Amity School of Engineering & Technology
Amity University

Object Oriented System Design

Code: CSE416

Prepared By

Hari Mohan Pandey
Assistant Professor, CSE Department
hmpandey@amity.edu

Module Notes

Department of Computer Science & Engineering
ODD Semester 2014

UNIT-3
DIAGRAMS

3.1 Sequence diagrams

A sequence diagram is an interaction diagram that details how operations are carried out -- what messages are sent and when. Sequence diagrams are organized according to time. The time progresses as you go down the page. The objects involved in the operation are listed from left to right according to when they take part in the message sequence.

Below is a sequence diagram for making a hotel reservation. The object initiating the sequence of messages is a Reservation window.

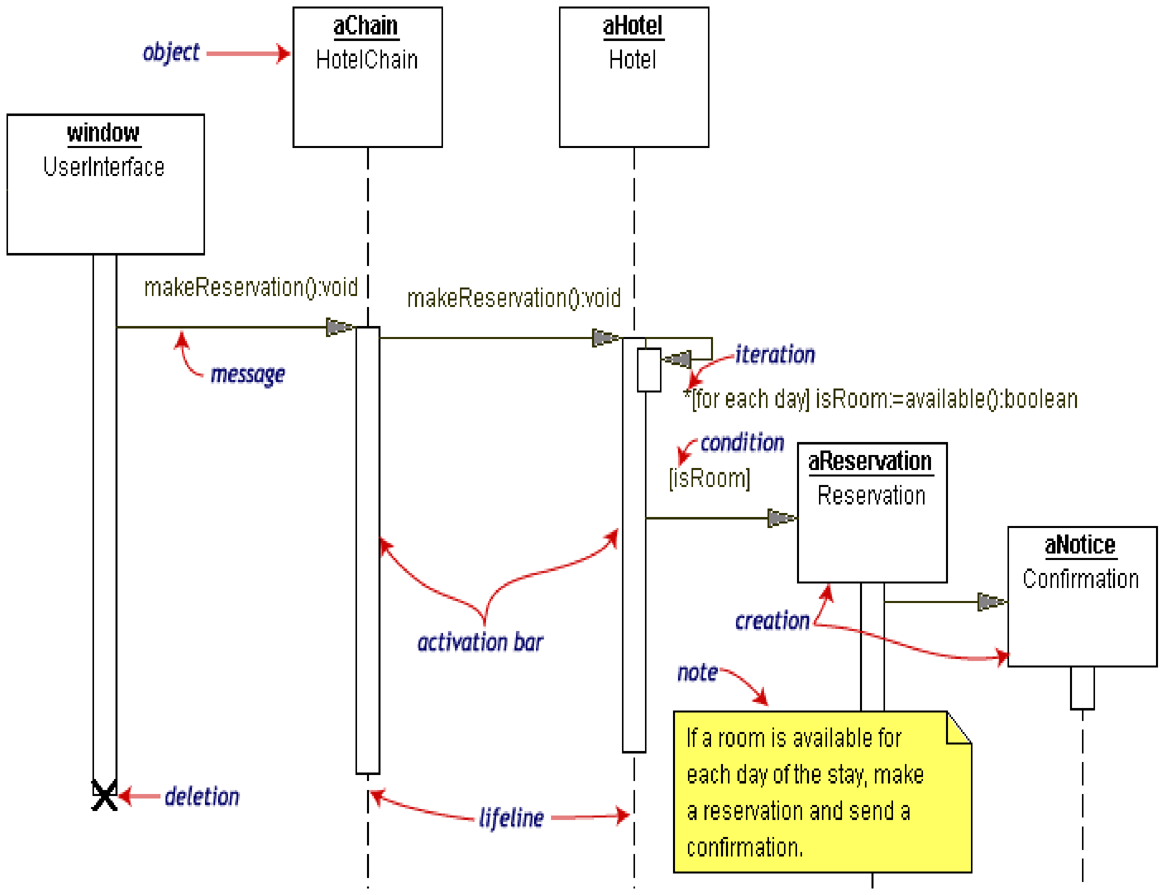


Figure: Sequence Diagram-1

The Reservation window sends a **makeReservation ()** message to a **HotelChain.** The **HotelChain** then sends a **makeReservation ()** message to a Hotel. If the Hotel has available rooms, then it makes a Reservation and a Confirmation.

Within a sequence diagram, on object is available in the box at the top of a dotted vertical line. Each vertical dotted line is a lifeline, representing the time that an object exists. Each arrow is a message call. An arrow goes from the sender to the top of the activation bar of the message on the receiver's lifeline. The activation bar represents the duration of execution of the message. Each message is labeled at minimum with the message name.
In our diagram, the Hotel issues a self call to determine if a room is available. If so, then the Hotel creates a Reservation and a Confirmation. The asterisk on the self call means iteration (to make sure there is available room for each day of the stay in the hotel). The expression in square brackets, [ ], is a condition.

The diagram has a clarifying note, which is text inside a dog-eared rectangle. Notes can be put into any kind of UML diagram.

Another Example

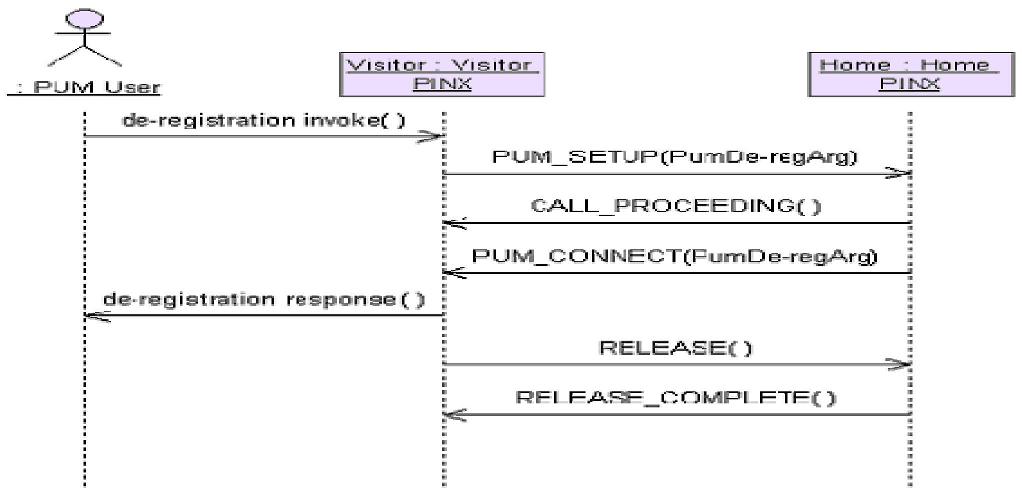
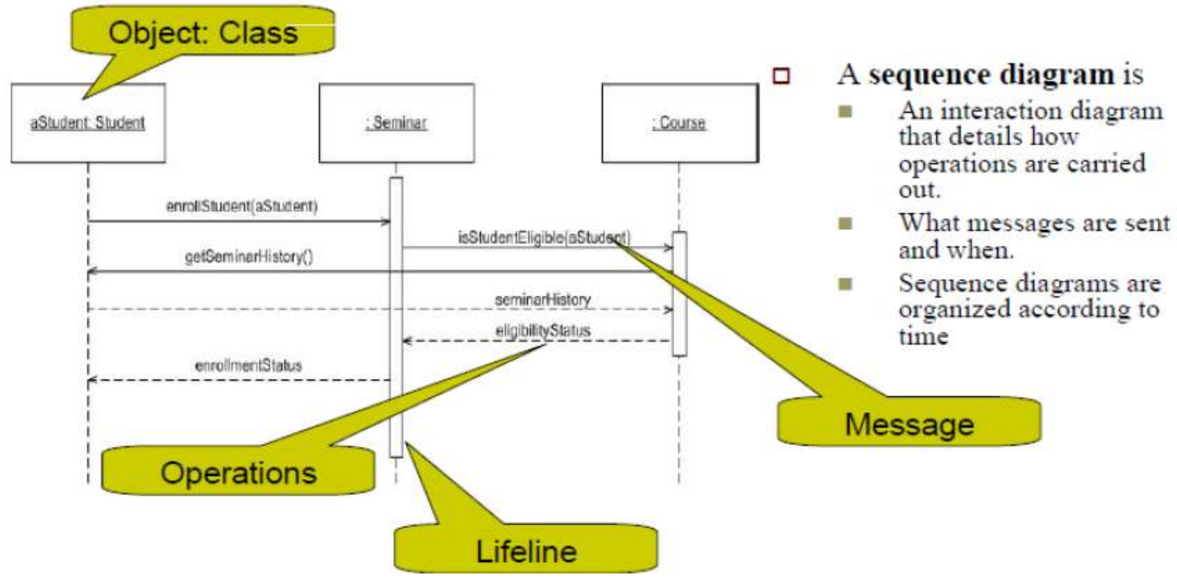


Figure: Sequence Diagram-2

Guide lines for Drawing Sequence Diagrams

1. An actor that initiates the interaction is often shown on the left.

2. The vertical dimension represents time.
3. A vertical line, called a lifeline, is attached to each object or actor.
4. The lifeline becomes a broad box, called an activation box during the live activation period.
5. A message is represented as an arrow between activation boxes of the sender and receiver.



- A **sequence diagram** is
  - An interaction diagram that details how operations are carried out.
  - What messages are sent and when.
  - Sequence diagrams are organized according to time

3.2 Collaboration diagrams

Collaboration Diagrams describe interactions among classes and associations. These interactions are modeled as exchanges of messages between classes through their associations. Collaboration diagrams are a type of interaction diagram. Collaboration diagrams contain the following elements:

• **Class roles,** which represent roles that objects may play within the interaction.

• **Association roles,** which represent roles that links may play within the interaction.

• **Message flows,** which represent messages sent between objects via links. Links transport or implement the delivery of the message.

3.3 State chart diagrams

Objects have behaviors and state. The state of an object depends on its current activity or condition. A state chart diagram shows the possible states of the object and the transitions that cause a change in state.

UML state machine diagrams depict the various states that an object may be in and the transitions between those states. In fact, in other modeling languages, it is common for this type of a diagram to be called a state-transition diagram or even simply a state diagram. A state represents a stage in the behavior pattern of an object, and like UML activity diagrams it is possible to have initial states and final states. An initial state, also called a creation state, is the one that an object is in when it is first created, whereas a final state is one in which no transitions lead out of. A transition is a progression from one state to another and will be triggered by an event that is either internal or external to the object.

Top-level state machine diagram is shown below for seminar enrollment, teaching and final exams for a specific subject. The arrows represent transitions, progressions from one state to another.

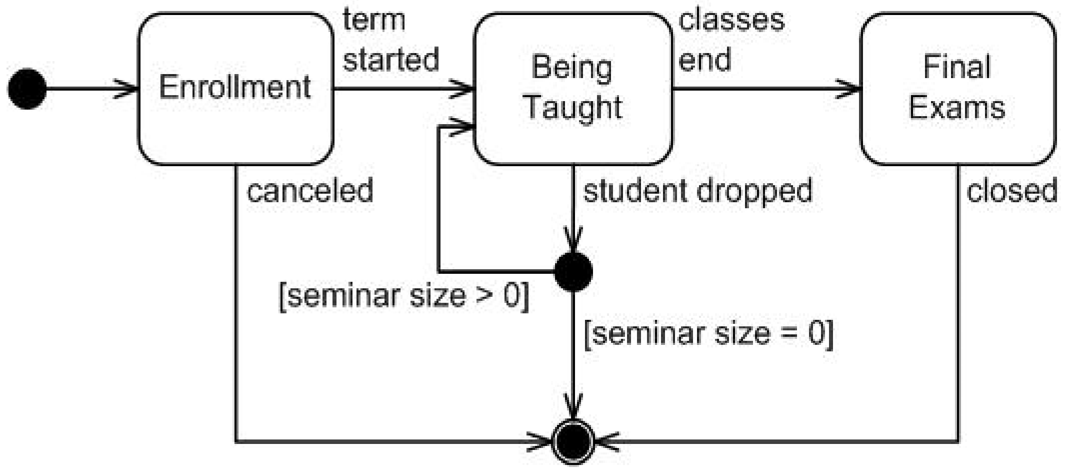


Figure: State Chart Diagram-1

Following figure presents sub-states of enrollment state during seminar class registration.

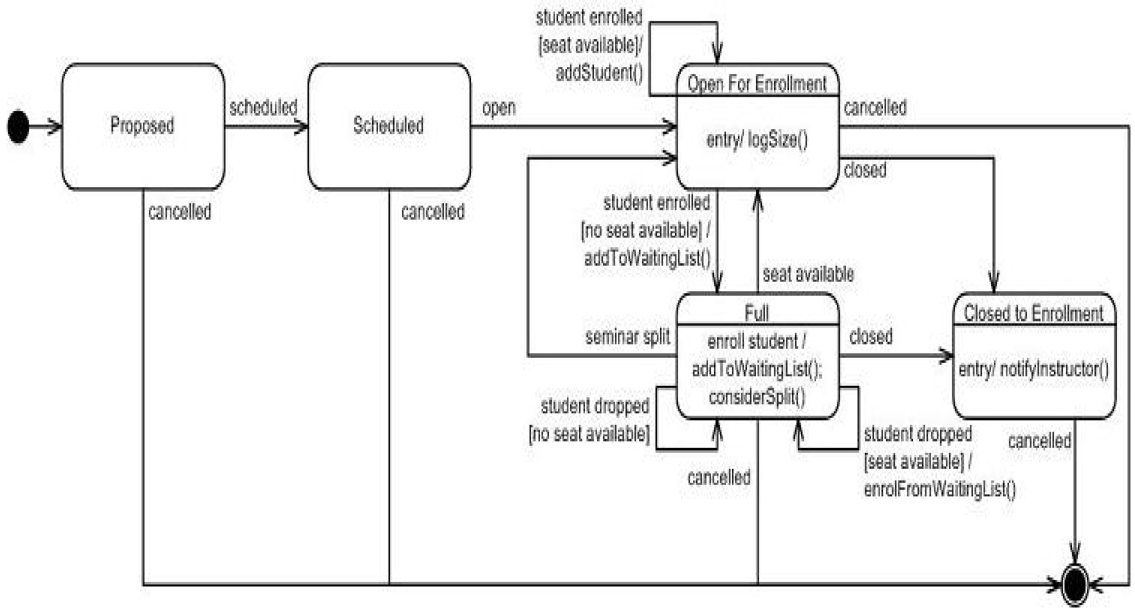


Figure: State Chart Diagram-2

3.4 Realizing Use Cases in Sequence Diagrams

- Realizing use cases by means of sequence diagrams is an important part of our analysis.
- It ensures that we have an accurate and complete class diagram.
- The sequence diagrams increase the completeness and understandability of our analysis model.
- The behavior is associated with the class the first time it is required, and then the behavior is reused for every other use case that requires the behavior.
- When assigning behaviors or responsibilities to a class while mapping a use case to the analysis model, you must take special care to assign the responsibility to the correct class.
- The responsibility or behavior belongs to the class if it is something you would do to the thing the class represents.

Example

- Suppose you were asked to read the first paragraph of three chapters of book. First, you would need to know where to go to get the book. We might state that all books we are referring to are available at the Fourth St. library. You might then know to first go to the library, but the library has thousands of books. You might next have to consult the card catalog to determine where the book is located and then retrieve the book. Next, you might look at the book's table of contents to determine which pages concern you and then turn to those pages. We could consider the library as the whole and the books as the part of the whole-part relationship. The relationship between the book and the pages could also be viewed as a whole-part relationship. When we determined where to look and then proceeded to find that point, we were navigating the whole-part relationship.

- Every time you find yourself navigating the whole-part relationship to find the appropriate class, you will need to assign responsibilities to the classes you are navigating to ensure that you can, in fact, find the appropriate class.
- Said another way, the navigating behavior must be a method on the class representing the whole.
- It is not unusual that this requires returning to the class that represents the system itself.

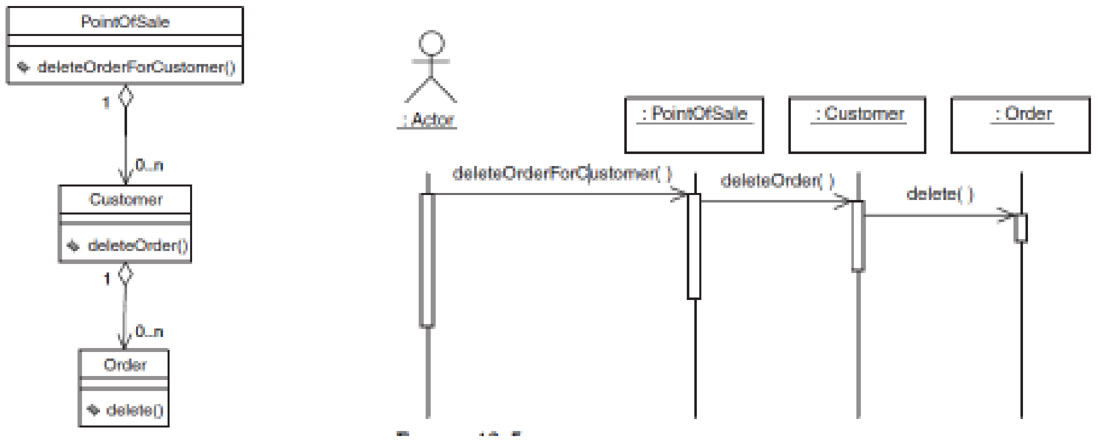


Figure: class diagram and sequence diagram for the class diagram

- Suppose the system receives a message from an actor requesting that you delete a given order belonging to a given customer.
- The sequence diagram might look like as shown previously.
- The sequence diagram requires the system to navigate the whole-part relationships to delete the order specified by the object of type *ACTOR.*
- *The* sequence of events begins when the object of type *ACTOR requests that a specific* order be deleted for a specific customer.
- There is no way for the object of type *ACTOR to call the delete method on the object of the type ORDER* because the object of type *ACTOR does not have a reference to the specific* order.
- It is appropriate for the object of type *ACTOR to have a reference to the* object of type *POINTOFSALE.*
- *This follows because there is only one object of* type *POINTOFSALE and it can, therefore, be referenced by name.*
- *The logical* starting point for all interaction with the actor is the object of type *POINTOFSALE.*
- The objects of type *ACTOR can then traverse the whole-part relationships* to arrive at the specific object of type *ORDER for which the action is* intended.

- The responsibilities for navigating the whole-part relationship result in assigning behaviors to the object of type *POINTOFSALE and the* object of type *CUSTOMER.*

3.5 Logical Architecture and UML Package Diagrams

3.5.1 Logical Architecture and Layers

- Logical architecture: the large-scale organization of software classes into packages, subsystems, and layers.
  - "Logical" because no decisions about deployment are implied. (See Chap. 37.)
- Layer: a very coarse-grained grouping of classes, packages, or subsystems that has cohesive responsibility for a major aspect of the system.

3.5.2 Layered Architectures

- Typical layers in an OO system:
  - User Interface
  - Application Logic and Domain Objects
  - Technical Services
    - Application-independent, reusable across systems.
- Relationships between layers:
  - Strict layered architecture: a layer only calls upon services of the layer directly below it.
  - Relaxed layered architecture: a higher layer calls upon several lower layers.

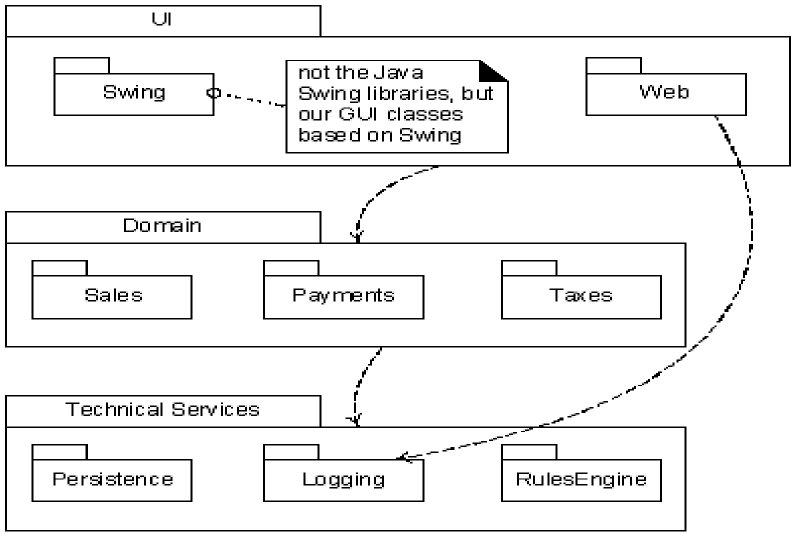


Figure: Layers shown with UML package diagrams

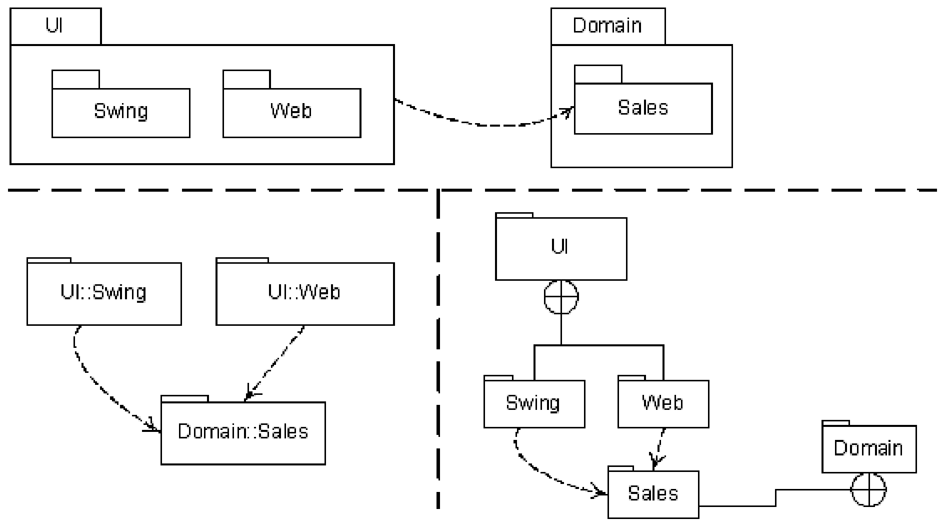


Figure: Various UML notations for packages nesting

3.5.3 Design with Layers

- Organize the large-scale logical structure of a system into discrete layers of distinct, related responsibilities.
  - Cohesive separation of concerns.
  - Lower layers are general services.
  - Higher layers are more application-specific.
- Collaboration and coupling is from higher to lower layers.
  - Lower-to-higher layer coupling is avoided.

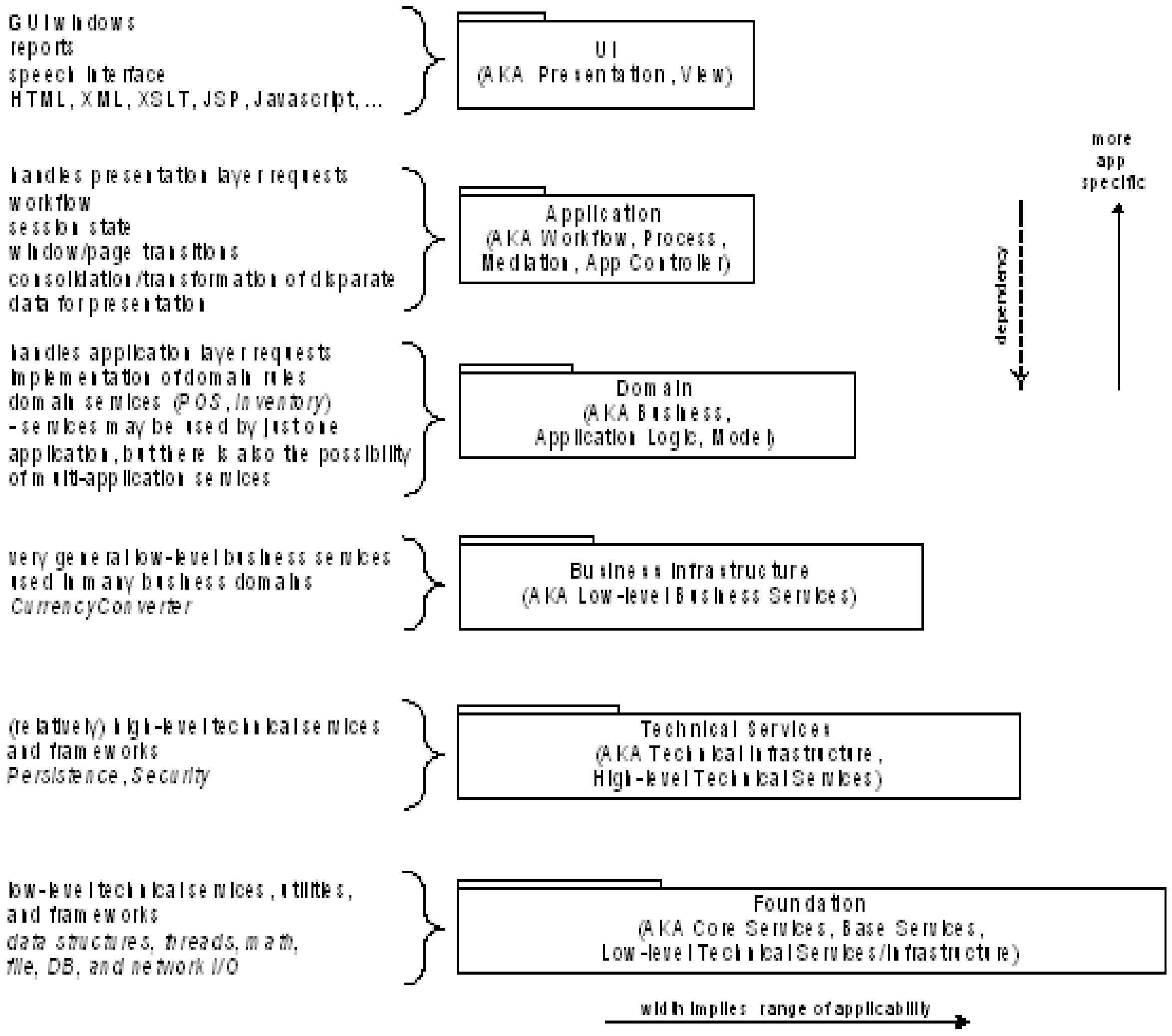

GUI windows
reports
speech interface
HTML, XML, XSLT, JSP, Javascript, ...
UI
(AKA Presentation, View)
handles presentation layer requests
workflow
session state
window/page transitions
consolidation/transformation of disparate data for presentation
Application
(AKA Workflow, Process, Mediation, App Controller)
handles application layer requests
implementation of domain rules
domain services (POS, Inventory)
- services may be used by just one application, but there is also the possibility of multi-application services
Domain
(AKA Business, Application Logic, Model)
very general low-level business services used in many business domains
CurrencyConverter
Business Infrastructure
(AKA Low-level Business Services)
(relatively) high-level technical services and frameworks
Persistence, Security
Technical Services
(AKA Technical Infrastructure, High-level Technical Services)
low-level technical services, utilities, and frameworks
data structures, threads, math, file, DB, and network I/O
Foundation
(AKA Core Services, Base Services, Low-level Technical Services/Infrastructure)
more app specific
dependency
width implies range of applicability

Figure: Common layers in an IS logical architecture

3.5.4 Benefits of a Layered Architecture

- Separation of concerns:

  E.g., UI objects should not do application logic (a window object should not calculate taxes) nor should a domain layer object create windows or capture mouse events.

  - Reduced coupling and dependencies.
  - Improved cohesion.
  - Increased potential for reuse.
  - Increased clarity.
- Related complexity is encapsulated and decomposable.
- Some layers can be replaced with new implementations.
- Lower layers contain reusable functions.
- Some layers can be distributed.

    – Especially Domain and Technical Services.
- Development by teams is aided by logical segmentation.

Designing the Domain Layer

How do we design the application logic with objects?

- Create software objects with names and information similar to the real-world domain.
- Assign application logic responsibilities to these domain objects.
    - E.g., a *Sale* object is able to calculate its total.

The application logic layer is more accurately called a domain layer when designed this way.

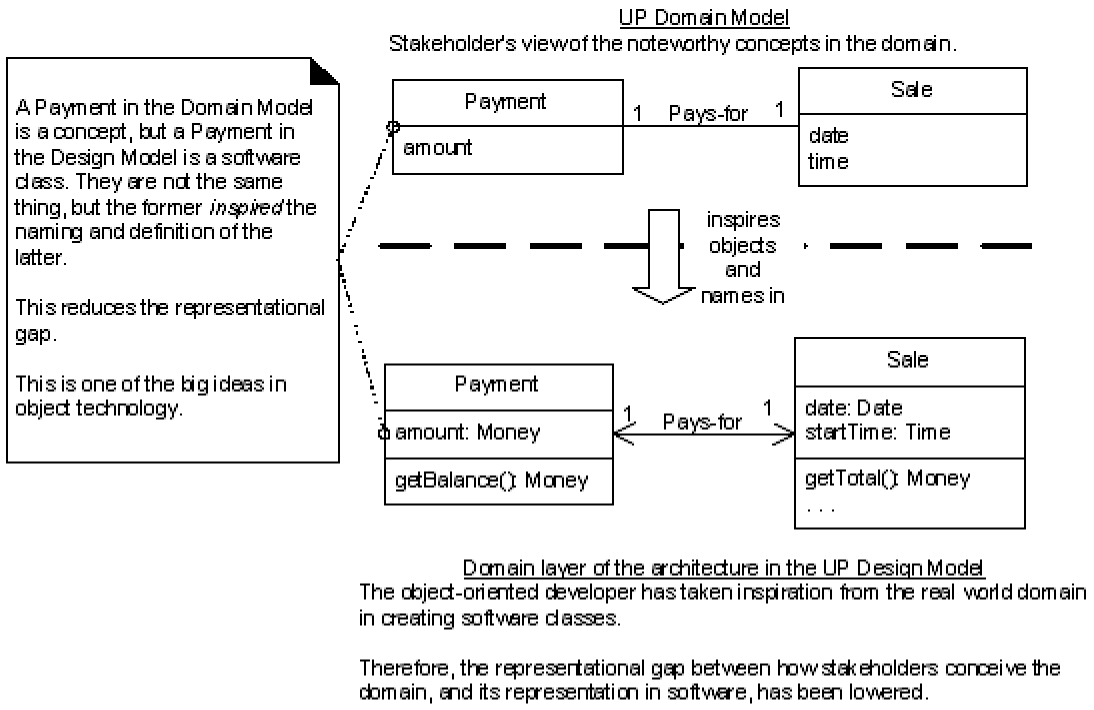

UP Domain Model
Stakeholder's view of the noteworthy concepts in the domain.
A Payment in the Domain Model is a concept, but a Payment in the Design Model is a software class. They are not the same thing, but the former *inspired* the naming and definition of the latter.
This reduces the representational gap.
This is one of the big ideas in object technology.
Payment
amount
1
Pays-for
1
Sale
date
time
inspires objects and names in
Payment
amount: Money
getBalance(): Money
1
Pays-for
1
Sale
date: Date
startTime: Time
getTotal(): Money
. . .
Domain layer of the architecture in the UP Design Model
The object-oriented developer has taken inspiration from the real world domain in creating software classes.
Therefore, the representational gap between how stakeholders conceive the domain, and its representation in software, has been lowered.

Figure: Domain Model Related to Domain Layer

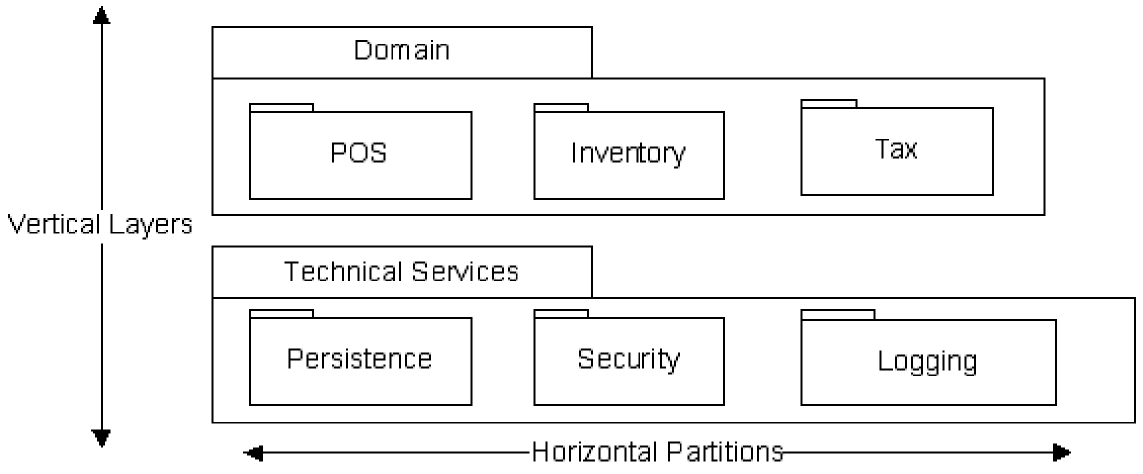


Figure: Layers vs. Partitions

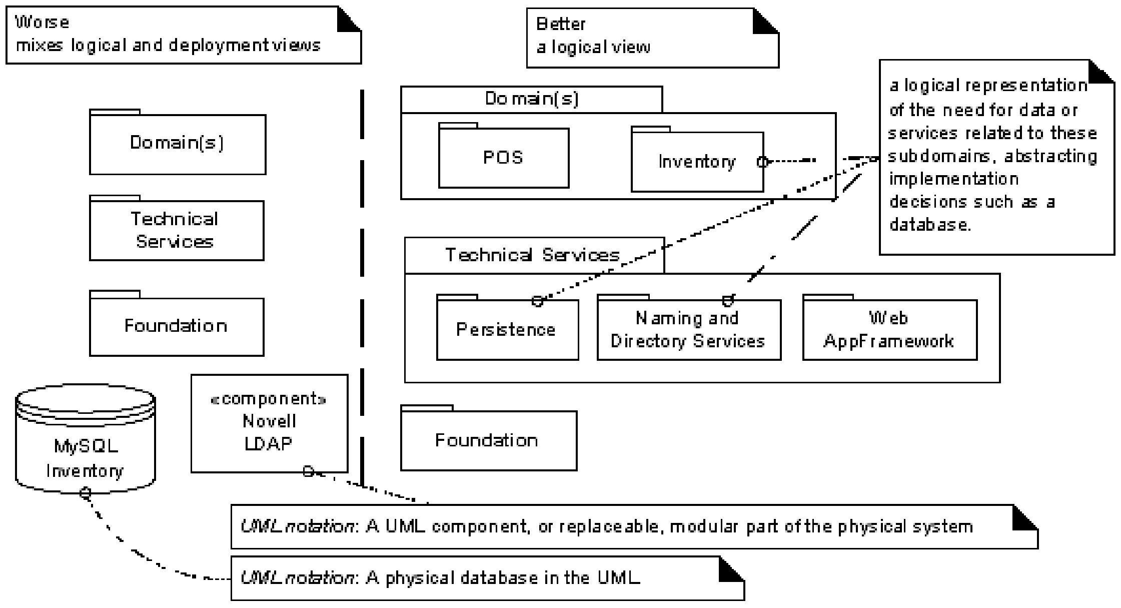

Worse
mixes logical and deployment views
Better
a logical view
Domain(s)
Technical Services
Foundation
MySQL Inventory
«component» Novell LDAP
Domain(s)
POS
Inventory
a logical representation of the need for data or services related to these subdomains, abstracting implementation decisions such as a database.
Technical Services
Persistence
Naming and Directory Services
Web AppFramework
Foundation
UML notation: A UML component, or replaceable, modular part of the physical system
UML notation: A physical database in the UML

Figure: Don’t mix logical and deployment views

3.5.5 The Model-View Separation Principle

- Model: the domain layer of objects.
- View: user interface (UI) objects.
- Model objects should not have direct knowledge of view objects.
  - Do not connect or couple non-UI objects directly to UI objects.
    - E.g., don’t let a *Sale* object have a reference to a Java Swing *JFrame* window object.
  - Do not put application logic in a UI object.

- UI objects should receive UI events and delegate requests for application logic to non-UI objects.

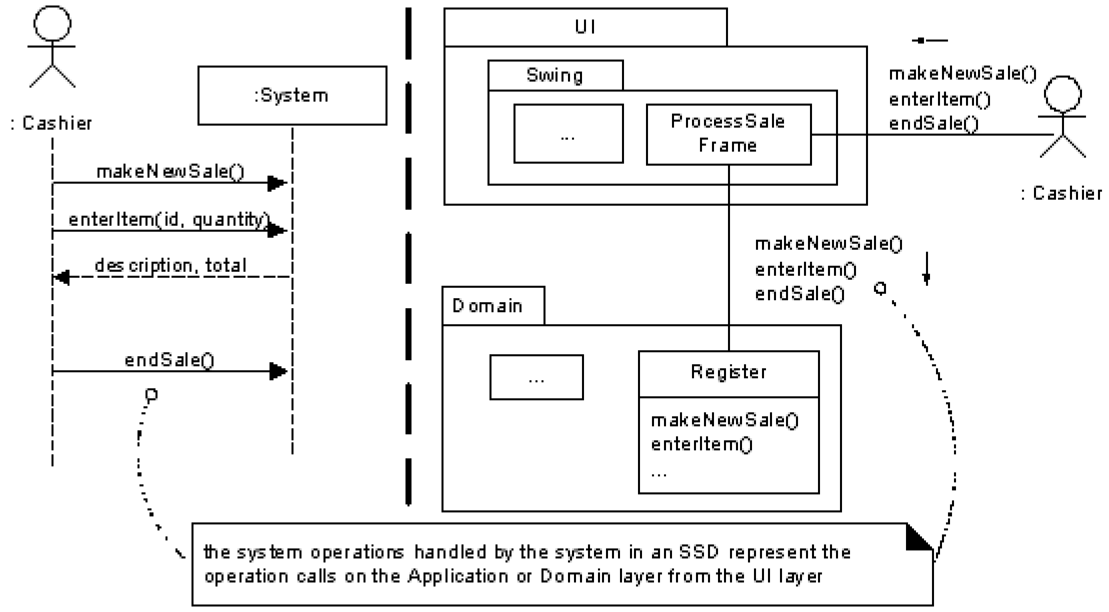


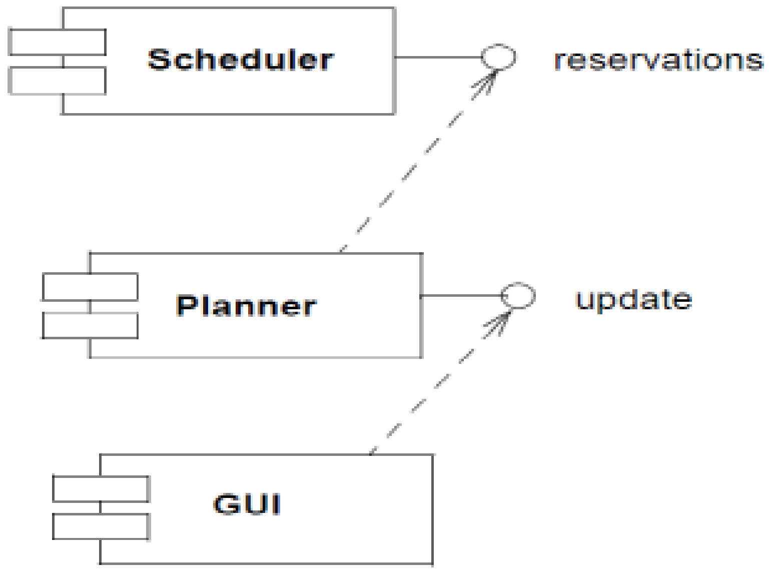
Figure: Messages from UI layer to domain layer

3.5.6 The Observer Pattern

- If model (domain) objects do not have direct knowledge of view (UI) objects, how can a *Register* or *Sale* object get a window to refresh its display when a total changes?
- The *Observer* pattern (p. 463) allows domain objects to send messages to UI objects viewed only in terms of an interface.
  - E.g., known not as concrete window class, but as implementation of *PropertyListener* interface.
- Allows replacement of one view by another.

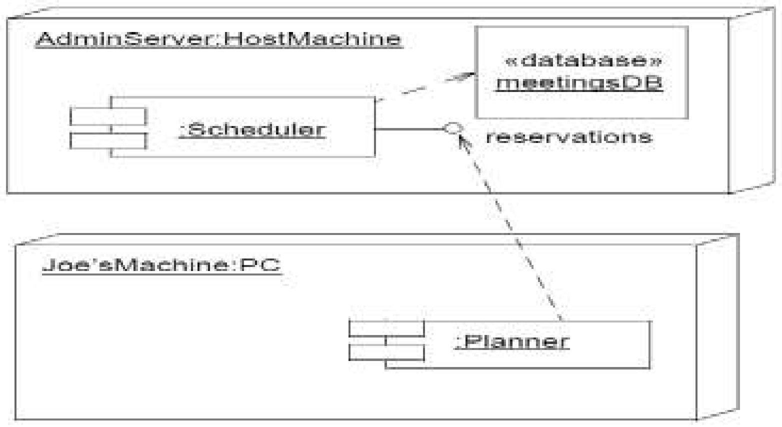
3.6 Component Diagram

A component diagram shows the dependencies among software components, including source code components, binary code components, and executable components. A software module may be represented as a component type. Some components exist at compile time, some exist at link time, some exist at run time, and some exist at more than one time. A compile-only component is one that is only meaningful at compile time. The run-time component in this case would be an executable program. A component diagram has only a type form, not an instance form. To show component instances, use a deployment diagram (possibly a degenerate one without nodes).



3.7 Deployment Diagram

Deployment diagrams show the configuration of run-time processing elements and the software components, processes, and objects that live on them. Software component instances represent run-time manifestations of code units. Components that do not exist as run-time entities (because they have been compiled away) do not appear on these diagrams, they should be shown on component diagrams.



3.8 Collaboration diagram

Collaboration Diagrams describe interactions among classes and associations. These interactions are modeled as exchanges of messages between classes through their associations. Collaboration diagrams are a type of interaction diagram. Collaboration diagrams contain the following elements:

• **Class roles,** which represent roles that objects may play within the interaction.

• **Association roles,** which represent roles that links may play within the interaction.

• **Message flows,** which represent messages sent between objects via links. Links transport or implement the delivery of the message.

3.9 Design Patterns

3.9.1 Reusing object oriented design

Software designers are in a similar position to architects and civil engineers, particularly those concerned with the design of large heterogeneous constructions, such as towns and industrial plants. It therefore seems natural that we should turn to these subjects for ideas about how to attack the design problem.

Subsystems created by the composition of objects do not conform to any accepted notion of structure and are very hard to characterize, though they do determine subsystems that exhibit reusable regularities of interface behavior. The term pattern is used to denote reusable regularities of behavior exhibited by interactive subsystems created by the composition of interaction**.**

3.9.2 What is a design pattern?

They originate from the work of Christopher Alexander, a building architect in the 1970‟s. Alexander‟s idea was to improve the quality of the buildings of the time by utilising proven „patterns‟ of good architectural design. „Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem.‟

A design pattern is defined as „a description of communicating objects and classes that are customized to solve a general design problem in a particular context‟.

- Patterns capture good design principles and communicate them to others.
- Design patterns represent the first legitimate attempt at design reusability.

3.9.3 Design Patterns: Essentials

- Patterns are found through trial and error and by observation.
- In general a design pattern has four essential elements:
- The pattern name
- The problem the pattern is used to solve
- The solution or template for implementing the pattern
- The consequences or results of applying the pattern.

3.9.4 Design Patterns: Characteristics

Smart

- Design patterns are elegant solutions that would not necessarily be apparent to designer without significant experience

Generic

- Patterns are normally generic for a specific problem (a bit like generic containers).
- Design patterns are not normally dependent on a specific system type, programming language, or application domain.

Well-proven

- Design patterns have been identified from real, object-oriented systems. They have not just been thought up, they have been successfully used and tested in several systems.

Simple

- Design patterns usually only consist of a small number of classes, so they are quite small. Combining patterns allows the building of more complex systems.

Reusable

- Patterns are well documented so they are easy to reuse. They are generic so they can be used in a variety of different types of system. It is worth noting that the reuse is at the design level, not at the code level; the classes are not in libraries.

Object Oriented

- Design patterns conform to the usual object-oriented concepts of classes, objects, inheritance and polymorphism.
- The most widely known work on design patterns is that of Gamma, Helm, Johnson and Vlissides. „The gang of four" as they are commonly referred to. Their book „Design Patterns: Elements of Reusable Object-Oriented Software" was published in 1994. It contains a description of the concepts of patterns, plus a catalog of 23 design patterns with their full documentation.

3.9.5 Types of Design Patterns

Creational Patterns: - All of the creational patterns deal with the best way to create instances of classes. Creational patterns separate the operation of an application from how its objects are created. This is important because your program should not depend on how objects are created and arranged. In Java, of course, the simplest way to create an instance of an object is by using the **new** operator.

Fred = new Fred(); //instance of Fred class

However, this really amounts to hard coding how you create the object within your program. In many cases, the exact nature of the object that is created could vary with the needs of the program from time to time and abstracting the creation process into a special "creator" class can make your program more flexible and general.

Creational patterns abstract the object instantiation process. They hide how objects are created and help make the overall system independent of how its objects are created and composed. l Class creational patterns focus on the use of inheritance to decide the object to be instantiated Patterns become important as systems evolve to depend more on object composition than class inheritance. Thus creating objects with particular behaviors requires more than simply instantiating a class.

As an example we consider the creational pattern, Singleton, which can be used to ensure that only one instance of a class is created. Singleton pattern offers several advantages but also has disadvantages

Advantages

It provides controlled access to the sole object instance as the singleton encapsulates the instance.

The namespace is not unnecessarily extended with global variables.

The singleton class may be sub classed. At system start up user selected options may determine which of the subclasses instantiated is when the singleton class is first accessed.

A variation of this pattern can be used to create a specified number of instances if required.

Disadvantages

Using the pattern introduces some additional message passing. To access the singleton instance, the class scope operation getInstance() has to be accessed first rather than accessing the instance directly.

The pattern limits the flexibility of the application.

The singleton pattern is quite well known and developers are tempted to use it in circumstances that are inappropriate. Patterns must be used with care.

Structural Patterns: Structural patterns describe how classes and objects can be combined to form larger structures. Structural patterns offer effective ways of using object oriented concepts such as inheritance, aggregation and composition to satisfy particular requirements. The difference between class patterns and object patterns is that class patterns describe how inheritance can be used to provide more useful program interfaces. Object patterns, on the other hand, describe how objects can be composed into

larger structures using object composition, or the inclusion of objects within other objects.
As an example we consider the structural pattern, composite pattern

Behavioural Patterns: Behavioral patterns are those patterns that are most specifically concerned with communication between objects. This pattern addresses the problems that arise when responsibilities are assigned to classes and in designing algorithms. Behavioural patterns not only suggest particular static relationships between objects and classes but also describe how the objects communicate. Behavioural patterns may use inheritance structures to spread behaviour across the subclasses or they may use aggregation and composition to build complex behaviour from simpler components. The state pattern uses both of these techniques

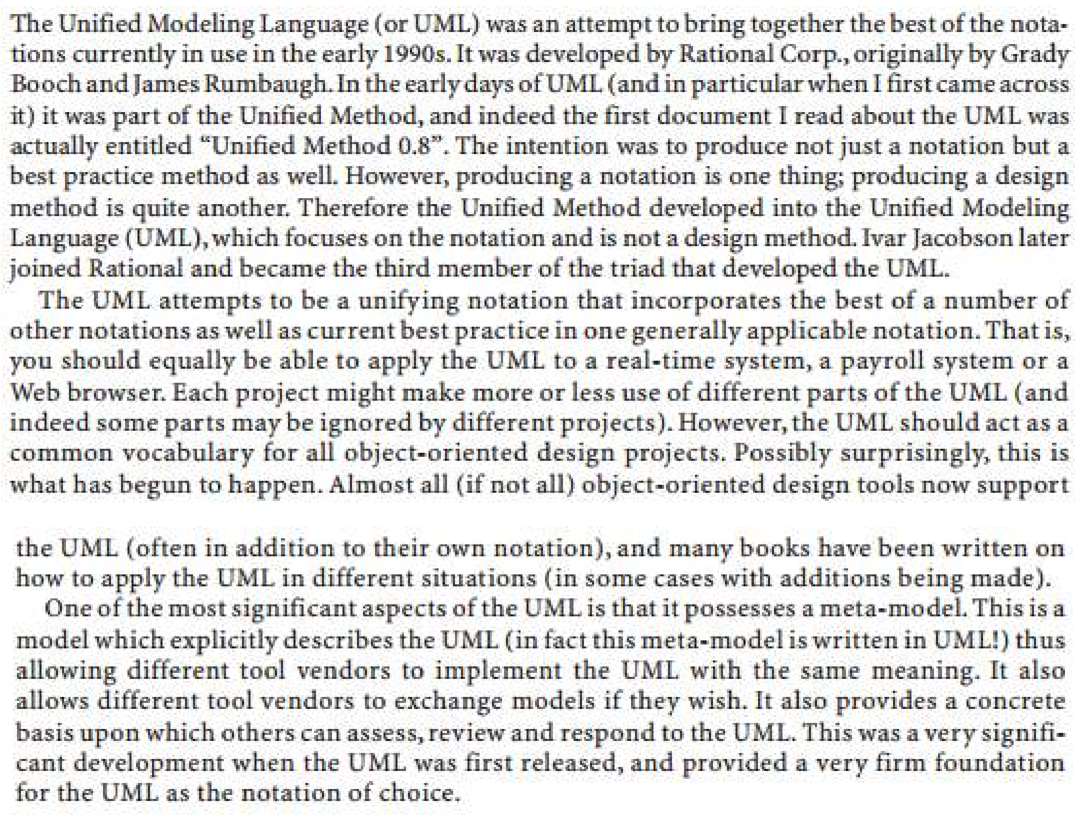
Unit-4

Object Oriented System Design

4.1 Design Issues

- UML as a Model Can't Work in Isolation
- Large Scale System Design/Development Involves
  - Team-Oriented Efforts
  - Software Architectural Design
  - System Design, Implementation, Integration
- The Unified Process by Rational is
  - Iterative and Incremental
  - Use Case Driven
  - Architecture-Centric

4.2 Unified Modeling Language



The Unified Modeling Language (or UML) was an attempt to bring together the best of the notations currently in use in the early 1990s. It was developed by Rational Corp., originally by Grady Booch and James Rumbaugh. In the early days of UML (and in particular when I first came across it) it was part of the Unified Method, and indeed the first document I read about the UML was actually entitled "Unified Method 0.8". The intention was to produce not just a notation but a best practice method as well. However, producing a notation is one thing; producing a design method is quite another. Therefore the Unified Method developed into the Unified Modeling Language (UML), which focuses on the notation and is not a design method. Ivar Jacobson later joined Rational and became the third member of the triad that developed the UML.

The UML attempts to be a unifying notation that incorporates the best of a number of other notations as well as current best practice in one generally applicable notation. That is, you should equally be able to apply the UML to a real-time system, a payroll system or a Web browser. Each project might make more or less use of different parts of the UML (and indeed some parts may be ignored by different projects). However, the UML should act as a common vocabulary for all object-oriented design projects. Possibly surprisingly, this is what has begun to happen. Almost all (if not all) object-oriented design tools now support

the UML (often in addition to their own notation), and many books have been written on how to apply the UML in different situations (in some cases with additions being made).

One of the most significant aspects of the UML is that it possesses a meta-model. This is a model which explicitly describes the UML (in fact this meta-model is written in UML!) thus allowing different tool vendors to implement the UML with the same meaning. It also allows different tool vendors to exchange models if they wish. It also provides a concrete basis upon which others can assess, review and respond to the UML. This was a very significant development when the UML was first released, and provided a very firm foundation for the UML as the notation of choice.

4.3 History of the UML

The UML was not developed overnight (see Figure 3.1). It has gone through an extensive development process which started in the mid-1990s. As stated earlier, my first encounter with what was to become the UML was when it was first documented as part of the Unified Method (release 0.8) in October 1995. At this point in time its heaviest influences were OMT (where I was coming from) and the Booch method. This was primarily because the two key architects at this time were Rumbaugh and Booch. However, OMT has had many influences and has taken many elements from other design methods (see Figure 3.2).

At this time the Unified Method was an impressive exercise, as it had only been under development for the best part of a year. However, things did not stand still, and by the middle of the next year (1996) version 0.9 was released (and version 0.91 three months later). The name had at this point changed and the release was now called the Unified

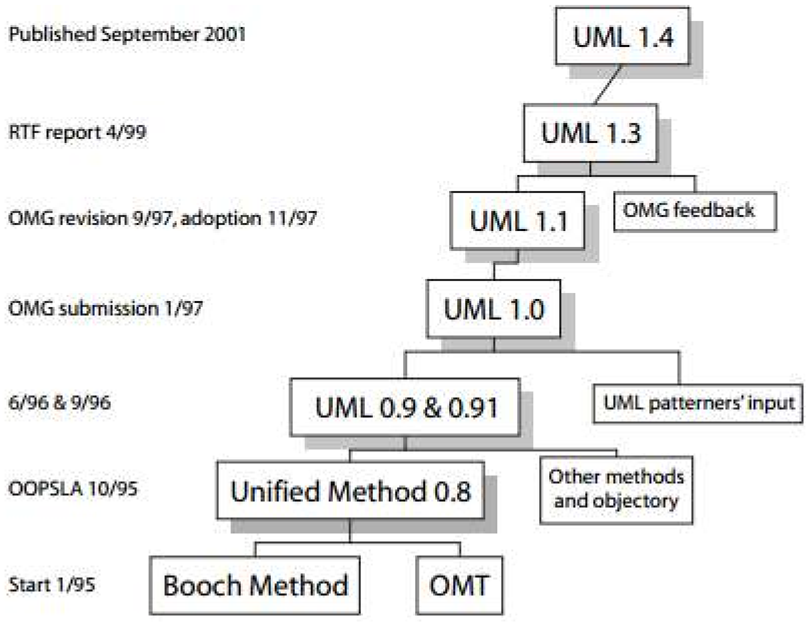


Figure: A potted history of the UML

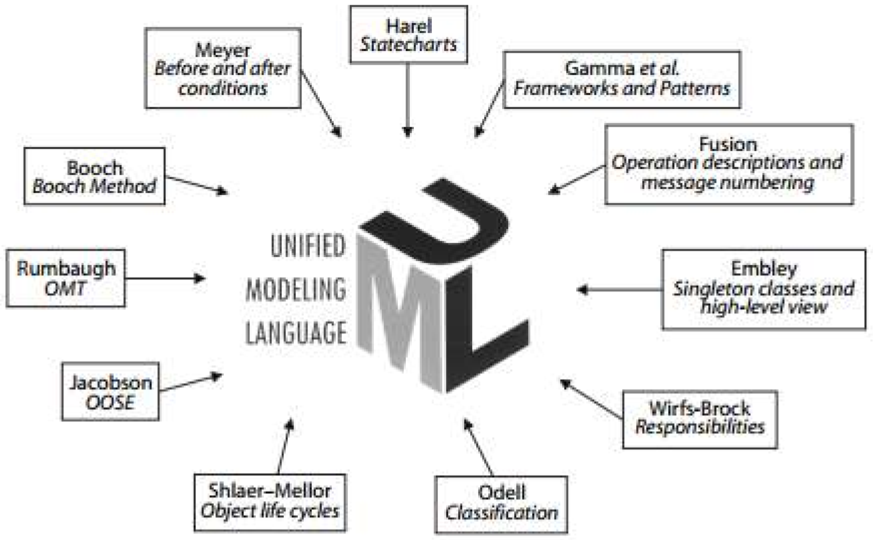


Figure: Influences on the UML

Modeling Language. This release focused on the notation and mostly ignored the process. However, much had happened during this time. Not only had many people worldwide commented on the 0.8 release (as it was freely available for download from Rational's Web site – `http://www.rational.com/`), but the influence of Ivar Jacobson was now being felt. Jacobson had been one of the key architects of the Objectory method, which was most notable for its use of use cases. He had joined Booch and Rumbaugh at Rational at just about the same time as the 0.8 version was released. In UML 0.9 use cases were seen for the first time.

Other partners were now becoming involved in the UML development process, ensuring that a wide variety of backgrounds, expertise and experience was brought to bear. Companies such as IBM, Hewlett-Packard and Microsoft all contributed. Then, at the beginning of 1997, the UML 1.0 standard was presented to the OMG for acceptance as an OMG standard. Version 1.1 of the UML was promulgated as a standard by the OMG towards the end of 1997.

Development of the UML has not stood still, however, although it is now under the control of the OMG. Rather it has continued to develop, and we are now at version 1.4 of the UML (September 2001). At the time of writing there were at least three initiatives looking at developments to the UML, including UML 1.4 with Action Semantics, which adds to UML the syntax and semantics of executable actions and procedures, including their run-time semantics. These semantics are contained within one package, labelled `Actions`, which defines the various kinds of actions that may compose a procedure. In addition the RTF has been published for UML 1.4.1.

UML is now almost universal as the language used to describe object-oriented models. Indeed, incorporating UML as an OMG standard has ensured that many organizations have adopted it as a non-proprietary standard and that the standard has maintained pace with current developments in computer science – the Internet and Java in particular.

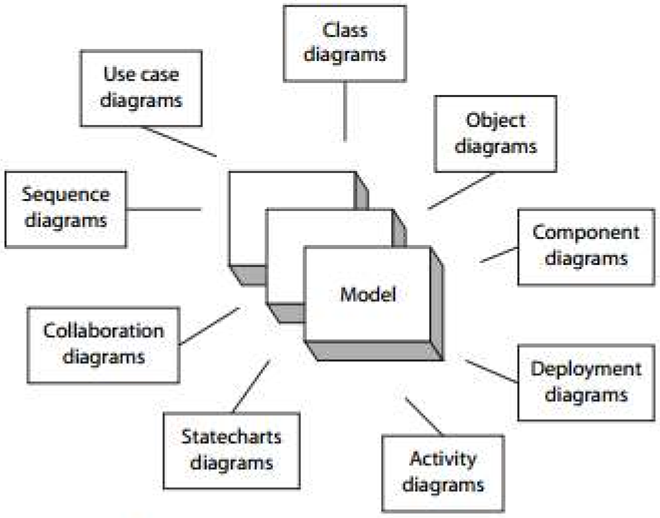


Figure 3.3 Relationship between diagrams and models.

- *Statecharts.* A statechart, or state diagram, illustrates the states an object can be in and the transitions which move the object between states.
- *Component diagrams.* These diagrams are used to illustrate the physical structure of the code in terms of the source code. In Java this means the class files and Java Archive Files (JAR), as well as items such as Web Archive Files (WAR) and Enterprise Archive Files (EAR) in the Java 2 Enterprise Edition architecture.
- *Deployment diagrams.* Deployment diagrams illustrate the physical architecture of the system in terms of processes, networks and the location of components.

4.4 The Unified Approach to Design

The Unified Process is a design framework which guides the tasks, people and products of the design process. It is a framework because it provides the inputs and outputs of each activity, but

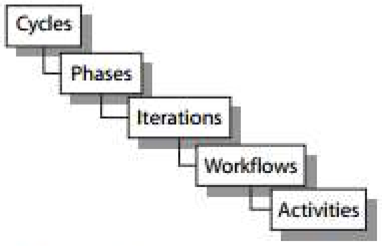


Figure 3.4 Key building blocks of the Unified Process.

does not restrict how each activity must be performed. Different activities can be used in different situations, some being left out, others being replaced or augmented (this is discussed in more detail later in this book). Why then is the Unified Process call a process and not the Unified Framework? It is called a process because its primary aim is to define:

- Who is doing what.
- When they do it.
- How to reach a certain goal (i.e. each activity).
- The inputs and outputs of each activity.

It is thus an engineered process. In fact, it is comprised of a number of different hierarchical elements (see Figure 3.4).

The Unified Process actually comprises low-level activities (such as finding classes), which are combined together into disciplines (formerly known as workflows) which describe how one activity feeds into another). These disciplines are organized into iterations. Each iteration identifies some aspect of the system to be considered. How this is done is considered in more detail later. Iterations themselves are organized into phases. Phases focus on different aspects of the design process, for example requirements, analysis, design and implementation. In turn phases can be grouped into cycles. Cycles focus on the generation of successive releases of a system (for example, version 1.0, version 1.1 etc.).

There are four key elements to the philosophy behind the Unified Process. These four elements:

- are iterative and incremental
- are use case-driven
- are architecture-centric
- acknowledge risk

Iterative and Incremental

The Unified Process is iterative and incremental, as it does not try to complete the whole design task in one go. One of the features of the waterfall model of software engineering used by many design methods (see Figure 3.5) is that it primarily assumes that you will complete the require-

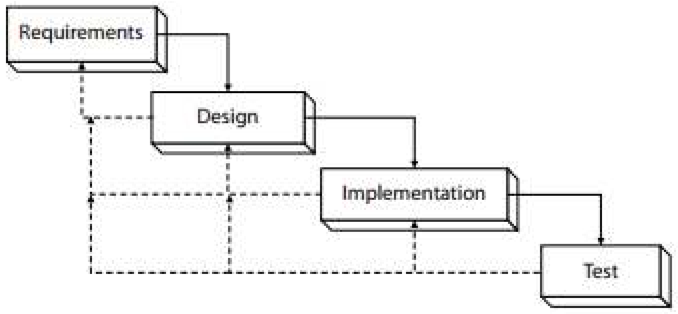


Figure 3.5 The waterfall model.

ments analysis before you start the design phase. In turn, you will complete the design phase before you start the implementation phase and so on. It does accept that there may be some feedback of information from one phase to any preceding phases and that this feedback may have an impact on the products of the preceding phases. However, this is a secondary issue and the assumption is that you will be able to complete the vast majority of one phase before ever considering the next phase. This may be true if this is the fifth or sixth system you have built in the same domain for the same type of application. It is unlikely to be the case with your first application in a new domain (such as your first e-commerce project!).

In contrast to the waterfall model, the Unified Process has an iterative and incremental model. That is, the design process is based on iterations which either address different aspects of the design process or move the design forward in some way (this is the incremental aspect of the model). This does not mean that the Unified Process is a process based on rapid prototyping. Any prototypes that are developed in the Unified Process are used to explore some aspect of the design. This could be to verify some architectural issue for which the design options are similar. Indeed, the use of an iterative and incremental approach in the Unified Process requires more planning (rather than less planning) compared with approaches such as those based on the waterfall model.

Essentially the following holds with the iterative approach in the Unified Process:

- You plan a little.
- You specify, design and implement a little.
- You integrate, test and run.
- You obtain feedback before next iteration.

The end result is that you incrementally produce the system being designed. While you do this you explicitly identify the risks to your design/system up front and deal with them early on (see later). Notice that this does not mean that you are hacking the system together; nor are you carrying out some form of rapid prototyping. However, it does mean that a great deal of planning is required, both initially and as the design develops.

Use Case-driven

The Unified Process is also use case-driven. Remember from earlier that use cases help to identify who uses the system and what they need to do with the system (i.e. the top-level functionality).

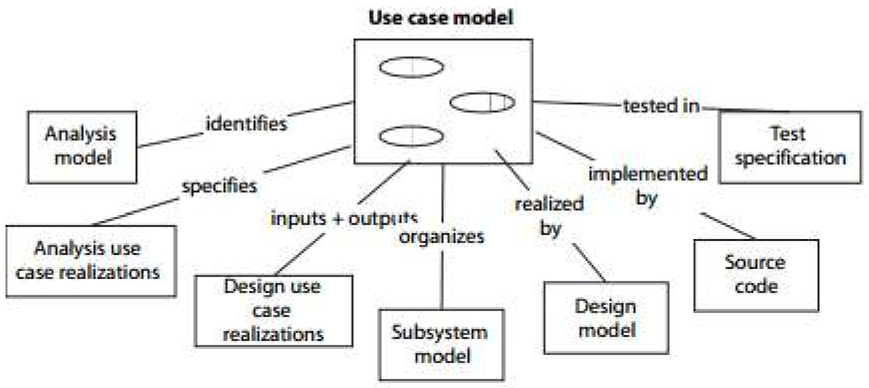


Figure 3.6 The role of use cases.

Thus use cases help identify the primary requirements of the system. One problem with many traditional approaches is that once the requirements have been identified there is no traceability of those requirements through the design to the implementation. Instead, designers (and possibly implementers) must refer back implicitly to the requirements specification and make sure that they have done what is required of them. This is then verified by testing (by which time it is often too late to make any major modifications if the functionality is either wrong or missing).

In the Unified Process use cases are used to ensure that the evolving design is always relevant to what the user required. Indeed, the use cases act as the one consistent thread throughout the whole of the development process, as illustrated in Figure 3.6. For example, at the beginning of the design phase one of the two primary inputs to this phase is the use case model. Then, explicitly within the design model, there are use case realizations which illustrate how each use case is supported by the design. Any use case which does not have a use case realization is not currently supported by the design (in turn, any design elements which do not in some way partake in a use case realization do not support the required functionality of the system!).

To summarize the role of use cases they:

- identify the users of the system and their requirements
- aid in the creation and validation of the system's architecture
- help produce the definition of test cases and procedures
- direct the planning of iterations
- drive the creation of user documentation
- direct the deployment of the system
- synchronize the content of different models
- drive traceability throughout models

Architecture-Centric

One problem with having an iterative and incremental approach is that while one group may be working on part of the implementation another group may be working on part of the design. To ensure that all the various parts fit together there needs to be something. That something is an

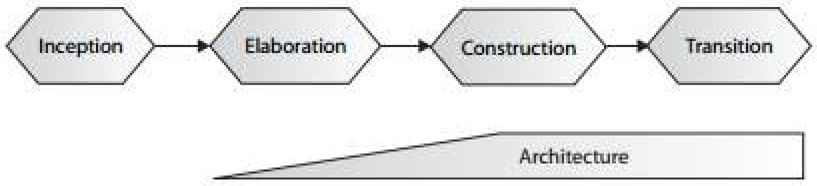


Figure 3.7 The development of the architecture.

architecture. An architecture is the skeleton on which the muscles (functionality) and skin (the user interface) of the system will be hung. A good architecture will be resilient to change and to the evolving design. The Unified Process explicitly acknowledges the need for this architecture by being architecture-centric. It describes how you identify what should be part of the architecture and how you go about designing and implementing the architecture (Figure 3.7). The remainder of the Unified Process then refers back to that architecture.

Obviously, the generation of this architecture is both critical and very hard. Therefore the Unified Process prescribes the successive refinement of the executable architecture, thereby attempting to ensure that the architecture remains relevant.

Acknowledges Risk

Finally, the Unified Process explicitly acknowledges the risk inherent in software design and development. It does this by highlighting unknown aspects in the system being designed and other areas of concern. These areas are then targeted as either being critical to the system and therefore part of the architecture, or areas of risk which need to be addressed early on in the design process (when there is more time) rather than later on (when time tends to be short). Thus it tries to force the riskiest aspects of the system to be designed and implemented early on, hence ensuring that the risk in the system is addressed and managed in a professional manner. Note that it is typically the areas of a design which we do not really understand which end up having the biggest impact on an architecture or the final system. This is often because we do not realize the impact that such areas will have and therefore do not take into account how to deal with their requirements. This is why late on in projects, when such areas are addressed, the system either needs to leave out that functionality or requires major modifications to incorporate the functionality.

4.5 Life Cycle Phase of Unified Process

The Unified Process is composed of four distinct phases. These four phases (presented in Figure 3.8) focus on different aspects of the design process. The four phases are Inception, Elaboration, Construction and Transition.

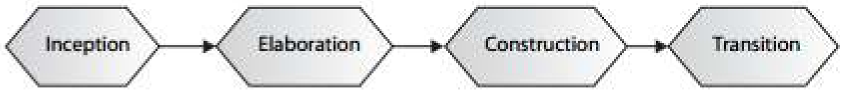


Figure 3.8 The four phases of the Unified Process.

The four phases and their roles are outlined below:

- *Inception*. This phase defines the scope of the project and develops the business case for the system. It also establishes the feasibility of the system to be built. Various prototypes may be developed during this phase to ensure the feasibility of the proposal. Note that we do not focus on the development of the business case in this book: it is assumed that the system to be designed is required and that a business case has already been made.
- *Elaboration*. This phase captures the functional requirements of the system. It should also specify any non-functional requirements to ensure that they are taken into account. The other primary task for this phase is the creation of the architecture to be used throughout the remainder of the Unified Process.
- *Construction*. This phase concentrates on completing the analysis of the system, performing the majority of the design and the implementation of the system. That is, it essentially builds the product.
- *Transition*. The transition phase moves the system into the user's environment. This involves activities such as deploying the system and maintaining it.

Each phase has a set of major milestones that are used to judge the progress of the overall Unified Process (of course, with each phase there are numerous minor milestones to be achieved). The primary milestones (or products) of the four phases are illustrated in Figure 3.9.

A milestone is the culmination of a phase and comprises a set of artefacts (such as specific models) which are the product of the disciplines (and thus activities) in that phase. The primary milestones for each phase are:

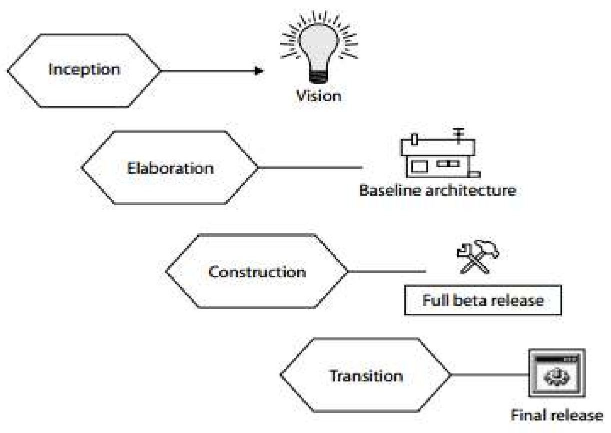


Figure 3.9 The major deliverables of each phase.

- *Inception*. The output of this phase is the vision for the system. This includes a very simplified use case model (to identify what the primary functionality of the system is) and a very tentative architecture, and the most important or significant risks are identified and the elaboration phase is planned.
- *Elaboration*. The primary output of this phase is the architecture, along with a detailed use case model and a set of plans for the construction phase.
- *Construction*. The end result of this phase is the implemented product which includes the software as well as the design and associated models. The product may not be without defects, as some further work has yet to be completed in the transition phase.
- *Transition*. The transition phase is the last phase of a cycle. The major milestone met by this phase is the final production-quality release of the system.

4.6 Applying the Unified Process

When it comes to applying the Unified Process to a real-world project, you should notice that it is a framework (see Figure 3.14). This means that there is no universal process which will always be applicable in its entirety. Instead, the Unified Process is designed for flexibility and extensibility. It allows a variety of life cycle strategies and also allows the selection of what artefacts should be produced. It defines what activities should be performed when and which workers should

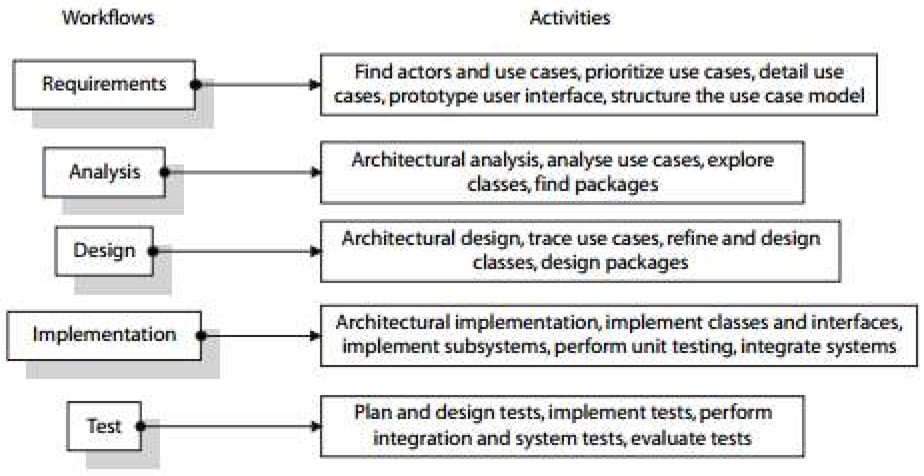

Workflows
Activities
Requirements
Find actors and use cases, prioritize use cases, detail use cases, prototype user interface, structure the use case model
Analysis
Architectural analysis, analyse use cases, explore classes, find packages
Design
Architectural design, trace use cases, refine and design classes, design packages
Implementation
Architectural implementation, implement classes and interfaces, implement subsystems, perform unit testing, integrate systems
Test
Plan and design tests, implement tests, perform integration and system tests, evaluate tests

Figure 3.13 Disciplines are comprised of activities.

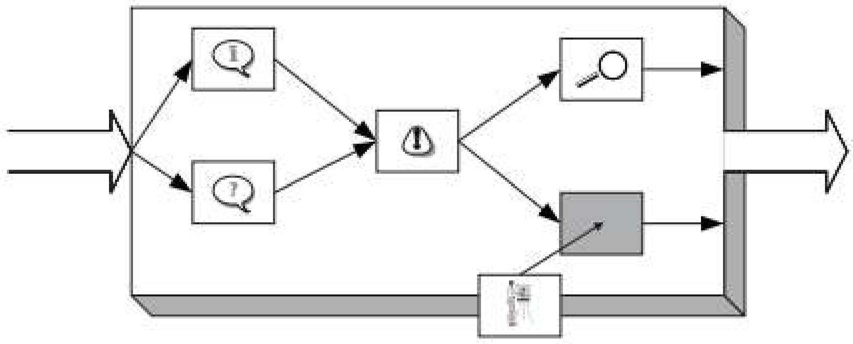


Figure 3.14 The Unified Process is a framework.

perform those activities. Thus it is possible to leave out those elements that don't fit the current project. For example, you might leave out deployment diagrams if you are only deploying on one processor, or if you are working with a batch processing oriented system you may decide to ignore some of the dynamic elements produced, such as statechart diagrams.

In turn, you can add in additional elements if they are required. For example, you may decide to incorporate some real-time extensions into the UML, and some activities to support them. You might decide to incorporate a security view of your system. You might also feel the need to incorporate additional processes. For example, you might incorporate additional activities to help identify an initial set of classes, attributes and relationships. In fact, you may even decide to leave out whole phases, iterations and disciplines as appropriate: for example, a simple system may not need an explicit analysis model!

4.7 Analysis model Partitioning for the design

Software design deals with transforming the customer requirements, as described in the SRS document, into a form (a set of documents) that is suitable for implementation in a programming language. A good software design is seldom arrived by using a single step procedure but rather through several iterations through a series of steps. Design activities can be broadly classified into two important parts: Preliminary (high-level) design and detailed design. The meaning and scope of two design activities (i.e. high level and detailed design) tend to vary considerably from one methodology to another. High-level design means identification of different modules and the control relationships among them and the definition of the interfaces among these modules. The outcome of high-level design is called the program structure or software architecture. Many different types of notations have been used to represent a high-level design. A popular way is to use a tree-like diagram called the structure chart to represent the control hierarchy in a high-level design. However, other notations such as Jackson diagram or

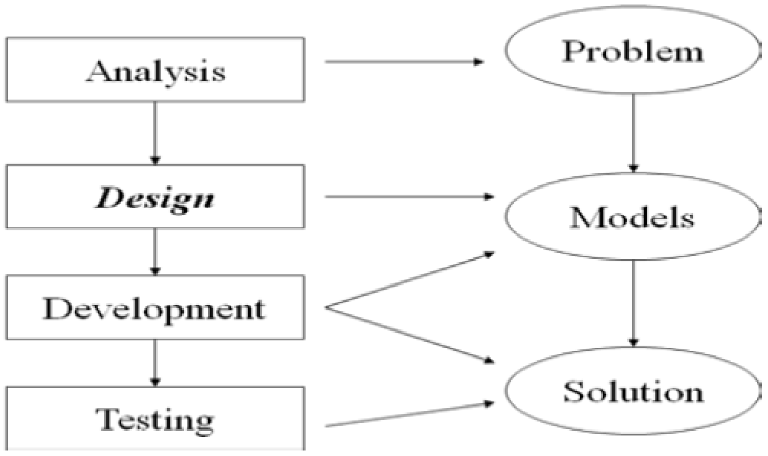


Figure: Analysis and design view

Warnier-Orr diagram can also be used. During detailed design, the data structure and the algorithms of the different modules are designed.

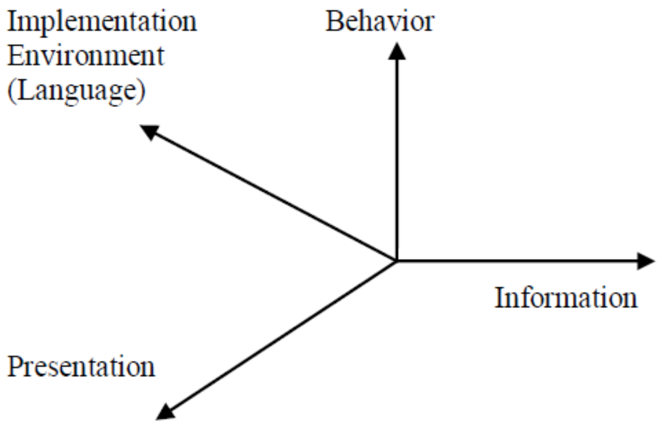


Figure: dimensions of analysis and design

The analysis model is refined and formalized to get a design model. During design modeling, we try to adapt to the actual implementation environment. In design space, yet another new dimension has been added to the analysis space to include the implementation environment. This is show in figure above. This means that we want to adopt our analysis model to fit in the implementation model at the same time as we refine it.

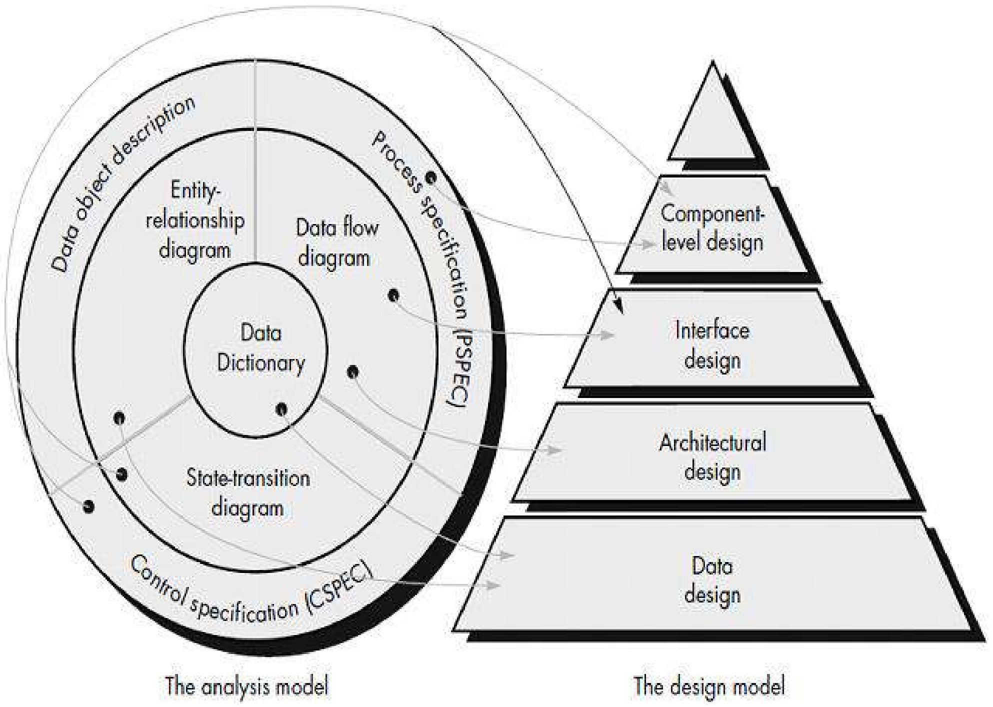


Figure: Component of analysis model and its mapping to the design model.

Map the information from the analysis model to the design representations - data design, architectural design, interface design, procedural design.

A. The design process should not suffer from tunnel vision

A good designer should consider alternative approaches, judging each based on the requirements of the problem, the resources available to do the job.

B. The design should be traceable to the analysis model

Because a single element of the design model often traces to multiple requirements, it is necessary to have a means for tracking how requirements have been satisfied by the design model.

C. The design should minimize the intellectual distance between the software and the problem as it exists in the real world.

That is, the structure of the software design should(whenever possible) mimic the structure of the problem domain.

D. The design should exhibit uniformity and integration

A design is uniform if it appears that one person developed the entire thing. Rules of style and format should be defined for a design team before design work begins. A design is integrated if care is taken in defining interfaces between design components.

E. Design is not coding, coding is not design

Even when detailed procedural designs are created for program components, the level of abstraction of the design model is higher than source code. The only design decisions made at the coding level address the small implementation details that enable the procedural design to be coded.

F. The design should be assessed for quality

A variety of design concepts and design measures are available to assist the designer in assessing quality.

G. The design should be reviewed to minimize conceptual errors

There is sometimes a tendency to focus on minutiae when the design is reviewed, missing the forest for the trees. A design team should ensure that major conceptual elements of the design (omissions, ambiguity, and inconsistency) have been addressed before worrying about the syntax of the design model.

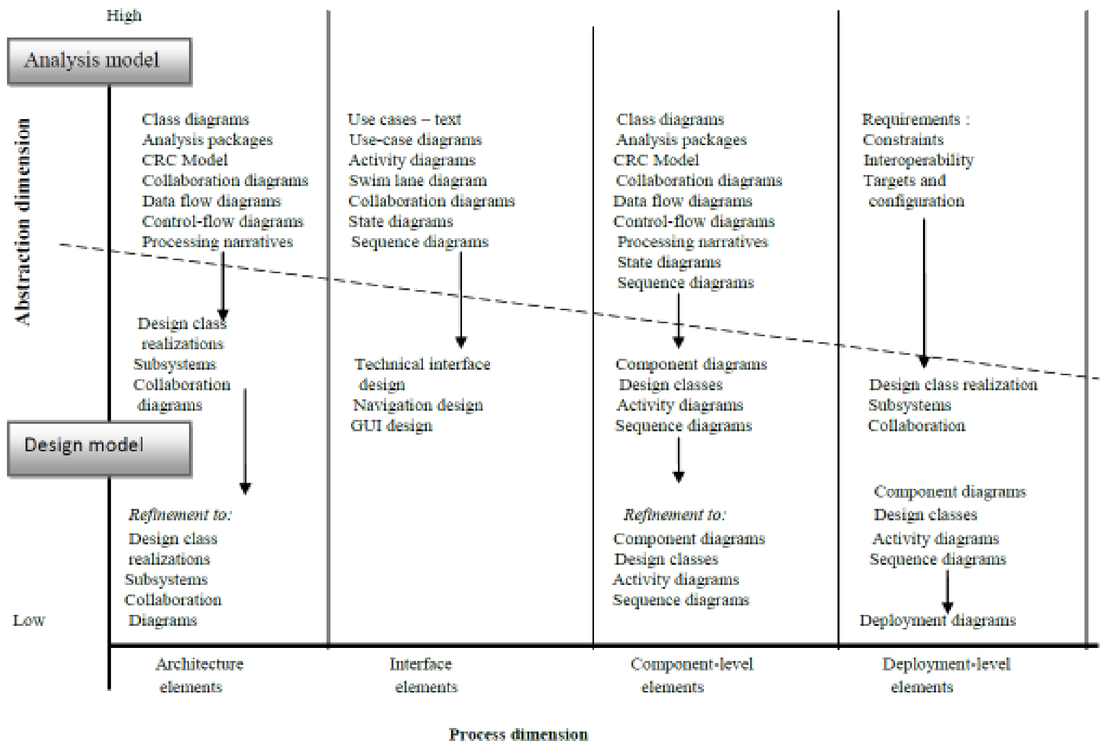
In the construction process, we construct the system using both the analysis model and requirements model. We design and implement the system. Firstly, a design model is made where each object will be fully specified. This model will then form an input data for the rest process.

H. When to do this transition?

The transition from the analysis model to the design model should be made when the Consequences of the implementation environment start to show. This is with adaptation of DBMS distributed environment, real-time adaptations etc. then it is fine to be quite formal in the analysis model.

But if these circumstances will strongly affect the system structure, then the transition should be made quite early. The goal is not to redo any work in a later phase that has done in an earlier phase. We try to keep an ideal analysis model of a system during the entire system life cycle.

A design model is a Specialization of the analysis model for a specific implementation environment [1]. Such changes are then easily incorporated because it is the same analysis model that will form should not affect the analysis model as we do not want changes due to design decisions to be illustrated in the analysis model.



I. When Changes should be made?

If a change of the design model comes from a logical change in the system, then such changes should also be made in the analysis model.

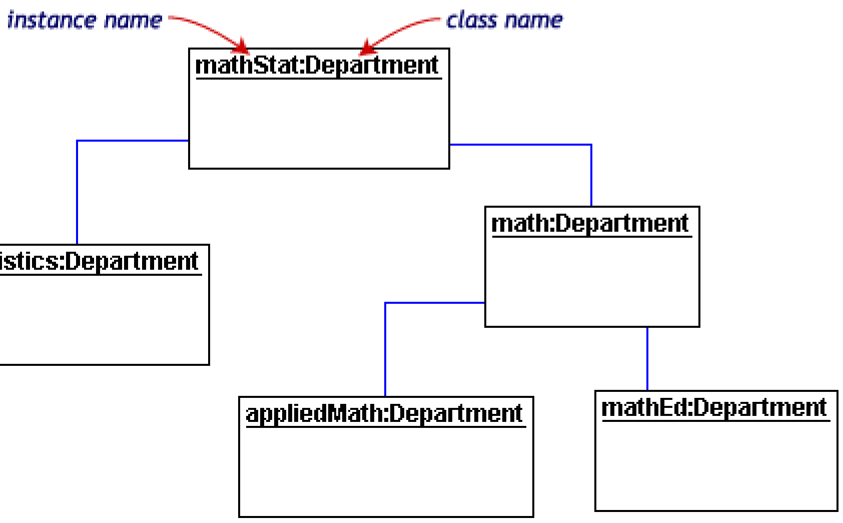
We use a concept of block now to describe the intention of how the code should be produced. The blocks are the design objects. One block normally tries to implement should not affect the analysis model as we do not want changes due to design decisions to be illustrated in the analysis model.

4.8 Object diagrams

Object diagrams show instances instead of classes. They are useful for explaining small pieces with complicated relationships, especially recursive relationships.

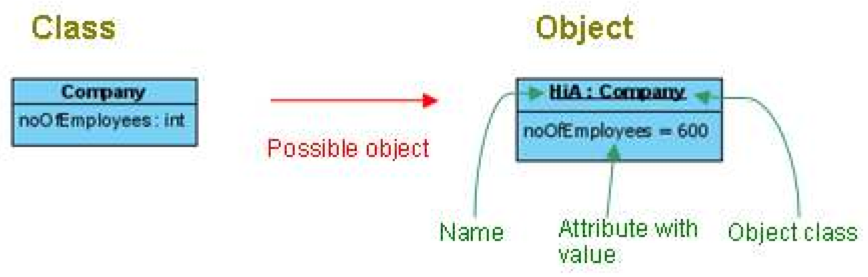
This small class diagram shows that a University Department can contain lots of other Departments.

The object diagram below instantiates the class diagram, replacing it by a concrete example.



Each rectangle in the object diagram corresponds to a single instance. Instance names are underlined in UML diagrams. Class or instance names may be omitted from object diagrams as long as the diagram meaning is still clear.

A class contains a class name, properties and functions. An object shows the class name it is instantiated from preceded by a colon (:), then optionally preceded by the object name. The class in a class diagram displays properties and functions, whereas the object in an object diagram shows only properties, along with their values at the moment of interest to the modeller or viewer. It uses the similar notation as the class diagram. Although less important from a system documentation point of view, object diagrams are handy for documenting a current state of a system. This would include the current values of all documented attributes as shown in the figure



4.9 State chart diagrams

Objects have behaviors and state. The state of an object depends on its current activity or condition. A state chart diagram shows the possible states of the object and the transitions that cause a change in state.

UML state machine diagrams depict the various states that an object may be in and the transitions between those states. In fact, in other modeling languages, it is common for this type of a diagram to be called a state-transition diagram or even simply a state diagram. A state represents a stage in the behavior pattern of an object, and like UML activity diagrams it is possible to have initial states and final states. An initial state, also called a creation state, is the one that an object is in when it is first created, whereas a final state is one in which no transitions lead out of. A transition is a progression from one state to another and will be triggered by an event that is either internal or external to the object.

Top-level state machine diagram is shown below for seminar enrollment, teaching and final exams for a specific subject. The arrows represent transitions, progressions from one state to another.

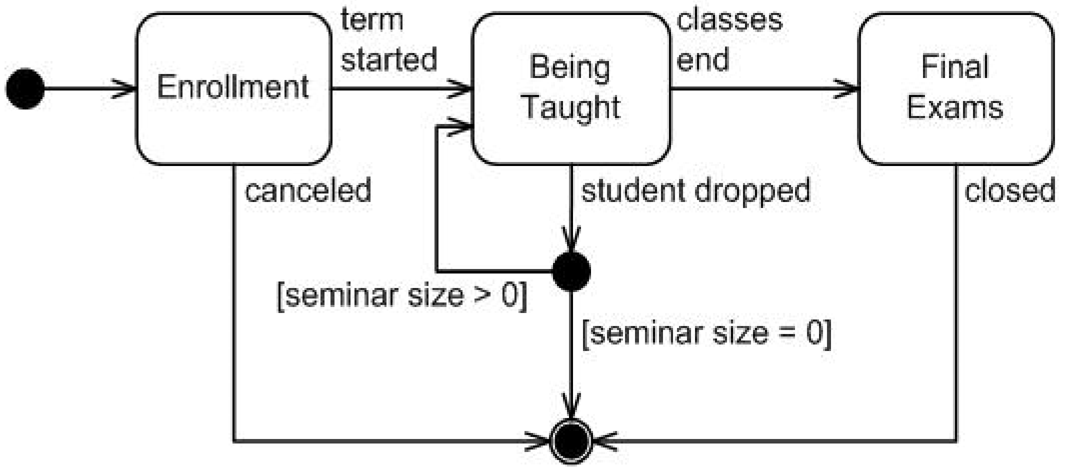


Figure 3.7: State Chart Diagram-1

Following figure presents sub-states of enrollment state during seminar class registration.

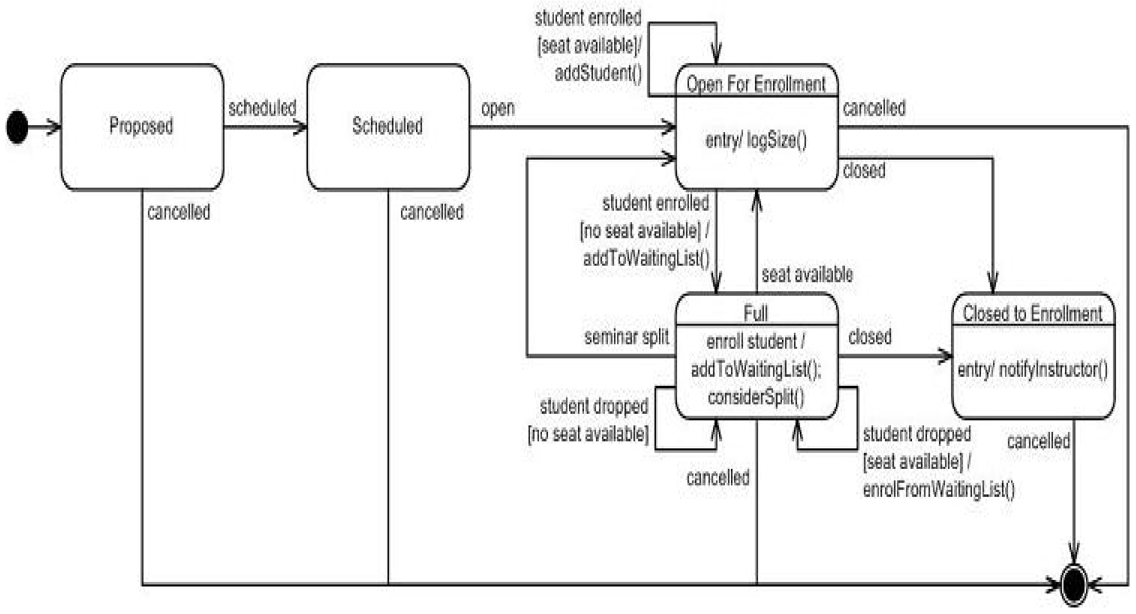

Proposed
scheduled
Scheduled
open
student enrolled [seat available]/ addStudent()
Open For Enrollment
entry/ logSize()
cancelled
closed
cancelled
cancelled
student enrolled [no seat available] / addToWaitingList()
seat available
Full
enroll student / addToWaitingList(); considerSplit()
seminar split
closed
Closed to Enrollment
entry/ notifyInstructor()
student dropped [no seat available]
student dropped [seat available] / enrolFromWaitingList()
cancelled
cancelled

Figure 3.8: State Chart Diagram-2

4.10 Introduction to Modeling

If you are building a new addition to your house, you probably won"t start by just buying a bunch of wood and nailing it together until it looks about right. You will want some blue prints to follow so you can plan and structure the addition before you start working. Models do the same thing for us in the software world. They are the blue prints for systems. A blue print helps you plan an addition before you build it. It can help you be sure the design is sound, the requirements have been met and system can withstand even requirement changes.

4.10.1 Principles of Modeling

- A model is a simplification of reality.

If you want to build a dog house, you can pretty much start with a pile of lumber, some nails, and a few basic tools, such as a hammer, saw, and tape measure. In a few hours, with little prior planning, you'll likely end up with a dog house that's reasonably functional, and you can probably do it with no one else's help. As long as it's big enough and doesn't leak too much, your dog will be happy. If it doesn't work out, you can always start over, or get a less demanding dog.

If you want to build a house for your family, you can start with a pile of lumber, some nails, and a few basic tools, but it's going to take you a lot longer, and your family will certainly be more demanding than the dog. In this case, unless you've already done it a few dozen times before, you'll be better served by doing some detailed planning before you pound the first nail or lay the foundation. At the very least, you'll want to make some sketches of how you want the house to look. If you want to build a quality house that meets the needs of your family and of local building codes, you'll need to draw some blueprints as well, so that you can think through the intended use of the rooms and the practical details of lighting, heating, and plumbing. Given these plans,

If you really want to build the software equivalent of a house or a high rise, the problem is more than just a matter of writing lots of software--in fact, the trick is in creating the right software and in figuring out how to write less software. This makes quality software development an issue of architecture and process and tools. Even so, many projects start out looking like dog houses but grow to the magnitude of a high rise simply because they are a victim of their own success. There comes a time when, if there was no consideration given to architecture, process, or tools that the dog house, now grown into a high rise, collapses of its own weight. The collapse of a dog house may annoy your dog; the failure of a high rise will materially affect its tenants.

- Every model may be expressed at different levels of precision.

If you are building a high rise, sometimes you need a 30,000-foot view--for instance, to help your investors visualize its look and feel. Other times, you need to get down to the level of the studs--for instance, when there's a tricky pipe run or an unusual structural element.

The same is true with software models. Sometimes, a quick and simple executable model of the user interface is exactly what you need; at other times, you have to get down and dirty with the bits, such as when you are specifying cross-system interfaces or wrestling with networking bottlenecks. In any case, the best kinds of models are those that let you choose your degree of detail, depending on who is doing the viewing and why they need to view it. An analyst or an end user will want to focus on issues of what; a developer will want to focus on issues of how. Both of these stakeholders will want to visualize a system at different levels of detail at different times.

- The best models are connected to reality.

A physical model of a building that doesn't respond in the same way as do real materials has only limited value; a mathematical model of an aircraft that assumes only ideal conditions and perfect manufacturing can mask some potentially fatal characteristics of the real aircraft. It's best to have models that have a clear connection to reality, and where that connection is weak, to know exactly how those models are divorced from the real world. All models simplify reality; the trick is to be sure that your simplifications don't mask any important details.

- No single model is sufficient. Every nontrivial system is best approached through a small set of nearly independent models.

If you are constructing a building, there is no single set of blueprints that reveal all its details. At the very least, you'll need floor plans, elevations, electrical plans, heating plans, and plumbing plans.

The operative phrase here is "nearly independent." In this context, it means having models that can be built and studied separately but that are still interrelated. As in the case of a building, you can study electrical plans in isolation, but you can also see their mapping to the floor plan and perhaps even their interaction with the routing of pipes in the plumbing plan.

Visual modeling the process of taking the information from the model and displaying it graphically using some sort of standard set of graphical elements. A standard is vital to realizing one of the benefits of visual modeling –Communication. Communication between users, developers, analysts, testers, managers and anyone else involved with a project is the primary purpose of visual modeling. By producing visual models of a system, we can show how the system works on several levels. We can model the interactions between the users and a system, can model the interaction between different objects within a system.

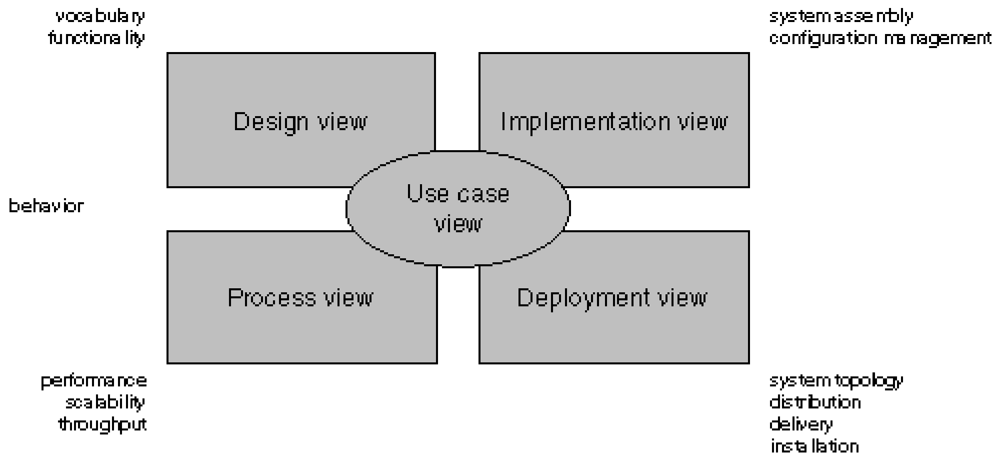
4.10.2 Different views of a system

A model is a semantically closed abstraction of a system, meaning that it represents a complete and self-consistent simplification of reality, created to better understand the system.

Architecture is the set of significant decisions about

- The organization of a software system
- The selection of the structural elements and their interfaces by which the system is composed
- Their behavior, as specified in the collaborations among those elements
- The composition of these structural and behavioral elements into progressively larger subsystems
- The architectural style that guides this organization: the static and dynamic elements and their interfaces, their collaborations, and their composition

The architecture of a software-intensive system can best be described by five interlocking views. Each view is a projection into the organization and structure of the system, focused on a particular aspect of that system



The ***use case view*** of a system encompasses the use cases that describe the behavior of the system as seen by its end users, analysts, and testers. This view doesn't really specify the organization of a software system. Rather, it exists to specify the forces that shape the system's architecture. With the UML, the static aspects of this view are captured in use case diagrams; the dynamic aspects of this view are captured in interaction diagrams, statechart diagrams, and activity diagrams.

The ***design view*** of a system encompasses the classes, interfaces, and collaborations that form the vocabulary of the problem and its solution. This view primarily supports the functional requirements of the system, meaning the services that the system should provide to its end users. With the UML, the static aspects of this view are captured in class diagrams and object diagrams; the dynamic aspects of this view are captured in interaction diagrams, statechart diagrams, and activity diagrams.

The ***process view*** of a system encompasses the threads and processes that form the system's concurrency and synchronization mechanisms. This view primarily addresses the performance, scalability, and throughput of the system. With the UML, the static and dynamic aspects of this view are captured in the same kinds of diagrams as for the design view, but with a focus on the active classes that represent these threads and processes.

The ***implementation*** *view* of a system encompasses the components and files that are used to assemble and release the physical system. This view primarily addresses the configuration management of the system's releases, made up of somewhat independent components and files that can be assembled in various ways to produce a running system. With the UML, the static aspects of this view are captured in component diagrams; the dynamic aspects of this view are captured in interaction diagrams, state chart diagrams, and activity diagrams.

The ***deployment view*** of a system encompasses the nodes that form the system's hardware topology on which the system executes. This view primarily addresses the distribution, delivery, and installation of the parts that make up the physical system. With the UML, the static aspects of this view are captured in deployment diagrams; the dynamic aspects of this view are captured in interaction diagrams, statechart diagrams, and activity diagrams.

Inheritance model: Inheritance model was best described with class diagram and it can be modeled by making use of generalization relationship among classes in class diagram. This model is comprised of super class as well as sub classes. This will support the idea of re-usability.

4.11 Use Case Modeling

The Use Case model is about describing WHAT our system will do at a high-level and with a user focus for the purpose of scoping the project and giving the application some structure. The Use Cases are the unit of estimation and also the smallest unit of delivery. Each increment that is planned and delivered is described in terms of the Use Cases that will be delivered in that increment.

Use Cases are not a functional decomposition model. Use Cases are not intended to capture all of the system requirements. Use Cases do not capture HOW the system will do anything - nor do they capture anything the actor does that does not involve the system. All of these things are better modeled using other modeling techniques that were developed for those purposes. The Object Model to capture the static structure of the system and the composition of the classes. Object Sequence Diagrams and State Transition Diagrams to capture the detailed dynamic behaviour of the system - the HOW. The Business Process Model to capture the overall business processes - both computerized and manual.

Use Cases are not an inherently object-oriented modeling technique. There is no fundamental reason why they couldn't be used as the front-end to a structured development method - but they're not because the methods gurus are concentrating on the development of OO methods.

A Use Case represents a kind of task. UML standard calls this a coherent unit of functionality. A system comprises of 1 or more Use Cases. represented by the elliptical elements. Titles may be anything but important you understand what they symbolise.

4.12 Design Model

Relationships between classes

Now that we can produce class diagrams we can look at how classes can be related to each other.

When we are drawing class diagrams to give an overview of the relationships between classes we may omit the attributes and methods for simplicity sake. You will see that this is the way that the diagrams have been drawn below. The detail of the attributes and methods can be added later.

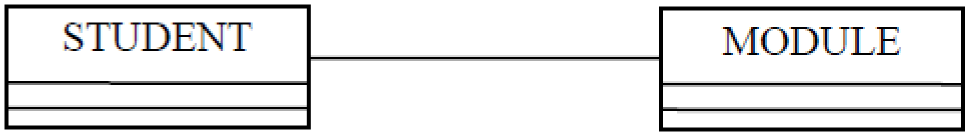
There are three relationships possible:

- association
- aggregation
- generalization (or inheritance)

Association

This is the loosest relationship. It simply means that there will be some communication between the classes but nothing special. This will mean more to you when we have looked at the other two types of relationship.

An example would be Student to Module

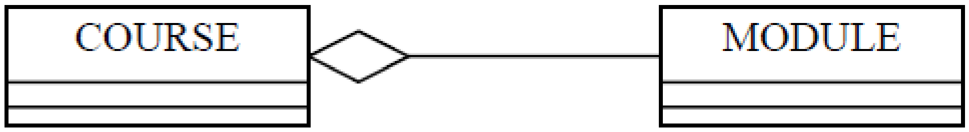


Aggregation

This means that one class is „part of" another class.

This is tighter than an association, in other words the classes are more closely related. To determine whether the relationship is an aggregation it is useful to ask yourself if one class is part of another class. If you can answer „yes" then it probably is an aggregation.

An example would be Course to Module



The way that I test to make sure that this is correct is to ask myself

- can I say „module is a part of course"
- is it essential for course to exist before module can exist

If I can answer yes to both (or at least yes to the first) then I am fairly satisfied.

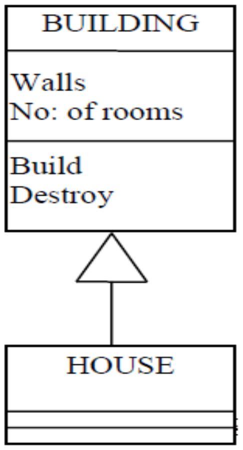
In this example module is definitely a part of a course. If there was no course then it would be possible to have a module but it could never be offered and would never have any students, so module is dependent on course and so is a part of it.

Generalization (Generalization/Specialization)

The third relationship is inheritance. This is extremely important to object oriented systems and is a very powerful feature. The rule that applies here is to ask whether one class is „a kind of"another class.

Super Classes

The real power and impact of inheritance becomes clearer when we start to add attributes and methods. Let us first add some attributes and methods to the class BUILDING.

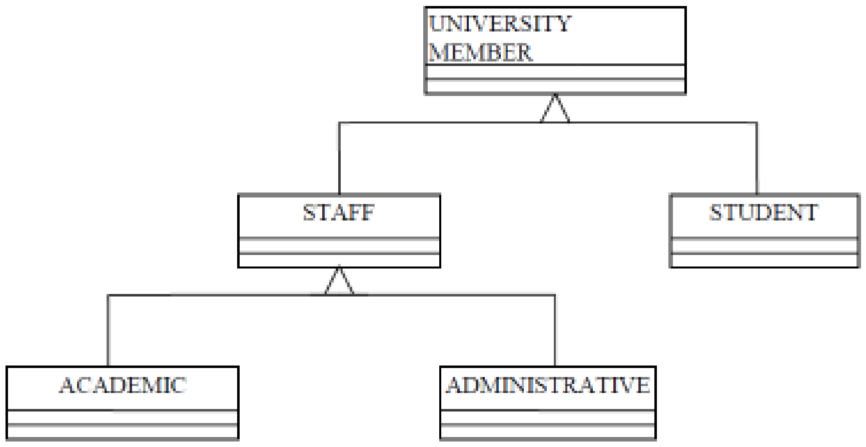


There are clearly lot more attributes and methods that we could add to the class BUILDING, but we must remember that they must apply to *all* BUILDINGs.
What is really important is that HOUSE now inherits all the *generalized* attributes and methods from BUILDING, and then we can add the *specialized* attributes that make it special.

Sub classes
A class will inherit all the attributes and methods from all its ancestors.

If you look at the structure of many of the object oriented languages you will see that there is a base class. This is more clearly demonstrated by looking at an example.

Let us imagine all the people that one might find in a university; we will call them university members. These may be students or members of staff and staff may be academic or administrative.
In outline this would look like this:



Unit-5: GRASP and UML

5.1 GRASP (object-oriented design)

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

The different patterns and principles used in GRASP are: Controller, Creator, Indirection, Information Expert, High Cohesion, Low Coupling, Polymorphism, Protected Variations, and Pure Fabrication. All these patterns answer some software problem, and in almost every case these problems are common to almost every software development project. These techniques have not been invented to create new ways of working, but to better document and standardize old, tried-and-tested programming principles in object-oriented design.

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

5.2 Patterns

5.2.1 Controller

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

A use case controller should be used to deal with *all* system events of a use case, and may be used for more than one use case (for instance, for use cases *Create User* and *Delete User*, one can have a single *UserController*, instead of two separate use case controllers).
It is defined as the first object beyond the UI layer that receives and coordinates ("controls") a system operation. The controller should delegate the work that needs to be done to other objects; it coordinates or controls the activity. It should not do much work itself. The GRASP Controller can be thought of as being a part of the Application/Service

layer (assuming that the application has made an explicit distinction between the application/service layer and the domain layer) in an object-oriented system with Common layers in an information system logical architecture

5.2.2 Creator

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

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

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

5.2.3 High Cohesion

High Cohesion is an evaluative pattern that attempts to keep objects appropriately focused, manageable and understandable. High cohesion is generally used in support of Low Coupling. High cohesion means that the responsibilities of a given element are strongly related and highly focused. Breaking programs into classes and subsystems is an example of activities that increase the cohesive properties of a system. Alternatively, low cohesion is a situation in which a given element has too many unrelated responsibilities. Elements with low cohesion often suffer from being hard to comprehend, hard to reuse, hard to maintain and averse to change.

5.2.4 Indirection

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

5.2.4 Information Expert

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

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

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

5.2.5Low Coupling

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

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

5.2.6 Polymorphism

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

5.2.7 Protected Variations

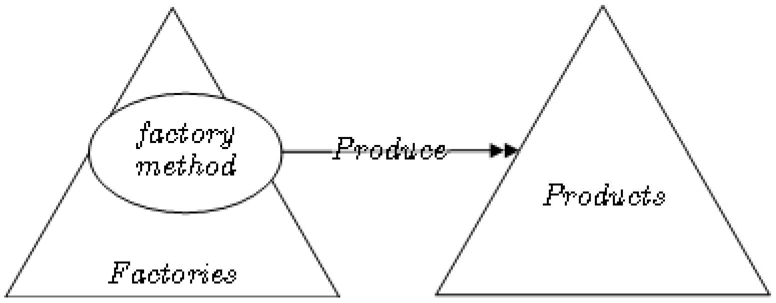
The **Protected Variations** pattern protects elements from the variations on other elements (objects, systems, subsystems) by wrapping the focus of instability with an interface and using polymorphism to create various implementations of this interface.

5.2.8 Pure Fabrication

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

5.3 Factory (object-oriented programming)

In object-oriented programming, a **factory** is an object for creating other objects – formally a factory is simply an object that returns an object from some method call, which is assumed to be "new". More broadly, a subroutine that returns a "new" object may be referred to as a "factory", as in *factory method* or *factory function*. This is a basic concept in OOP, and forms the basis for a number of related software design patterns.



In class-based programming, a factory is an abstraction of a constructor of a class, while in prototype-based programming a factory is an abstraction of a prototype object. A constructor is concrete in that it creates objects as instances of a single class, and by a specified process (class instantiation), while a factory can create objects by instantiating various classes, or by using other allocation schemes such as an object pool. A prototype object is concrete in that it is used to create objects by being cloned, while a factory can create objects by cloning various prototypes, or by other allocation schemes.

Factories may be invoked in various ways, most often a method call (a *factory method*), sometimes by being called as a function if the factory is a function object (a *factory* *function*). In some languages factories are generalizations of constructors, meaning constructors are themselves factories and these are invoked in the same way. In other languages factories and constructors are invoked differently, for example using the keyword `new` to invoke constructors but an ordinary method call to invoke factories; in these languages factories are an abstraction of constructors but not strictly a generalization, as constructors are not themselves factories.

Terminology differs as to whether the concept of a factory is itself a design pattern – in the seminal book *Design Patterns* there is no "factory pattern", but instead two patterns (factory method pattern and abstract factory pattern) that use factories. Some sources refer to the concept as the **factory pattern**, while others consider the concept itself aprogramming idiom, reserving the term "factory pattern" or "factory patterns" to more complicated patterns that use factories, most often the factory method pattern; in this context, the concept of a factory itself may be referred to as a **simple factory.** In other contexts, particularly the Python language, "factory" itself is used, as in this article. More broadly, "factory" may be applied not just to an object that returns objects from some method call, but to a *subroutine* that returns objects, as in a *factory function* (even if functions are not objects) or *factory method.* Because in many languages factories are invoked by calling a method, the general concept of a factory is often confused with the specific factory method pattern design pattern.

Using factories instead of constructors or prototypes allows one to use polymorphism for object creation, not only object use. Specifically, using factories provides encapsulation, and means the code is not tied to specific classes or objects, and thus the class hierarchy or prototypes can be changed or refactored without needing to change code that uses them – they abstract from the class hierarchy or prototypes.

OOP provides polymorphism on object *use* by method dispatch, formally subtype polymorphism via single dispatch determined by the type of the object on which the method is called. However, this does not work for constructors, as constructors *create* an object of some type, rather than *using* an existing object. More concretely, when a constructor is called, there is no object yet on which to dispatch.

More technically, in languages where factories generalize constructors, factories can usually be used anywhere constructors can be, meaning that interfaces that accept a constructor can also in general accept a factory – usually one only need something that creates an object, rather than needing to specify a class and instantiation.

For example, in Python, the `collections.defaultdict` class has a constructor which creates an object of type `defaultdict` whose default values are produced by invoking a factory. The factory is passed as an argument to the constructor, and can itself be a constructor, or anything that behaves like a constructor – a callable object that returns an object, i.e., a factory. For example, using the `list` constructor for lists:

```
# collections.defaultdict([default_factory[, ...]])
d = defaultdict(list)
```

Factory objects are used in situations where getting hold of an object of a particular kind is a more complex process than simply creating a new object, notably if complex allocation or initialization is desired. Some of the processes required in the creation of an object include determining which object to create, managing the lifetime of the object, and managing specialized build-up and tear-down concerns of the object. The factory object might decide to create the object's class (if applicable) dynamically, return it from an object pool, do complex configuration on the object, or other things. Similarly, using this definition, a singleton implemented by the singleton pattern is a formal factory – it returns an object, but does not create new objects beyond the single instance.

Example

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

```
def f():
  return A()
```

A simple factory function implementing the singleton pattern is:

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

f.obj = None
```

This will create an object when first called, and always return the same object thereafter.

5.4 Delegation pattern

In software engineering, the **delegation pattern** is a design pattern in object-oriented programming where an object, instead of performing one of its stated tasks, delegates that task to an associated helper object. There is an *Inversion of Responsibility* in which a helper object, known as a **delegate**, is given the responsibility to execute a task for the **delegator**. The delegation pattern is one of the fundamental abstraction patterns that underlie other software patterns such as composition (also referred to as aggregation), mixins and aspects.

Examples

Java examples

In this Java example, the Printer class has a print method. This print method, rather than performing the print itself, delegates to class RealPrinter. To the outside world it appears that the Printer class is doing the print, but the RealPrinter class is the one actually doing the work.

Delegation is simply passing a duty off to someone/something else. Here is a simple example:

```
class RealPrinter { // the "delegate"
   void print() {
     System.out.println("something");
   }
}

class Printer { // the "delegator"
   RealPrinter p = new RealPrinter(); // create the delegate
   void print() {
     p.print(); // delegation
   }
}

public class Main {
   // to the outside world it looks like Printer actually prints.
   public static void main(String[] args) {
      Printer printer = new Printer();
      printer.print();
   }
}
```

C++ example

This example is a C++ version of the complex Java example above. Since C++ does not have an interface construct, a pure virtual class plays the same role. The advantages and disadvantages are largely the same as in the Java example.

```
#include <iostream>
using namespace std;
```

```
class I {
 public:
  virtual void f() = 0;
  virtual void g() = 0;
  virtual ~I() {}
};

class A : public I {
 public:
  void f() { cout << "A: doing f()" << endl; }
  void g() { cout << "A: doing g()" << endl; }
  ~A() { cout << "A: cleaning up." << endl; }
};

class B : public I {
 public:
  void f() { cout << "B: doing f()" << endl; }
  void g() { cout << "B: doing g()" << endl; }
  ~B() { cout << "B: cleaning up." << endl; }
};

class C : public I {
 public:
  // construction/destruction
  C() : i( new A() ) { }
  virtual ~C() { delete i; }

 private:
  // delegation
  I* i;

 public:
  void f() { i->f(); }
  void g() { i->g(); }

  // normal attributes
  void toA() { delete i; i = new A(); }
  void toB() { delete i; i = new B(); }
};

int main() {
  C c;
  c.f();   //A: doing f()
  c.g();   //A: doing g()
  c.toB(); //A: cleaning up.
  c.f();   //B: doing f()
  c.g();   //B: doing g()
}
```

5.5 Applying GOF Design Pattern

Design patterns represent the best practices used by experienced object-oriented software developers. Design patterns are solutions to general problems that software developers faced during software development. These solutions were obtained by trial and error by numerous software developers over quite a substantial period of time.

What is Gang of Four (GOF)?

In 1994, four authors Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides published a book titled **Design Patterns - Elements of Reusable Object-Oriented Software** which initiated the concept of Design Pattern in Software development.

These authors are collectively known as **Gang of Four (GOF)**. According to these authors design patterns are primarily based on the following principles of object orientated design.

- Program to an interface not an implementation
- Favor object composition over inheritance

Usage of Design Pattern

Design Patterns have two main usages in software development.

COMMON PLATFORM FOR DEVELOPERS

Design patterns provide a standard terminology and are specific to particular scenario. For example, a singleton design pattern signifies use of single object so all developers familiar with single design pattern will make use of single object and they can tell each other that program is following a singleton pattern.

BEST PRACTICES

Design patterns have been evolved over a long period of time and they provide best solutions to certain problems faced during software development. Learning these patterns helps un-experienced developers to learn software design in an easy and faster way.

Types of Design Pattern

As per the design pattern reference book **Design Patterns - Elements of Reusable Object-Oriented Software** , there are 23 design patterns. These patterns can be classified in three categories: Creational, Structural and behavioral patterns. We'll also discuss another category of design patterns: J2EE design patterns.

S.N.	Pattern & Description
1	**Creational Patterns** These design patterns provides way to create objects while hiding the creation logic, rather than instantiating objects directly using new opreator. This gives program more flexibility in deciding which objects need to be created for a given use case.
2	**Structural Patterns** These design patterns concern class and object composition. Concept of inheritance is used to compose interfaces and define ways to compose objects to obtain new functionalities.
3	**Behavioral Patterns** These design patterns are specifically concerned with communication between objects.
4	**J2EE Patterns** These design patterns are specifically concerned with the presentation tier. These patterns are identified by Sun Java Center.

Name of the 23 design pattern
Factory Pattern
Abstract Factory Pattern
Singleton Pattern
Builder Pattern
Prototype Pattern
Adapter Pattern
Bridge Pattern
Filter/Criteria Pattern
Composite Pattern
Decorator Pattern
Facade Pattern
Flyweight Pattern
Proxy Pattern
Chain of Responsibility Pattern
Command Pattern
Interpreter Pattern
Iterator Pattern
Mediator Pattern
Memento Pattern
Observer Pattern
State Pattern

Null Object Pattern
Strategy Pattern
Template Pattern
Visitor Pattern
MVC Pattern
Business Delegate Pattern
Composite Entity Pattern
Data Access Object Pattern
Front Controller Pattern
Intercepting Filter Pattern
Service Locator Pattern
Transfer Object Pattern

5.5.1 Factory Pattern

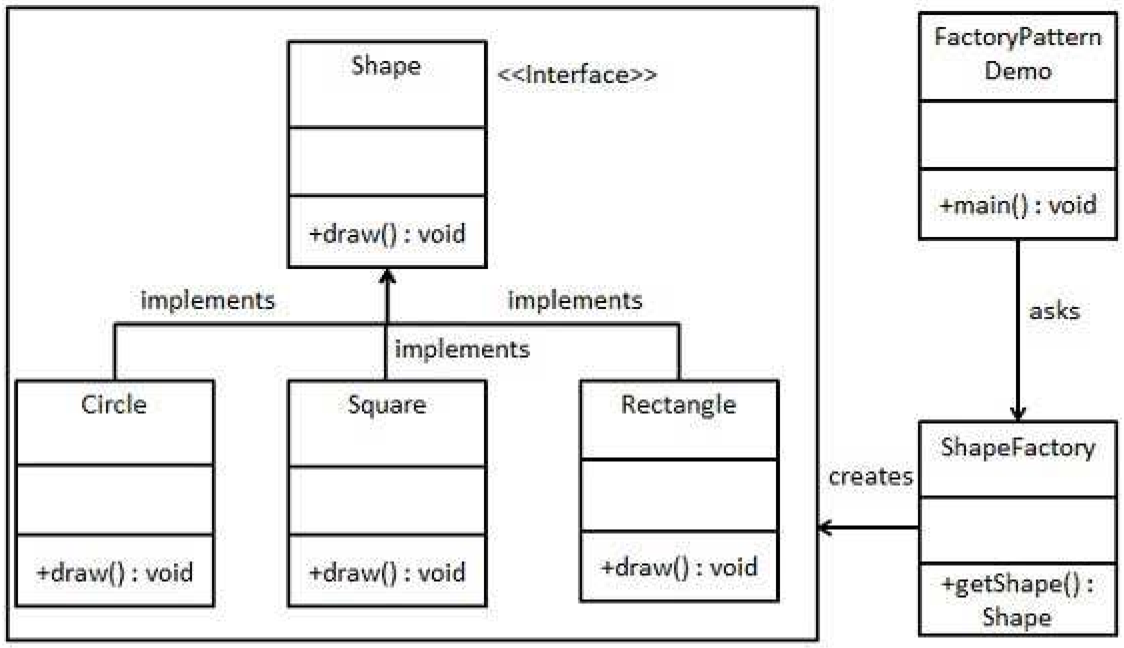
Factory pattern is one of most used design pattern in Java. This type of design pattern comes under creational pattern as this pattern provides one of the best ways to create an object.

In Factory pattern, we create object without exposing the creation logic to the client and refer to newly created object using a common interface.

Implementation

We're going to create a *Shape* interface and concrete classes implementing the *Shape* interface. A factory class *ShapeFactory* is defined as a next step.

FactoryPatternDemo, our demo class will use *ShapeFactory* to get a *Shape* object. It will pass information (*CIRCLE / RECTANGLE / SQUARE*) to *ShapeFactory* to get the type of object it needs.



Steps involved

Step 1

Create an interface.

Shape.java

```
public interface Shape {
  void draw();
}
```

Step 2

Create concrete classes implementing the same interface.

Rectangle.java

```
public class Rectangle implements Shape {

  @Override
  public void draw() {
    System.out.println("Inside Rectangle::draw() method.");
  }
}
```

Square.java

```
public class Square implements Shape {

  @Override
  public void draw() {
    System.out.println("Inside Square::draw() method.");
  }
}
```

Circle.java

```
public class Circle implements Shape {

  @Override
  public void draw() {
    System.out.println("Inside Circle::draw() method.");
  }
}
```

Step 3

Create a Factory to generate object of concrete class based on given information.

ShapeFactory.java

```
public class ShapeFactory {

  //use getShape method to get object of type shape
  public Shape getShape(String shapeType){
      if(shapeType == null){
        return null;
      }
      if(shapeType.equalsIgnoreCase("CIRCLE")){
        return new Circle();
      } else if(shapeType.equalsIgnoreCase("RECTANGLE")){
        return new Rectangle();
      } else if(shapeType.equalsIgnoreCase("SQUARE")){
        return new Square();
      }
      return null;
   }
}
```

Step 4

Use the Factory to get object of concrete class by passing an information such as type.

FactoryPatternDemo.java

```
public class FactoryPatternDemo {

   public static void main(String[] args) {
      ShapeFactory shapeFactory = new ShapeFactory();

      //get an object of Circle and call its draw method.
      Shape shape1 = shapeFactory.getShape("CIRCLE");

      //call draw method of Circle
      shape1.draw();
      //get an object of Rectangle and call its draw method.
      Shape shape2 = shapeFactory.getShape("RECTANGLE");

      //call draw method of Rectangle
      shape2.draw();

      //get an object of Square and call its draw method.
      Shape shape3 = shapeFactory.getShape("SQUARE");

      //call draw method of circle
      shape3.draw();
   }
}
```

Step 5

Verify the output.

```
Inside Circle::draw() method.
Inside Rectangle::draw() method.
Inside Square::draw() method.
```

5.5.2 Singleton Pattern

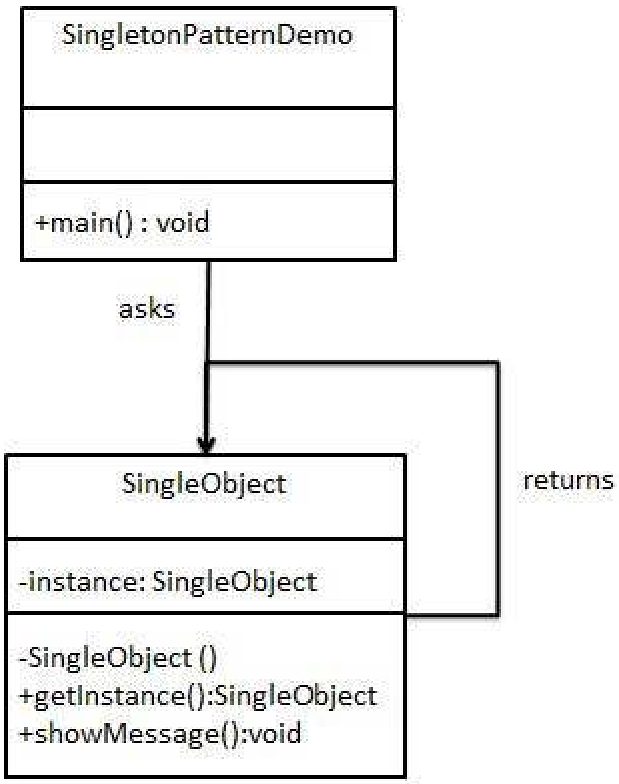
Singleton pattern is one of the simplest design patterns in Java. This type of design pattern comes under creational pattern as this pattern provides one of the best way to create an object.

This pattern involves a single class which is responsible to creates own object while making sure that only single object get created. This class provides a way to access its only object which can be accessed directly without need to instantiate the object of the class.

Implementation

We're going to create a *SingleObject* class. *SingleObject* class have its constructor as private and have a static instance of itself.

SingleObject class provides a static method to get its static instance to outside world.*SingletonPatternDemo*, our demo class will use *SingleObject* class to get a *SingleObject* object.



Step 1

Create a Singleton Class.

SingleObject.java

```
public class SingleObject {

   //create an object of SingleObject
   private static SingleObject instance = new SingleObject();

   //make the constructor private so that this class cannot be
   //instantiated
   private SingleObject(){}

   //Get the only object available
   public static SingleObject getInstance(){
      return instance;
   }

   public void showMessage(){
      System.out.println("Hello World!");
   }
}
```

Step 2

Get the only object from the singleton class.

SingletonPatternDemo.java

```
public class SingletonPatternDemo {
   public static void main(String[] args) {

      //illegal construct
      //Compile Time Error: The constructor SingleObject() is not visible
      //SingleObject object = new SingleObject();
      //Get the only object available
      SingleObject object = SingleObject.getInstance();

      //show the message
      object.showMessage();
   }
}
```

Step 3

Verify the output.

```
Hello World!
```

5.5.3 Adapter Pattern

Adapter pattern works as a bridge between two incompatible interfaces. This type of design pattern comes under structural pattern as this pattern combines the capability of two independent interfaces.

This pattern involves a single class which is responsible to join functionalities of independent or incompatible interfaces. A real life example could be a case of card reader which acts as an adapter between memory card and a laptop. You plugins the memory card into card reader and card reader into the laptop so that memory card can be read via laptop.

We are demonstrating use of Adapter pattern via following example in which an audio player device can play mp3 files only and wants to use an advanced audio player capable of playing vlc and mp4 files.

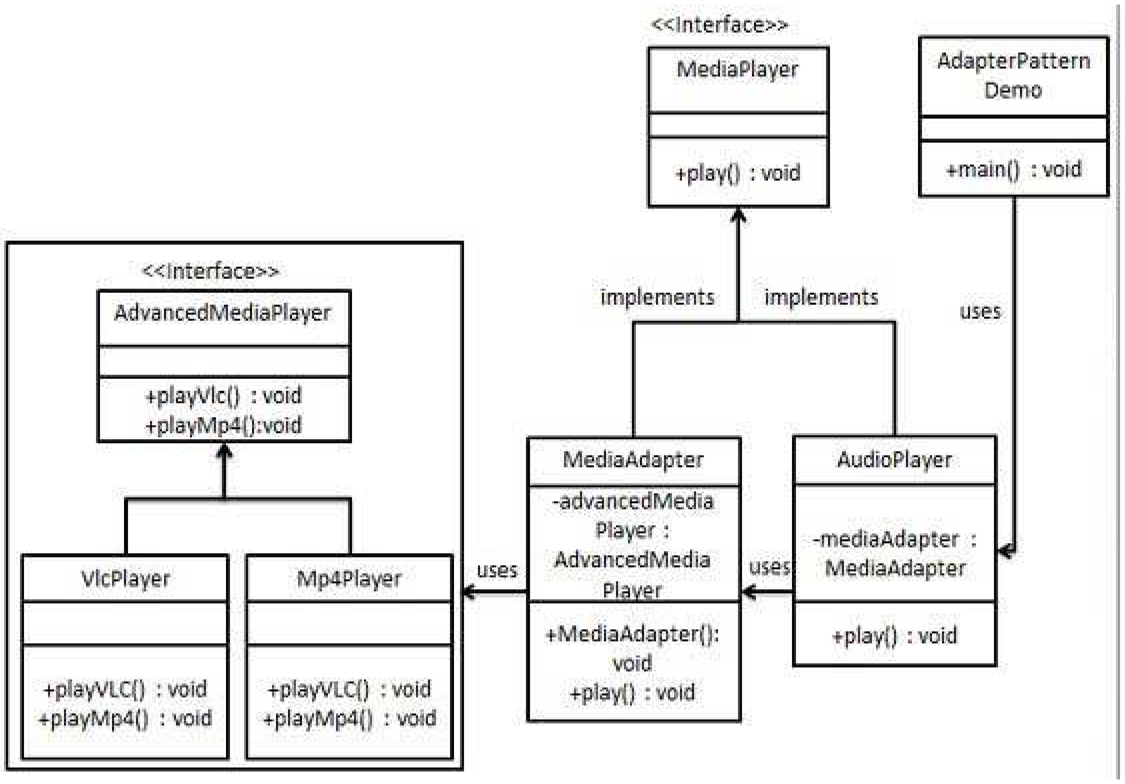
Implementation

We've an interface *MediaPlayer* interface and a concrete class *AudioPlayer* implementing the*MediaPlayer* interface. *AudioPlayer* can play mp3 format audio files by default.

We're having another interface *AdvancedMediaPlayer* and concrete classes implementing the*AdvancedMediaPlayer* interface.These classes can play vlc and mp4 format files.

We want to make *AudioPlayer* to play other formats as well. To attain this, we've created an adapter class*MediaAdapter* which implements the *MediaPlayer* interface and uses *AdvancedMediaPlayer* objects to play the required format.

AudioPlayer uses the adapter class *MediaAdapter* passing it the desired audio type without knowing the actual class which can play the desired format. *AdapterPatternDemo*, our demo class will use*AudioPlayer* class to play various formats.



Step 1

Create interfaces for Media Player and Advanced Media Player.

MediaPlayer.java

```java
public interface MediaPlayer {
   public void play(String audioType, String fileName);
}
```

AdvancedMediaPlayer.java

```java
public interface AdvancedMediaPlayer {
   public void playVlc(String fileName);
   public void playMp4(String fileName);
}
```

Step 2

Create concrete classes implementing the *AdvancedMediaPlayer* interface.

VlcPlayer.java

```java
public class VlcPlayer implements AdvancedMediaPlayer{
   @Override
   public void playVlc(String fileName) {
      System.out.println("Playing vlc file. Name: "+ fileName);
   }

   @Override
   public void playMp4(String fileName) {
      //do nothing
   }
}
```

Mp4Player.java

```java
public class Mp4Player implements AdvancedMediaPlayer{

   @Override
   public void playVlc(String fileName) {
      //do nothing
```

```
  }

  @Override
  public void playMp4(String fileName) {
    System.out.println("Playing mp4 file. Name: "+ fileName);
  }
}
```

Step 3

Create adapter class implementing the *MediaPlayer* interface.

MediaAdapter.java

```
public class MediaAdapter implements MediaPlayer {

  AdvancedMediaPlayer advancedMusicPlayer;

  public MediaAdapter(String audioType){
    if(audioType.equalsIgnoreCase("vlc") ){
      advancedMusicPlayer = new VlcPlayer();
    } else if (audioType.equalsIgnoreCase("mp4")){
      advancedMusicPlayer = new Mp4Player();
    }
  }

  @Override
  public void play(String audioType, String fileName) {
    if(audioType.equalsIgnoreCase("vlc")){
      advancedMusicPlayer.playVlc(fileName);
    }else if(audioType.equalsIgnoreCase("mp4")){
      advancedMusicPlayer.playMp4(fileName);
    }
  }
}
```

Step 4

Create concrete class implementing the *MediaPlayer* interface.

AudioPlayer.java

```
public class AudioPlayer implements MediaPlayer {
  MediaAdapter mediaAdapter;

  @Override
  public void play(String audioType, String fileName) {

    //inbuilt support to play mp3 music files
    if(audioType.equalsIgnoreCase("mp3")){
      System.out.println("Playing mp3 file. Name: "+ fileName);
    }
    //mediaAdapter is providing support to play other file formats
    else if(audioType.equalsIgnoreCase("vlc")
      || audioType.equalsIgnoreCase("mp4")){
      mediaAdapter = new MediaAdapter(audioType);
      mediaAdapter.play(audioType, fileName);
    }
    else{
      System.out.println("Invalid media. "+
        audioType + " format not supported");
    }
  }
}
```

Step 5

Use the AudioPlayer to play different types of audio formats.

AdapterPatternDemo.java

```
public class AdapterPatternDemo {
  public static void main(String[] args) {
    AudioPlayer audioPlayer = new AudioPlayer();

    audioPlayer.play("mp3", "beyond the horizon.mp3");
    audioPlayer.play("mp4", "alone.mp4");
    audioPlayer.play("vlc", "far far away.vlc");
    audioPlayer.play("avi", "mind me.avi");
  }
}
```

Step 6

Verify the output.

```
Playing mp3 file. Name: beyond the horizon.mp3
Playing mp4 file. Name: alone.mp4
Playing vlc file. Name: far far away.vlc
Invalid media. avi format not supported
```

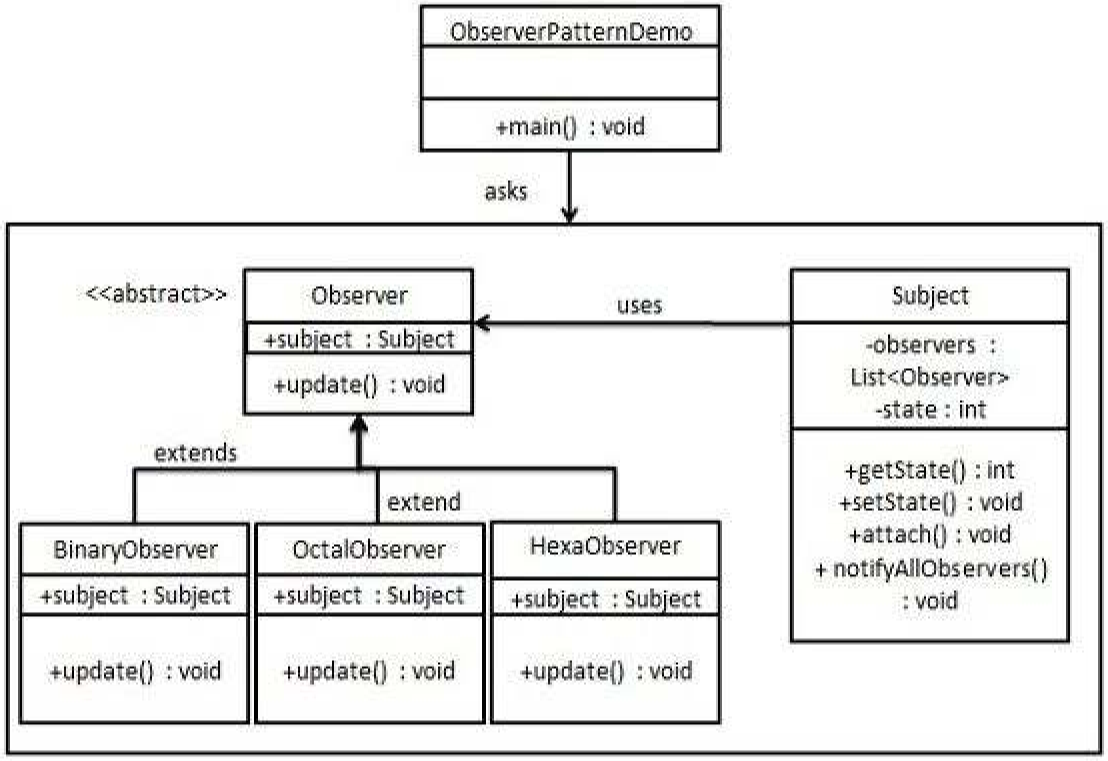
5.5.4 Observer Pattern

Observer pattern is used when there is one to many relationship between objects such as if one object is modified, its depenedent objects are to be notified automatically. Observer pattern falls under behavioral pattern category.

Implementation

Observer pattern uses three actor classes. Subject, Observer and Client. Subject, an object having methods to attach and de-attach observers to a client object. We've created classes *Subject*, *Observer*abstract class and concrete classes extending the abstract class the *Observer*.

ObserverPatternDemo, our demo class will use *Subject* and concrete class objects to show observer pattern in action.



Step 1

Create Subject class.

Subject.java

```
import java.util.ArrayList;
import java.util.List;

public class Subject {
  private List<Observer> observers
    = new ArrayList<Observer>();
  private int state;

  public int getState() {
    return state;
  }

  public void setState(int state) {
    this.state = state;
    notifyAllObservers();
  }

  public void attach(Observer observer){
    observers.add(observer);
  }

  public void notifyAllObservers(){
    for (Observer observer : observers) {
      observer.update();
    }
  }
}
```

Step 2

Create Observer class.

Observer.java

```
public abstract class Observer {
  protected Subject subject;
  public abstract void update();
}
```

Step 3

Create concrete observer classes

BinaryObserver.java

```
public class BinaryObserver extends Observer{

  public BinaryObserver(Subject subject){
    this.subject = subject;
    this.subject.attach(this);
  }

  @Override
  public void update() {
    System.out.println( "Binary String: "
    + Integer.toBinaryString( subject.getState() ) );
  }
}
```

OctalObserver.java

```
public class OctalObserver extends Observer{

  public OctalObserver(Subject subject){
    this.subject = subject;
    this.subject.attach(this);
  }

  @Override
  public void update() {
    System.out.println( "Octal String: "
    + Integer.toOctalString( subject.getState() ) );
  }
}
```

HexaObserver.java

```
public class HexaObserver extends Observer{

  public HexaObserver(Subject subject){
    this.subject = subject;
    this.subject.attach(this);
  }

  @Override
  public void update() {
    System.out.println( "Hex String: "
    + Integer.toHexString( subject.getState() ).toUpperCase() );
  }
}
```

Step 4

Use *Subject* and concrete observer objects.

ObserverPatternDemo.java

```
public class ObserverPatternDemo {
  public static void main(String[] args) {
    Subject subject = new Subject();

    new HexaObserver(subject);
    new OctalObserver(subject);
    new BinaryObserver(subject);
```

```
      System.out.println("First state change: 15");
      subject.setState(15);
      System.out.println("Second state change: 10");
      subject.setState(10);
   }
}
```

Step 5

Verify the output.

```
First state change: 15
Hex String: F
Octal String: 17
Binary String: 1111
Second state change: 10
Hex String: A
Octal String: 12
Binary String: 1010
```