detail estimates for the elaboration phase that will immediately follow)
- Estimating potential risks (the source of unpredictability)
- Preparing the supporting environment for the project.

2.36. Essential Activities

The essential activities of the Inception phase 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 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.

2.37. Size of the Phase

The size of the phase depends upon the project. An ecommerce project may well need to hit the market as quickly as possible, and the only activities in

Inception might be to define the vision and get finance from a bank via the business plan.
By contrast, a defense project could well require requirement analysis, project definition, previous studies, invites to tender, etc, etc. It all depends on the project.
Detail descriptions of following documents templates are available in appendix C.

- Business Case
- Project Vision with an example
- Development Case with an example

The Elaboration Phase

The goal of the elaboration phase is to baseline the architecture of the system to provide stable basis for the bulk of the design and implementation effort in the construction phase. The architecture evolves out of a consideration of the most significant requirements (those that have a great impact on the architecture of the system) and an assessment of risk. The stability of the architecture is evaluated through one or more architectural prototypes.

2.38. Objectives

The primary objectives of the elaboration phase include:

- To ensure that the architecture, requirements and plans are stable enough and the risks sufficiently mitigated to be able to predictably determine the cost and schedule for the completion of the development. For most projects, passing this milestone also corresponds to the transition from a light-and-fast, low risk operation to a high cost, high risk operation with substantial organizational inertia.
- To address all architecturally significant risk of the project
- To establish a base lined architecture derived from addressing the architecturally significant scenarios, which typically expose the top technical risks of the project.
- To produce and evolutionary prototype of production-quality components, as well as possible one or more exploratory, throw away prototypes to mitigate specific risk such as:
    - design/requirements tradeoffs
    - component reuse
    - product feasibility or demonstrations to investors, customers, and end-users

  We will discuss prototypes in detail after phase introduction paragraphs.

-

- To demonstrate that the base lined architecture will support the requirements of the system at a reasonable cost and in a reasonable time.
- To establish a supporting environment.

In order to achieve these primary objectives, it is equally important to set up the supporting environment for the project. This includes creating a development case, preparing templates, guidelines, and setting up tools.

2.39. Essential Activities

The essential activities of the Elaboration phase include:

- **Defining, validating and base lining the architecture** as rapidly as practical.

- **Refining the Vision**, based on new information obtained during the phase, establishing a solid understanding of the most critical use cases that drive the architectural and planning decisions.
- **Creating and base lining detailed iteration plans for the construction phase**.
- **Refining the development case and putting in place the development environment**, including the process, tools and automation support required to support the construction team.
- **Refining the architecture and selecting components**. Potential components are evaluated and the make/buy/reuse decisions sufficiently understood to determine the construction phase cost and schedule with confidence. The selected architectural components are integrated and assessed against the primary scenarios. Lessons learned from these activities may well result in a redesign of the architecture, taking into consideration alternative designs or reconsideration of the requirements.

In accordance with RUP explanations, we can categorize the Elaboration phase activities as follows.

- Refining the Business Case, Vision and Development Case
- Use Case Modeling
- Conceptual Model
- Prototyping

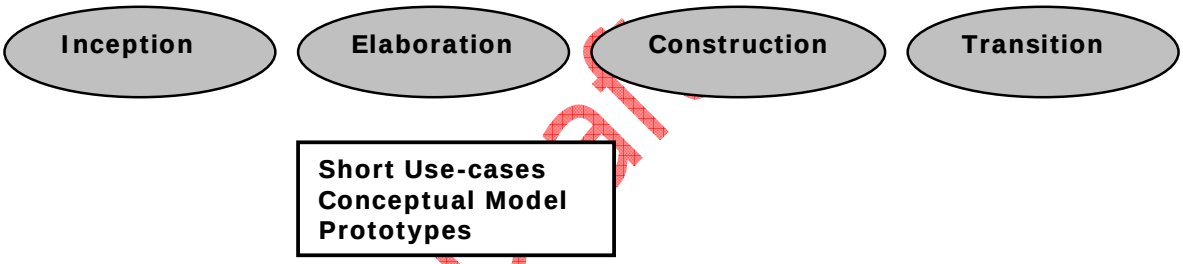


Figure 9: Elaboration phase activities

2.40. Use Case Modeling

2.40.1. Explanation

A use case model is a model of the system's intended functions and its surroundings, and serves as a contract between the customer and the developers. Use cases serve as a unifying thread throughout system development. The same use case model is the result of the Requirements discipline, and is used as input to Analysis & Design and Test disciplines. The most important purpose of a Use case model is to communicate the system's behavior to the customer or end user. It means that the model must be easy to understand.

2.40.2. Timing

Even though, we discussed Use case model at the elaboration phase, it can be used early in the inception phase to outline the scope of the system. The use case model is refined by more detailed flows of events during the construction phase. The use case model is continuously kept consistent with the design model. Because it is a very powerful planning instrument, the use case model is generally used in all phases of the development.

2.40.3. Responsibility

A System Analyst is responsible for the integrity of the use case model, and ensures that the use case model as a whole is correct, consistent, and readable.

2.40.4. Purpose

The following people use the use case model:

- The customer approves the use case model. When you have that approval, you know the system is what the customer wants. You can also use the model to discuss the system with the customer during development.
- Potential users use the use case model to better understand the system.
- The software architect uses the use case model to identify architecturally significant functionality.
- Designers use the use case model to get a system overview. When you refine the system, for example, you need documentation on the use case model to aid that work.
- The manager uses the use case model to plan and follow up the use case modeling and also subsequent design.
- People outside the project but within the organization, executives, and steering committees, use the use case model to get an insight into what has been done.
- People review the use case model to give appropriate feedback to developers on a regular basis.
- Designers use the use case model as a basis for their work.
- Testers use the use case model to plan testing activities (use case and integration testing) as early as possible.
- Those who develop the next version of the system use the use case model to understand how the existing version works.
- Documentation writers use the use cases as a basis for writing the system's user guides.

2.41. The Resources of the Use Case model

The Use case model takes advantage of three different view points to fully describe each requirement. The first and simplest resource is the ***use case diagram***. The ***Use case narrative*** and ***use case scenarios*** make up the remainder of the model.

2.41.1. The Use Case diagram

The use case diagram consists of five very simple graphics that represent the system, actors, use cases, and dependencies of the project. The goal of the diagram is to provide a high level explanation of the relationship between the system and the outside world. It is a very flat diagram (that is, it provides only a surface level, or black box, view of the system).

2.41.2. The elements of the use case diagram

Of the three elements that comprise the use case model, the only one actually defined by the UML is the use case diagram.