Object Oriented Paradigm

Definition: The *object-oriented paradigm* is an approach to the solution of problems in which all computations are performed in the context of objects. The objects are instances of programming constructs, normally called classes, which are data abstractions and which contain procedural abstractions that operate on the objects.

In the object-oriented paradigm, a running program can be seen as a collection of objects collaborating to perform a given task.

Figure 2.1 summarizes the essential difference between the object-oriented and procedural paradigms. In the procedural paradigm (shown on the left), the code is organized into procedures that each manipulate different types of data. In the object-oriented paradigm (shown on the right), the code is organized into classes that each contain procedures for manipulating instances of that class alone. Later on, we will explain how the classes themselves can be organized into hierarchies that provide even more abstraction.

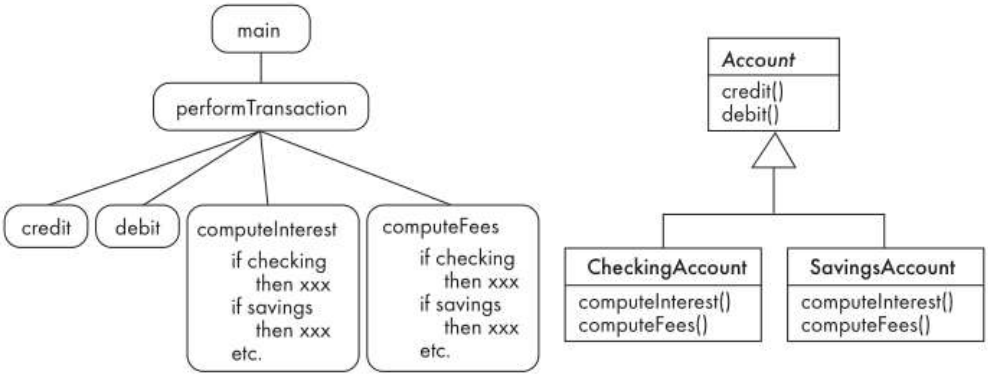


Figure 2.1 Organizing a system according to the procedural paradigm (left) or the object-oriented paradigm (right). The UML notation used in the right-hand diagram will be discussed in more detail later

OOA

Grady Booch has defined OOA as, *"Object-oriented analysis is a method of analysis that examines requirements from the perspective of the classes and objects found in the vocabulary of the problem domain"*.

Object-oriented analysis

It is method of analysis that examines requirements from the perspective of the classes and objects found in the vocabulary of the problem domain

A thorough investigation of the problem domain is done during this phase by asking the **WHAT** questions (rather than how a solution is achieved)

During OO analysis, there is an **emphasis on finding and describing the objects** (or concepts) in the problem domain.

For example, concepts in a Library Information System include Book, and Catalog.

- Object Oriented Analysis:
  - OOA is the procedure of identifying software engineering requirements and developing software specifications in terms of a software system's object model, which comprises of interacting objects.
  - The main difference between object-oriented analysis and other forms of analysis is that in object-oriented approach, requirements are organized around objects, which integrate both data and functions.
  - They are modeled after real-world objects that the system interacts with.
  - In traditional analysis methodologies, the two aspects - functions and data - are considered separately.

OOD

Booch defines OOD as follows: "Object-oriented design is a method of design encompassing the process of object-oriented decomposition and a notation for depicting both logical and physical a well as static and dynamic models of the system under design." In other words, OOD uses classe and objects to structure systems, as opposed to algorithmic abstractions used by structured design. It also uses a notation that expresses classes and objects (the logical decomposition) as well as modules and process (the physical decomposition).

- Object Oriented Design:
  - involves implementation of the conceptual model produced during object-oriented analysis.
  - In OOD, concepts in the analysis model, which are technology–independent, are mapped onto implementing classes, constraints are identified and interfaces are designed, resulting in a model for the solution domain, i.e., a detailed description of how the system is to be built on concrete technologies.
  - The implementation details generally include:
    - Restructuring the class data (if necessary),
    - Implementation of methods, i.e., internal data structures and algorithms,
    - Implementation of control, and
    - Implementation of associations.

OOA	OOD
Elaborate a problem	Provide conceptual solution
WHAT type of questions asked	HOW type of questions asked
During Analysis phase questions asked include Q. What is required in the Library Information System? A. Authentication!!	Later during Design phase questions asked include Q. How is Authentication in the Library Information System achieved? A. Thru Smart Card!! Or Fingerprint!!

UML

The Unified Modeling Language is a visual language for specifying, constructing and documenting the artifacts of systems [OMG03a].

The UML has emerged as the de facto and de jure standard diagramming notation for object-oriented modeling.

It started as an effort by Grady Booch and Jim Rumbaugh in 1994 to combine the diagramming notations from their two popular methods—the Booch and OMT (Object Modeling Technique) methods.

The UML is a language for

-visualizing
-constructing
-specifying
-documenting

Behavioral UML Diagram

- Activity Diagram
- Use Case Diagram
- Interaction Overview Diagram
- Timing Diagram
- State Machine Diagram
- Communication Diagram
- Sequence Diagram

Structural UML Diagram

- Class Diagram
- Object Diagram
- Component Diagram
- Composite Structure Diagram
- Deployment Diagram
- Package Diagram
- Profile Diagram

Structural Diagrams

A. Structural Diagrams

Static aspects of a house: walls, doors, windows, pipes, wires, and vents,

Static aspects of a software system : classes, interfaces, components, and nodes.

Used to describe the **building blocks** of the system

– features **that do not change** with time.

These diagrams answer the question – **What's there?**

Structure diagram shows static structure of the system and its parts on different abstraction and implementation levels and how those parts are related to each other.

The elements in a structure diagram represent the meaningful concepts of a system, and may include abstract, real world and implementation concepts.

Structure diagrams are not utilizing time related concepts, do not show the details of dynamic behavior. However, they may show relationships to the behaviors of the classifiers exhibited in the structure diagrams.

Class Diagram

A **class diagram** describes the types of objects in the system and the various kinds of static relationships that exist among them. Class diagrams also show the properties and operations of a class and the constraints that apply to the way objects are connected. The UML uses the term **feature** as a general term that covers properties and operations of a class.

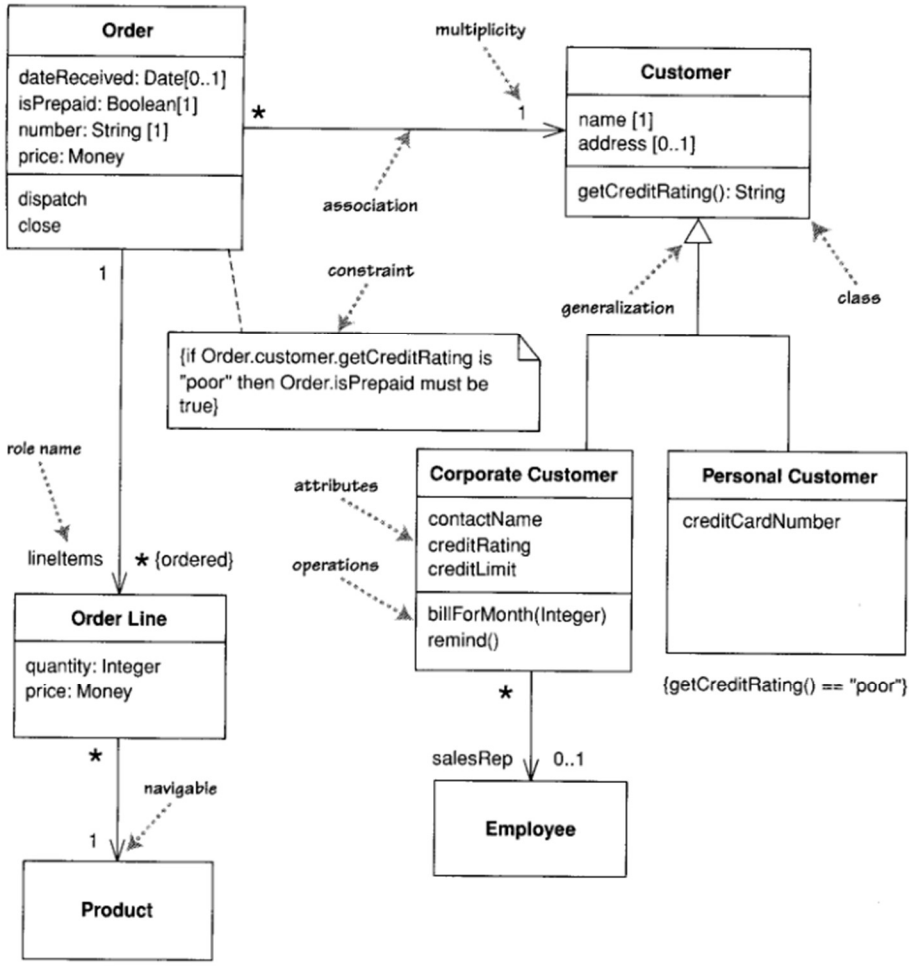


Figure 3.1 *A simple class diagram*

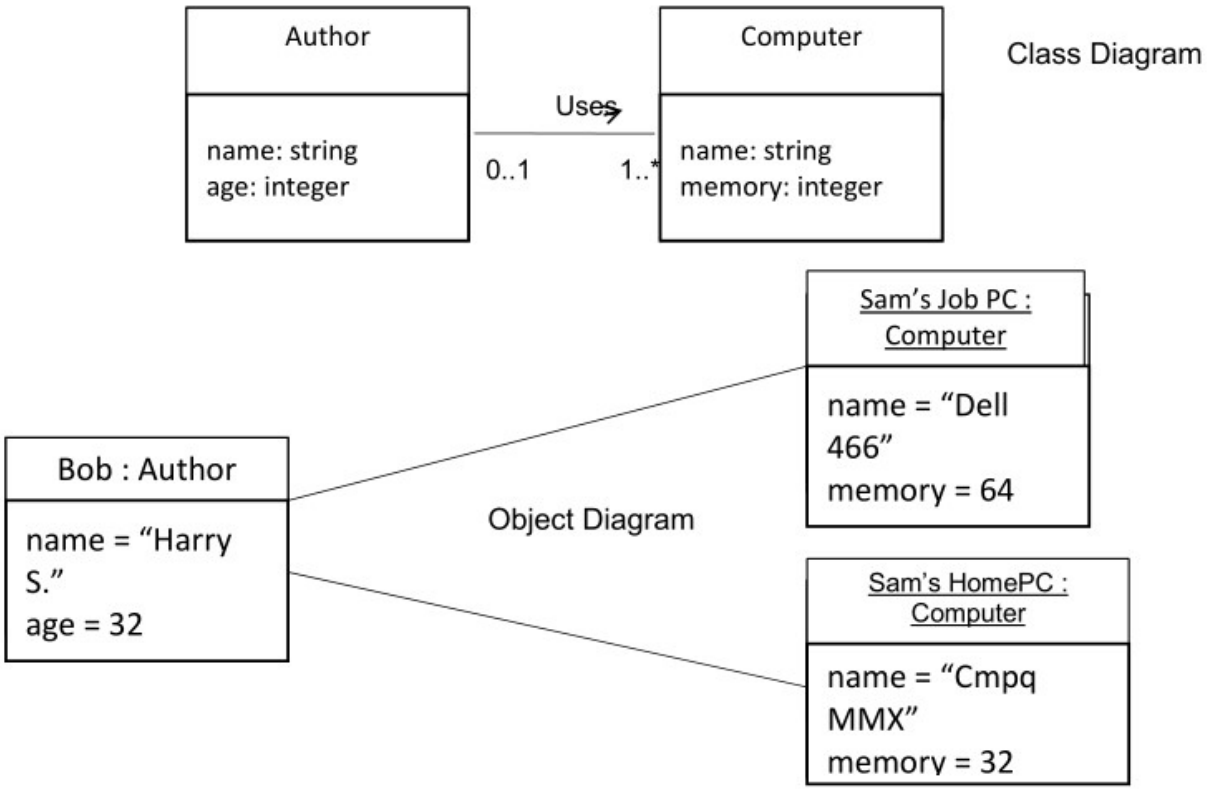
2. Object Diagram

Now obsolete, is defined as "a graph of instances, including objects and data values. A static object diagram is an instance of a class diagram; it shows a snapshot of the detailed state of a system at a point in time."

An *object diagram* shows a set of **objects and their relationships**.

Used to illustrate **data structures, the static snapshots of instances** of the things found in class diagrams.

Object diagrams address the static design view or static process view of a system just as do class diagrams, but from the **perspective of real** or **prototypical cases.**



B. Behavioral Diagrams

The UML's behavioral diagrams are used to visualize, specify, construct, and document **the dynamic aspects of a system**.

Dynamic aspects of a system represent **its changing parts.**

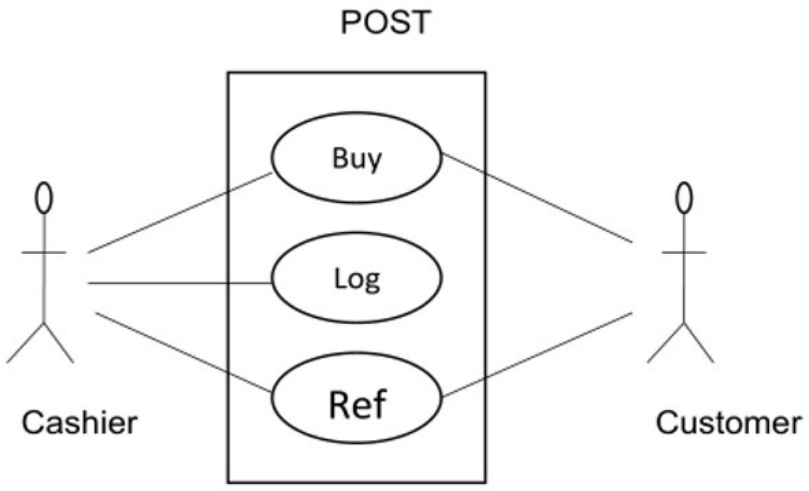
They show the dynamic behavior of the objects in a system, which can be described as a series of changes to the system over time.

Used to show **how** the system evolves over time (responds to requests , events etc)
The UML's behavioral diagrams are roughly organized around the major ways you can model the dynamics of a system.

1. Use Case Diagram

Use case diagrams are UML's notation for showing the relationships among a set of use cases and actors. They help a software engineer to convey a high-level picture of the functionality of a system.

It is not necessary to create a use case diagram for every system or subsystem. For a small system, or a system with just one or two actors, a simple list of use cases will suffice.



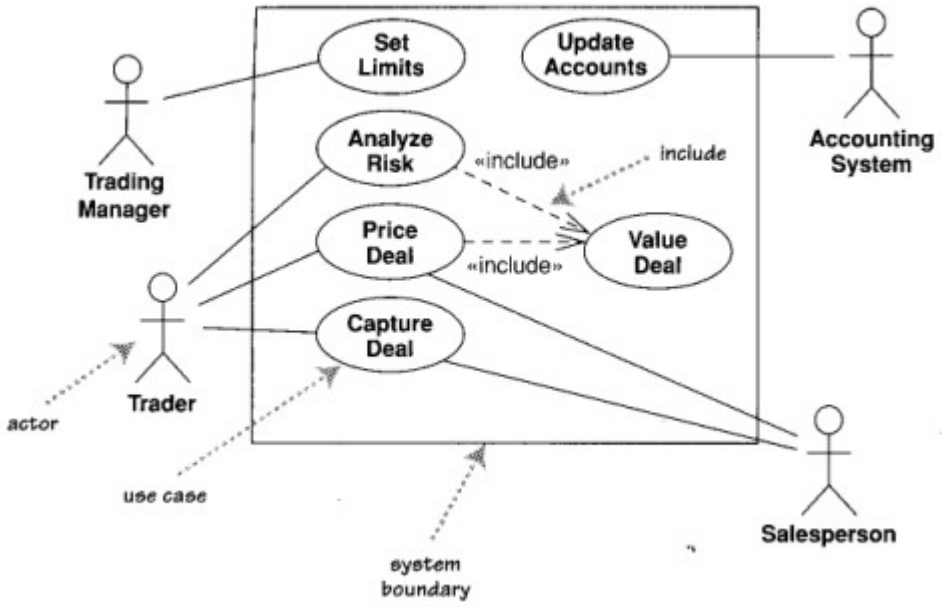
Emphasis on what *a system does* **rather than how.**

Scenario – Shows what happens **when someone interacts with system.**

Actor – A user or another system that interacts with the modeled system.

The best way to think of a use case diagram is that it's a graphical table of contents for the use case set. It's also similar to the context diagram used in structured methods, as it shows the system boundary and the interactions with the outside world. The use case diagram shows the actors, the use cases, and the relationships between them:

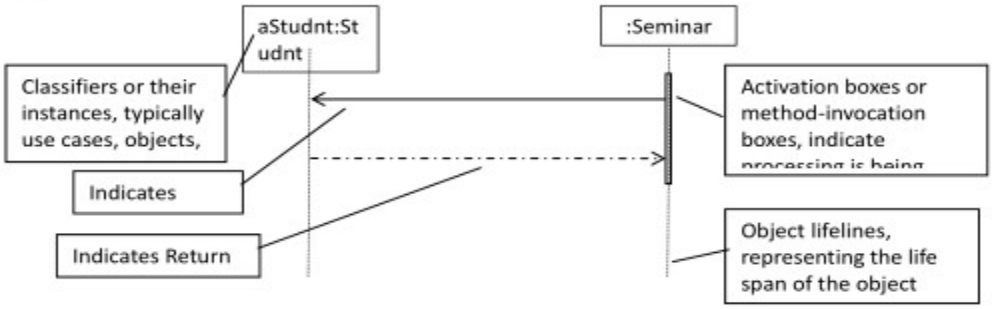
- Which actors carry out which use cases
- Which use cases include other use cases



Sequence Diagram

A *sequence diagram* is an **interaction diagram** that **emphasizes the time ordering of messages** and shows a set of objects and the messages sent and received by those objects.

The objects are typically named or anonymous instances of classes, but may also represent instances of other things, such as collaborations, components, and nodes.



A *sequence diagram* is an **interaction diagram** that **emphasizes the time ordering of messages.**

A sequence diagram shows a set of objects and the messages sent and received by those objects.

The objects are typically named or anonymous instances of classes, but may also represent instances of other things, such as collaborations, components, and nodes.

Waterfall Model:

The Waterfall Model was first Process Model to be introduced. It is also referred to as a **linear-sequential life cycle model**. It is very simple to understand and use. In a waterfall model, each phase must be completed fully before the next phase can begin. This type of software development model is basically used for the for the project which is small and there are no uncertain requirements. At the end of each phase, a review takes place to determine if the project is on the right path and whether or not to continue or discard the project. In this model software testing starts only after the development is complete. In waterfall model phases do not overlap.

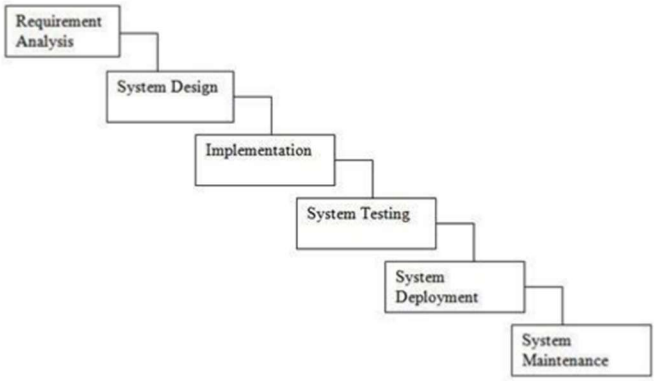


Figure : Waterfall Model

Disadvantages of waterfall model:

- Once an application is in the testing stage, it is very difficult to go back and change something that was not well-thought out in the concept stage.
- No working software is produced until late during the life cycle.
- High amounts of risk and uncertainty.
- Not a good model for complex and object-oriented projects.
- Poor model for long and ongoing projects.
- Not suitable for the projects where requirements are at a moderate to high risk of changing.

Iterative and Evolutionary (Incremental) Development

In this lifecycle approach, development is organized into a series of short, fixed-length (for example, three-week) mini-projects called iterations; the outcome of each is a tested, integrated, and executable partial system. Each iteration includes its own requirements analysis, design, implementation, and testing activities.

The iterative lifecycle is based on the successive enlargement and refinement of a system through multiple iterations, with cyclic feedback and adaptation as core drivers to converge upon a suitable system. The system grows incrementally over time, iteration by iteration, and thus this approach is also known as iterative and incremental development (see Figure below). Because feedback and adaptation evolve the specifications and design, it is also known as iterative and evolutionary development.

Early iterative process ideas were known as spiral development and evolutionary Development [Boehm]

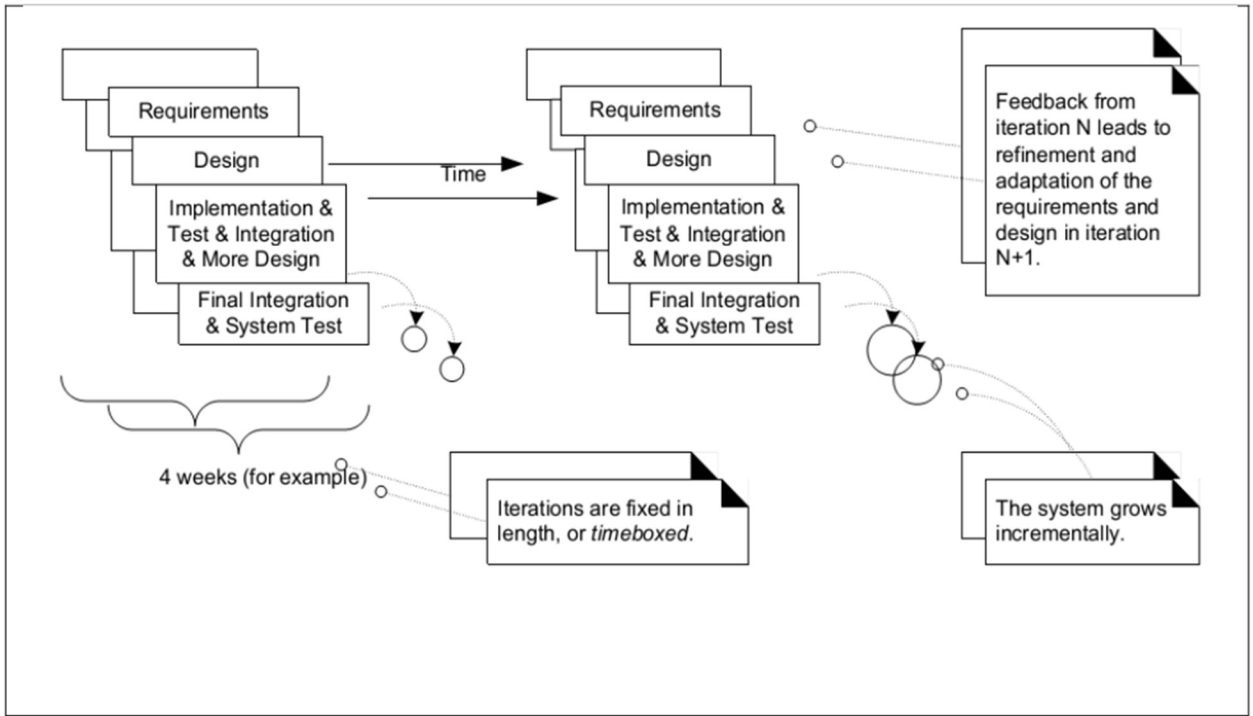


Figure : Iterative and evolutionary development.

Benefits include:

- less project failure, better productivity, and lower defect rates; shown by research into iterative and evolutionary methods

- early rather than late mitigation of high risks (technical, requirements, objectives, usability, and so forth) early visible progress

- early feedback, user engagement, and adaptation, leading to a refined system that more closely meets the real needs of the stakeholders

- managed complexity; the team is not overwhelmed by "analysis paralysis" or very long and complex steps

- the learning within an iteration can be methodically used to improve the development process itself, iteration by iteration

The Unified Process

Until recently, three of the most successful object-oriented methodologies were

– Booch's method

– Jacobson's Objectory

– Rumbaugh's OMT (Object Modeling Technique)

• Today, the unified process is usually the primary choice for object-oriented software production. That is, the unified process is the primary object-oriented methodology

In 1999, Booch, Jacobson, and Rumbaugh published a complete object-oriented analysis and design methodology, which is called **Unified Process**. It unified their three separate methodologies

– **Original name**: Rational Unified Process (RUP)

– Next name: Unified Software Development Process (USDP)

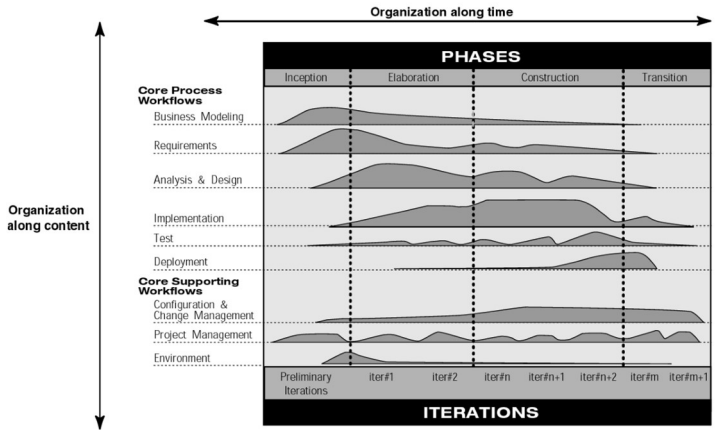
What is the Rational Unified Process?

The Rational Unified Process® is a ***Software Engineering Process***. It provides a disciplined approach to assigning tasks and responsibilities within a development organization. Its goal is to ensure the production of high-quality software that meets the needs of its end-users, within a predictable schedule and budget. [11, 13]

Two Dimensions

The process can be described in two dimensions, or along two axis:

- the horizontal axis represents time and shows the dynamic aspect of the process as it is enacted, and it is expressed in terms of cycles, phases, iterations, and milestones.
- the vertical axis represents the static aspect of the process: how it is described in terms of activities, artifacts, workers and workflows.



The Iterative Model graph shows how the process is structured along two dimensions

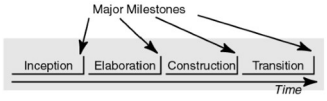
Phases and Iterations - The Time Dimension

This is the dynamic organization of the process along time.

The software lifecycle is broken into *cycle*s, each cycle working on a new generation of the product. The Rational Unified Process divides one development cycle in four consecutive *phases* [10]

- Inception phase
- Elaboration phase
- Construction phase
- Transition phase

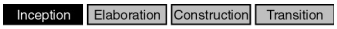
Each phase is concluded with a well-defined *milestone*—a point in time at which certain critical decisions must be made, and therefore key goals must have been achieved [2].



The phases and major milestones in the process.

Each phase has a specific purpose.

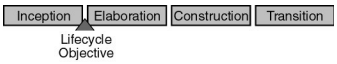
Inception Phase



During the inception phase, you establish the business case for the system and delimit the project scope. To accomplish this you must identify all external entities with which the system

will interact (actors) and define the nature of this interaction at a high-level. This involves identifying all use cases and describing a few significant ones. The business case includes success criteria, risk assessment, and estimate of the resources needed, and a phase plan showing dates of major milestones. [10, 14] The outcome of the inception phase is:

- A vision document: a general vision of the core project's requirements, key features, and main constraints.
- A initial use-case model (10% -20%) complete).
- An initial project glossary (may optionally be partially expressed as a domain model).
- An initial business case, which includes business context, success criteria (revenue projection, market recognition, and so on), and financial forecast.
- An initial risk assessment.
- A project plan, showing phases and iterations.
- A business model, if necessary.
- One or several prototypes. *Milestone : Lifecycle Objectives*



At the end of the inception phase is the first major project milestone: the Lifecycle Objectives Milestone. The evaluation criteria for the inception phase are:

- Stakeholder concurrence on scope definition and cost/schedule estimates.
- Requirements understanding as evidenced by the fidelity of the primary use cases.
- Credibility of the cost/schedule estimates, priorities, risks, and development process.
- Depth and breadth of any architectural prototype that was developed.
- Actual expenditures versus planned expenditures.

The project may be cancelled or considerably re-thought if it fails to pass this milestone.

Elaboration Phase

The purpose of the elaboration phase is to analyze the problem domain, establish a sound architectural foundation, develop the project plan, and eliminate the highest risk elements of the project. To accomplish these objectives, you must have the “mile wide and inch deep” view of the system. Architectural decisions have to be made with an understanding of the whole system: its scope, major functionality and nonfunctional requirements such as performance requirements.

It is easy to argue that the elaboration phase is the most critical of the four phases. At the end of this phase, the hard “engineering” is considered complete and the project undergoes its most important day of reckoning: the decision on whether or not to commit to the construction and transition phases. For most projects, this also corresponds to the transition from a mobile, light and nimble, low-risk operation to a high-cost, high-risk operation with substantial inertia. While the process must always accommodate changes, the elaboration phase activities ensure that the architecture, requirements and plans are stable enough, and the risks are sufficiently mitigated, so you can predictably determine the

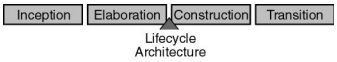
cost and schedule for the completion of the development. Conceptually, this level of fidelity would correspond to the level necessary for an organization to commit to a fixed-price construction phase.

In the elaboration phase, an executable architecture prototype is built in one or more iterations, depending on the scope, size, risk, and novelty of the project. This effort should at least address the critical use cases identified in the inception phase, which typically expose the major technical risks of the project. While an evolutionary prototype of a production-quality component is always the goal, this does not exclude the development of one or more exploratory, throwaway prototypes to mitigate specific risks such as design/requirements trade-offs, component feasibility study, or demonstrations to investors, customers, and end-users.

The outcome of the elaboration phase is:

- A use-case model (at least 80% complete) — all use cases and actors have been identified, and most usecase descriptions have been developed.
- Supplementary requirements capturing the non functional requirements and any requirements that are not associated with a specific use case.
- A Software Architecture Description.
- An executable architectural prototype.
- A revised risk list and a revised business case.
- A development plan for the overall project, including the coarse-grained project plan, showing iterations" and evaluation criteria for each iteration.
- An updated development case specifying the process to be used.
- A preliminary user manual (optional).

Milestone : Lifecycle Architecture



At the end of the elaboration phase is the second important project milestone, the Lifecycle Architecture
Milestone. At this point, you examine the detailed system objectives and scope, the choice of architecture, and the resolution of the major risks.

The main evaluation criteria for the elaboration phase involves the answers to these questions:

- Is the vision of the product stable?
- Is the architecture stable?
- Does the executable demonstration show that the major risk elements have been addressed and credibly resolved?
- Is the plan for the construction phase sufficiently detailed and accurate? Is it backed up with a credible basis of estimates?
- Do all stakeholders agree that the current vision can be achieved if the current plan is executed to develop the complete system, in the context of the current architecture?
- Is the actual resource expenditure versus planned expenditure acceptable?

The project may be aborted or considerably re-thought if it fails to pass this milestone.

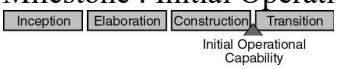
Construction Phase

During the construction phase, all remaining components and application features are developed and integrated into the product, and all features are thoroughly tested. The construction phase is, in one sense, a manufacturing process where emphasis is placed on managing resources and controlling operations to optimize costs, schedules, and quality. In this sense, the management mindset undergoes a transition from the development of intellectual property during inception and elaboration, to the development of deployable products during construction and transition.

Many projects are large enough that parallel construction increments can be spawned. These parallel activities can significantly accelerate the availability of deployable releases; they can also increase the complexity of resource management and workflow synchronization. A robust architecture and an understandable plan are highly correlated. In other words, one of the critical qualities of the architecture is its ease of construction. This is one reason why the balanced development of the architecture and the plan is stressed during the elaboration phase. The outcome of the construction phase is a product ready to put in hands of its end-users. At minimum, it consists of:

- The software product integrated on the adequate platforms.
- The user manuals.
- A description of the current release.

Milestone : Initial Operational Capability



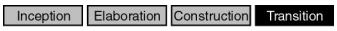
At the end of the construction phase is the third major project milestone (Initial Operational Capability Milestone). At this point, you decide if the software, the sites, and the users are ready to go operational, without exposing the project to high risks. This release is often called a “beta” release.

The evaluation criteria for the construction phase involve answering these questions:

- Is this product release stable and mature enough to be deployed in the user community?
- Are all stakeholders ready for the transition into the user community?
- Are the actual resource expenditures versus planned expenditures still acceptable?

Transition may have to be postponed by one release if the project fails to reach this milestone.

Transition Phase



The purpose of the transition phase is to transition the software product to the user community. Once the product has been given to the end user, issues usually arise that require you to develop new releases, correct some problems, or finish the features that were postponed.

The transition phase is entered when a baseline is mature enough to be deployed in the end-user domain. This typically requires that some usable subset of the system has been completed to an acceptable level of quality and that user documentation is available so that the transition to the user will provide positive results for all parties.

This includes:

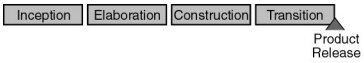
- “beta testing” to validate the new system against user expectations
- parallel operation with a legacy system that it is replacing
- conversion of operational databases
- training of users and maintainers
- roll-out the product to the marketing, distribution, and sales teams

The transition phase focuses on the activities required to place the software into the hands of the users. Typically, this phase includes several iterations, including beta releases, general availability releases, as well as bug-fix and enhancement releases. Considerable effort is expended in developing user-oriented documentation, training users, supporting users in their initial product use, and reacting to user feedback. At this point in the lifecycle, however, user feedback should be confined primarily to product tuning, configuring, installation, and usability issues.

The primary objectives of the transition phase include:

- Achieving user self-supportability
- Achieving stakeholder concurrence that deployment baselines are complete and consistent with theevaluation criteria of the vision
- Achieving final product baseline as rapidly and cost effectively as practical

This phase can range from being very simple to extremely complex, depending on the type of product. For example, a new release of an existing desktop product may be very simple, whereas replacing a nation's air-traffic control system would be very complex. *Milestone: Product Release*



At the end of the transition phase is the fourth important project milestone, the Product Release Milestone. At this point, you decide if the objectives were met, and if you should start another development cycle. In some cases, this milestone may coincide with the end of the inception phase for the next cycle.

The primary evaluation criteria for the transition phase involve the answers to these questions:

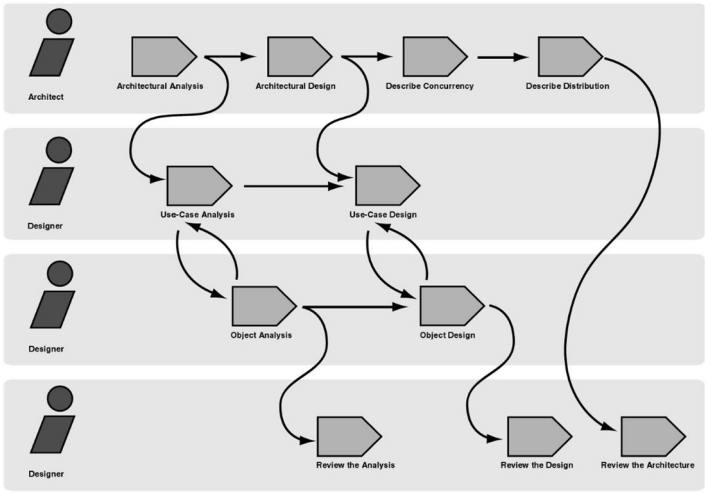
- Is the user satisfied?
- Are the actual resources expenditures versus planned expenditures still acceptable?

Workflows

A mere enumeration of all workers, activities and artifacts does not quite constitute a process. We need a way to describe meaningful sequences of activities that produce some valuable result, and to show interactions between workers.

A *workflow* is a sequence of activities that produces a result of observable value.

In UML terms, a workflow can be expressed as a sequence diagram, a collaboration diagram, or an activity diagram. We use a form of activity diagrams in this white paper.



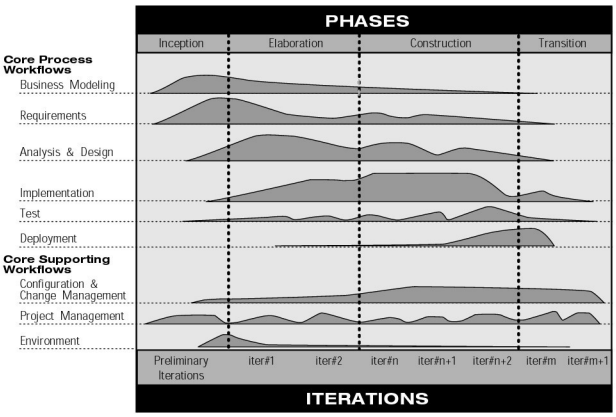
Example of workflow

Note that it is not always possible or practical to represent all of the dependencies between activities. Often two activities are more tightly interwoven than shown, especially when they involve the same worker or the same individual. People are not machines, and the workflow cannot be interpreted literally as a program for people, to be followed exactly and mechanically.

In the next section we will discuss the most essential type of workflows in the process, called Core Workflows.

Core workflows

There are nine *core process workflows* in the Rational Unified Process, which represent a partitioning of all workers and activities into logical groupings.



The nine core process workflows

The core process workflows are divided into six core "engineering" workflows:

1. Business modeling workflow
2. Requirements workflow
3. Analysis & Design workflow
4. Implementation workflow
5. Test workflow
6. Deployment workflow

And three core "supporting" workflows:

1. Project Management workflow
2. Configuration and Change Management workflow
3. Environment workflow

Although the names of the six core engineering workflows may evoke the sequential phases in a traditional waterfall process, we should keep in mind that the phases of an iterative process are different and that these workflows are revisited again and again throughout the lifecycle. The actual complete workflow of a project interleaves these nine core workflows, and repeats them with various emphasis and intensity at each iteration.

Business Modeling

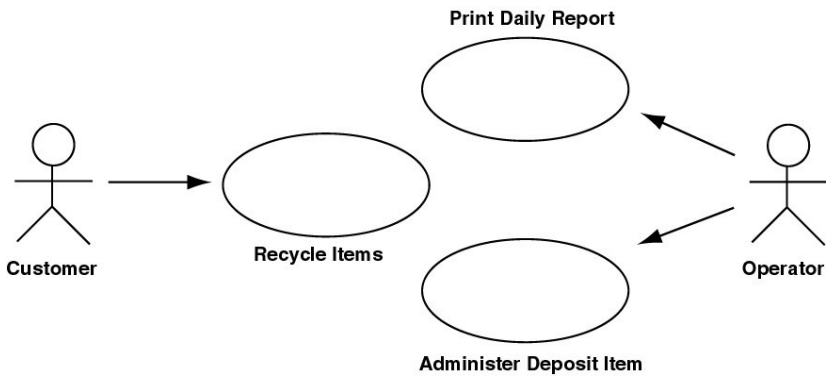
One of the major problems with most business engineering efforts, is that the software engineering and the business engineering community do not communicate properly with each other. This leads to the output from business engineering is not being used properly as input to the software development effort, and vice-versa. The Rational Unified Process addresses this by providing a common language and process for both communities, as well as showing how to create and maintain direct traceability between business and software models.

In Business Modeling we document business processes using so called business use cases. This assures a common understanding among all stakeholders of what business process needs to be supported in the organization. The business use cases are analyzed to understand how the business should support the business processes. This is documented in a business object-model. Many projects may choose not to do business modeling.

Requirements

The goal of the Requirements workflow is to describe *what* the system should do and allows the developers and the customer to agree on that description. To achieve this, we elicit, organize, and document required functionality and constraints; track and document tradeoffs and decisions.

A Vision document is created, and stakeholder needs are elicited. *Actors* are identified, representing the users, and any other system that may interact with the system being developed. *Use cases* are identified, representing the behavior of the system. Because use cases are developed according to the actor's needs, the system is more likely to be relevant to the users. The following figure shows an example of a use-case model for a recycling-machine system.



An example of use-case model with actors and use cases.

Each use case is described in detail. The *use-case description* shows how the system interacts step by step with the actors and what the system does. Non-functional requirements are described in *Supplementary* Specifications.

The use cases function as a unifying thread throughout the system's development cycle. The same use-case model is used during requirements capture, analysis & design, and test.

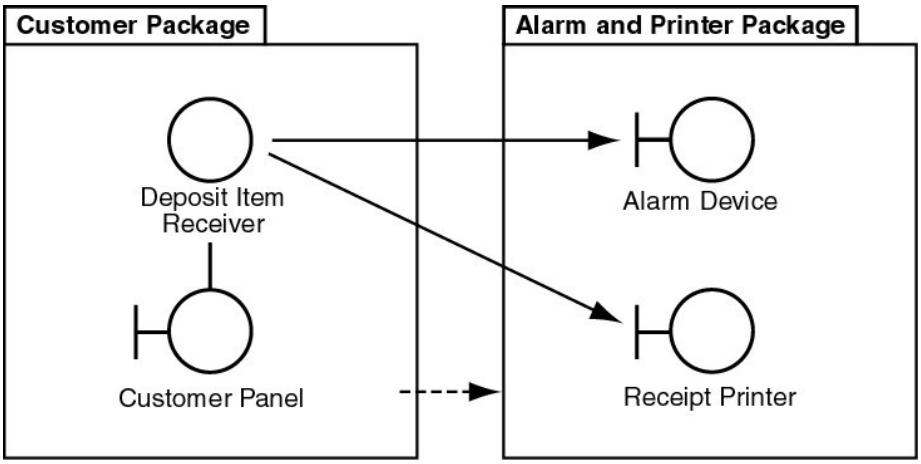
Analysis & Design

The goal of the Analysis & Design workflow is to show *how* the system will be *realized* in the implementation phase. You want to build a system that:

- Performs—in a specific implementation environment—the tasks and functions specified in the use-case descriptions.
- Fulfills all its requirements.
- Is structured to be robust (easy to change if and when its functional requirements change).

Analysis & Design results in a *design model* and optionally an *analysis model*. The design model serves as an abstraction of the source code; that is, the design model acts as a 'blueprint' of how the source code is structured and written.

The design model consists of design classes structured into design packages and design subsystems with welldefined interfaces, representing what will become components in the implementation. It also contains descriptions of how objects of these design classes collaborate to perform use cases. The next figure shows part of a sample design model for the recycling-machine system in the use-case model shown in the previous figure.



Part of a design model with communicating design classes, and package group design classes.

The design activities are centered around the notion of *architecture.* The production and validation of this architecture is the main focus of early design iterations. Architecture is represented by a number of architectural views [9]. These views capture the major structural design decisions. In essence, architectural views are abstractions or simplifications of the entire design, in which important characteristics are made more visible by leaving details aside. The architecture is an important vehicle not only for developing a good design model, but also for increasing the quality of any model built during system development.

Implementation

The purpose of implementation is:

- To define the organization of the code, in terms of implementation subsystems organized in layers.
- To implement classes and objects in terms of components (source files, binaries, executables, and others).
- To test the developed components as units.
- To integrate the results produced by individual implementers (or teams), into an executable system.

The system is realized through implementation of components. The Rational Unified Process describes how you reuse existing components, or implement new components with well defined responsibility, making the system easier to maintain, and increasing the possibilities to reuse.

Components are structured into Implementation Subsystems. Subsystems take the form of directories, with additional structural or management information. For example, a subsystem can be created as a directory or a folder in a file system, or a subsystem in Rational/Apex for C++ or Ada, or packages using Java.™

Test

The purposes of testing are:

- To verify the interaction between objects.
- To verify the proper integration of all components of the software.
- To verify that all requirements have been correctly implemented.
- To identify and ensure defects are addressed prior to the deployment of the software.

The Rational Unified Process proposes an iterative approach, which means that you test throughout the project. This allows you to find defects as early as possible, which radically reduces the cost of fixing the defect. Tests are carried out along three quality dimensions reliability, functionality, application performance and system performance. For each of these quality dimensions, the process describes how you go through the test lifecycle of planning, design, implementation, execution and evaluation.

Strategies for when and how to automate test are described. Test automation is especially important using an iterative approach, to allow regression testing at then end of each iteration, as well as for each new version of the product.

Deployment

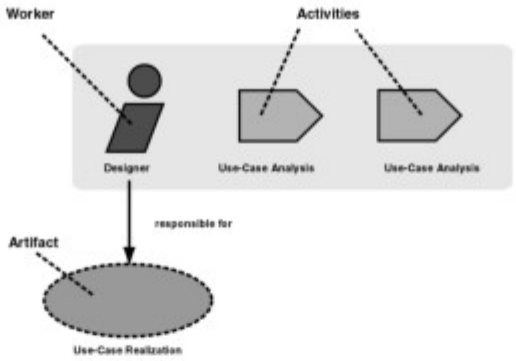
The purpose of the deployment workflow is to successfully produce product releases, and deliver the software to its end users. It covers a wide range of activities including:

- Producing external releases of the software.
- Packaging the software.
- Distributing the software.
- Installing the software.
- Providing help and assistance to users.
- In many cases, this also includes activities such as:
- Planning and conduct of beta tests.
- Migration of existing software or data.
- Formal acceptance.

Although deployment activities are mostly centered around the transition phase, many of the activities need to be included in earlier phases to prepare for deployment at the end of the construction phase.
The Deployment and Environment workflows of the Rational Unified Process contain less detail than other workflows.

Components of RUP: Activities, Disciplines, Artifacts, Role or Worker

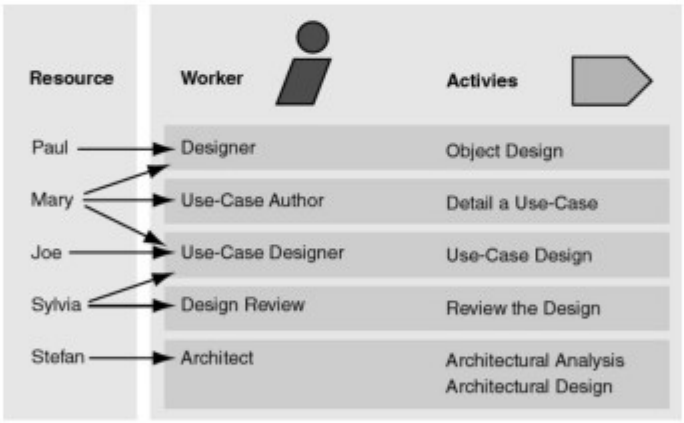
Activities, Artifacts, and Workers



Workers, activites, and artifacts.

Worker

A ***worker*** defines the behavior and responsibilities of an individual, or a group of individuals working together as a team. You could regard a worker as a "hat" an individual can wear in the project. One individual may wear many different hats. This is an important distinction because it is natural to think of a worker as the individual or team itself, but in the Unified Process the worker is more the role defining how the individuals should carry out the work. The responsibilities we assign to a worker includes both to perform a certain set of activities as well as being owner of a set of artifacts.



People and Workers

Activity

An **activity** of a specific worker is a unit of work that an individual in that role may be asked to perform.
The activity has a clear purpose, usually expressed in terms of creating or updating some artifacts, such as a model, a class, a plan. Every activity is assigned to a specific worker. The granularity of an activity is generally a few hours to a few days, it usually involves one worker, and affects one or only a small number of artifacts. An activity should be usable as an element of planning and progress; if it is too small, it will be neglected, and if it is too large, progress would have to be expressed in terms of an activity's parts.

Example of activities:

- **Plan an iteration**, for the Worker: Project Manager
- **Find use cases and actors**, for the Worker: System Analyst
- **Review the design**, for the Worker: Design Reviewer
- **Execute performance test**, for the Worker: Performance Tester

Artifact

An artifact is a piece of information that is produced, modified, or used by a process. Artifacts are the tangible products of the project, the things the project produces or uses while working towards the final product. Artifacts are used as input by workers to perform an activity, and are the result or output of such activities. In object-oriented design terms, as activities are operations on an active object (the worker), artifacts are the parameters of these activities.

- Artifacts may take various shapes or forms:
- A model, such as the Use-Case Model or the Design Model
- A model element, i.e. an element within a model, such as a class, a use case or a subsystem
- A document, such as Business Case or Software Architecture Document
- Source code
- Executables

The Role of Disciplines in the Software Engineering Process and the RUP

According to one of the historical definitions, the term *discipline* is a field of study that allows us to learn about that field in detail. Software engineering can be considered one of the many disciplines of engineering. In the RUP, however, a discipline refers to a specific area of concern (or a field of study, as mentioned earlier) within software engineering. For instance, Analysis and Design is one of the disciplines in RUP, which is itself a field of study and requires dedicated learning and distinct skill-sets. In addition, disciplines in RUP allow you to govern the activities you perform within that discipline. A discipline in RUP gives you all the guidance you require to learn not only when to perform a given activity but also how to perform it. Therefore, disciplines in RUP allow you to bring closely related activities under control.

Rational Unified Process – Main Characteristics

- Iterative and incremental
- Use-case-driven
- Architecture-centric
- Uses UML as its modeling notation
- Process framework
  - Comprehensive set of document templates, process workflow templates, and process guidelines
  - Distributed by IBM/Rational on a CD

Four Ps of Software Development

1) Explain management spectrum or explain 4 p's of software system.

Effective software project management focuses on the four P's: people, product, process, and project.

The People

- People factor is very much important in the process of software development.
- There are following areas for software people like, recruiting, selection, performance management, training, compensation, career development, organization and work design, and team/culture development.
- Organizations achieve high levels of maturity in the people management area.

The Product

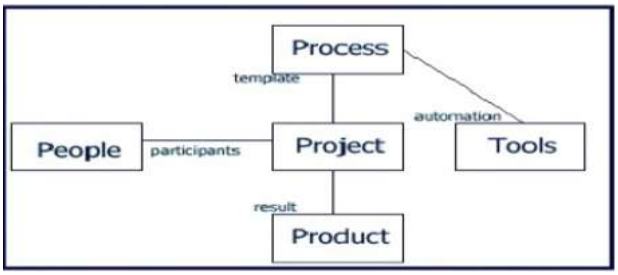
- Before a project can be planned, product objectives and scope should be established, alternative solutions should be considered and technical and management constraints should be identified.
- Without this information, it is impossible to define reasonable estimates of the cost, an effective assessment of risk, a realistic breakdown of project tasks, or a manageable project schedule.
- Objectives identify the overall goals for the product without considering how these goals will be achieved.
- Scope identifies the primary data, functions and behaviours that characterize the product.
- Once the product objectives and scope are understood, alternative solutions are considered. From the available various alternatives, managers and practitioners select a "best" approach.

The Process

- A software process provides the framework from which a comprehensive plan for software development can be established.
- A small number of frame-work activities are applicable to all software projects, regardless of their size or complexity.
- A number of different tasks, milestones, work products and quality assurance points enable the framework activities to be adapted to the characteristics of the software project and the requirements of the project team.
- Finally, umbrella activities such as software quality assurance, software configuration management, and measurement overlay the process model.

The Project

- We conduct planned and controlled software projects for one primary reason it is the only known way to manage complexity.
- A software project manager and the software engineers who build the product must avoid a set of common warning signs, understand the critical success factors that lead to good project management, and develop a common sense approach for planning, monitoring and controlling the project.



Design Pattern

According to GoF definition of design patterns:
"In software engineering, a software design pattern is a general reusable solution to a commonly occurring problem within a given context in software design. Design patterns are formalized best practices that the programmer can use to solve common problems when designing an application or system."

A design pattern is a general repeatable solution to a commonly occurring problem in software design. A design pattern isn't a finished design that can be transformed directly into code. It is a description or template for how to solve a problem that can be used in many different situations. The pattern is not a specific piece of code, but a general concept for solving a particular problem. We can follow the pattern details and implement a solution that suits the realities of our own program

Programming Paradigm Vs Design Pattern:
Programming paradigm is a method, a way, a principle of programming. It describes the programming process, which is the way programs are made. It explains core structure of program written in certain paradigm, everything that program consists of and its components. Some of programming paradigms: Procedural paradigm, functional paradigm, object-oriented paradigm, modular programming, and structured paradigm etc.

According to GoF definition of design patterns:
"In software engineering, a software design pattern is a general reusable solution to a commonly occurring problem within a given context in software design. Design patterns are formalized best practices that the programmer can use to solve common problems when designing an application or system."
Design patterns are formalized solutions to common programming problems. They mostly refer to object oriented programming, but some of solutions can be applied in various paradigms.

Importance of Design Patterns:

1. Off-the-shelf solution-forms for the common problem-forms.
2. Teaches how to solve all sorts of problems using the principles of object-oriented design.
3. Helps in learning design and rationale behind project, enhances communication and insight.
4. Reusing design patterns helps to prevent subtle issues that can cause major problems.
5. Improves code readability for coders and architects familiar with the patterns.
6. Design patterns provide general solutions, documented in a format that doesn't require specifics tied to a particular problem.
7. Speed up the development process by providing well tested, proven development/design paradigm.

In software engineering, **creational design patterns** are design patterns that deal with object creation mechanisms, trying to create objects in a manner suitable to the situation. The basic form of object creation could result in design problems or in added complexity to the design. Creational design patterns solve this problem by somehow controlling this object creation.

Creational design patterns are composed of two dominant ideas. One is encapsulating knowledge about which concrete classes the system uses. Another is hiding how instances of these concrete classes are created and combined.[1]

Creational design patterns are further categorized into object-creational patterns and Class-creational patterns, where Object-creational patterns deal with Object creation and Class-creational patterns deal with Class-instantiation. In greater details, Object-creational patterns defer part of its object creation to another object, while Class-creational patterns defer its object creation to subclasses.[2]

Five well-known design patterns that are parts of creational patterns are the

- Abstract factory pattern, which provides an interface for creating related or dependent objects without specifying the objects' concrete classes.[3]
- Builder pattern, which separates the construction of a complex object from its representation so that the same construction process can create different representations.
- Factory method pattern, which allows a class to defer instantiation to subclasses.[4]
- Prototype pattern, which specifies the kind of object to create using a prototypical instance, and creates new objects by cloning this prototype.
- Singleton pattern, which ensures that a class only has one instance, and provides a global point of access to it.[5]

In software engineering, **structural design patterns** are design patterns that ease the design by identifying a simple way to realize relationships among entities.

Examples of Structural Patterns include:

- Adapter pattern: 'adapts' one interface for a class into one that a client expects
  - Adapter pipeline: Use multiple adapters for debugging purposes.[1]
  - Retrofit Interface Pattern:[2][3] An adapter used as a new interface for multiple classes at the same time.
- Aggregate pattern: a version of the Composite pattern with methods for aggregation of children
- Bridge pattern: decouple an abstraction from its implementation so that the two can vary independently
  - Tombstone: An intermediate "lookup" object contains the real location of an object.[4]
- Composite pattern: a tree structure of objects where every object has the same interface
- Decorator pattern: add additional functionality to an object at runtime where subclassing would result in an exponential rise of new classes
- Extensibility pattern: a.k.a. Framework - hide complex code behind a simple interface
- Facade pattern: create a simplified interface of an existing interface to ease usage for common tasks
- Flyweight pattern: a large quantity of objects share a common properties object to save space
- Marker pattern: an empty interface to associate metadata with a class.
- Pipes and filters: a chain of processes where the output of each process is the input of the next

In software engineering, **behavioral design patterns** are design patterns that identify common communication patterns among objects and realize these patterns. By doing so, these patterns increase flexibility in carrying out this communication.

Examples of this type of design pattern include:

- Blackboard design pattern: provides a computational framework for the design and implementation of systems that integrate large and diverse specialized modules, and implement complex, non-deterministic control strategies
- Chain of responsibility pattern: Command objects are handled or passed on to other objects by logic-containing processing objects
- Command pattern: Command objects encapsulate an action and its parameters
- "Externalize the stack": Turn a recursive function into an iterative one that uses a stack[1]
- Interpreter pattern: Implement a specialized computer language to rapidly solve a specific set of problems
- Iterator pattern: Iterators are used to access the elements of an aggregate object sequentially without exposing its underlying representation
- Mediator pattern: Provides a unified interface to a set of interfaces in a subsystem

In software engineering, **concurrency patterns** are those types of design patterns that deal with the multi-threaded programming paradigm. Examples of this class of patterns include:

- Active Object[1][2]
- Balking pattern
- Barrier
- Double-checked locking
- Guarded suspension
- Leaders/followers pattern
- Monitor Object
- Nuclear reaction
- Reactor pattern

Structural Design Pattern

Structural Design Patterns

- concerned with how classes and objects can be composed, to form larger structures.
- simplifies the structure by identifying the relationships.
- focus on, how the classes inherit from each other and how they are composed from other classes.

Adapter Pattern

converts the interface of a class into another interface that a client wants i.e. to provide the interface according to client requirement while using the services of a class with a different interface.

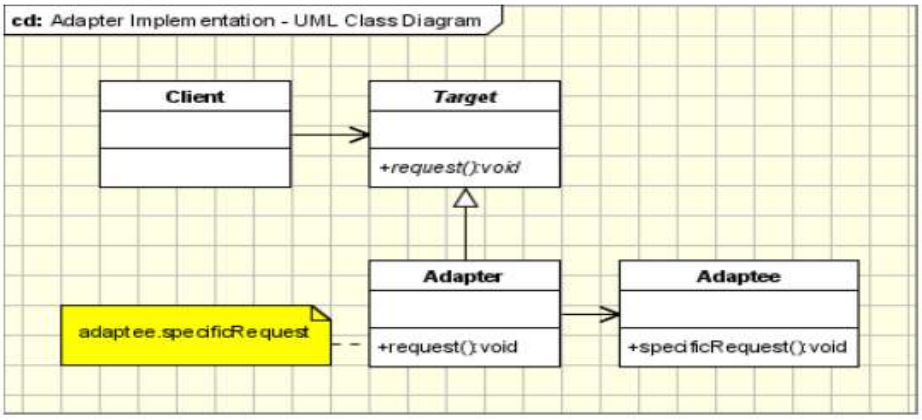
Advantages

- It allows two or more previously incompatible objects to interact.
- It allows reusability of existing functionality.

Uses

It is used:

- When an object needs to utilize an existing class with an incompatible interface.
- When you want to create a reusable class that cooperates with classes which don't have compatible interfaces.



The classes/objects participating in adapter pattern:

- **Target** - defines the domain-specific interface that Client uses.
- **Adapter** - adapts the interface Adaptee to the Target interface.
- **Adaptee** - defines an existing interface that needs adapting.
- **Client** - collaborates with objects conforming to the Target interface.

Composite Pattern

allow clients to operate in generic manner on objects that may or may not represent a hierarchy of objects

The intent of this pattern is to compose objects into tree structures to represent part-whole hierarchies.

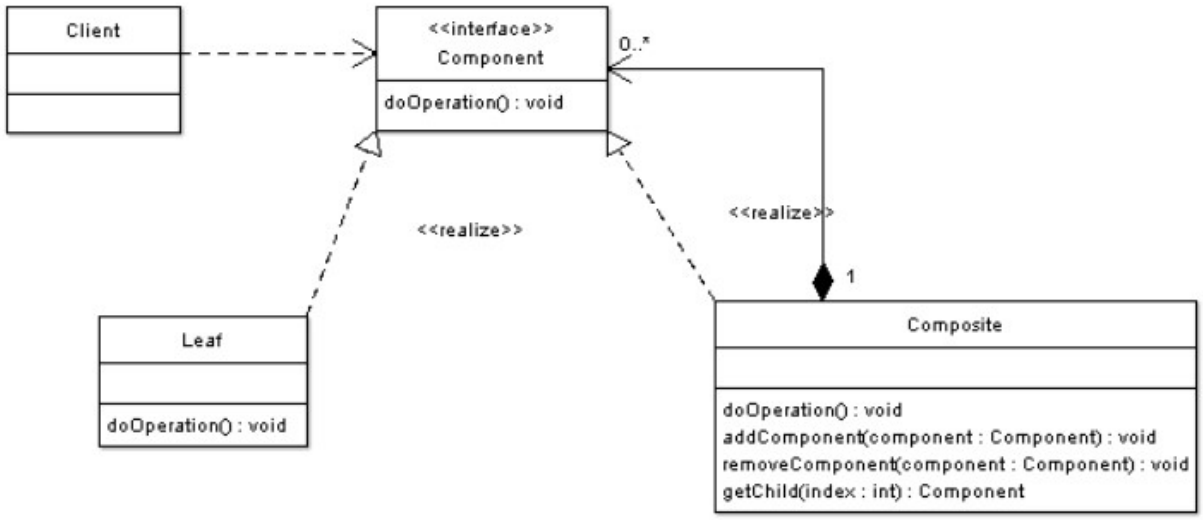
Composite lets clients treat individual objects and compositions of objects uniformly.

Advantages

- It defines class hierarchies that contain primitive and complex objects.
- It makes easier to you to add new kinds of components.
- It provides flexibility of structure with manageable class or interface.

Uses

- When you want to represent a full or partial hierarchy of objects.
- When the responsibilities are needed to be added dynamically to the individual objects without affecting other objects. Where the responsibility of object may vary from time to time.



Documenting and Describing Patterns

- **Pattern Name and Classification:** A descriptive and unique name that helps in identifying and referring to the pattern.
- **Intent:** A description of the goal behind the pattern and the reason for using it.
- **Also Known As:** Other names for the pattern.
- **Motivation (Forces):** A scenario consisting of a problem and a context in which this pattern can be used.
- **Applicability:** Situations in which this pattern is usable; the context for the pattern.
- **Structure:** A graphical representation of the pattern. Class diagrams and Interaction diagrams may be used for this purpose.
- **Participants:** A listing of the classes and objects used in the pattern and their roles in the design.
- **Collaboration:** A description of how classes and objects used in the pattern interact with each other.
- **Consequences:** A description of the results, side effects, and trade offs caused by using the pattern.
- **Implementation:** A description of an implementation of the pattern; the solution part of the pattern.
- **Sample Code:** An illustration of how the pattern can be used in a programming language
- **Known Uses:** Examples of real usages of the pattern.
- **Related Patterns:** Other patterns that have some relationship with the pattern; discussion of the differences between the pattern and similar patterns.

Criticism

The concept of design patterns has been criticized by some in the field of computer science.

Does not differ significantly from other abstractions

Some authors allege that design patterns don't differ significantly from other forms of abstraction, and that the use of new terminology (borrowed from the architecture community) to describe existing phenomena in the field of programming is unnecessary.

The Model-View-Controller paradigm is touted as an example of a "pattern" which predates the concept of "design patterns" by several years. It is further argued by some that the primary contribution of the Design Patterns community (and the Gang of Four book) was the use of Alexander's pattern language as a form of documentation; a practice which is often ignored in the literature.

Leads to inefficient solutions

The idea of a design pattern is an attempt to standardize what are already accepted best practices. In principle this might appear to be beneficial, but in practice it often results in the unnecessary duplication of code. It is almost always a more efficient solution to use a well-factored implementation rather than a "just barely good enough" design pattern.

Lacks formal foundations

The study of design patterns has been excessively ad hoc, and some have argued that the concept sorely needs to be put on a more formal footing. At OOPSLA 1999, the Gang of Four were (with their full cooperation) subjected to a show trial, in which they were "charged" with numerous crimes against computer science. They were "convicted" by ⅔ of the "jurors" who attended the trial.

Targets the wrong problem

The need for patterns results from using computer languages or techniques with insufficient abstraction ability. Under ideal factoring, a concept should not be copied, but merely referenced. But if something is referenced instead of copied,

then there is no "pattern" to label and catalog. Paul Graham writes in the essay Revenge of the Nerds.

Peter Norvig provides a similar argument. He demonstrates that 16 out of the 23 patterns in the Design Patterns book (which is primarily focused on C++) are simplified or eliminated (via direct language support) in Lisp or Dylan.

Inefficient solutions

Patterns try to systematize approaches that are already widely used. This unification is viewed by many as a belief and they implement patterns "to the point", without adapting them to the context of their project.

Unjustified use

"If all you have is a hammer, everything looks like a nail."

This is the problem that haunts many novices who have just familiarized themselves with patterns. Having learned about patterns, they try to apply them everywhere, even in situations where simpler code would do just fine.

2. **Structural Pattern:** explain how to assemble objects and classes into larger structures, while keeping the structures flexible and efficient.
    - Adapter
      Match interfaces of different classes
    - Bridge
      Separates an object's interface from its implementation
    - Composite
      A tree structure of simple and composite objects
    - Decorator
      Add responsibilities to objects dynamically
    - Facade
      A single class that represents an entire subsystem
    - Flyweight
      A fine-grained instance used for efficient sharing
    - Proxy
      An object representing another object
    - Private Class Data
      Restricts accessor/mutator access

Introduction to software architecture:

Software architecture is the process of converting software characteristics such as flexibility, scalability, feasibility, reusability, and security into a structured solution that meets the technical and the business expectations.

Software architecture is the process of designing the global organization of a software system, including dividing software into subsystems, deciding how these will interact, and determining their interfaces.

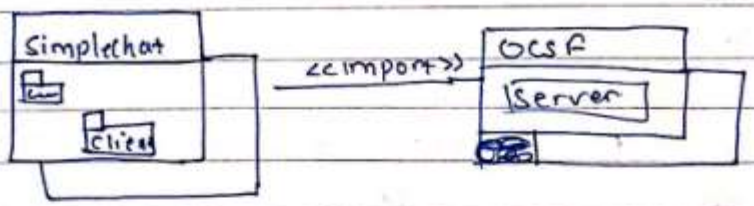
The architecture is the core of the design, so all software engineers need to understand it.

The architectural model will often constrain the overall efficiency, reusability and maintainability of the system.

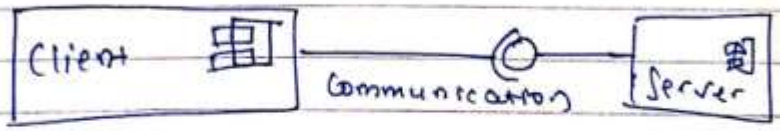
Poor decisions made while creating this model will constrain subsequent design.

* Describing an architecture Using UML:

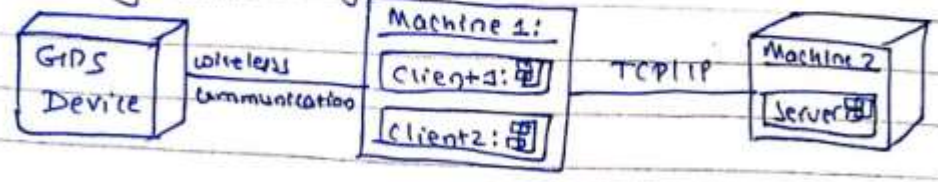
- Package Diagram



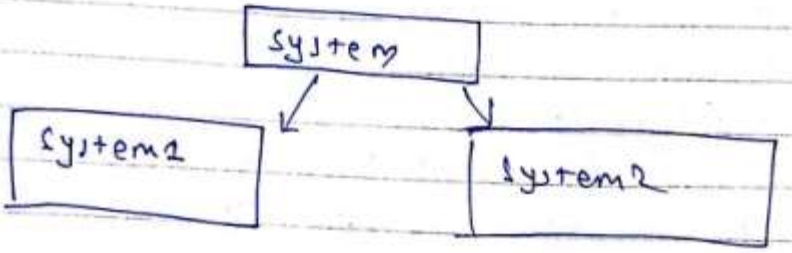
- Component Diagram



- Deployment Diagram



- Subsystem Diagram



4(b) Describe suitable design pattern for following problem. "What is the best way to represent related objects (occurence) in a class diagram?"

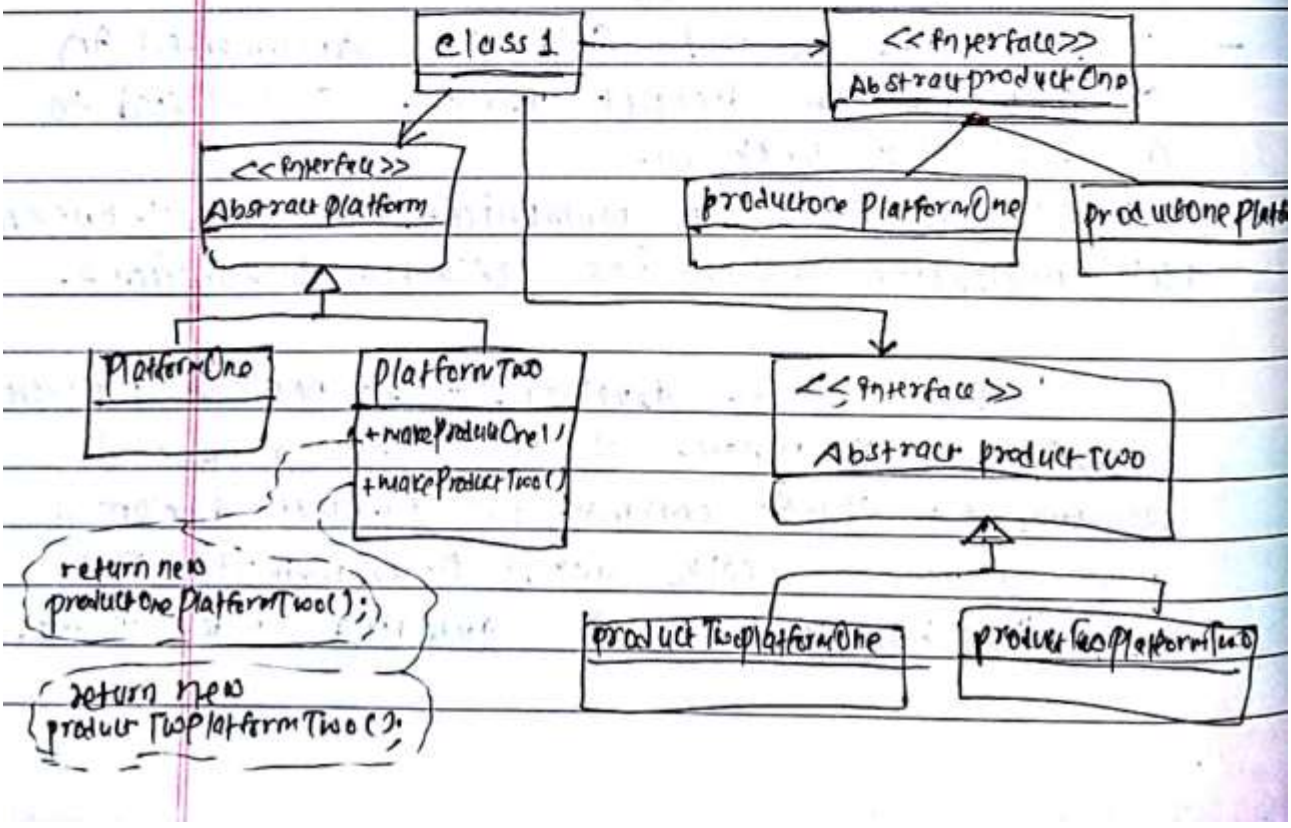
Sol? As looking at the problem, the best solution or design pattern is Abstract Factory. It is so because

→ It provides an interface for creating families of related or dependent objects without specifying their concrete classes.

→ A hierarchy that encapsulates: many possible "platforms" and the construction of a suite of "products".

→ The new operator considered harmful.

So, the structure will be,



The Model View Controller Architectural Patterns

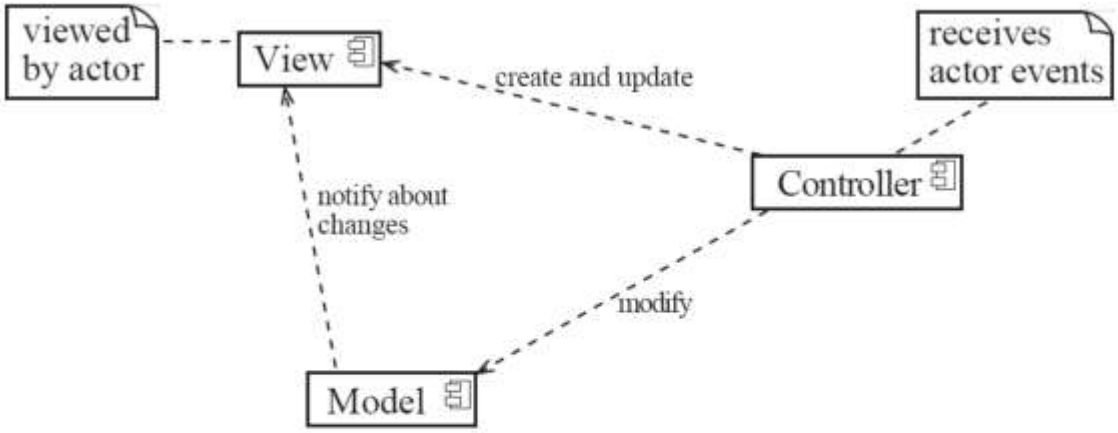
This pattern, also known as MVC pattern, divides an interactive application in to 3 parts as,

1. **model** — contains the core functionality and data, contains the underlying classes whose instances are to be viewed and manipulated.
2. **view** — displays the information to the user (more than one view may be defined), contains objects used to render the appearance of the data from the model in the user interface.
3. **controller** — handles the input from the user, contains the objects that control and handle the user's interaction with the view and the model.

This is done to separate internal representations of information from the ways information is presented to, and accepted from, the user. It decouples components and allows efficient code reuse.

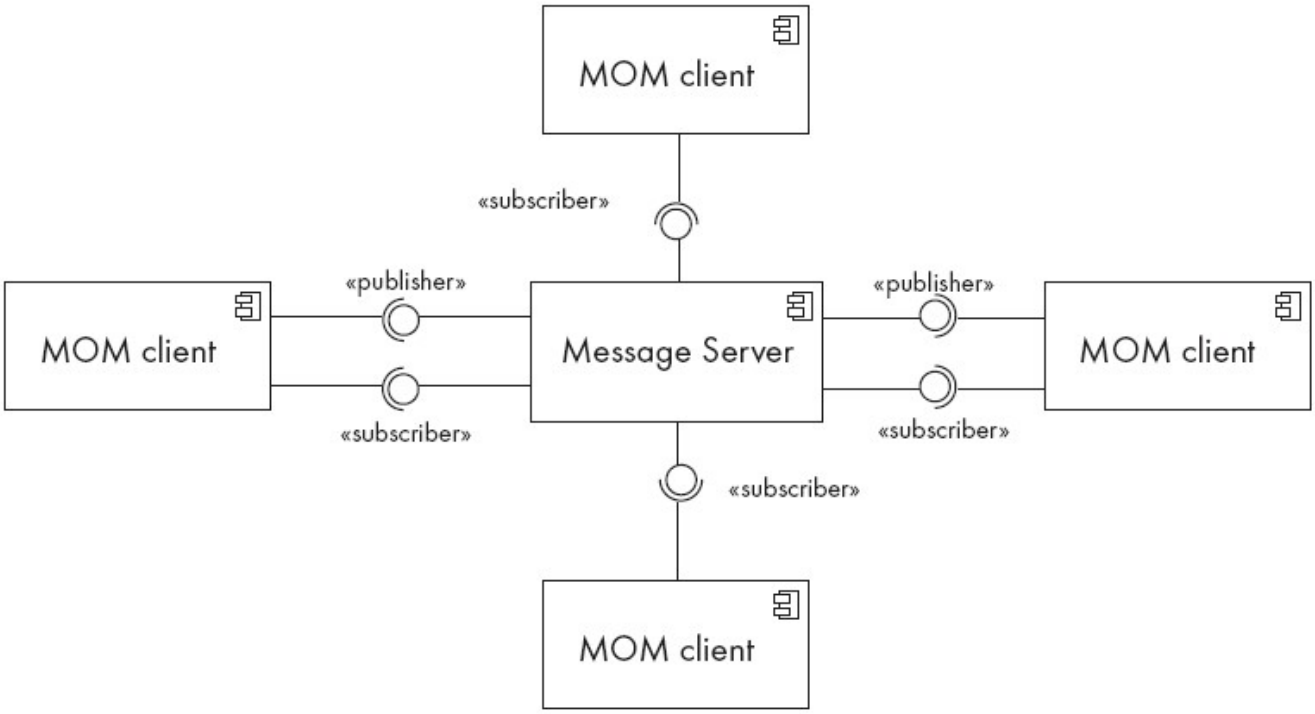
Usage

- Architecture for World Wide Web applications in major programming languages.
- Web frameworks such as **Django** and **Rails**.



Example of MVC in web architecture

- The *View* component generates the HTML code to be displayed by the browser.
- The *Controller* is the component that interprets 'HTTP post' transmissions coming back from the browser.
- The *Model* is the underlying system that manages the information.



The Message Oriented Architectural Patterns

Also known as Message-oriented Middleware (MOM).

This architecture is based on the idea that since humans can communicate and collaborate to accomplish some task by exchanging emails or instant messages, then software applications should also be able to operate in a similar manner i.e. the different sub-systems communicate and collaborate to accomplish some task only by exchanging messages.

The core of this architecture is an application-to-application messaging system

Senders and receivers need only to know what are the message formats