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.

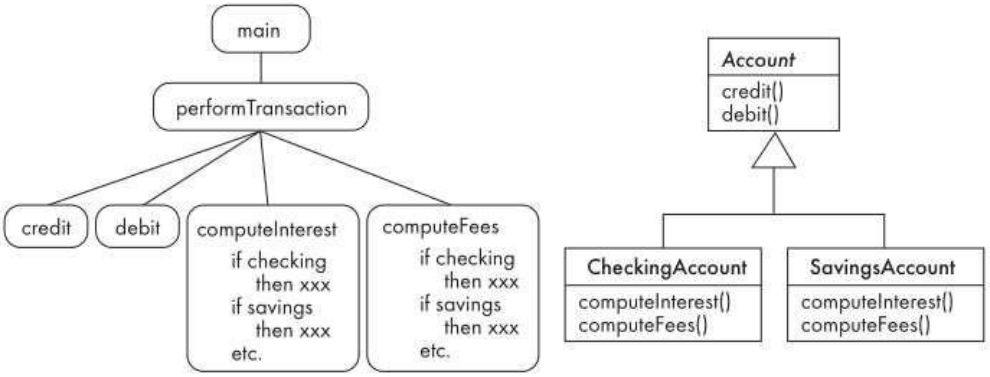


Figure 2.1 **Organizing a system according to the procedural paradigm (left) or the object-oriented paradigm (right). The UML notation used in the right-hand diagram will be discussed in more detail later**

1. Activities performed in object-oriented analysis

1. Modeling the functions of the system.
2. Finding and identifying the business objects.
3. Organizing the objects and identifying their relationships.
4. Modeling the behavior of the objects.

Designing concepts

- Defining objects, creating class diagram from conceptual diagram: Usually map entity to class.
- Identifying attributes.
- Use design patterns (if applicable): A design pattern is not a finished design, it is a description of a solution to a common problem, in a context.[1] The main advantage of using a design pattern is that it can be reused in multiple applications. It can also be thought of as a template for how to solve a problem that can be used in many different situations and/or applications. Object-oriented design patterns typically show relationships and interactions between classes or objects, without specifying the final application classes or objects that are involved.
- Define application framework (if applicable): Application framework is usually a set of libraries or classes that are used to implement the standard structure of an application for a specific operating system. By bundling a large amount of reusable code into a framework, much time is saved for the developer, since he/she is saved the task of rewriting large amounts of standard code for each new application that is developed.
- Identify persistent objects/data (if applicable): Identify objects that have to last longer than a single runtime of the application. If a relational database is used, design the object relation mapping.
- Identify and define remote objects (if applicable).

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.

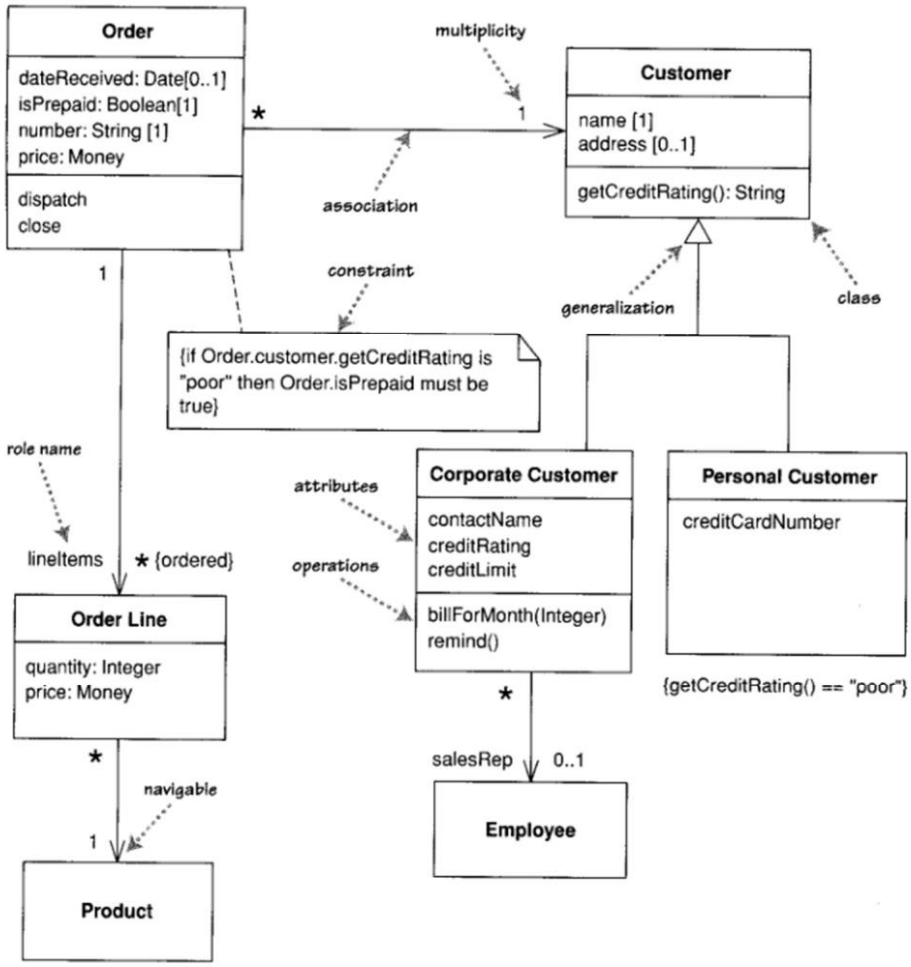


Figure 3.1 *A simple class diagram*

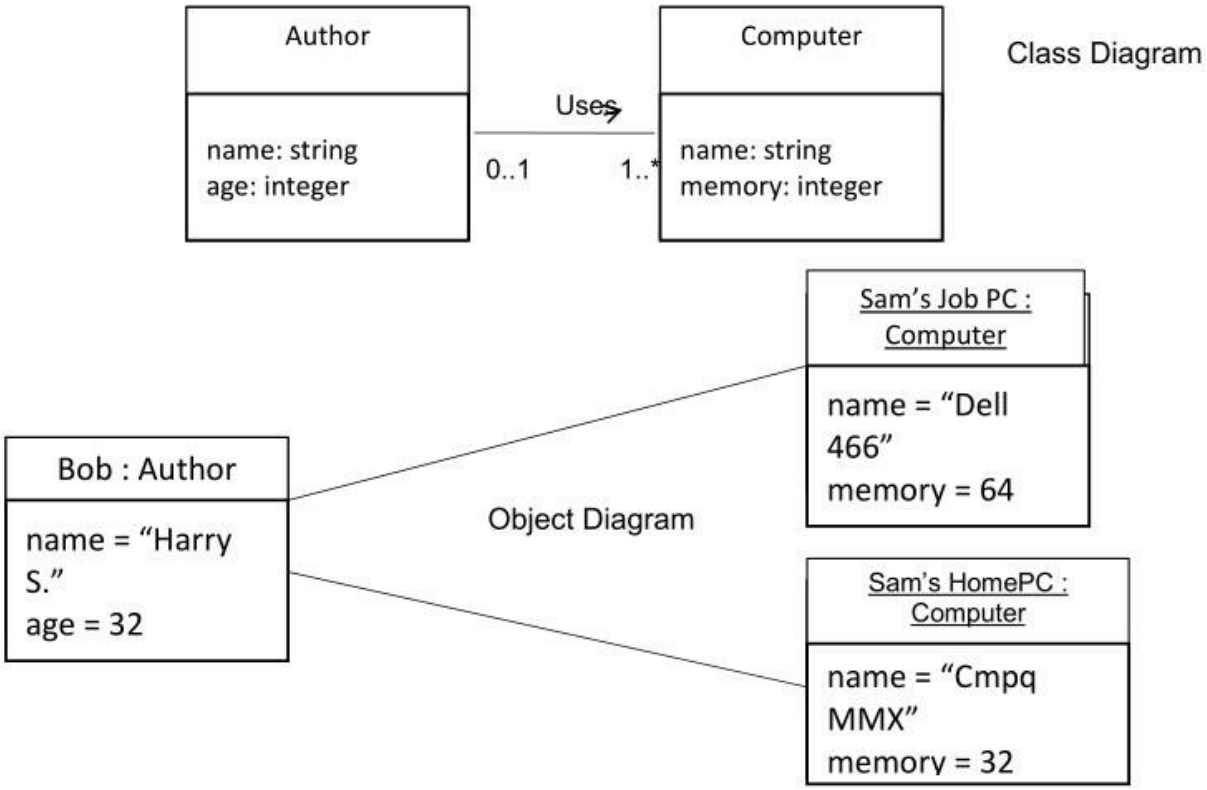
2. Object Diagram

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.**

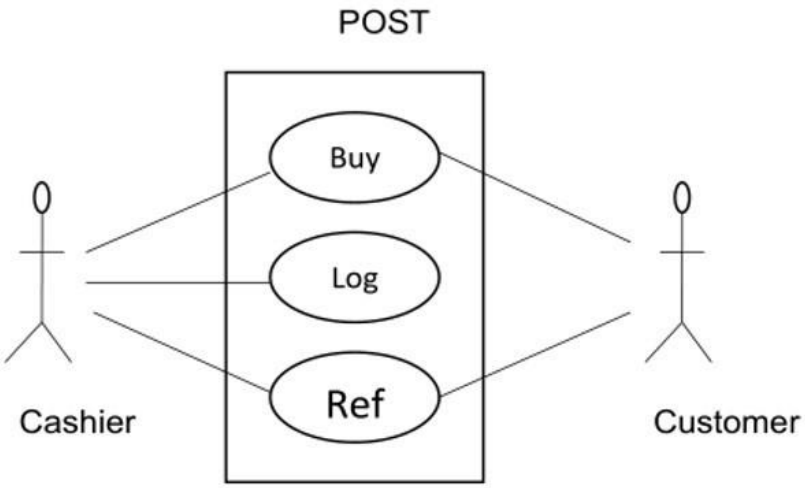
They show the dynamic behavior of the objects in a system, which can be described as a series of changes to the system over time.

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.

2. 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.



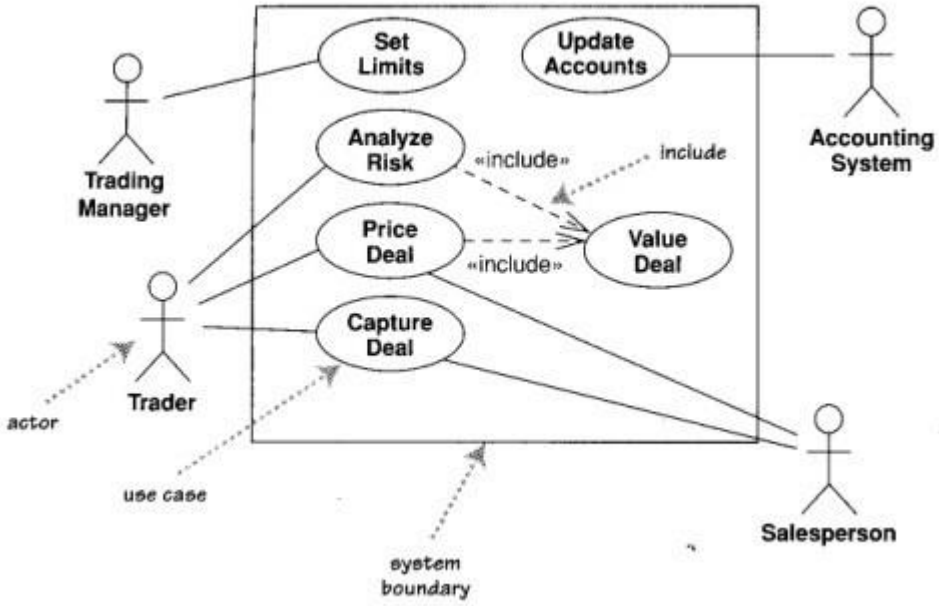
Emphasis on what *a system does* **rather than how.**

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:

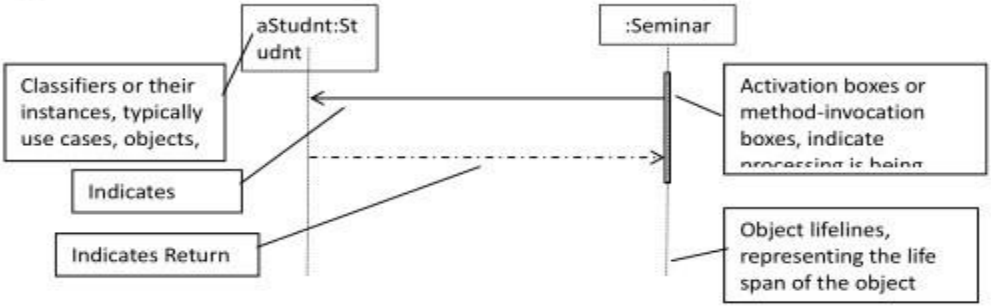
- Which actors carry out which use cases
- Which use cases include other use cases



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.

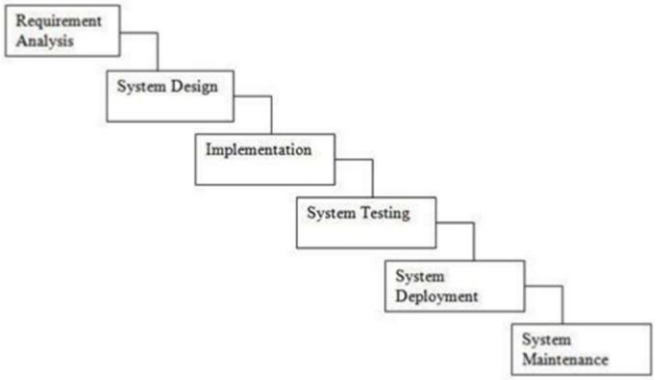


Figure : Waterfall Model

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]

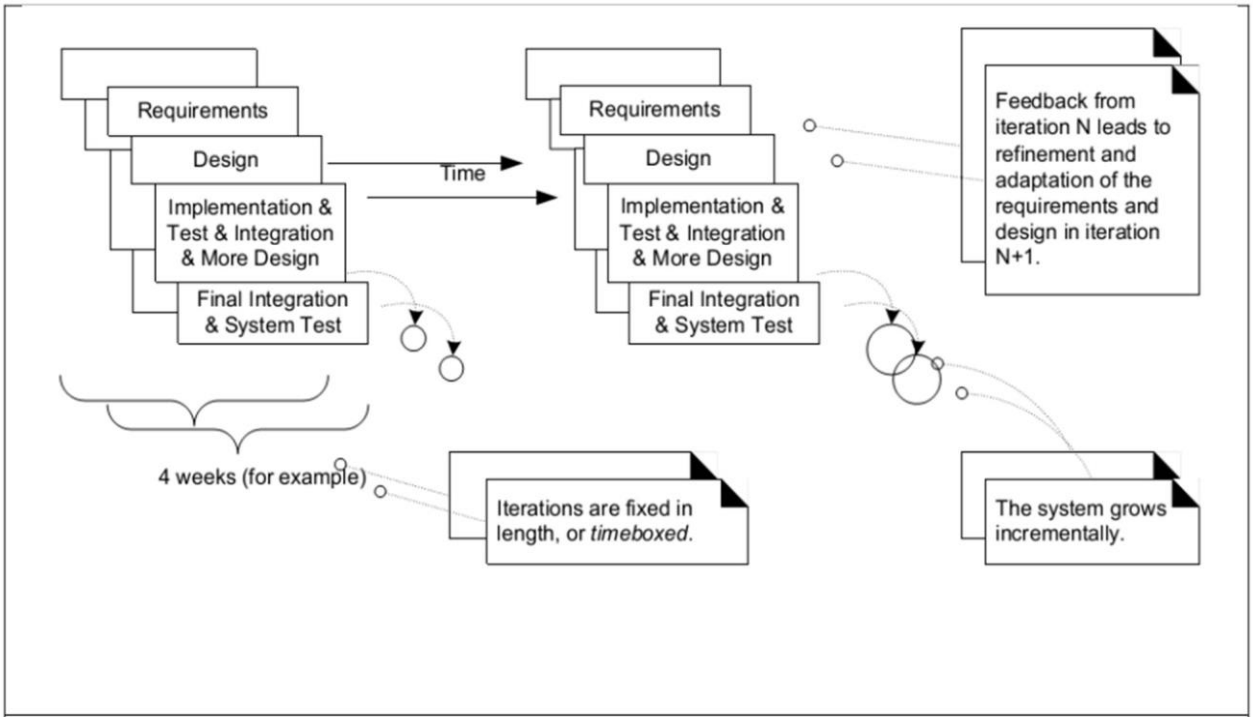


Figure : Iterative and evolutionary development.

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)

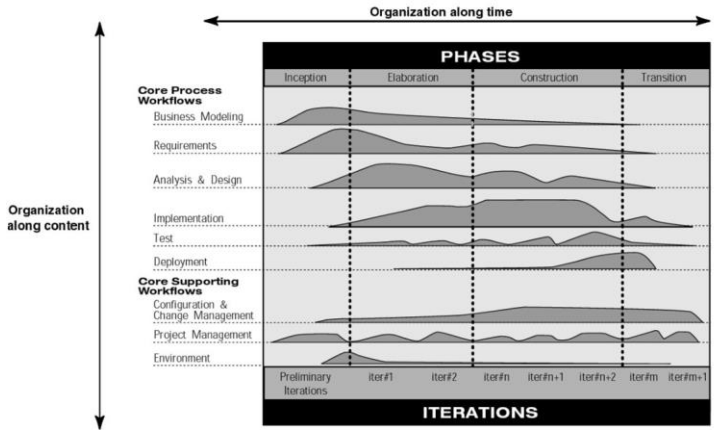
– Next name: Unified Software Development Process (USDP)

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

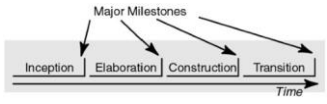
Phases and Iterations - The Time Dimension

This is the dynamic organization of the process along time.

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

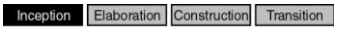
- Inception phase
- Elaboration phase
- Construction phase
- Transition phase

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(ARTIFACTS) 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

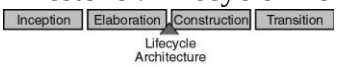
operation to a high-cost, high-risk operation with substantial inertia. While the process must always accommodate changes, the elaboration phase activities ensure that the architecture, requirements and plans are stable enough, and the risks are sufficiently mitigated, so you can predictably determine 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 use case 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.

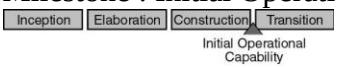
Construction Phase

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



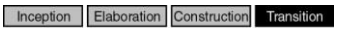
At the end of the construction phase is the third major project milestone (Initial Operational Capability Milestone). At this point, you decide if the software, the sites, and the users are ready to go operational, without exposing the project to high risks. This release is often called a “beta” release.

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:

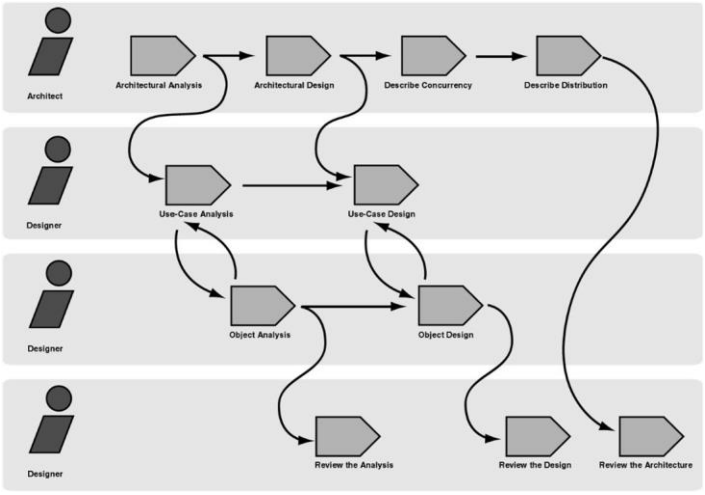
- Is the user satisfied?
- Are the actual resources expenditures versus planned expenditures still acceptable?

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.



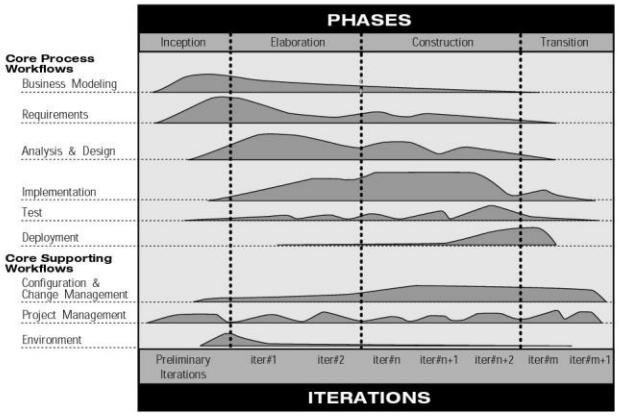
Example of workflow

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

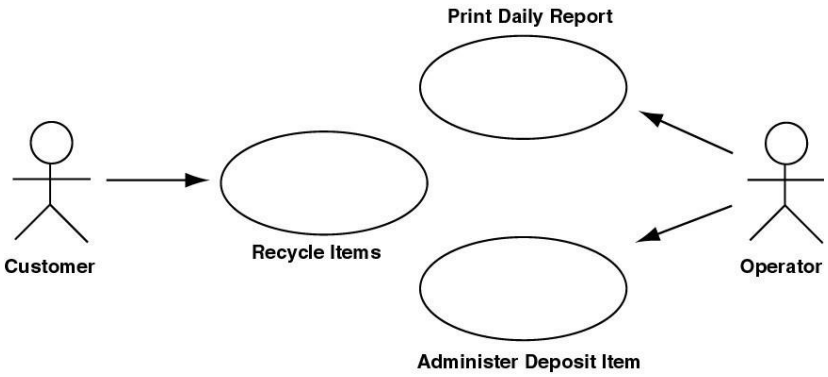
One of the major problems with most business engineering efforts, is that the software engineering and the business engineering community do not communicate properly with each other. This leads to the output from business engineering is not being used properly as input to the software development effort, 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 usecase model is used during requirements capture, analysis & design, and test.

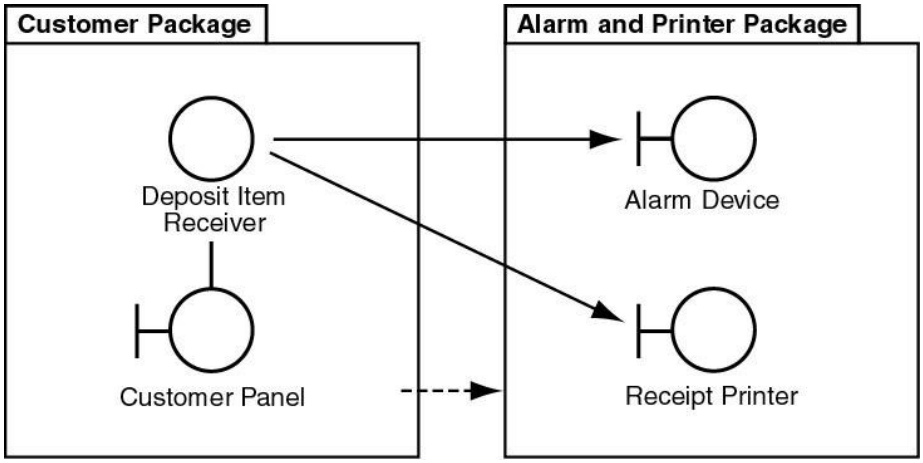
Analysis & Design

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:

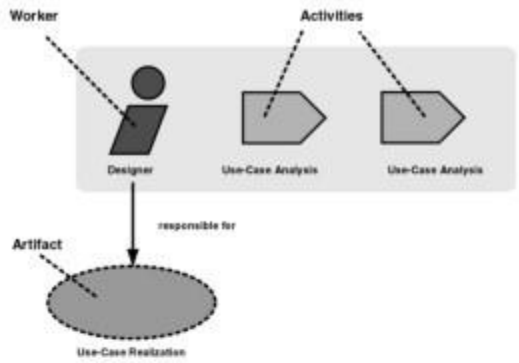
- Producing external releases of the software.
- Packaging the software.
- Distributing the software.
- Installing the software.
- Providing help and assistance to users.
- In many cases, this also includes activities such as:
- Planning and conduct of beta tests.
- Migration of existing software or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

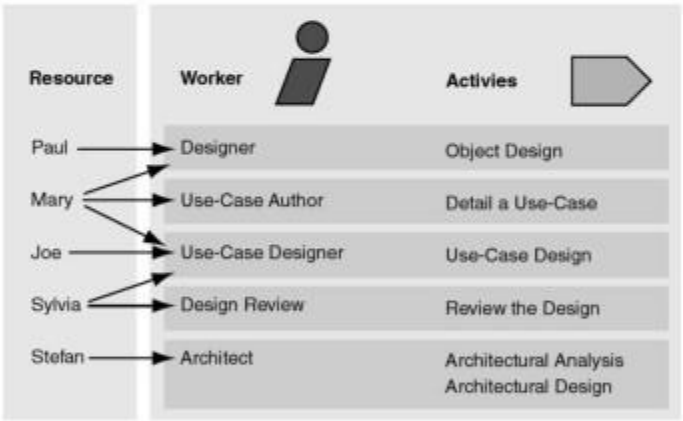
Activities, Artifacts, and Workers



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

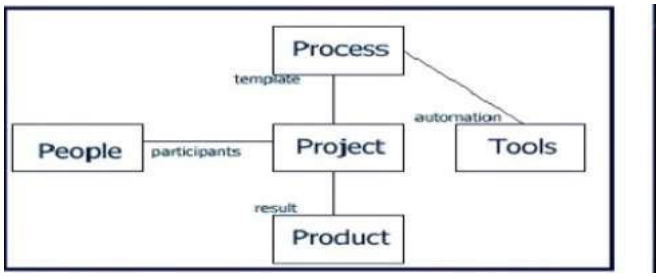
- Before a project can be planned, product objectives and scope should be established, alternative solutions should be considered and technical and management constraints should be identified.
- Without this information, it is impossible to define reasonable estimates of the cost, an effective assessment 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.

Project Management

11.1 What is project management?

Project management encompasses all the activities needed to plan and execute a project. The following are specific activities often done by a project manager:

1. **Deciding what needs to be done**. Finding customers; working with customers to determine their problem and the scope of the project; prioritizing the work; selecting the overall processes that will be followed, and negotiating contracts.
2. **Estimating costs**. The most important aspect of this is estimating the amount of elapsed time and effort that will be required to complete the project. We will discuss this in Section 11.3.
3. **Ensuring there are suitable people to undertake the project**. This includes finding people, and ensuring that people have appropriate training. It can also include firing people who are not performing adequately.
4. **Defining responsibilities**. Determining how people will work together in teams and who will be responsible for what. Teams are the subject of Section 11.4.
5. **Scheduling**. Determining the sequence of tasks, plus setting deadlines for when tasks must be complete. Major deadlines are called *milestones*. Scheduling is discussed in Section 11.5.
6. **Making arrangements for the work**. Initiating the paperwork involved in hiring or subcontracting; setting up training courses; finding office space; ensuring that hardware and software is available; ensuring that people have the requisite security clearance, etc.
7. **Directing**. Telling subordinates and contractors what to do. Many of the other activities in this list involve making decisions; but acting on those decisions by ordering people to do things is a distinct activity. Directing is not as simple as issuing orders – you have to get people to commit to deliver what they promise.

8. **Being a technical leader**. Giving advice about engineering problems; helping people solve problems by leading discussions; pointing people to appropriate sources of information; acting as a mentor, and making high-level decisions about requirements and design.

9. **Reviewing and approving decisions made by others**. In certain types of projects, the project manager will have to take the ultimate legal responsibility for declaring that proper engineering practice has been followed, and that the manager believes the resulting system will be safe. However, a certain amount of reviewing and approving is a part of every project.

10. **Building morale and supporting staff**. Helping resolve interpersonal conflicts; ensuring that people feel rewarded, respected and motivated; giving people feedback to help them improve their work; and ensuring that people always have somebody to talk to about problems.

11. **Monitoring and controlling**. Finding out what is going on, determining how the plans need to change, and taking action to keep the project on track. The risk management process, which we have talked about throughout this book, is a central aspect of this.

12. **Co-ordinating the work with managers of other projects**.

13. **Reporting**. Telling customers and higher-level managers what they need and want to know.

14. **Continually striving to improve the process**.

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.

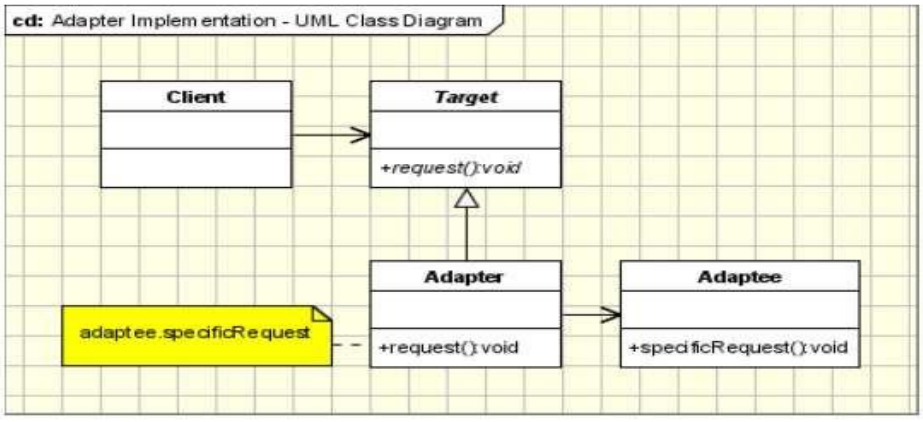
Advantages

- 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.

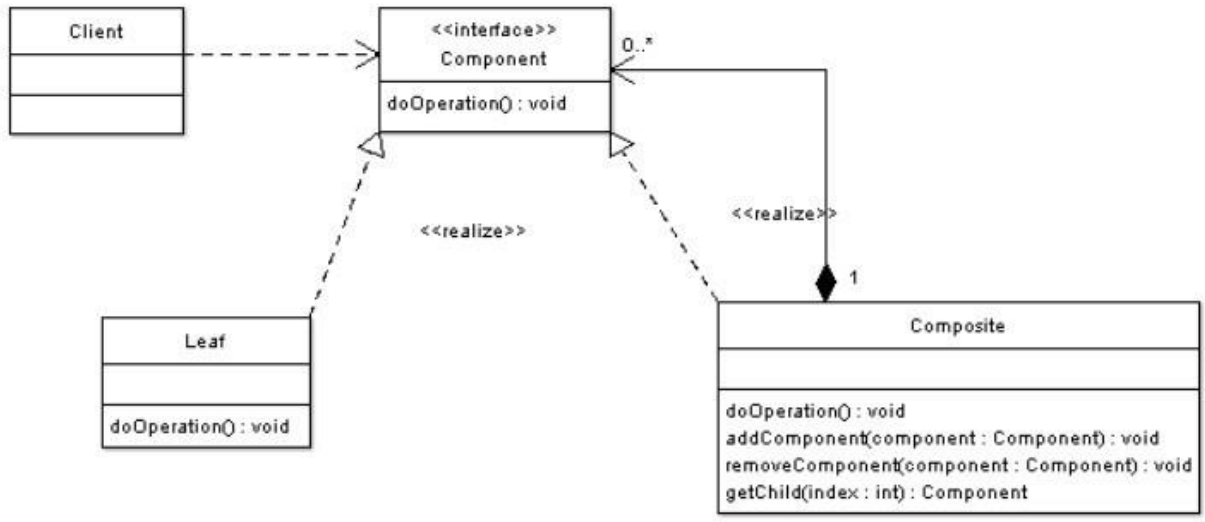
Composite lets clients treat individual objects and compositions of objects uniformly.

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.



Software Architecture

Definition: *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.

There are four main reasons why you need to develop an architectural model:

- **To enable everyone to better understand the system**. As a system becomes more and more complex, making it understandable is an increasing challenge. This is especially true for large, distributed systems that use sophisticated technology. A good architectural model allows people to understand how the system as a whole works; it also defines the terms that people use when they communicate with each other about lower-level details.
- **To allow people to work on individual pieces of the system in isolation**. The work of developing a complex software system must be distributed among a large number of people. The architecture allows the planning and co-ordination of this distributed work. The architecture should provide sufficient information so that the work of the individual people or teams can later on be integrated to form the final system. It is for that reason that the interfaces and dynamic interactions among the subsystems are an important part of the architecture.
- **To prepare for extension of the system**. With a complete architectural model, it becomes easier to plan the evolution of the system. Subsystems that are envisioned to be part of a future release can be included in the architecture, even though they are not to be developed immediately. It is then possible to see how the new elements will be integrated, and where they will be connected to the system. Architects designing buildings often use this technique – their drawings show not only the proposed building but also its future extensions (Phase I, Phase II, etc.). Specialists like electrical engineers can then plan the cabling to take into account the future needs of the foreseen extension.
- **To facilitate reuse and reusability**. The architectural model makes each system component visible. This is an important benefit since it encourages reuse. By analyzing the architecture, you can discover those components that can be obtained from past projects or from third parties. You can also identify components that have high potential reusability. Making the architecture as generic as possible is a key to ensuring reusability.

Contents of a good architectural model

A system's architecture will often be expressed in terms of several different *views*. These can include:

- The logical breakdown into subsystems. This is often shown using package diagrams, which we will describe later. The interfaces among the subsystems must also be carefully described.
- The dynamics of the interaction among components at run time, perhaps expressed using interaction or activity diagrams.
- The data that will be shared among the subsystems, typically expressed using class diagrams.
- The components that will exist at run time, and the machines or devices on which they will be located. This information can be expressed using component and deployment diagrams, which are discussed later.

Developing an architectural model

Start by sketching an outline of the architecture

- Based on the principal requirements and use cases
- Determine the main components that will be needed
- Choose among the various architectural patterns
- *Suggestion*: have several different teams independently develop a first draft of the architecture and merge together the best ideas

Refine the architecture

- Identify the main ways in which the components will interact and the interfaces between them
- Decide how each piece of data and functionality will be distributed among the various components
- Determine if you can re-use an existing framework, if you can build a framework

Consider each use case and adjust the architecture to make it realizable

Mature the architecture

Describing an architecture using UML

All UML diagrams can be useful to describe aspects of the architectural model. Remember that the goal of architecture is to describe the system at a very high level, with emphasis on software components and their interfaces. Use case diagrams can provide a good summary of the system from the user's perspective. Class diagrams can be used to indicate the services offered by components and the main data to be stored. Interaction diagrams can be used to define the protocol used when two components communicate with each other.

In addition to the UML diagrams we have already studied in this book, three other types of UML diagram are particularly important for architecture modeling: package diagrams, component diagrams and deployment diagrams. These are used to describe different aspects of the organization of the system. In the next three subsections we will survey the essentials of these types of diagram.

Packages

Breaking a large system into subsystems is a fundamental principle of software development. A good decomposition helps make the system more understandable and therefore facilitates its maintainability.

In UML, a *package* is a collection of modeling elements that are grouped together because they are logically related. Note that a UML package is not quite the same thing as a Java package, which is a collection containing only classes. However, a very common use of UML packages is to represent Java packages.

A package in UML is shown as a box, with a smaller box attached above its top left corner. The packages of the SimpleChat system are illustrated in Figure 9.6. Inside the box you can put practically anything, including classes, instances, text or other packages.

When you define a package, you should apply the principles of cohesion and coupling discussed earlier. Increasing cohesion means ensuring that a package only has related classes; decreasing coupling means decreasing the number of dependencies as much as possible.

Figure 9.6 **An example package diagram**

You show dependencies between packages using a dashed arrow. A dependency exists if there is a dependency between an *element* in one of the packages and an *element* in another. To use a package, it is required to have access to packages that it depends on. Also, changes made to the interface of a package will require modification to packages that depend on it.

A package that depends on many others will be difficult to reuse, since using it will also necessitate importing its dependent packages. Circular dependencies among packages are particularly important to avoid. Finally, making the interface of a package as simple as possible greatly simplifies its use and testing. The Façade pattern can help to simplify a package interface.

Component diagrams

A component diagram shows how a system's components – that is, the physical elements such as files, executables, etc. – relate to each other. The UML symbol for a component is a box with a little 'plug' symbol in the top-right corner. An example is shown in Figure 9.7; many more examples can be found in the next section.

Figure 9.7 **An example component diagram**

A component provides one or more interfaces for other components to use. The same 'lollipop' symbol is used for an interface as was introduced in Chapter 5. To show a component using an interface provided by another, you use a semi-circle at the end of a line. Figure 9.7 show how these two symbols plug together.

Various relationships can exist among components, for example:

- A component may *execute* another component, or a method in the other component.
- A component may *generate* another component.
- Two components may *communicate* with each other using a network.

It is easy initially to confuse component diagrams with package diagrams. The difference is that package diagrams show *logical groupings* of design elements, whereas component diagrams show relationships among types of *physical* components.

Deployment diagrams

A deployment diagram describes the hardware where various instances of components reside at run time. An example is shown in Figure 9.8. A node in a deployment diagram represents a computational unit such as a computer, a processing card, a sensor or a device. It appears as a three-dimensional box.

Figure 9.8 An example deployment diagram

The links between nodes show how communication takes place. Each node of a deployment diagram can include one or several run-time software components. Various artifacts such as files can also be shown inside nodes.

Architectural Pattern

Architectural Pattern

An **architectural pattern** is a general, reusable solution to a commonly occurring problem in software architecture within a given context. Architectural patterns are similar to software design pattern but have a broader scope.

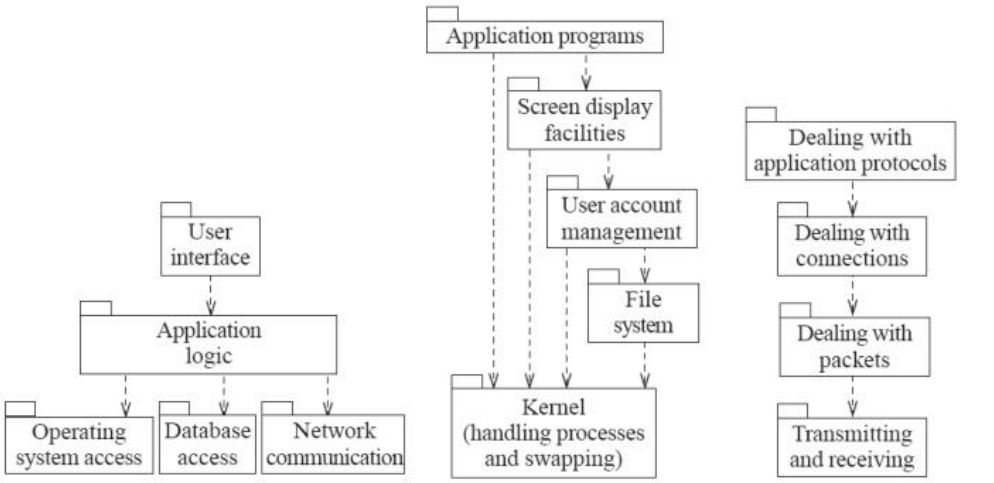
Allows you to design flexible systems using components where the components are as independent of each other as possible.

Multi-Layer architectural pattern

In layered systems each layer communicates only with the layers below it .Each layer has a well-defined API, defining the services it provides.

A complex system can be built by superimposing layers at increasing levels of abstraction. The Multi-Layer architectural pattern makes it possible to replace a layer by an improved version, or one with a different set of capabilities.

It is important to have a separate layer for the UI.



(a) Typical layers in an application program

(b) Typical layers in an operating system

(c) Simplified view of layers in a communication system

Layers immediately below the UI layer provide the application functions determined by the use-cases.
Bottom layers provide general services.

The most commonly found 4 layers of a general information system are as follows.

- **Presentation layer** (also known as **UI layer**)
- **Application layer** (also known as **service layer**)
- **Business logic layer** (also known as **domain layer**)
- **Data access layer** (also known as **persistence layer**)

Usage:

- General desktop applications.
- E commerce web applications.

The Multi-Layer architectural pattern and design principles:

***1. Divide and conquer*:** The layers can be independently designed.
2. *Increase cohesion*: Well-designed layers have layer cohesion.
3. *Reduce coupling*: Well-designed lower layers do not know about the higher layers and the only connection between layers is through the API.
4. *Increase abstraction*: you do not need to know the details of how the lower layers are implemented.
5. *Increase reusability*: The lower layers can often be designed generically.
6. *Increase reuse*: You can often reuse layers built by others that provide the services you need.
7. Increase flexibility: you can add new facilities built on lower-level services, or replace higher-level layers.
8. Anticipate obsolescence: By isolating components in separate layers, the system becomes more resistant to obsolescence.
9. Design for portability: All the dependent facilities can be isolated in one of the lower layers.
10. Design for testability: Layers can be tested independently.
11. Design defensively: The APIs of layers are natural places to build in rigorous assertion-checking.

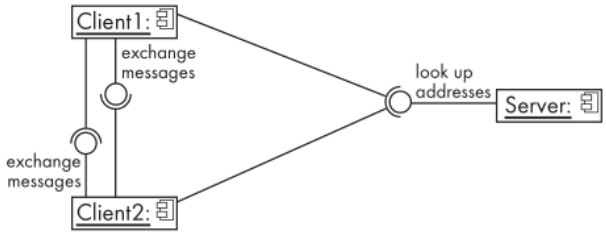
The Client/Server and other distributed architectural patterns

This pattern consists of two parties; a ***server*** and multiple ***clients***. The server component will provide services to multiple client components. Clients request services from the server and the server provides relevant services to those clients. Furthermore, the server continues to listen to client requests.

- There is at least one component that has the role of *server*, waiting for and then handling connections.
- There is at least one component that has the role of *client*, initiating connections in order to obtain some service.
- A further extension is the Peer-to-Peer pattern.
    - A system composed of various software components that are distributed over several hosts.

Usage

- Online applications such as email, document sharing and banking.



A peer-to-peer architecture for instant messaging, retaining a central server only to look up the addresses of clients

The Distributed architecture and design principles

1. *Divide and conquer*: Dividing the system into client and server processes is a strong way to divide the system.

- Each can be separately developed.

2. *Increase cohesion*: The server can provide a cohesive service to clients.

3. *Reduce coupling*: There is usually only one communication channel exchanging simple messages.

4. *Increase abstraction*: Separate distributed components are often good abstractions.

6. *Increase reuse*: It is often possible to find suitable frameworks on which to build good distributed systems

- However, client-server systems are often very application specific.

7. *Design for flexibility*: Distributed systems can often be easily reconfigured by adding extra servers or clients.

9. *Design for portability*: You can write clients for new platforms without having to port the server.

10. *Design for testability*: You can test clients and servers independently.

11. *Design defensively*: You can put rigorous checks in the message handling code.

The Model–View–Controller (MVC) architectural pattern

Model–View–Controller, or MVC, is an architectural pattern used to help separate the user interface layer from other parts of the system. Not only does MVC help enforce layer cohesion of the user interface layer, but it also helps reduce the coupling between that layer and the rest of the system, as well as between different aspects of the UI itself.

The MVC pattern separates the functional layer of the system (the *model*) from two aspects of the user interface, the *view* and the *controller*. This is illustrated in Figure 9.13. Although the three components are normally

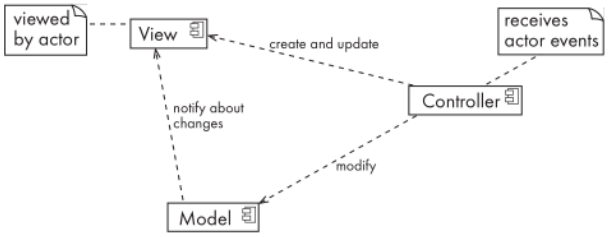


Figure 9.13 The Model–View–Controller (MVC) architectural pattern for user interfaces

instances of classes, we use a component diagram to emphasize the fact that the components could also be separate threads or processes.

The *model* contains the underlying classes whose instances are to be viewed and manipulated.

The *view* contains objects used to render the appearance of the data from the model in the user interface. The view also displays the various controls with which the user can interact.

The *controller* contains the objects that control and handle the user's interaction with the view and the model. It has the logic that responds when the user types into a field or clicks the mouse on a control.

The model does not know what views and controllers are attached to it. In particular, the Observer design pattern is normally used to separate the model from the view. The MVC architectural pattern therefore exhibits layer cohesion and is a special case of the Multi-Layer architectural pattern.

The MVC architectural pattern allows us to adhere to the following design principles:

1. *Divide and conquer.* The three components can be somewhat independently designed.

2. *Increase cohesion*. The components have stronger layer cohesion than if the view and controller were together in a single UI layer.

3. *Reduce coupling*. The communication channels between the three components are minimal and easy to find.

6. *Increase reuse*. The view and controller normally make extensive use of reusable components for various kinds of UI controls. The UI, however will become application specific, therefore it will not be easily reusable.

7. *Design for flexibility.* It is usually quite easy to change the UI by changing the view, the controller, or both.

10. *Design for testability.* You can test the application separately from the UI.

The Service-Oriented architectural pattern

Now that the Internet has become so pervasive, a new architectural pattern has become prominent: *Service-oriented architecture*. This architecture organizes an application as a collection of services that communicate with each other through well-defined interfaces. In the context of the Internet, the services in question are called *Web services*.

A Web service is an application accessible through the Internet that can be integrated with other web services to form a *Web-based application*. To use a web service, you send a correctly formatted http request to an http server. Unlike normal http requests this is done 'behind the scenes' by the Web application program, not a browser being used by an end user. The server will run the service application and return the response as a document, typically structured using a language called *XML*. This is illustrated in Figure 9.14.

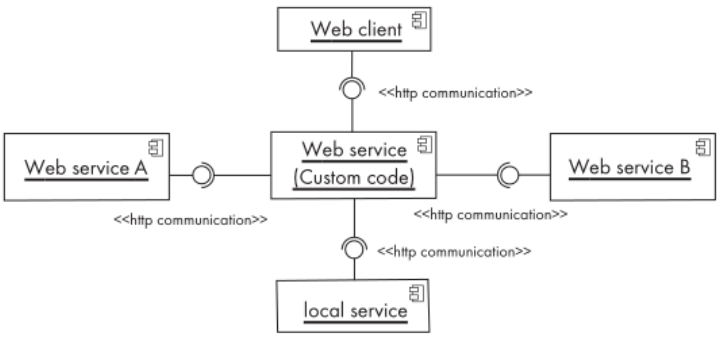


Figure 9.14 The Service-oriented architectural pattern

Web services can perform a wide range of tasks. Some may handle simple requests for information while others can perform more complex business processing. Enterprises can use Web services to automate and improve their operations. For example, an electronic commerce application can make use of web services to:

- access the product databases of several suppliers;
- process credit cards using a Web service offered by a bank;
- arrange for delivery using a Web service offered by a shipping company.

The Service Oriented Architecture and design principles

1. *Divide and conquer*: The application is made of independently designed services.

2. *Increase cohesion*: The Web services are structured as layers and generally have good functional cohesion.

3. *Reduce coupling*: Web-based applications are loosely coupled built by binding together distributed components.

5. *Increase reusability*: A Web service is a highly reusable component.

6. *Increase reuse*: Web-based applications are built by reusing existing Web services.

8. *Anticipate obsolescence*: Obsolete services can be replaced by new implementation without impacting the applications that use them.

9. *Design for portability*: A service can be implemented on any platform that supports the required standards.

10. *Design for testability*: Each service can be tested independently.

11. *Design defensively*: Web services enforce defensive design since different applications can access the service.

The Message-Oriented architectural pattern

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. The core of the architecture is an application-to-application messaging system. Senders and receivers need only to know what are the message formats; that is, a receiving (or sending) application does not have to know anything about the software component that sent (or received) the message. Moreover, the two communicating applications do not even have to be available at the same time. This is illustrated in Figure 9.15.

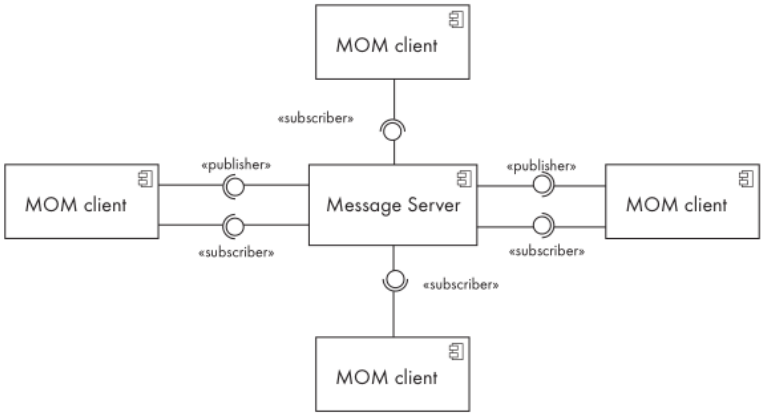


Figure 9.15 The Message-Oriented architectural pattern

The application can choose to ignore a received message or react to it by, for instance, sending a reply containing requested information. The exchange of these messages is governed by two important principles. First, message delivery is completely asynchronous; that means the exact moment at which a given message is delivered to a given subscribing application is unknown. Secondly, reliability mechanisms are in place such that the messaging system can offer the guarantee that a given message is delivered once and only once.

Text messaging using cellular phones can be seen as a simple message oriented application, but more complex systems can also adopt this architecture.

Clearly, the effectiveness of system depends on the messaging system that is used to deliver the messages. Two approaches can be taken when designing such a system. The first is to use a centralized architecture where all messages transit through a message server that is responsible for delivering messages to the subscribers. This is the simpler model, but all functionality then relies on the server. The alternative is to use a decentralized architecture where message routing is delegated to the network layer and where some of the server functionality is distributed among all message clients.

The Message Oriented Architecture and design principles

1. *Divide and conquer*: The application is made of isolated software components.
3. *Reduce coupling*: The components are loosely coupled since they share only data format.
4. *Increase abstraction*: The prescribed formats of the messages are generally simple to manipulate, all the application details being hidden behind the messaging system.
5. *Increase reusability*: A component will be reusable is the message formats are flexible enough.
6. *Increase reuse*: The components can be reused as long as the new system adheres to the proposed message formats.
7. *Design for flexibility*: The functionality of a message-oriented system can be easily updated or enhanced by adding or replacing components in the system.
10. *Design for testability*: Each component can be tested independently.
11. *Design defensively*: Defensive design consists simply of validating all received messages before processing them.

Introduction:

- ✓ 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.

A good pattern should be as general as possible, containing a solution that has been proven to solve the problem effectively in the indicated context. The pattern must be described in an easy-to-understand form so that people can determine when and how to use it. Studying patterns is an effective way to learn from the experience of others.

Each pattern should have a name; it should also have the following information:

- **Context:** the general situation in which the pattern applies.
- **Problem:** a sentence or two explaining the main difficulty being tackled.
- **Forces:** the issues or concerns that you need to consider when solving the problem. These include criteria for evaluating a good solution.
- **Solution:** the recommended way to solve the problem in the given context. The solution is said to 'balance the forces'; in other words, it has a good combination of advantages, with few counterbalancing disadvantages.
- **Antipatterns:** (optional) solutions that are inferior or do not work in this context. The reason for their rejection should be explained. The antipatterns may be valid solutions in other contexts, or may never be valid. They often are mistakes made by beginners.
- **Related patterns:** (optional) patterns that are similar to this pattern. They may represent variations, extensions or special cases.
- **References:** acknowledgements of those who developed or inspired the pattern.

A pattern should normally be illustrated using a simple diagram and should be written using a narrative writing style.

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."

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.

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.

6.15 Difficulties and risks when using design patterns

The following are the key difficulties to anticipate when designing and using design patterns:

- **Patterns are not a panacea**. Whenever you see an indication that a pattern should be applied, you might be tempted to apply the pattern blindly. However, this can lead to unwise design decisions. For example, you do not always need to apply the Façade pattern in every subsystem; adding the extra class might make the overall design more complex, especially if instances of many of the classes in the subsystem are passed as data to methods outside the subsystem.
  Resolution. Always understand in depth the forces that need to be balanced, and when other patterns better balance the forces. Also, make sure you justify each design decision carefully.
- **Developing patterns is hard**. Writing a good pattern takes considerable work. A poor pattern can be hard for other people to apply correctly, and can lead

them to make incorrect decisions. It is particularly hard to write a set of forces effectively.
Resolution. Do not write patterns for others to use until you have considerable experience both in software design and in the use of patterns. Take an in-depth course on patterns. Iteratively refine your patterns, and have them peer reviewed at each iteration.

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.

6.2 The Abstraction–Occurrence pattern

Context This modeling pattern is most frequently found in class diagrams that form part of a system domain model.

Often in a domain model you find a set of related objects that we will call *occurrences*; the members of such a set share common information but also differ from each other in important ways.

Examples of sets of occurrences include:

- All the episodes of a television series. They have the same producer and the same series title, but different story-lines.

Problem What is the best way to represent such sets of occurrences in a class diagram?

Forces You want to represent the members of each set of occurrences without duplicating the common information. Duplication would consume unnecessary space and would require changing all the occurrences when the common information changes. Furthermore, you want to avoid the risk of inconsistency that would result from changing the common information in some objects but not in others. Finally you want a solution that maximizes the flexibility of the system.

Solution Create an «*Abstraction*» class that contains the data that is common to all the members of a set of occurrences. Then create an «*Occurrence*» class representing the occurrences of this abstraction. Connect these classes with a one-to-many association. This is illustrated in Figure 6.1.

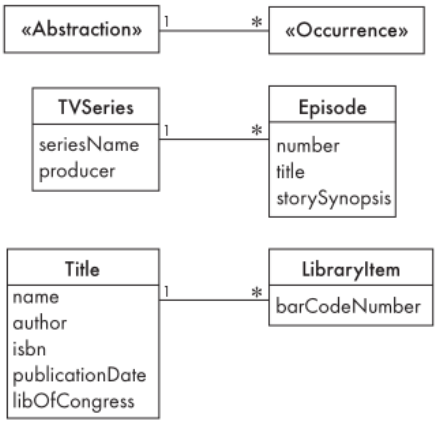
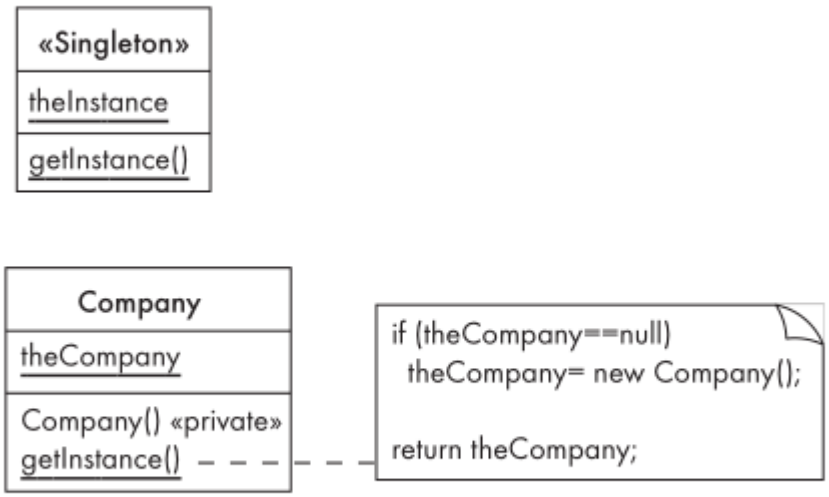


Figure 6.1 Template and examples of the Abstraction–Occurrence pattern

Examples You might create an «Abstraction» class called `TVSeries`; an instance of this might be the children's series 'Sesame Street'. You would then create an «Occurrence» class called `Episode`. Similarly, you might create an «Abstraction» class called `Title` which will contain the author and name of a book or similar publication. The corresponding «Occurrence» class might be called `LibraryItem`. These examples are illustrated in Figure 6.1.

Creational design patterns



6.7 Template and example of the Singleton design pattern

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]

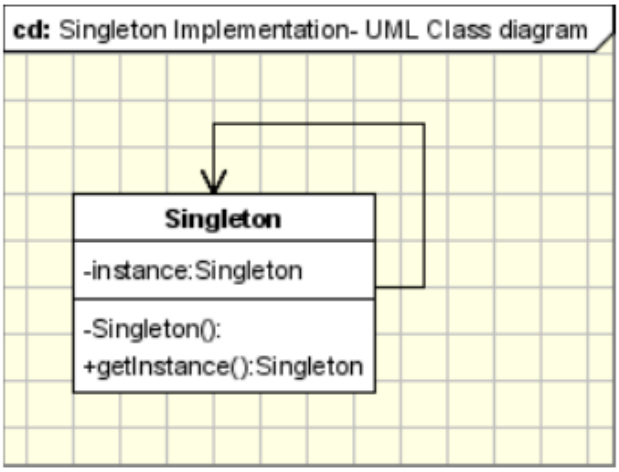
Five well-known design patterns that are parts of creational patterns are the

- 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]

Singleton Pattern

It is a creational design pattern that lets you ensure that a class has only one instance, while providing a global access point to this instance.

The implementation involves a static member in the "Singleton" class, a private constructor and a static public method that returns a reference to the static member.



The Singleton Pattern defines a getInstance operation which exposes the unique instance which is accessed by the clients. getInstance() is responsible for creating its class unique instance in case it is not created yet and to return that instance.

```
public class SingletonClass {

    private static SingletonClass instance = new SingletonClass();

    private SingletonClass() {}

    public static SingletonClass getInstance() {
        return instance;
    }

    public void showMessage() {
        System.out.println("I'm a singleton object!");
    }
}
```

Here

- This class is creating a static object of itself, which represents the global instance.
- By providing a private constructor, the class cannot be instantiated.
- A static method getInstance() is used as a global access point for the rest of the application.

```
public class Main {
    public static void main(String[] args) {
        SingletonClass singletonClass = SingletonClass.getInstance();
        singletonClass.showMessage();
    }
}
```

Output : `I'm a singleton object!`

Structural patterns are concerned with how classes and objects are composed to form larger structures.

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

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.

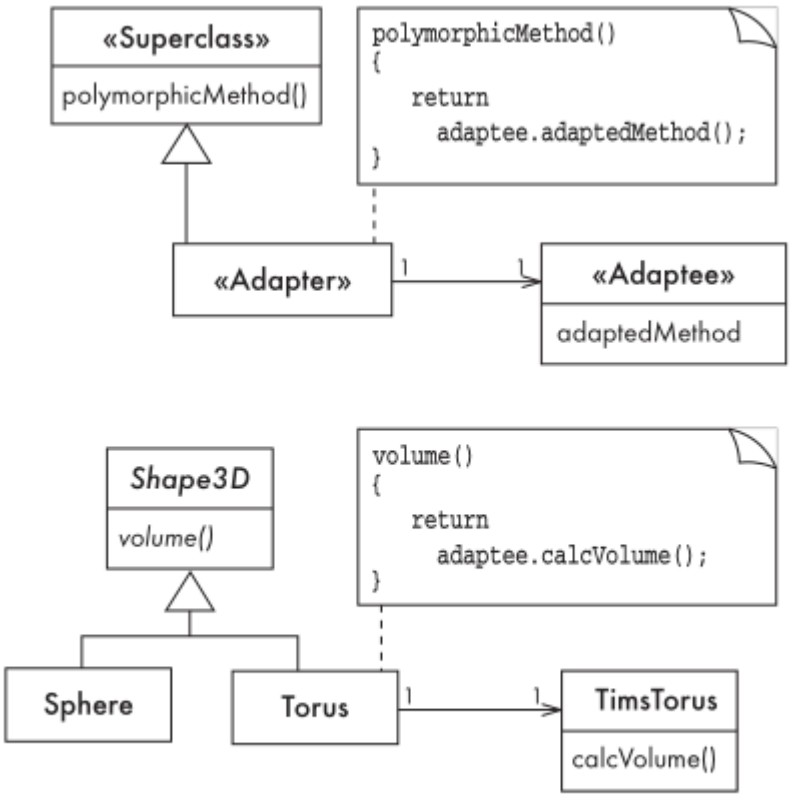
Advantages

- 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.



Template and example of the Adapter design pattern

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.

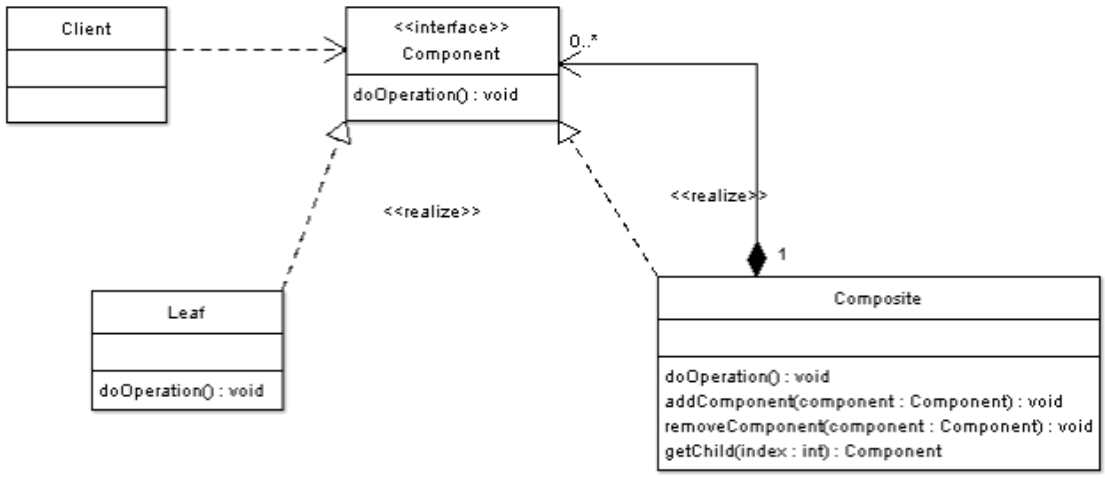
Composite lets clients treat individual objects and compositions of objects uniformly.

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.



Elements of composite patterns:

- **Component** - Component is the abstraction for leafs and composites. It defines the interface that must be implemented by the objects in the composition. For example a file system resource defines move, copy, rename, and getSize methods for files and folders.
- **Leaf** - Leafs are objects that have no children. They implement services described by the Component interface. For example a file object implements move, copy, rename, as well as getSize methods which are related to the Component interface.
- **Composite** - A Composite stores child components in addition to implementing methods defined by the component interface. Composites implement methods defined in the Component interface by delegating to child components. In addition composites provide additional methods for adding, removing, as well as getting components.
- **Client** - The client manipulates objects in the hierarchy using the component interface.

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

Behavioral patterns are concerned with algorithms and the assignment of responsibilities between objects. Behavioral patterns describe not just patterns of objects or classes but also the patterns of communication between them.

Chain of Responsibility

In chain of responsibility, sender sends a request to a chain of objects.

The request can be handled by any object in the chain.

A Chain of Responsibility Pattern says that just **"avoid coupling the sender of a request to its receiver by giving multiple objects a chance to handle the request".** For example, an ATM uses the Chain of Responsibility design pattern in money giving process.

In other words, we can say that normally each receiver contains reference of another receiver.

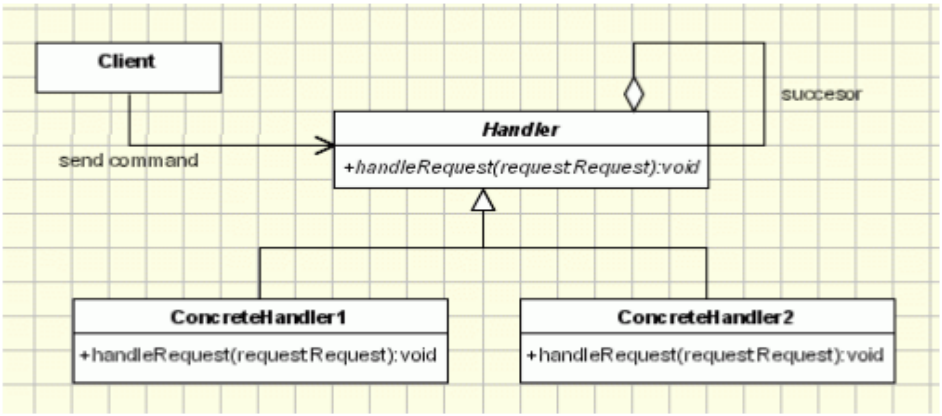
If one object cannot handle the request then it passes the same to the next receiver and so on.

Advantages

- It reduces the coupling.
- It adds flexibility while assigning the responsibilities to objects.
- It allows a set of classes to act as one; events produced in one class can be sent to other handler classes with the help of composition.

Uses

- When more than one object can handle a request and the handler is unknown.
- When the group of objects that can handle the request must be specified in dynamic way.



Handler - defines an interface for handling requests
RequestHandler - handles the requests it is responsible for

- If it can handle the request it does so, otherwise it sends the request to its successor

Client - sends commands to the first object in the chain that may handle the command

the Client in need of a request to be handled sends it to the chain of handlers, which are classes that extend the Handler class. Each of the handlers in the chain takes its turn at trying to handle the request it receives from the client. If ConcreteHandler_i can handle it, then the request is handled, if not it is sent to the handler ConcreteHandler_i+1, the next one in the chain.

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

In the UML, the basic unit of concurrency is the thread. Threads are associated with a stereotype of objects, called «active» objects; that is, an «active» object is special in the sense that it is the root of a thread of control.
The recommended way of adding concurrency into an object model is to identify the desired threads through the application of task identification strategies (see [7] for a list of several such strategies). Once a set of threads is identified, the developer creates an «active» object for each. The "passive" objects are then

added to the «active» objects via the composition (strong aggregation) relation. The role of the «active»
object is to run when appropriate and call or delegate actions to the passive objects that it owns. The passive objects execute in the thread of their «active» owner.

The **active object** design pattern decouples method execution from method invocation for objects that each reside in their own thread of control.[1] The goal is to introduce concurrency, by using asynchronous method invocation and a scheduler for handling requests.[2]

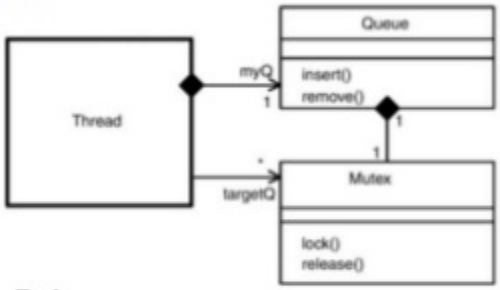
- The *Monitor Object* design pattern (399) synchronizes concurrent method execution to ensure that only one method at a time runs within an object. It also allows an object's methods to schedule their execution sequences cooperatively.

GlobalLogic

14

Message Queuing Pattern

- The Message Queuing Pattern uses asynchronous communications, implemented via queued messages, to synchronize and share information among tasks.
- Any information shared among threads is passed by value to the separate thread.
- Relatively heavyweight approach



Collaboration Roles

- **Thread** - it is the root of an operating system thread, task, or process. It can both create messages to send to other Threads and receive and process messages when it runs.
- **Queue -** container that can hold a number of messages.
- **Mutex -** provides protection from multiple access to the Queue contents

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.

Principles that lead to good design

Design Principle 1: Divide and conquer

The divide and conquer principle dates back to the earliest days of organized human activity. Trying to deal with something big all at once is normally much harder than dealing with a series of smaller things. Military campaigns are waged this way: commanders try to avoid fighting on all fronts at once. Cars are also built using the divide and conquer strategy: some people design the engines while others design the body, etc. Furthermore, the task of assembling the car is also divided into smaller, more manageable chunks – each assembly-line worker will focus on one small task.

In software engineering, the divide and conquer principle is applied in many ways. We have already seen how the process of development is divided into activities such as requirements gathering, design and testing. In this section we will look at how software systems themselves can be divided.

Dividing a software system into pieces has many advantages:

- Separate people can work on each part. The original development work can therefore be done in parallel.
- An individual software engineer can specialize in his or her component, becoming expert at it. It is possible for someone to know everything about a small part of a system, but it is not possible to know everything about an entire system.
- Each individual component is smaller, and therefore easier to understand.
- When one part needs to be replaced or changed, this can hopefully be done without having to replace or extensively change other parts.
- Opportunities arise for making the components reusable.

A software system can be divided in many ways:

- A distributed system is divided up into clients and servers.
- A system is divided up into subsystems.
- A subsystem can be divided up into one or more packages.
- A package is composed of classes.
- A class is composed of methods.

Design Principle 2: Increase cohesion where possible

The cohesion principle is an extension of the divide and conquer principle – divide and conquer simply says to divide things up into smaller chunks. Cohesion says to do it intelligently: yes, divide things up, but keep things together that belong together.

A subsystem or module has high cohesion if it keeps together things that are related to each other, and keeps out other things. This makes the system as a whole easier to understand and change.

Listed below are several important types of cohesion that designers should try to achieve. Table 9.1 summarizes these types of cohesion, starting with the most desirable.

Table 9.1 **The different types of cohesion, ordered from highest to lowest in terms of the precedence you should normally give them when making design decisions**

Cohesion type	*Comments*
Functional	Facilities are kept together that perform only *one computation* with no *side effects*. Everything else is kept out
Layer	*Related services* are kept together, everything else is kept out, and there is a *strict hierarchy* in which higher-level services can access only lower-level services. Accessing a service may result in side effects
Communicational	Facilities for operating on the *same data* are kept together, and everything else is kept out. Good classes exhibit communicational cohesion
Sequential	A set of procedures, which work in sequence to perform some computation, is kept together. *Output from one is input to the next*. Everything else is kept out
Procedural	A set of procedures, which are called *one after another*, is kept together. Everything else is kept out
Temporal	Procedures used in the *same general phase* of execution, such as initialization or termination, are kept together. Everything else is kept out
Utility	*Related utilities* are kept together, when there is no way to group them using a stronger form of cohesion

Design Principle 3: Reduce coupling where possible

Coupling occurs when there are interdependencies between one module and another. Figure 9.5 illustrates the concept of a tightly coupled and loosely coupled system.

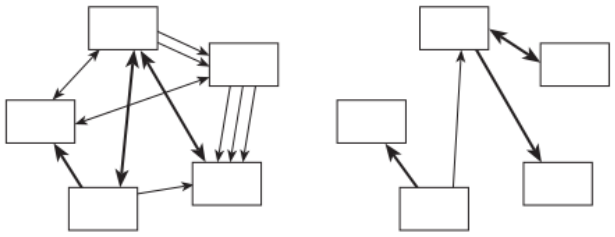


Figure 9.5 **Abstract examples of a tightly coupled system (left) and a loosely coupled system (right). The boldness of the arrows indicates the strength of the coupling**

Design Principle 4: Keep the level of abstraction as high as possible

You should ensure that your designs allow you to hide or defer consideration of details, thus reducing complexity. The general term given to this property of designs is *abstraction*. Abstractions are needed because the human brain can process only a limited amount of information at any one time.

We have discussed many types of abstractions in earlier chapters. In Chapter 2 we introduced procedural abstraction and data abstraction – hiding the details of procedures and data, respectively. In Section 2.7 we discussed several types of abstraction present in object-oriented programs.

Some abstractions, like classes and methods, are supported directly by the programming language. Others, like associations, are present purely in models used by the designer.

Abstractions work by allowing you to understand the essence of something and make important decisions without knowing unnecessary details. The details can be provided in several ways:

- At a later stage of design. For example, when creating class diagrams, you often initially leave out the data types of attributes, and you do not show the implementation details of associations.
- By the compiler or run-time system. For example, dynamic binding takes care of which methods will run.
- By the use of default values. For example, a draw operation that always makes the background white unless some explicit action is taken to change the default.

Design Principle 5: Increase reusability where possible

There are two complementary principles that relate to reuse; the first is to design *for* reuse, and the second is to design *with* reuse. We introduced both of these principles in Chapter 3.

Designing for reusability means designing various aspects of your system so that they can be used again in other contexts, both in your system and in other systems. As discussed in Chapter 3, you can build reusability into algorithms, classes, procedures, frameworks and complete applications. Mechanisms whereby components can be reused include calling procedures and inheriting a superclass.

Important strategies for increasing reusability are as follows:

- Generalize your design as much as possible. As you design a potentially reusable component, imagine several other systems that could use this component. Then design your component so that it could work with the other systems too. For example, if you are creating a facility to draw a particular kind of diagram, why not design it so that it could be used to draw other kinds of diagrams for other applications? Better yet, forget the specific application and focus on the reusable component alone. For example, if you need to create a method to save instances of `Employee` to a binary file, instead consider the problem of saving instances of *any* class to a binary file.
- Follow the preceding three design principles. Increasing cohesion increases reusability since the component has a well-defined purpose. Reducing coupling increases reusability because the component can stand alone. Increasing abstraction increases reusability since abstractions are naturally more general.
- Design your system to contain hooks. As discussed in Chapter 2, a hook is an aspect of the design deliberately added to allow other designers to add additional functionality. One of the barriers to reuse occurs when a component does most of what someone else needs, but not quite everything. If a component has effective hooks, then other people can easily extend it to do what they want. For example, the OCSF system has hooks such as `connectionClosed` that allow application designers to choose to do something interesting when a connection is closed.

Design Principle 6: Reuse existing designs and code where possible

Designing with reuse is complementary to designing for reusability. Actively reusing designs or code allows you to take advantage of the investment you or others have made in reusable components.

Cloning should normally *not* be seen as an effective form of reuse. Cloning involves copying code from one place to another; it should be avoided since, when there are two or more occurrences of the same or similar code in the system, any changes made (e.g. to fix defects) will have to be made in all clones. Unfortunately, maintainers are often not aware of all the clones that exist, and hence only make the change in one place. The bug thus remains, even though the maintainer thinks it is fixed.

In general, it can be acceptable to clone a single line of code; perhaps a line that contains a complicated call to a method with many arguments. However, any time you are tempted to clone more than a couple of lines of code, it is normally best to encapsulate the code in a separate method and call it from all the places it is needed. See the sidebar 'Tolerating Clones?' for a discussion of exceptions to this rule.

Design Principle 9: Design for portability

Designing for portability shares many things in common with anticipating obsolescence, although the objective is different. Anticipating obsolescence has, as its primary objective, the survival of the software. Design for portability has, as its prime objective, the ability to have the software run on as many platforms as possible, although sometimes this might also be a necessity for survival.

Design Principle 10: Design for testability

During design you can take steps to make testing easier. Testing, which is the subject of the next chapter, can be performed both manually and automatically. Automatic testing involves writing a program that will provide various inputs to the system in order to test it thoroughly. Therefore it pays to design a system so that automatic testing is made easy.

The most important way to design for testability is to ensure that all the functionality of the code can be executed without going through the graphical user interface. You can achieve this by carefully separating the UI from the functional layer of the system. A test harness can then be written that calls the API of the functional layer. Another good strategy is to provide a command-line version of your system, such as the command-line version of SimpleChat that we presented at the beginning of this book. This will allow you to write a test program that automatically issues commands to your application.

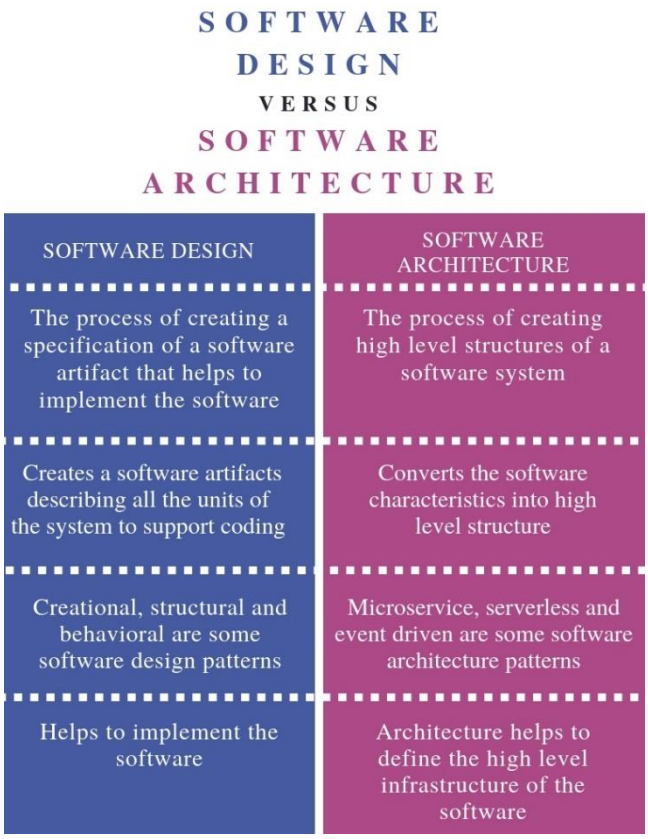
In order to design a Java class for testability you can create a `main` method in each class. Such `main` methods simply exercise the other methods of a class and report any problems.

Design Principle 11: Design defensively

You should never trust how others will try to use a component you are designing. Just like automobile drivers are taught not to trust other drivers, and therefore to *drive* defensively, a software designer should not trust other designers or programmers, and so should *design* defensively. In other words, in order to increase the reliability of your system, you not only need to make sure you don't add any defects yourself, but you must also properly handle all cases where other code attempts to use your component inappropriately.

#	Type of Comparison	Architectural Patterns	Design Patterns	D/S
1	**Occurrences**	Always comes before Design Patterns selection	Always comes after Architecture Patterns selection	Difference
2	**Deals with**	Deals with behavior of the whole system	Deals with classes, interfaces, objects, abstract classes	Difference
3	**Counts**	Few	Many	Difference
4	**Level**	Higher-level	Lower-level	Difference
5	**System Impact**	Have a global impact on the whole implementation of a system	Have a local impact on specific parts of the implementation of a system	Difference
6	**Organizations**	Large-scale systems	Small-scale systems	Difference
7	**Aspect**	Concerned with strategic aspects of the whole system	Concerned with technical aspects of an implementation	Difference
8	**View**	Fundamental structural organization for software systems	Solves reoccurring problems in software construction	Difference
9	**Example**	Model-View-Controller (MVC) Pattern	Singleton Pattern	Difference
10	**Format**	Follow similar "pattern" format	Follow similar "pattern" format	Similarity
11	**Problem**	Basic target is to solve the problem	Basic target is to solve the problem	Similarity
12	**Level of Problem**	Solves Architecture level problems	Solves Design level problems	Difference

Architectural pattern vs design pattern



Application Architecture describes the overall architecture of the software. For instance a web-based programs typically use a layered architecture where functionality is divided to several layers, such as user interface (html generation, handling commands from users), business logic (rules how the functions of the software are executed) and database (for persistent data). In contrast, a data processing application could use a so-called pipes and filters architecture, where a piece of data passes through a pipeline where different modules act on the data.

Design Patterns are a much lower level tool, providing proven models on how to organize code to gain specific functionality while not compromising the overall structure. Easy examples might include a Singleton (how to guarantee the existence of a single instance of a code) or a Facade (how to provide a simple external view to a more complex system).

On the other hand paradigms are the other extreme, guiding the principles on how code is actually laid out, and they each require quite different mindsets to apply. For instance, procedural programming is mainly concerned about dividing the program logic into functions and bundling those functions into modules. Object-oriented programming aims to encapsulate the data and the operations that manipulate the data into objects. Functional programming emphasizes the use of functions instead of separate statements following one another, avoiding side-effects and state changes.

Objective-C is mostly an object-oriented extension to C, design patterns and architecture are not language-specific constructs.

Requirement Analysis

4.4 What is a requirement?

Definition: a *requirement* is a statement describing either 1) an aspect of what the proposed system must do, or 2) a constraint on the system's development. In either case, it must contribute in some way towards adequately solving the customer's problem; the set of requirements as a whole represents a negotiated agreement among all stakeholders.

In systems engineering and software engineering, **requirements analysis** focuses on the tasks that determine the needs or conditions to meet the new or altered product or project, taking account of the possibly conflicting requirements of the various stakeholders, *analyzing, documenting, validating and managing* software or system requirements.

Activities for Requirement Analysis

Requirements analysis is critical to the success or failure of a systems or software project. The requirements should be documented, actionable, measurable, testable, traceable, related to identified business needs or opportunities, and defined to a level of detail sufficient for system design. Conceptually, requirements analysis includes four types of activity:

1. **Eliciting requirements**: the task of communicating with customers and users to determine what their requirements are. This is sometimes also called requirements gathering.
2. **Analyzing requirements**: determining whether the stated requirements are unclear, incomplete, ambiguous, or contradictory, and then resolving these issues.
3. **Requirements modeling**: Requirements might be documented in various forms, such as natural-language documents, use cases, user stories, or process specifications.
4. **Review and retrospective**: Team members reflect on what happened in the iteration and identifies actions for improvement going forward.

1.5 Software quality

Almost everybody says they want software to be of 'high quality'. But what does the word 'quality' really mean? There is no single answer to this question since, like beauty, quality is largely in the eye of the beholder.

Figure 1.1 shows what quality means to each of the stakeholders. They each consider the software to be of good quality if the outcome of its development and maintenance helps them meet their personal objectives.

Attributes of software quality

The following are five of the most important attributes of software quality. Software engineers try to balance the relative importance of these attributes so as to design systems with the best overall quality, as limited by the money and time available.

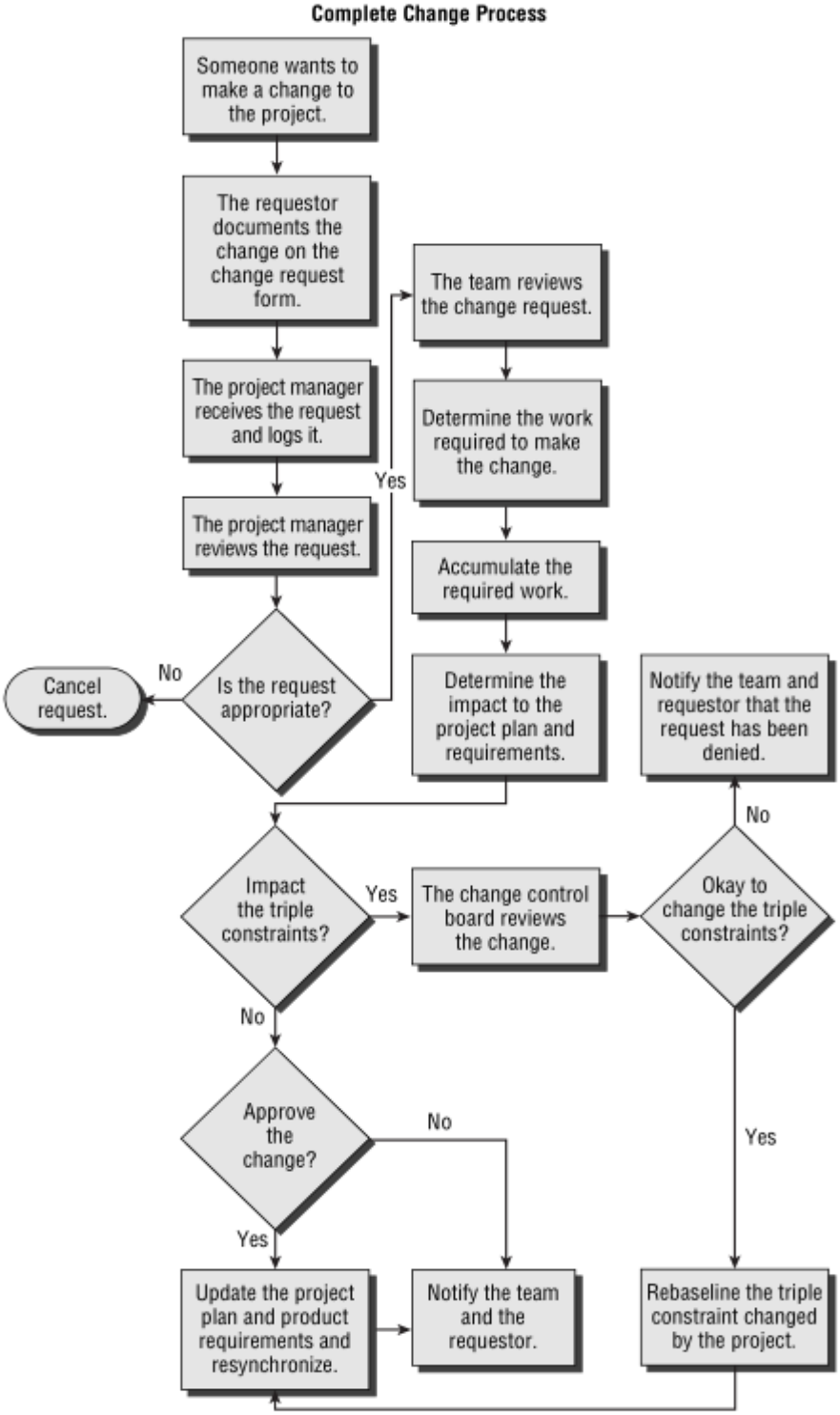
- **Usability**. The higher the usability of software, the easier it is for users to work with it. There are several aspects of usability, including learnability for novices, efficiency of use for experts, and handling of errors. We will discuss more about usability in Chapter 7.

Figure 1.1 What software quality means to different stakeholders

- **Efficiency**. The more efficient software is, the less it uses of CPU-time, memory, disk space, network bandwidth and other resources. This is important to customers in order to reduce their costs of running the software, although with today's powerful computers, CPU-time, memory and disk usage are less of a concern than in years gone by.
- **Reliability**. Software is more reliable if it has fewer failures. Since software engineers do not deliberately plan for their software to fail, reliability depends on the number and type of mistakes they make. Designers can improve reliability by ensuring the software is easy to implement and change, by testing it thoroughly, and also by ensuring that if failures occur, the system can handle them or can recover easily.
- **Maintainability**. This is the ease with which you can change the software. The more difficult it is to make a change, the lower the maintainability. Software engineers can design highly maintainable software by anticipating future changes and adding flexibility. Software that is more maintainable can result in reduced costs for both developers and customers.
- **Reusability**. A software component is reusable if it can be used in several different systems with little or no modification. High reusability can reduce the long-term costs faced by the development team. We will discuss reusable technology in Chapter 3.

Configuration management (**CM**) is a systems engineering process for establishing and maintaining consistency of a product's performance, functional, and physical attributes with its requirements, design, and operational information throughout its life

The **change management** process in systems engineering is the process of requesting, determining attainability, planning, implementing, and evaluating of changes to a system. Its main goals are to support the processing and traceability of changes to an interconnected set of factors.[1]



Design as 4 dimensional view of a system

Design as a series of decisions

Definition:	in the context of software, *design* is a problem-solving process whose objective is to find and describe a way to implement the system's functional requirements, while respecting the constraints imposed by the quality, platform and process requirements (including the budget and deadlines), and while adhering to general principles of good quality.

3(b) Design as four dimensional view of a system. Justify along with design concepts.

Sol'n → Software design is a process through which requirements are translated into a representation of software. From a project management point of view, software design can be conducted in two main steps!

(i) preliminary Design:

concerned with the transformation of requirements into data and software architecture.

(ii) Detail Design:

Focuses on refining the architectural representation and lead to detailed data structure and algorithmic representation of software.

The three main design activities concerned in Design phase are! Data Design, Architectural Design and procedural Design. In addition, many modern applications have a distinct interface design activity.

- Data design is used to transform the information domain into data structures.
- Architectural Design is used to develop a modular program structure and represent the control relationships between modules.
- procedural design is used to transform structural components into a procedural description of the software.
- Interface Design establishes the layout and interaction mechanisms for human-machine interaction.

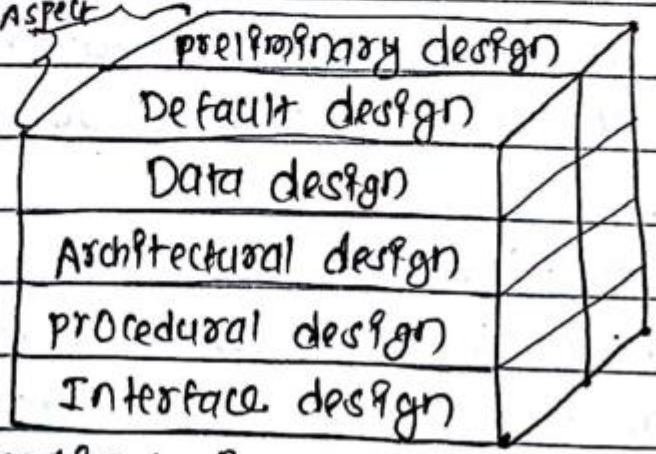


Figure: Relationship betn technical & Mgt. Aspect of design.

Use Case Driven

A **use case** is a sequence of actions, performed by one or more **actors** (people or non-human entities outside of the system) and by the system itself, that produces one or more results of value to one or more of the actors. One of the key aspects of the Unified Process is its use of use cases as a driving force for development. The phrase *use case driven* refers to the fact that the project team uses the use cases to drive all development work, from initial gathering and negotiation of requirements through code. (See "Requirements" later in this chapter for more on this subject.)

Use cases are highly suitable for capturing requirements and for driving analysis, design, and implementation for several reasons.

The benefits of basing software development on use cases

Use case analysis is an intuitive way to understand and organize what the system should do, since it is based on user tasks and expresses the tasks in natural language. It can also be used to *drive* the development process; in particular, use cases:

- Can help to define the *scope* of the system – that is, what the system must do and does not have to do.
- Are often used to plan the development process. The number of identified use cases is a good indicator of a project's size. Development progress can be measured in terms of the percentage of use cases that have been completed.
- Are used to both develop and validate the requirements. If some piece of proposed functionality does not support any use case, then it can be eliminated. Users and customers can also understand requirements better if they are expressed in terms of use cases. The use cases therefore can serve as part of the contract between the customers and the developer.
- Can form the basis for the definition of test cases (discussed in Chapter 10).
- Can be used to structure user manuals.

To quote, "A ***use - case realization*** describes how a particular use case is realized within the Design Model, in terms of collaborating objects" [RUP]. More precisely, a designer can describe the design of one or more *scenarios* of a use case; each of these is called a use case realization (though non - standard, perhaps better called a ***scenario realization*****).** *Use case realization* is a UP term used to remind us of the connection between the requirements expressed as use cases and the object design that satisfies the requirements

A use case realization provides a construct to organize artifacts which shown how the physical design of a system supports the logical business behavior outlined by a used case. To show this traceability between the logical and physical design, in the use case diagram each use case is depicted as an oval shown with a solid-line border. Then for each use case can you may show a use case realization as an oval with a dotted-line border. The use case and its corresponding use case realization are linked using a realization dependency which is shown in UML as a dashed line with a triangular arrowhead at the end corresponding to the realized element (the use case).

Each use case realization will define the physical design in terms of classes and collaborating objects which support the use case. Therefore, each use case realization typically is made up of a class diagram and a number of interaction diagrams, most commonly sequence diagrams, showing the collaboration or interaction between physical objects.

Use case realizations allow the analyst to clearly separate the concerns of the logical and physical design. Since a logical design (the business behavior and requirements) can be implemented or realized via a number of different physical implementations, if a physical design changes the logical use case can remain unaffected. Also, a use case realization is an excellent form of requirements traceability from the logical business requirements down to the physical implementation of the solution.

Player-Role Pattern

6.4 The Player–Role pattern

Context This modeling pattern can solve modeling problems when you are drawing many different types of class diagram. A *role* is a particular set of features

associated with an object in a particular context. An object may *play* different roles in different contexts.

For example, a student in a university can be either an undergraduate student or a graduate student at any point in time – and is likely to need to change from one of these roles to another. Similarly, a student can also be registered in his or her program full-time or part-time, as shown in Figure 5.16; in this case, a student may change roles several times. Finally, an animal may play several of the roles shown in Figure 5.15, although in this case the roles are unlikely to change.

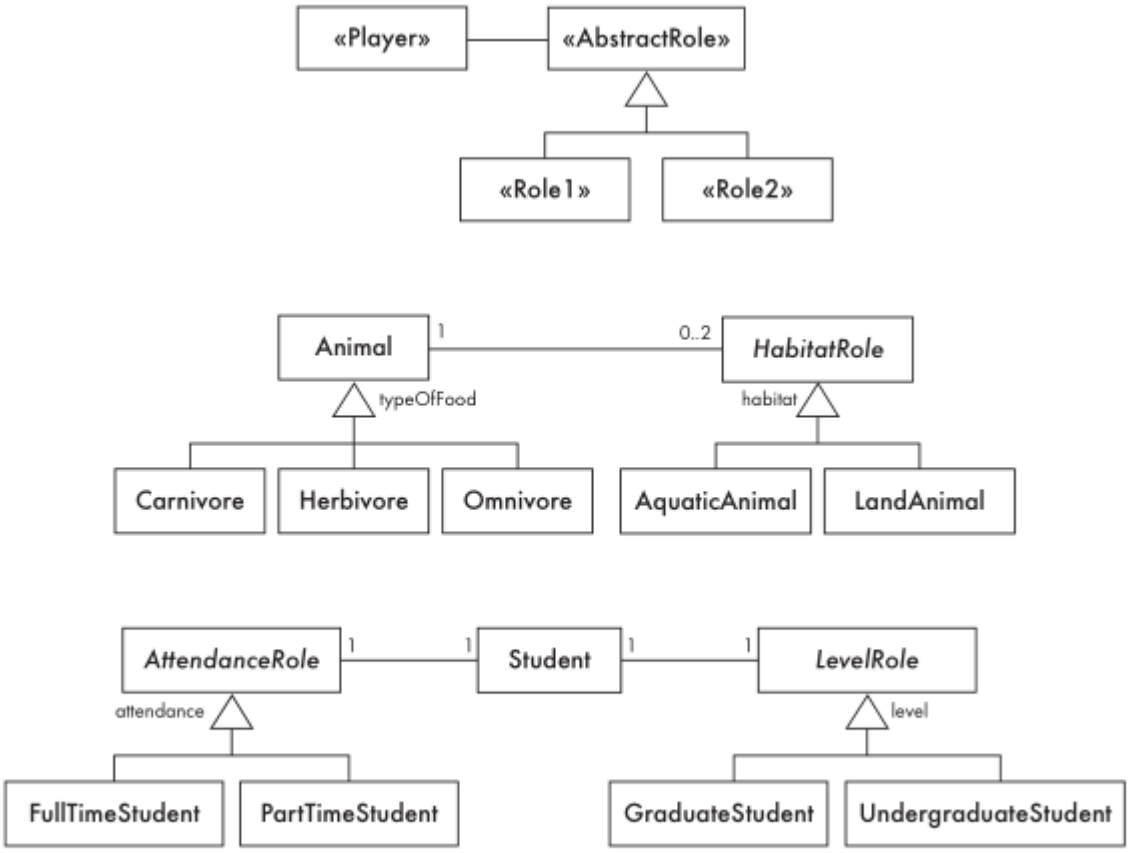
Problem How do you best model players and roles so that a player can change roles or possess multiple roles?

Forces It is desirable to improve encapsulation by capturing the information associated with each separate role in a class. However, as discussed in Chapter 5, you want to avoid multiple inheritance. Also, you cannot allow an instance to change class.

Solution Create a «*Player*» class to represent the object that plays different roles. Create an association from this class to an abstract «*Role*» class, which is the superclass of a set of possible roles. The subclasses of this «Role» class encapsulate all the features associated with the different roles.

If the «Player» can only play one role at a time, the multiplicity between «Player» and «Role» can be one-to-one, otherwise it will be one-to-many.

Instead of being an abstract class, the «Role» can be an interface. The only drawback to this variation is that the «Role» usually contains a mechanism, inherited by all its subclasses, allowing them to access information about the «Player». Therefore you should only make «Role» an interface if this mechanism is not needed.



gure 6.6 Template and examples of the Player–Role design pattern

Observer Pattern

Observer Pattern

An Observer Pattern says that "just define a one-to-one dependency so that when one object changes state, all its dependents are notified and updated automatically".

Advantages

- It describes the coupling between the objects and the observer.
- It provides the support for broadcast-type communication.

Uses

- When the change of a state in one object must be reflected in another object without keeping the objects tight coupled.
- When the framework we writes and needs to be enhanced in future with new observers with minimal changes.

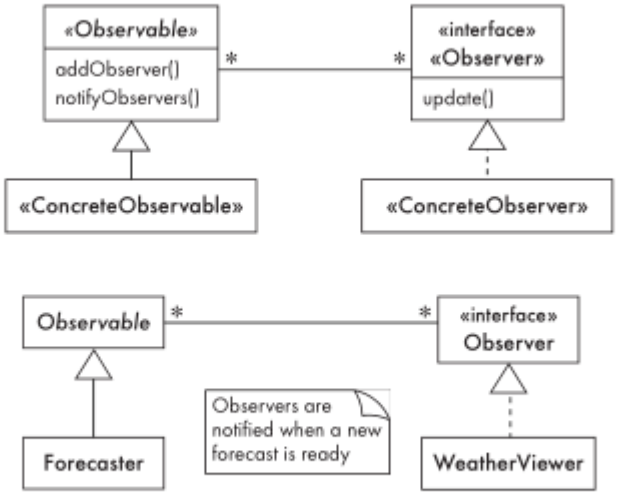
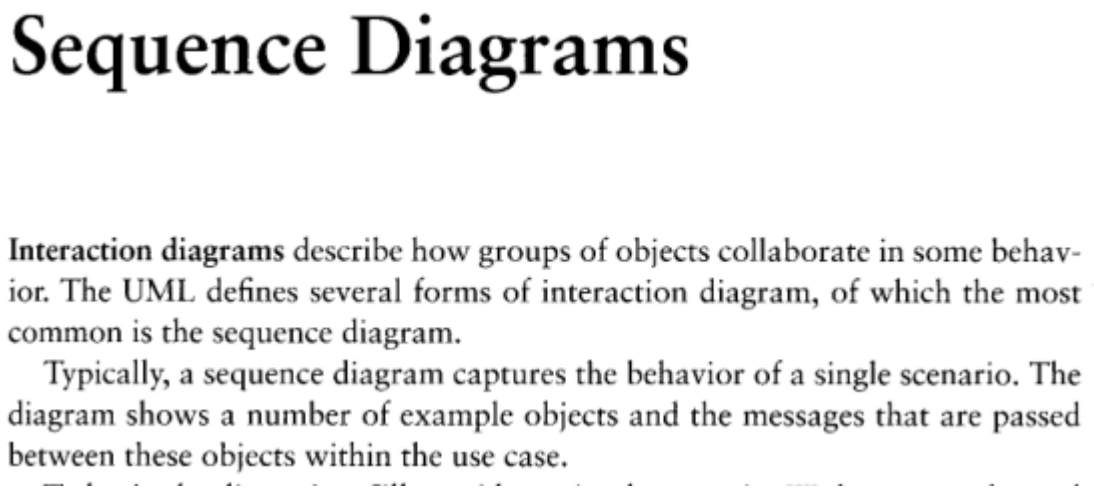


Figure 6.8 Template and example of the Observer design pattern

Interaction Diagram



Sequence Diagrams

Interaction diagrams describe how groups of objects collaborate in some behavior. The UML defines several forms of interaction diagram, of which the most common is the sequence diagram.

Typically, a sequence diagram captures the behavior of a single scenario. The diagram shows a number of example objects and the messages that are passed between these objects within the use case.

Reuse

Reusability: "... the ability of something to be used more than once ..."

Reuse: "... the action of using something more than once ..."

The difference is simple:

Reusability promotes and enables reuse but does not ensure it.

This semantic difference between ability and action is a key factor to consider within Enterprise IT Initiatives. The consequences of blindly promoting reusability are negative:

n computer science and software engineering, **reusability** is the use of existing *assets* in some form within the software product development process; these *assets* are products and by-products of the software development life cycle and include code, software components, test suites, designs and documentation. The opposite concept of *reusability* is **leverage**, which modifies existing assets as needed to meet specific system requirements. Because reuse implies the creation of a separately maintained version of the assets[*clarification needed*], it is preferred over leverage.[1]

Code reuse, also called **software reuse**, is the use of existing software, or software knowledge, to build new software,[1] following the reusability principles.

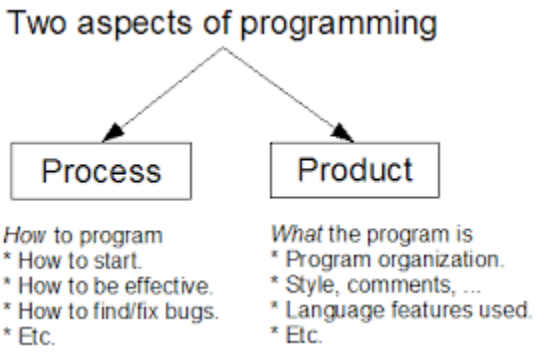
Designing with reuse is complementary to designing for reusability.

you can build reusability into algorithms,
classes, procedures, frameworks and complete applications whereas in reuse you just reuse.

Analysis Model vs Design Model

Analysis vs Design (USP)

Analysis Model	Design Model
Satisfies functional requirements.	Satisfies both functional and non-functional requirements.
A conceptual model, because it is an abstraction of the system and avoids implementation issues.	A physical model, because it is a blueprint for implementation.
Design-generic (applicable to several possible designs).	Specific for an implementation.
Based on three stereotyped classes that are conceptual in nature: Entity, Boundary, and Control.	Any number of physical class stereotypes and modules that may be implementation language dependent.
Less formal.	More formal.
Less effort to develop (1:5 ratio)	More effort to develop
Few layers	Many layers
Dynamic, but not much focus on sequence.	Dynamic with much more emphasis on sequence, concurrency, and distribution.
Outlines the design and architecture of the system.	Manifests or realizes the design (an instantiation of the Analysis Model).
May not be maintained throughout the lifecycle.	Should be maintained throughout the lifecycle.
Defines a structure that is an essential input to shaping the system including the Design Model.	Defines the structure of the system wrt both functional and non-functional requirements.



Enterprises are typically comprised of hundreds if not thousands of applications that are custom-built, acquired from a third-party, part of a legacy system, or a combination thereof, operating in multiple tiers of different operating system platforms. It is not uncommon to find an enterprise that has 30 different Websites, three instances of SAP and countless departmental solutions.

Integration styles and types [edit]

The book distinguishes four top-level alternatives for integration:

1. File Transfer
2. Shared Database
3. Remote Procedure Invocation
4. Messaging

The following integration types are introduced:

- Information Portal
- Data Replication
- Shared Business Function
- Service Oriented Architecture
- Distributed Business Process
- Business-to-Business Integration
- Tightly Coupled Interaction vs. Loosely Coupled Interaction

Messaging [edit]

- Message Channel
- Message
- Pipes and Filters
- Message Router
- Message Translator
- Message Endpoint

Message Channel [edit]

- Point-to-Point Channel
- Publish-Subscribe Channel
- Datatype Channel
- Invalid Message Channel
- Dead Letter Channel
- Guaranteed Delivery
- Channel Adapter
- Messaging Bridge
- Message Bus

Message Construction [edit]

- Command Message
- Document Message
- Event Message
- Request-Reply
- Return Address
- Correlation Identifier
- Message Sequence
- Message Expiration
- Format Indicator

Message Router [edit]

- Content-Based Router
- Message Filter
- Dynamic Router
- Recipient List
- Splitter
- Aggregator
- Resequencer
- Composed Message Processor
- Scatter-Gather
- Routing Slip
- Process Manager
- Message Broker

Message Transformation [edit]

- Envelope Wrapper
- Content Enricher
- Content Filter
- Claim Check
- Normalizer
- Canonical Data Model

Message Endpoint [edit]

- Messaging Gateway
- Messaging Mapper
- Transactional Client
- Polling Consumer
- Event-Driven Consumer
- Competing Consumers
- Message Dispatcher
- Selective Consumer
- Durable Subscriber
- Idempotent Receiver
- Service Activator

System Management [edit

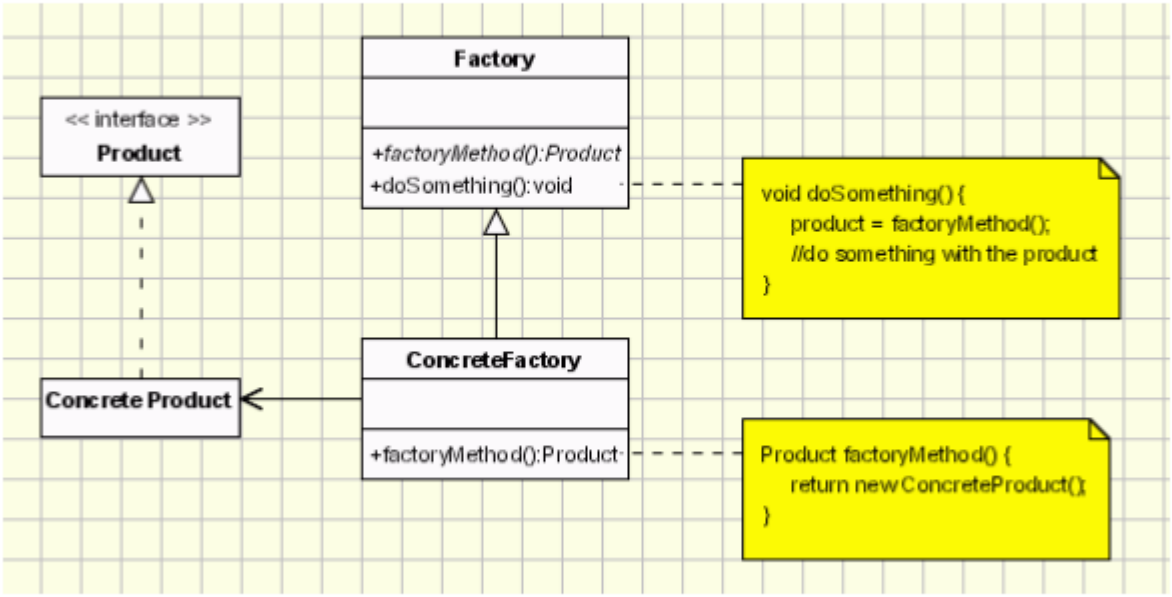
- Control Bus
- Detour
- Wire Tap
- Message History
- Message Store
- Smart Proxy
- Test Message
- Channel Purger

Real-time and embedded systems (RTE systems) must execute in a much more constrained environment than "traditional" computer systems such as desktop and mainframe computers. RTE systems must be highly efficient, optimally utilizing their limited processor and memory resources, and yet must often outperform systems with significantly more compute power. In addition, many RTE systems have important safety-critical and high-reliability requirements because they are often used in systems such as avionics flight control, nuclear power plant control, life support and medical instrumentation. The creation of RTE systems to meet these functional and quality of service requirements requires highly experienced developers with decades of experience. Yet, over the years, these developers have encountered the same problems over and over—maybe not exactly the same problems but common threads. The very best developers abstract these problems and their solutions into generalized approaches that have proved consistently effective. These generalized approaches are called design patterns. They are often best applied at the level of the system or software architecture—the sum of design decisions that affect the fundamental organization of the system. Real-Time Design Patterns is an attempt to capture in one place a set of architectural design patterns that are useful in the development of RTE systems.

Factory Method

Factory Method

It is a creational design pattern that provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created.



The participants classes in this pattern are:

- **Product** defines the interface for objects the factory method creates.
- **ConcreteProduct** implements the Product interface.
- **Creator**(also refered as **Factory** because it creates the Product objects) declares the method **FactoryMethod**, which returns a Product object. May call the generating method for creating Product objects
- **ConcreteCreator** overrides the generating method for creating ConcreteProduct objects

All concrete products are subclasses of the Product class, so all of them have the same basic implementation, at some extent. The Creator class specifies all standard and generic behavior of the products and when a new product is needed, it sends the creation details that are supplied by the client to the ConcreteCreator.

Applicability & Examples
The need for implementing the Factory Method is very frequent. The cases are the ones below:
when a class can't anticipate the type of the objects it is supposed to create
when a class wants its subclasses to be the ones to specific the type of a newly created object

Façade Pattern

6.9 The Façade pattern

Context Often, an application contains several complex packages. A programmer working with such packages has to manipulate many different classes.

Problem How do you simplify the view that programmers have of a complex package?

Forces It is hard for a programmer to understand and use an entire subsystem – in particular, to determine which methods are public. If several different application classes call methods of the complex package, then any modifications made to the package will necessitate a complete review of all these classes.

Solution Create a special class, called a «Façade», which will simplify the use of the package. The «Façade» will contain a simplified set of public methods such that most other subsystems do not need to access the other classes in the package. The net result is that the package as a whole is easier to use and has a reduced number of dependencies with other packages. Any change made to the package should only necessitate a redesign of the «Façade» class, not classes in other packages.

Example The airline system discussed in Chapter 5 has many classes and methods. Other subsystems that need to interact with the airline system risk being 'exposed' to any changes that might be made to it. We can therefore define the class `Airline` to be a «Façade», as shown in Figure 6.11. This provides access to the most important query and booking operations.

References This pattern is one of the 'Gang of Four' patterns.

Façade Pattern

just provide a unified and simplified interface to a set of interfaces in a subsystem, therefore it hides the complexities of the subsystem from the client

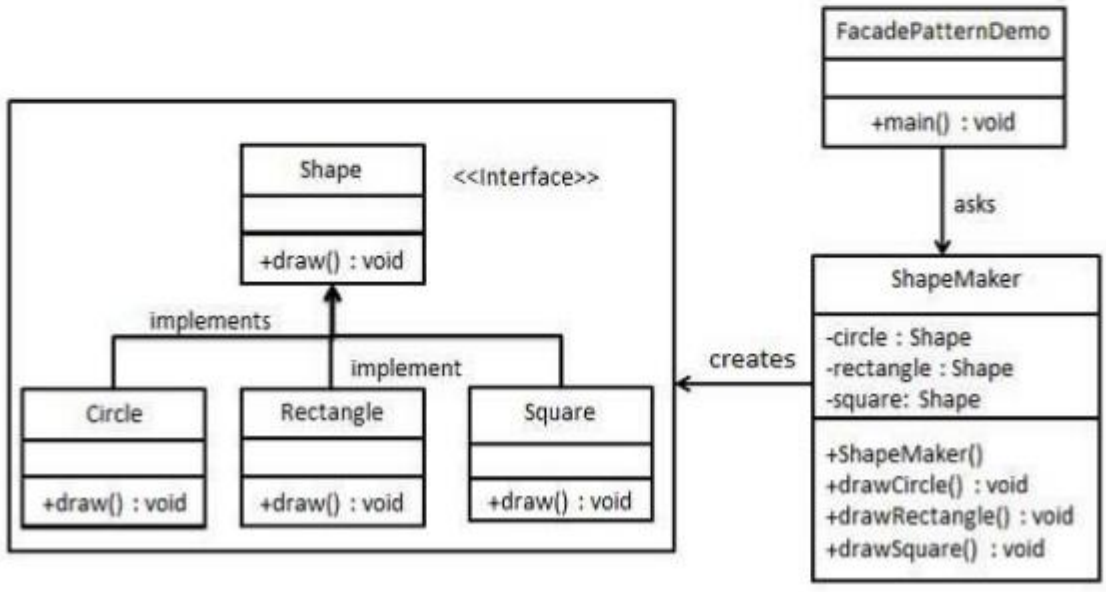
Facade Pattern describes a higher-level interface that makes the sub-system easier to use.

Advantages

- It shields the clients from the complexities of the sub-system components.
- It promotes loose coupling between subsystems and its clients.

Uses

- When you want to provide simple interface to a complex sub-system.
- When several dependencies exist between clients and the implementation classes of an abstraction.



Architecture Centric Process

Aspects of an architecture include static elements, dynamic elements, how those elements work together, and the overall architectural style that guides the organization of the system.

Architecture also addresses issues such as performance, scalability, reuse, and economic and technological constraints.

The Unified Process specifies that the architecture of the system being built, as the fundamental foundation on which that system will rest,must sit at the heart of project team's efforts to shape the system.

Need of architecture centric:

- Understanding the big picture
- Organizing development effort
- Facilitating the possibilities for reuse
- Guiding the use cases
- Evolving the system