



UNIT I

Object Oriented Methodologies: Software System Life Cycle – Traditional cycle models – Object Oriented approach – Rambaugh et al Object Modeling Technique – Booch Methodology – Jacobson et al methodology – Rational Unified Process (RUP) – Unified Modeling Language (UML) – UML Models.

1.1 THE SYSTEM LIFE CYCLE

Framework is very important for the development of a software system. An agreed framework for development brings many advantages:

1. A framework provides an overall picture of the development process; this picture is not cluttered by detail of what goes on at any stage in the process, but is useful as a high-level view of the major areas of activity, milestones and project deliverables.
2. A framework provides a basis for development and ensures a certain level of consistency in how the work is approached.
3. Consistency approach is very important when large of developers are involved in the project after it has started
4. A framework plays a significant role in ensuring quality, both of the development process and of the final system, by providing a structure for project management-planning, monitoring and controlling the development project.

In software system development, a framework has traditionally been known as a **system life cycle model**. The stages that have been referred for life cycle as requirements, analysis, design, implementation and installation. Each stage is concerned with particular issues and produces a set of outputs or deliverables shown in the below table

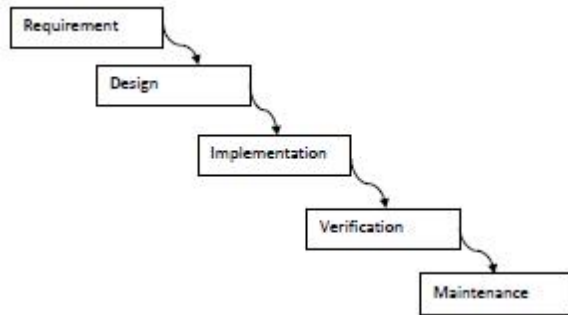
<i>Stage of life cycle</i>	<i>Issues addressed</i>	<i>Deliverables</i>
Requirements	What are the problems, needs and wishes of clients and users? What are the objectives and scope of the proposed system? What are the major risks involved?	List of requirements that can be used as a starting point for development. List of problem areas that fall within the scope of the proposed system. Assessment of risk factors.
Analysis	What does the system look like from the perspective of the clients and users?	A set of models, each taking a different view of the system, which together give a complete picture. The models may be text, diagrams or early prototypes.
Design	How can the system be constructed, so as to satisfy the requirements?	Models from the analysis stage, refined to illustrate the underlying architecture of the system. These models take account of technological considerations and constraints arising from the implementation environment.
Implementation	How can the models produced be translated into code?	A fully tested suite of programs.
Installation	What is needed to support clients and users so that they can use the new system effectively?	User manual, technical documentation, user training. Conversion from current system to new system.



1.2 TRADITIONAL LIFE CYCLE: The most important traditional life cycle models are:

1.2.1 Waterfall Model :

1. This is the early life cycle model; stages of development are straightforward sequence.
2. It describes a development method that is linear and sequence
3. It has distinct goals for each phase of development.
4. Once a phase of development is completed, the development proceeds to the next phase and is turning back.



Requirements: List of requirements for development.

Design: Process of problem solving and planning for a software solution.

Implementation: Coding

Testing: Make sure that the complete system meets software requirements.

Maintenance: modification of the product after deliver to correct faults.

1.2.2 V-model:

1. Stages are visualized in the form of the letter 'V'.
2. It emphasizes how later stages of development are related to earlier stages; for example, how testing should be derived from the activities that are carried out during requirements and analysis.

1.2.3 Spiral.

1. It incorporates iteration of life cycle stages and focuses on identifying and addressing the risks involved in development.
2. At each iteration around the cycle, the products are extensions of an earlier stage.

1.2.4 Prototyping.

1. In the prototyping life cycle, implementation takes place early in the development process.
2. The working model produced is subsequently refined and enhanced during a series of iterations until it is acceptable to the client.

1.2.5 Iterative Development:

1. This approach is closely related to the spiral model and to prototyping.
2. It covering the complete functionality of the system is produced and then refined as development progresses.

1.2.6 Incremental development.

1. In this life cycle model the system is partitioned according to areas of functionality.
2. Each major functional area is developed and delivered independently to the client.

1.3 THE OBJECT-ORIENTED APPROACH

One of the differences that is immediately obvious between traditional life cycle models and the object-oriented approach is the way that the various stages are named.

Traditional Life Cycle model	Object Oriented approach
------------------------------	--------------------------



Traditional model the name, such as ‘analysis’ or ‘implementation’, reflects the activities that are intended to be carried out in that stage.	A clear distinction is made between the activities and the stages (generally referred to as phases) of development.
	Phases are inception, elaboration, construction and transition. indicating the state of the system

1.3.1 Phases

o Inception:

It covers the initial work required to set up and agree terms for the project. It includes establishing the business case for the project, incorporating basic risk assessment and the scope of the system that is to be developed.

o Elaboration:

It deals with putting the basic architecture of the system in place and agreeing a plan for construction. During this phase a design is produced that shows that the system can be developed within the agreed constraints of time and cost.

o Construction:

It involves a series of iterations covering the bulk of the work on building the system; it ends with the beta release of the system, which means that it still has to undergo rigorous testing.

o Transition

It covers the processes involved in transferring the system to the clients and users. This includes sorting out errors and problems that have arisen during the development process.

In object-orientation, activities such as analysis or design are referred to as workflows. The below figure shows the different workflows that typically take place during a system development project.

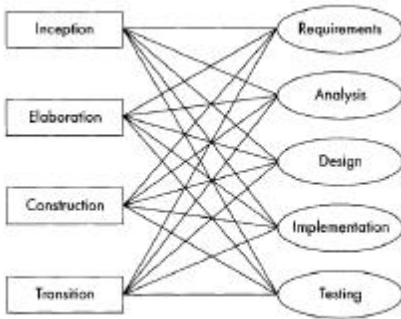


Workflows that take place during development of a system

It is recognized that a workflow may be carried out at more than one development phase and that developers may well engage in the whole range of workflows during every phase of building a system.

During the construction phase the main activities will be implementation and testing, but if bugs are found there will have to be some requirements and analysis as well.

The OO approach to development views the relationships between workflows and phases of development rather like the spider’s web in the below figure, where any phase may involve all workflows, and a workflow may be carried out during any phase.



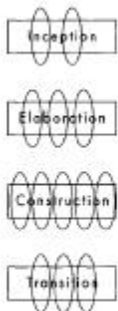
An object-oriented view of development phases and workflows

The object-oriented approach also recognizes fully the reality of iterative development. Activities at any phase do not take place in a neatly ordered fashion.

A developer may have to revisit a range of workflows several times during one phase of development, before it is possible to move on to the next phase.

The below figure illustrates the phases of the object-oriented life cycle with iteration of workflows at each phase.

In the diagram that iterations are most likely during construction, but can occur during any phase of development. Each ellipse represents a range of workflows.



Phases of object-oriented development with iterations of workflows

In addition to the emphasis on iterative development, the object-oriented approach also differs from traditional life cycle models in that it stresses the importance of a seamless development process.

This means that the separate phases are less distinct from each other than in a traditional system life cycle; it is not considered essential, nor is it often easy, to be able to say precisely when one phase is completed and another begins.

Although the traditional system life cycle was concerned about issues such as quality, ease of modification and potential reuse, it tended to regard them as add-ons to the core development process.

In the object-oriented approach such issues are regarded as central, and developers are encouraged to bear them in mind throughout the time they are working on the system.

1.3.1 RAMBAUGH ET AL OBJECT MODELING TECHNIQUE Object modelling techniques (OMT) presented by Jim Rumbaugh describes a method for the analysis, design and implementation of a system using OOT. It is a fast, intuitive approach for identifying and modelling all the object making up a system. This model lets you specify detailed state transitions their descriptions within in a system. It consists of 4 phases:

1. **Analysis:** The results are objects and dynamic and functional models.
 2. **System Design:** The results are structure of a basic architecture of the system along with the high-level strategy decisions.
 3. **Object Design:** This phase produce a design document, consisting of a detailed objects static, dynamic and functional models.
 4. **Implementation:** This activity produces reusable, extendible and robust code.
- OMT separates modeling into three different parts:

1. **Object model:** presented by the object model and data dictionary.
2. **Dynamic model:** presented by the state diagrams and event flow diagrams.
3. **Functional model:** presented by data flow and constraints.

1.3.1 THE OBJECT MODEL It describes structure of object in the system; their identity and relationship to other objects, attributes and operations. The figure below shows object model with graphical representation

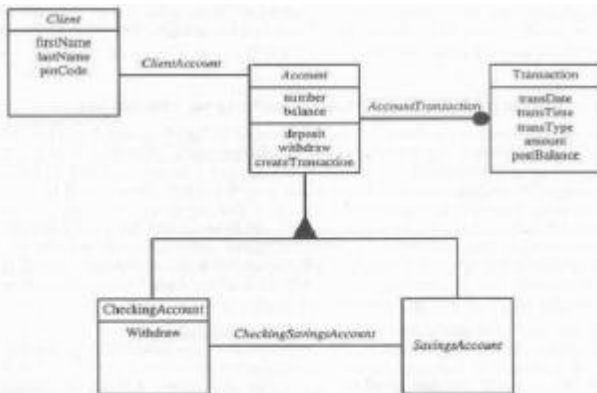


FIGURE 4-1
 The OMT object model of a bank system. The boxes represent classes and the filled triangle represents specialization. Association between Account and transaction is one to many; since one account can have many transactions, the filled circle represents many (zero or more). The relationship between Client and Account classes is one to one; A client can have only one account and account can belong to only one person (In this model joint accounts are not allowed).

1.3.2 THE DYNAMIC MODEL It provides detailed and comprehensive dynamic model, in addition to letting you depict states, transitions, events and actions. The below figure shows state transition is a network of states and events.

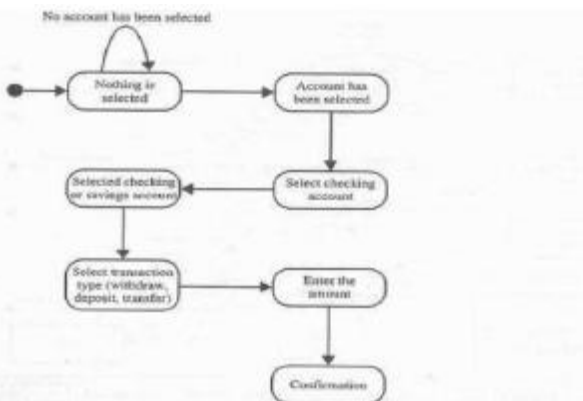


FIGURE 4-2
 State transition diagram for the bank application user interface. The round boxes represent states and the arrows represent transitions.

1.3.3 THE FUNCTIONAL MODEL It shows the flow of data between different processes in a business. The OMT DFD provides a simple and intuitive method for describing business processes without focusing on the details of computer systems. DFD use 4 primary symbols:

1. The *process* is any function being performed; *example* verifying Password/PIN in ATM.



2. The *data flow* shows the direction of data element movement; **example** PIN code.
3. The *data store* is a location where the data are stored; **example** account data store in ATM
4. The *external entity* is a source/ destination of a data element; **example** ATM card reader

1.4 THE RATIONAL UNIFIED PROCESS (RUP)

A life cycle provides a high-level representation of the stages that a development project must go through to produce a successful system.

A development method, on the other hand, is much more prescriptive, often setting down in detail the tasks, responsibilities, processes, prerequisites, deliverables and milestones for each stage of the project.

Nowadays, almost all object-oriented projects use the Unified Modeling Language as the principal tool in their development process.

Use of the UML has been approved by the Object Management Group (OMG), which controls issues of standardization in this area. This has resulted in conformity between projects in terms of notation and techniques.

The creators of the UML have proposed a generic object-oriented development The Unified Software Development Process (Jacobson et al., 1999) and this generic method has been adopted and marketed by the Rational Corporation under the name of the Rational Unified Process (RUP). RUP is based on the following six 'Best Practices'

- 1 Develop software iteratively
- 2 Manage requirements
- 3 Use component-based architectures
- 4 Visually model software
- 5 Verify software quality
- 6 Control changes to software.

1. Develop software iteratively

- RUP follows the phases of the generic object-oriented life cycle (inception, elaboration, construction and transition). It is built on the central concept of iterative development and each of its phases defines a series of activities that may be performed once or a number of times.
- Each iteration is defined as a complete development loop resulting in the release of an executable product that is a subset of the final system.
- In this way RUP supports incremental development- the frequent release of small packages of software that gradually build up to become the final system.
- Iteration and incremental development encourage involvement and feedback from clients and users; they make it easier to cope with changes, and reduce the risk factors associated with any development project.

2. Manage requirements

- RUP offers sound support for eliciting, organizing and recording requirements. Precise documentation of requirements facilitates traceability through the development process, which enhances the quality of the final system.
- The emphasis on the activities that take place early on in the life cycle provides a sound foundation for the later stages and results in systems that are robust, reliable and meet the needs of their users.



3. Use component-based architectures

- RUP prescribes the early identification and development of a system structure that is at the same time robust enough to ensure system reliability, and flexible enough to accommodate changes. This is achieved through the use of components subsystems that each have a single, well-defined function.
- RUP describes how to construct an architecture combining both new and previously existing components, thus encouraging the reuse of software as part of the development process.

4. Visually model software

- RUP is based around the Unified Modelling Language (UML) as a vehicle for development. UML has become an industry standard, and incorporates a wide range of techniques and tools to support developers. The techniques offered by UML bring with them all the advantages of visual modelling.
- *For example*, UML diagrams facilitate communication between developers and users and between members of the development team, they offer a number of different views of the system which combine to give a complete picture, they help developers to decompose the problem into smaller, more manageable chunks, and they provide a means of abstraction, concentrating on important information while hiding details that are currently irrelevant.

5. Verify software quality

- RUP provides the techniques to support quality assessment of functionality, reliability and performance throughout the development process.
- The RUP approach to quality is based on objective measures and criteria for success; it involves all members of the development team and applies to all the activities that are carried out as part of the system development.

6. Control changes to software

- Changes are the norm in a software development project, so an effective development process must be able to monitor and control them.
- RUP provides tools to do this, and also supports the work of developers by offering protection in one area of development from changes that occur in another.

1.5 UNIFIED MODELLING LANGUAGE (UML)

The Unified Modelling Language, or UML, is a set of diagrammatic techniques, which are specifically tailored for OOD, and which have become an industry standard for modelling object-oriented systems.

1.5.1 Modelling:

Software developers use specialized diagrams to model the system that they are working on throughout the development process. Each model produced represents part of the system or some aspect of it, such as the structure of the stored data, or the way that operations are carried out. Each model provides a view of the system, but not the whole picture.

1.5.2. Abstraction:



The characteristic of a model to provide some but not all the information about the person or thing being modelled is known as **abstraction**. Each of the modelling techniques in the Unified Modelling Language provides a particular view of the system as it develops; each UML model is an abstraction of the complete system. Abstraction, concentrates on only those aspects of the system that are currently of interest, and putting other details to the side for the time being.

1.5.3. Decomposition:

This is the breaking down of a large, complex problem or system into successively smaller parts, until each part is a 'brain-size' chunk and can be worked on as an independent unit. Traditionally software systems used to be decomposed according to their functions - the tasks that the system had to carry out. In OO, systems are decomposed according to the data that they have to store, access and manipulate.

1.6 UML MODELS

The UML is not a development method since it does not prescribe what developers should do, it is a diagrammatic language or notation, providing a set of diagramming techniques that model the system from different points of view.

The below table shows the principal UML models with a brief description of what each can tell us about the developing system.

The 4 + 1 view. The authors of UML, Booch et al., (1999), suggest the architecture of a system from five different perspectives or views:

- The use case view
- The design view
- The process view
- The implementation view
- The deployment view.

This is known as the 4 + 1 view (rather than the 5 views) because of the special role played by the use case view.

The Use Case view:

it specifies what the user wants the system to do; the other 4 views describe how to achieve this.

The use case view describes the external behavior of the system and is captured in the use case model

The Design view:

It sometimes called as **logical view**. Describes the logical structures required to provide the functionality specified in the use case view.

The design view describes the classes (including attributes and operations) of the system and their interactions.

Table 1.3: The principal UML diagrams with brief descriptions

Model	View of the system
Use case	How the system interacts with its users.
Class	The data elements in the system and the relationships between them.
Interaction (sequence and collaboration)	How the objects interact to achieve the functionality of a use case.
State	How the different objects of a single class behave through all the use cases in which the class is involved.
Activity	The sequence of activities that make up a process.
Component	The different software components of the system and the dependencies between them.
Deployment	The software and hardware elements of the system and the physical relationships between them.



The Process view:

It is concerned with describing concurrency in the system.

Sequence diagram can be used to achieve it.

The Implementation view:

It describes the physical software components of the system, such as executable files, class libraries and databases.

The view of the system can be modeled using component diagram

The Deployment view:

This view describes the hardware components of the system such as PCs, mainframes, printers and the way they are connected.

This view can also be used to show where software components are physically installed on the hardware elements.

OBJECT ORIENTED ANALYSIS AND DESIGN

UNIT IV

Object Oriented Design: Designing Classes, methods – access layer object storage and object interoperability –access layer for the ATM banking system. View layer – designing interface objects – prototyping User interface – view layer for the ATM banking system

1. How to Design Classes? Explain in detail.

The most important activity in designing an application is coming up with a set of classes that works together to provide the needed functionality. Underlying the functionality of any application is the quality of its design.

UML – OCL: UML is a graphical language with a set of rules & semantics in English in form of OCL. Object Constraint Language (OCL) is specification language that uses simple logic for specifying properties of a system. Syntax of some common navigational expressions is shown here

- *Item.selector:* The selector is name of an attribute in item. The result is the value of attribute
- *Item.selector [qualifier–value]:* The selector indicates a qualified association that qualifies the item. Result is related object selected by qualifier, eg., array indexing as form of qualification.
- *Set select (Boolean–expression):* Boolean expression is written in terms of objects within the set

The Process

During the design phase the classes identified in OOA must be revisited with a shift in focus to their implementation. New classes or attributes & methods are to added for implementation purposes & user interfaces.

The process consists of following activities

1. *Apply design axioms* to design classes, their attributes, methods, associations, structures & protocols. It constitutes two separate steps

⇒ *Refine & complete* the static UML class diagram by adding details. This steps consists of Refine attributes

Design methods & protocols by UML activity diagram to represent methods algorithm Refine associations between classes (if required)

Refine class hierarchy & design with inheritance (if required)

⇒ *Iterate* and refine again

Object oriented design is an iterative process. At each iteration, you can improve the design.

Class visibility

The main objective is designing well defined public, private & protected protocols

Public protocols define the functionality & external messages of an object; private protocols define implementation of an object.

Private Protocol (visibility) of class includes messages that normally should not be sent from other objects; it is accessible only to operations of that class.

In protected protocol (visibility), subclasses use method in addition to class itself. *Encapsulation leakage* – is a lack of well designed protocol

The problem of *encapsulation leakage* occurs when details about a class's internal implementation are disclosed through the interface.

Refining Attributes

The main goal of this activity is to refine existing attributes (identified in analysis) or add attributes that can elevate the system into implementation.

Attributes Types

The three basic types of attributes are

- Single-value attributes.
- Multiplicity or Multi-value attributes.
- Reference to another object, or instance connection.

Attributes represent the state of an object. When the state of the object changes, these changes are reflected in the value of attributes.

Single value attribute has only one value or state. (Eg). Name, address, salary.

Multiplicity or multivalued attribute can have a collection of many values at any time. (Eg) If we want to keep track of the names of people who have called a customer support line for help.

Instance connection attributes are required to provide the mapping needed by an object to fulfill its responsibilities.

(E.g.) A person may have one or more bank accounts.

A person has zero to many instance connections to Account(s). Similarly, an Account can be assigned to one or more person(s) (joint account). So an Account has zero to many instance connection to Person(s).

UML Attribute presentation: The following is the attribute presentation suggested by UML
Visibility name: type-expression = initial-

value where visibility is one of following

- + public visibility (accessibility to all classes)
- # protected visibility (accessibility to subclasses & operations of class)
- private visibility (accessibility only to operations of the class)

Type-expression is language-dependent specification of implementation type of an attribute. Initial value is language-dependent expression for initial value of newly created object and is optional.

2. How to Design Methods and Protocols? Explain in detail.

A class can provide several types of methods:

- Constructor: Method that creates instances (objects) of the class
- Destructor: The method that destroys instances
- Conversion Method: The method that converts a value from one unit of measure to another.
- Copy Method: The method that copies the contents of one instance to another instance
- Attribute set: The method that sets the values of one or more attributes
- Attribute get: The method that returns the values of one or more attributes
- I/O methods: The methods that provide or receive data to or from a device
- Domain specific: The method specific to the application.

Use private and protected protocols to define the functionality of the object. Remember five rules to avoid bad design:

If it looks messy, then its probably a bad design

If it looks too complex, then its probably a bad design

If it is too big, then its probably a bad design

If people don't like it, then its probably a bad

design If it doesn't work, then its probably a bad design

Apply design axioms and corollaries to avoid design pitfalls and use UML operation presentation which is similar to syntax of UML attribute representation

UML operation presentation: The following is the operation presentation suggested by UML

Visibility name: (parameter-list): return-type-expression

where visibility is one of following

- + public visibility (accessibility to all classes)
- # protected visibility (accessibility to subclasses & operations of class)
- private visibility (accessibility only to operations of the class)

parameter-list: is a list of parameters, separated by commas, each specified by name: type-expression = default value

return-type-expression: is a language dependent specification of the implementation of the value returned by the method.

Eg: +getName(): aName +getAccountNumber (account: type): account Number

3. Explain briefly the functions of Access Layer in detail. Object Storage and Object Interoperability

Object Storage and persistence:-

A database management system (DBMS) is a set of programs that enables the creation maintenance of a collection of related data. The fundamental purpose is to provide a reliable, persistent data storage facility & mechanisms for efficient, convenient data access & retrieval *Persistence* refers to the ability of some objects to outlive the programs that created them. A program will create a large amount of data throughout its execution. Each item of data will have a different life time. Atkinson et al. describe *six broad categories* of life time of data:

- Transient results to evaluation of expressions
- Variables involved in procedure activation (parameters & variables with a localized scope)
- Global variables & variables that are dynamically allocated
- Data that exist between the executions of a program
- Data that exist between the versions of a program
- Data that outlive a program

The first three categories are *transient data*, data that cease to exist beyond lifetime of creating process. The other three are non-transient or persistent data. A file or a database can provide a longer life for objects – longer than duration of process in which they were created. From a language perspective, this characteristic is called persistence. Essential elements in providing a persistent store are

1. Identification of persistent objects or reach ability (*object ID*).
2. Properties of objects & their interconnections. The store must be able to coherently manage non-pointer & pointer data (i.e., *inter-object references*).
3. Scale of the object store. The object store should provide a conceptually infinite store.
4. *Stability*: The system should be able to recover from unexpected failures and return the system to a

recent self-consistent state. This is similar to reliability requirements of a DBMS.

Data Base Management Systems: -

DBMS is a set of programs that enable the creation & maintenance of a collection of related data. They have number of properties that distinguish them from file-based data management approach

A *fundamental characteristic* of database approach is that DBMS contains not only data but complete definition of data formats it manages. This description is known as schema or meta-data containing a complete definition of data formats, such as data structures, types & constraints

Advantage of database approach is that it will provide a generic storage management mechanism.

Another one is program – data independence

Database Views: DBMS provides the database users with a conceptual representation that is independent of low-level details (physical view) of how the data are stored. The database can provide an abstract data model that uses logical concepts such as field, records, tables & their interrelationships. DBMS can provide multiple virtual views of data that are tailored to individual applications. This allows convenience of private data representation with advantage of globally managed information

Database Models: It is a collection of logical constructs used to represent the data structure & data relationships within the database. It is grouped into two categories:

Conceptual model is concerned with what is represented in the database & *Implementation model* is concerned with how it is represented. It can be stated as Hierarchical model, Network model, Relational model (*tuples – Primary key & foreign key*) Database

Interface: The interface on a database must include a data definition language (DDL), query and data manipulation language (DML)

- Database systems adopt *two approaches* for interfaces with system. One is to embed a database language (SQL) & other is to extend the host programming language with database constructs
- *Database Schema & DDL*: DDL is the language used to describe structure of & relationships between objects stored in a database. This structure of information is database schema
- *DML & Query Capabilities*: DML is language that allows users to access & manipulate data organization. SQL is standard DML for relational DBMSs. It is widely used for its query capabilities. The

Query usually specifies

The domain of the discourse over which to ask the query

The elements of general interest

The conditions or constraints that apply

UNIT – II

UML Diagrams: Use case diagram – UML class diagram – interaction diagram – state diagram – activity diagram – Requirements for ATM banking system – case study

UML DIAGRAMS: The UML is a language for specifying, constructing, visualizing and documenting the software system and its components. The UML is a graphical language with sets of rules and semantics, in a form known as Object constraint language (OCL). The primary goals in the design of the UML were as follows:

1. Provide users a ready-to-use, expensive visual modeling language so they can develop and exchange meaningful models.
2. Provide extensibility and specialization mechanisms to extend the core concepts.
3. Be independent of particular programming languages and development processes.
4. Provide a formal basis for understanding the modeling language.
5. Encourage the growth of the OO tools market.
6. Support higher-level development concepts.
7. Integrate best practices and methodologies.

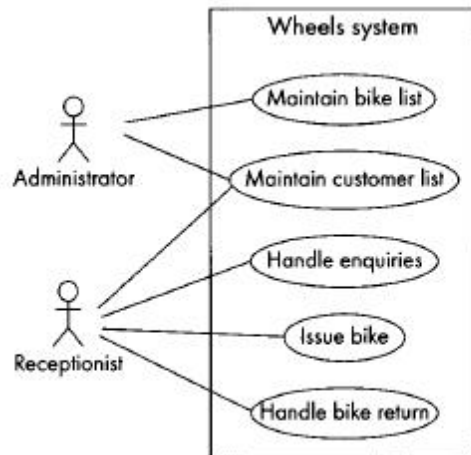
The UML defines 6 graphical diagrams:

1. Use-case diagram
2. Class Diagram
3. Interaction diagram
 - a. Sequence diagram
 - b. Collaboration diagram
4. State chart diagram
5. Activity diagram
6. Implementation diagram
 - a. Component diagram
 - b. Deployment diagram

USECASE DIAGRAM:

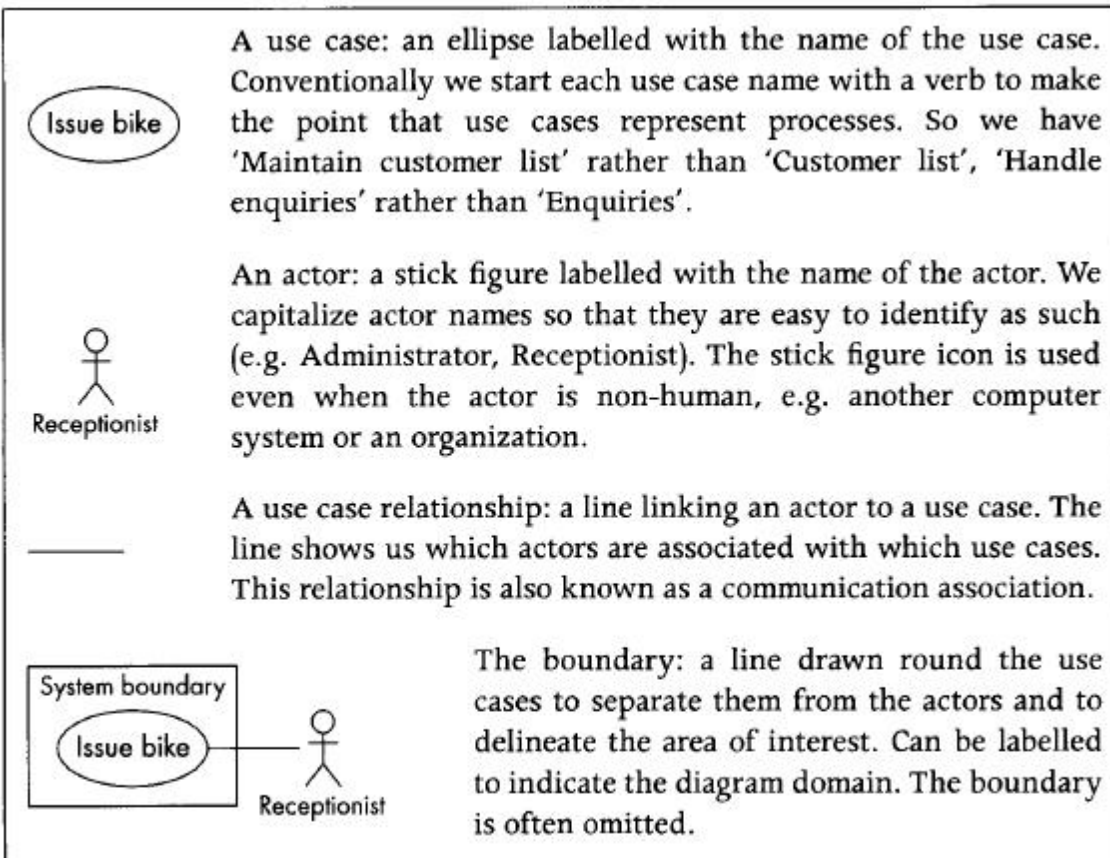
Use case Diagram concept was introduced by Ivar Jacobson in the OOSE method. This corresponds to a sequence of transactions, in which each transaction is invoked from outside the system (actors and engages internal objects to act with one another.

This is a graph of actors, a set of use cases enclosed by a system boundary, communication association between the actors and the use cases and generalization among the use cases.



IDENTIFYING USE CASES FROM THE ACTORS One is to identify the actors, the users of the system, and for each one, to establish how they use the system, what they use it to achieve. Scenarios belonging to the same use case have a common goal each scenario in the group

describes a different sequence of events involved in achieving (or failing to achieve) the use case goal.



The UML Symbols for use case diagram

These relationships are shown in a use case diagram:

1. **Communication:** Connecting actor symbol to the use case symbol with a solid path.
2. **Uses:** relationship between use cases is shown by a generalization arrow from the use case.
3. **Extends:** Used when one use case that is similar to another use case but does a bit more. It is like a subclass.

UML CLASS DIAGRAM:

The UML Class diagram, also referred to as object modeling, is the main static analysis diagram. This shows the static structure of the model.

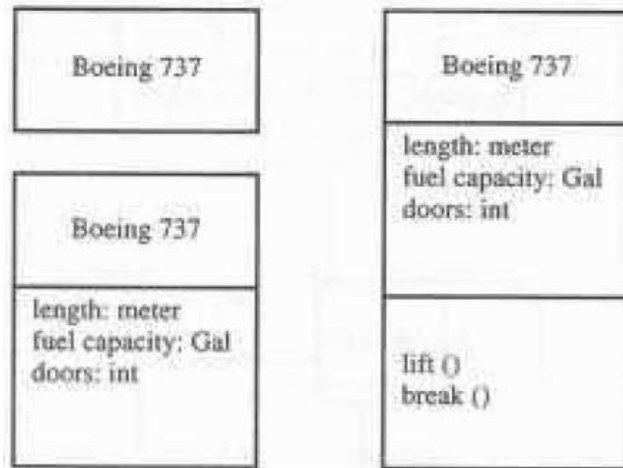
It is a collection of static modeling elements, such as classes and their relationships, connected as graph to each other and to their contents. It does not show temporal information, which is required in dynamic modeling.

The main task of object modeling is to graphically show what each object will do in the problem domain, describe the structure (such as class hierarchy) and relationship among the objects (such as associations) by visual notations, and determine what behavior fall within and outside the problem domain.

CLASS NOTATION: STATIC STRUCTURE

A class is drawn by a rectangle with three components separated by horizontal lines.

- The top name compartment holds the class name
- The general properties of the class, such as attributes are in the middle component.
- The bottom components contain the list of operations.



A separator line is not drawn for a missing compartment if a component is suppressed; no inference can be drawn about the presence or absence of elements in it.

In class notation, either or both the attributes and operation compartments may be suppressed.

OBJECT DIAGRAM

A static object diagram is an instance of a class diagram. It shows the detailed state of the system at a point in time. Class diagram contains object, so a class diagram with object and no classes is an object diagram

CLASS INTERFACE NOTATION

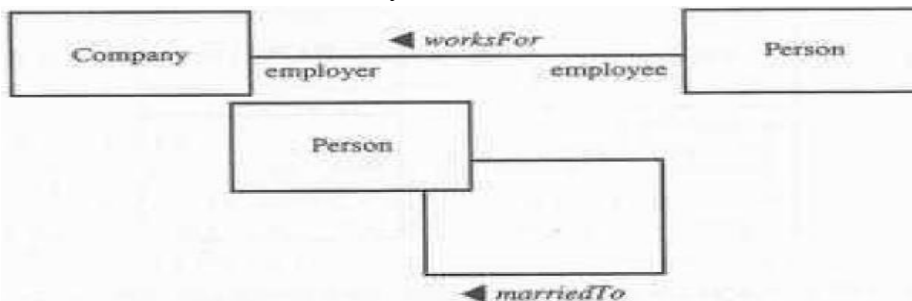
This is used to describe the externally visible behavior of a class; example, an operation with public visibility.

Interface notation of a class.



BINARY ASSOCIATION NOTATION

This is drawn as a solid path connecting two paths, or both ends may be connected to the same class. An association may have an association rule.



5. 0 Association Notation

Association name may have an optional black triangle in it, the point of the triangle indicating the direction in which to read the name, where it is connected to a class is called **association role**.

ASSOCIATION ROLE

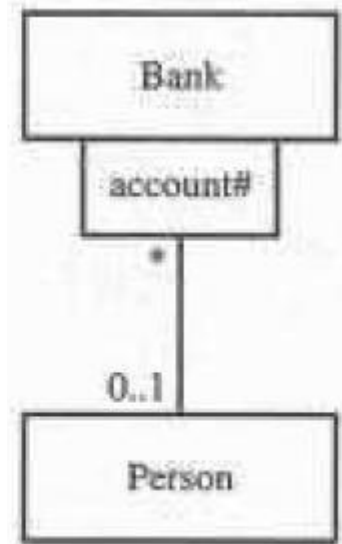
The technical term for it is binary association-is drawn as a solid line connecting two class symbols. The UML uses the term association navigation or navigability to specify a role affiliated with each end of an association relationship.

In fig 5.0 the association is navigable in only one direction, from the Bank Account to Person, but not the. Reverse

QUALIFIER

Qualifier is an association attribute. For example, a person object may be associate to a Bank object. An attribute of this association is account#. The account# is the qualifier of this association.

A qualifier is shown as small rectangle attached to end of an association path, between the final path segment and the symbol of the class to which it connects.

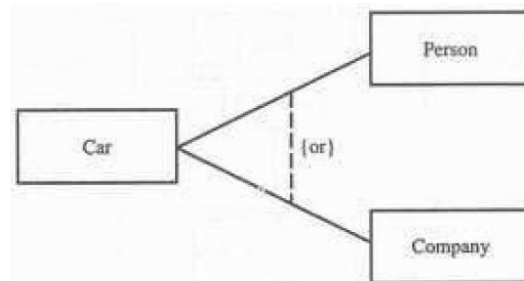


MULTIPLICITY

It specifies the range of allowable associated class. It is given for roles within association, parts with compositions, repetitions and other purposes. It shows as a text string comprising a period-separated sequence of integer intervals, where an interval represents a range of integers in the figure above. Example Lower bound.... Upper bound 0.....1 0..*

OR ASSOCIATION

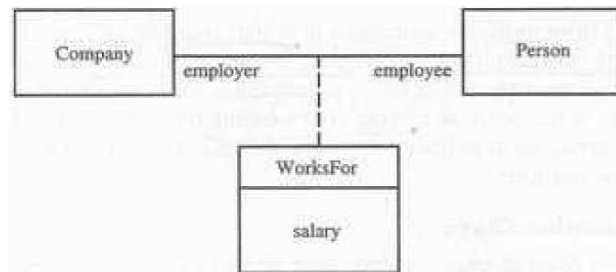
It indicates the situation in which only one of several potential associations may be instantiated at one time for any single object. This shown as dashed line connecting two or more associations, all of which have a class common, with the constraint string {or} labeling dashed line



ASSOCIATION CLASS

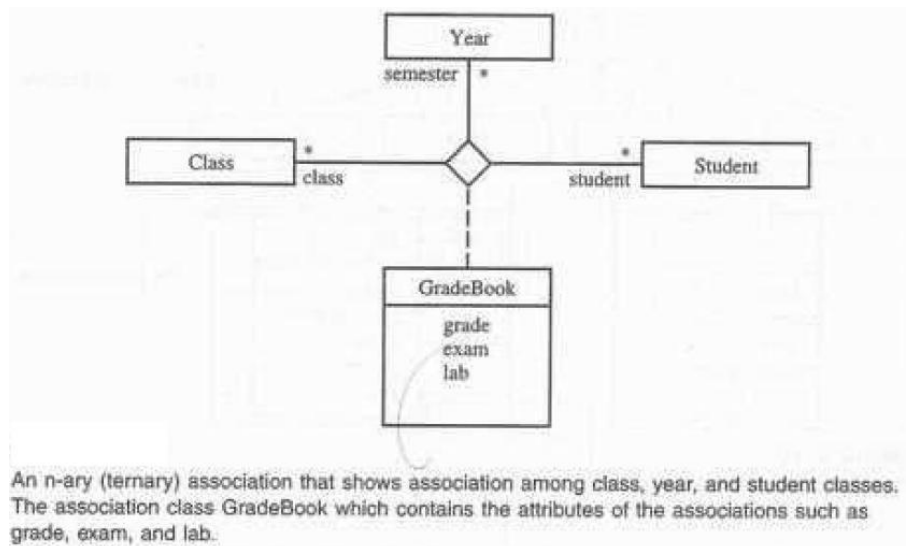
It is also has class properties. An association class is shown as a class symbol by a dashed line to an association path.

If an association class has attributes but no operation or other association, then the name may be displayed on the association path and omitted from the association class to emphasize its **“association nature”**. If it has operations and attributes, then the name may be omitted from the path and placed in the class rectangle to emphasize its **“class nature”**.



N-ARY ASSOCIATION

This is an association among more than two classes. N-ary is difficult to understand, it is better to convert it into binary association. It is shown as a large diamond with a path from diamond to each participant class. Multiplicity may be indication; however, qualifiers and aggregations are not permitted. In the below figure diamond by a dashed line, indicating N-ary association that has attributes, operations and associations.

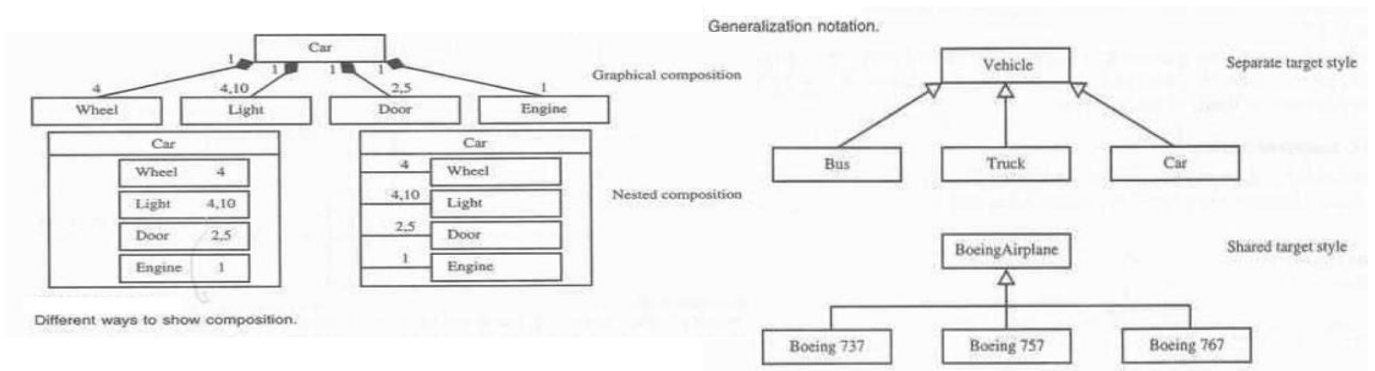


AGGREGATIONS AND COMPOSITION An aggregation is in the form of associations. A hollow diamond is attached at the end of the path indicates aggregation. Diamond may not be attached both ends.

Composition is also known as *a part-of*, is a form of aggregation with strong ownership to represent the component of a complex object. Composition also referred as *a part-whole relationship*

GENERALIZATION

Generalization is the relationship between a more general class and more specific class. It is displayed with a directed line with closed, hollow arrowhead at the super class end.



The UML allows discriminator label to be attached to a generalization of the superclass. In the above example Ellipse (...) indicates the generalization is incomplete and more subclass exist. The constructor complete indicates that generalization is complete and no more subclasses are needed.

INTERACTION DIAGRAM Interaction diagram describes how group of objects collaborate to get the job done. It captures the behavior of a single use case, showing the pattern of interaction among objects. There are two kinds of interaction models:

1. Sequence diagram
2. Collaboration diagram

UML SEQUENCE DIAGRAM Sequence diagram describes the behavior of the system viewing the interaction between the system and its environment. It shows an interaction arranged in a time sequence. It has two dimensions:

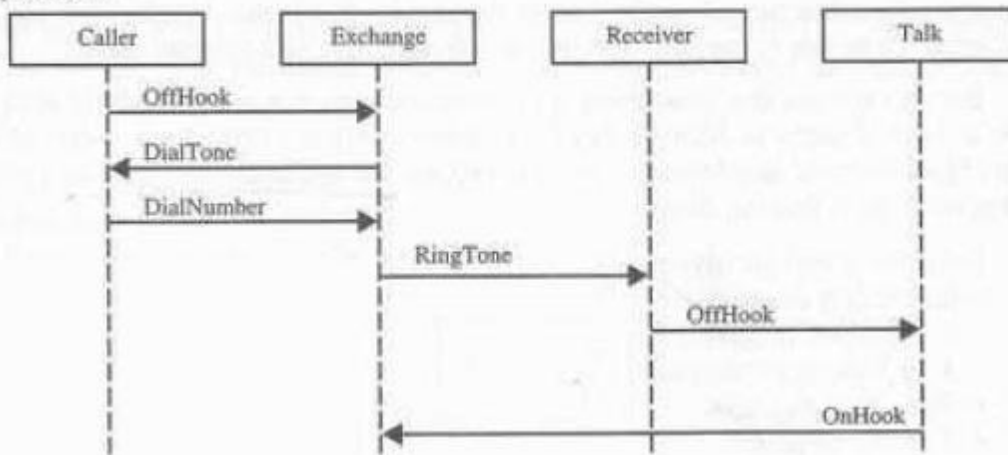
1. Vertical dimension represents time – called object *lifeline object existence during interactions*).
2. Horizontal dimension represents different objects.

The sequence of execution in an object-oriented program is complicated; it is hard to follow the flow of control as it is passed backwards and forwards between objects.

This demonstrates the usefulness of the sequence diagram as a map to guide us through the code. Sequence diagrams provide an overview of the inter-object messaging sequence; this is useful for software designers and for programmers, both when they are writing the code and when they are maintaining it.

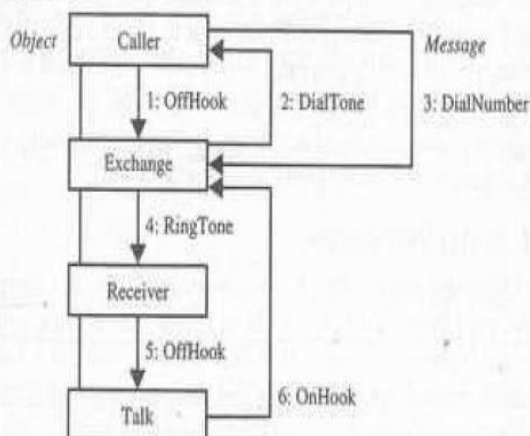
An example of a sequence diagram.

Telephone Call



A collaboration diagram with simple numbering.

Telephone Call



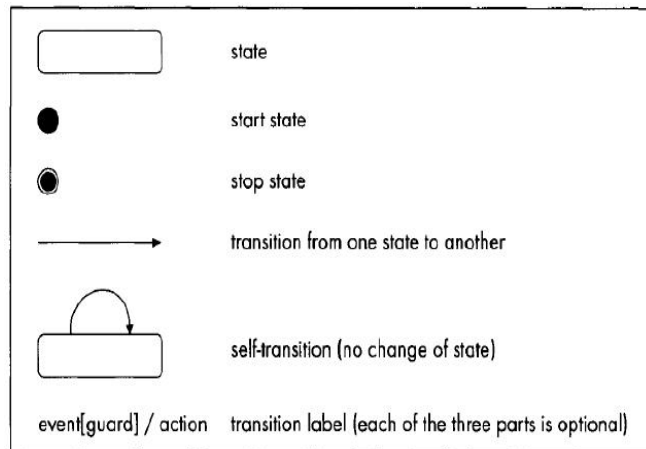
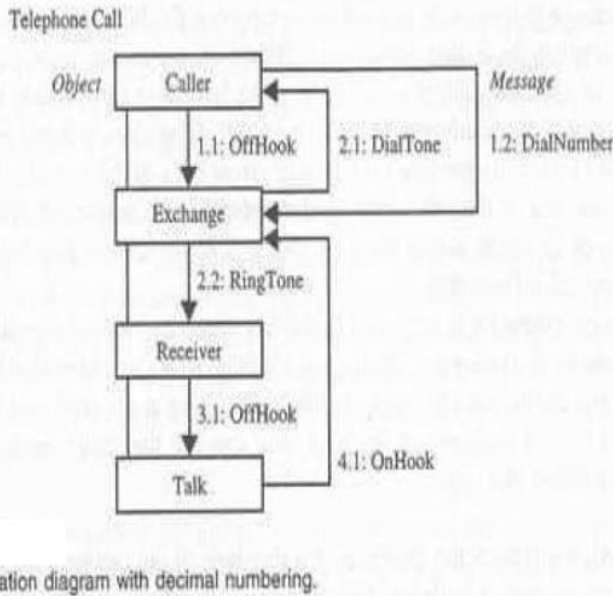
Each message is represented by an arrow between the lifelines of two objects. Each message is labeled with the message name. The sequence diagram is very simple and has immediate visual appeal- this is its great strength.

UML COLLABORATION DIAGRAM It represents a collaboration, which is a set of

objects related in a particular context, and interaction, which is a set of objects related in a particular context, and interaction,

which is a set of messages exchanged among the objects within the collaboration to achieve a desired outcome. In this diagram, the sequence is indicated by numbering the messages, makes it more difficult to see the sequence than drawing the lines on the page. This layout uses to indicate how objects are statistically connected. It is more compressed.

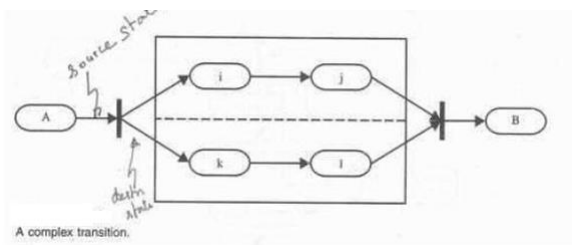
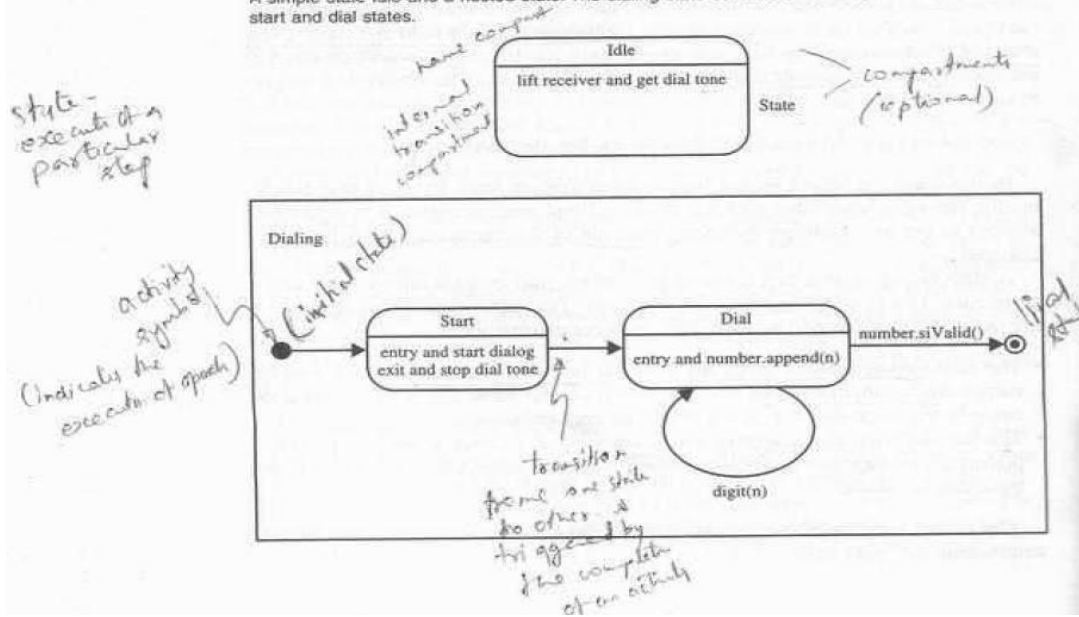
An interaction diagram is used to examine the behavior of objects within a single use case. It is good at showing collaboration among the objects but not so good at precise definition of the behavior. The interaction diagram loses its clarity with more complex conditional behavior. It is simplicity.



STATE DIAGRAM It shows the sequence of states that an object goes through during its life in response to outside stimuli and messages. The state is set of values describes an object in a specific point in time and is represented at state symbol and transitions are represented by arrows connecting the state symbols.

It may contain sub diagrams. It represents the state of the method execution (the state object executes the method), and activities in the diagram represents the activities of the object that performs the method.

A simple state Idle and a nested state. The dialing state contains substates, which consist of start and dial states.



A complex transition.

shown as a short heavy bar.

The transitions can be simple or complex. Events are processed one at a time. An event that triggers no transitions is simply ignored. A complex transition may have multiple source and target states. It represents synchronization or splitting of control into concurrent threads. A complex transition is

A bar may have one or more solid arrows from states to the bar (these are source states); It may also have one or more solid arrows from the bar to the states (these are destination states). State diagrams are useful, when you have a class that is very dynamic. It emphasizes the use of events and states to determine the overall activity of the system.

ACTIVITY DIAGRAM

An activity diagram, all the states are activities (i.e. a state of doing something) and the transitions between them are triggered by the completion of the activity, rather than by an external event. Activity diagrams show the internal flow of control in a process. Symbols used in activity diagram

Unlike state diagram that focus on the event occurring to a single object as it responds to messages, an activity diagram can be used to model an entire business process; used to

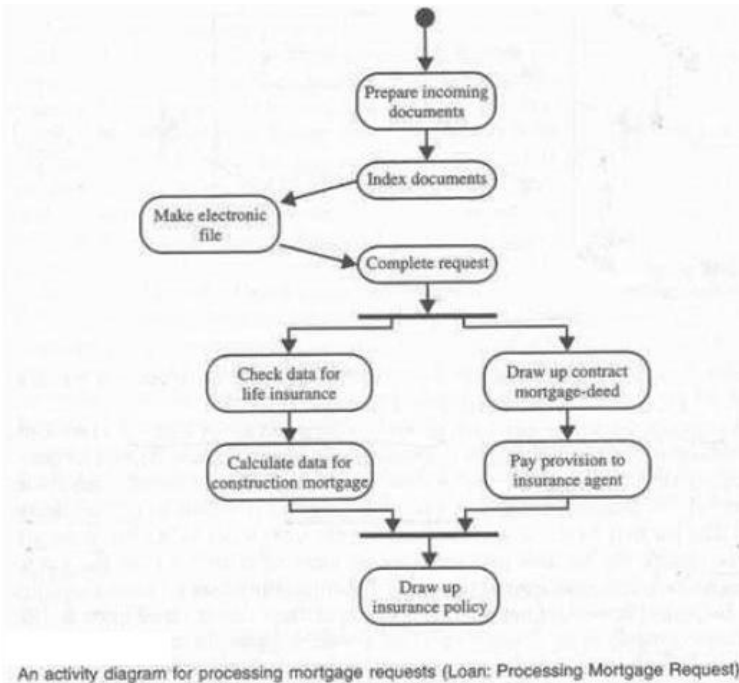
	activity
	start state: only one allowed on each activity diagram or subdiagram
	stop state: more than one can appear on the same diagram
	transition from one activity to another
	diamond indicates beginning (branch) and end (merge) of behaviour that depends on certain conditions being satisfied
	condition indicates whether a particular transition will be taken
	synchronization bar indicates start (fork) or end (join) of parallel processing
	swimlane indicates which agent, person or object is responsible for a set of activities

Activity diagram symbols

provide view of flow and what is going on inside the use case or among several classes

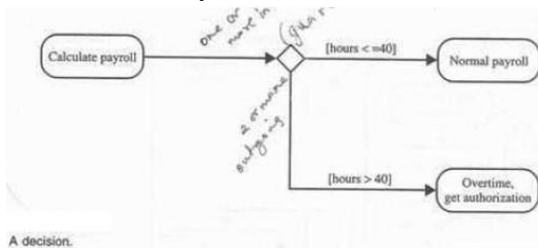
It is similar to state chart where a token represents operations. An operation appears

within the activity diagram indicates the execution of the operation. An outgoing solid arrow attached to an activity symbol indicates a transition triggered by the completion of the activity. The name of this implicit event need not be written, but the conditions that depend on the result of the activity or values may be included. If conditions are not disjoint, then the branch is nondeterministic. The concurrent control is represented by multiple arrows leaving a synchronization bar, which is represented by a short thick bar with incoming and outgoing bars. This diagram mostly to show internal state of an object, but external events may appear in them. An external event may appear when the object is in a “wait



An activity diagram for processing mortgage requests (Loan: Processing Mortgage Request).

state”(no internal activity by the object is waiting for external event to occur) there are 2 states: **wait and activity state** .A wait is a **“normal state”**



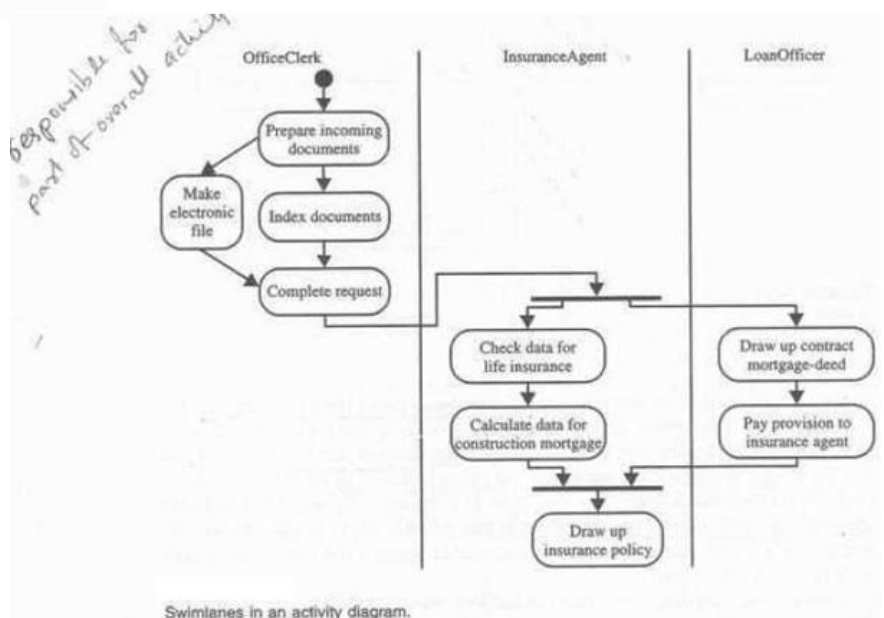
A decision.

The above diagram provided for a decision is the traditional diamond shape, with one or more incoming arrows or two or more outgoing arrows, each labeled by a distinct guard condition. This diagram may be organized into swimlanes, each

separated from neighboring swimlanes by vertical solid lines on both sides.

Swimlanes

Represents responsibility for part of the overall activity and may be implemented by one or more objects. The relative ordering of the swimlanes has no semantic significances but might indicate some affinity. Each action is assigned to one swimlane.



Swimlanes in an activity diagram.

UNIT – III

Object Oriented Analysis : Use case driven Object analysis – approaches for identifying classes – identifying objects, relationships attributes, methods for ATM banking system –Object oriented design process – design axioms.

OBJECT ORIENTED ANALYSIS

Analysis is the process of extracting the needs of a system and what the system must do to satisfy the users' requirement. The goal of object oriented analysis is to understand the domain of the problem and the system's responsibilities by understanding how the users use or will use the system. The main objective of the analysis is to capture a complete, unambiguous, and consistent picture of the requirements of the system and what the system must do to satisfy the users' requirements and needs.

Business Object Analysis: Understanding the Business Layer

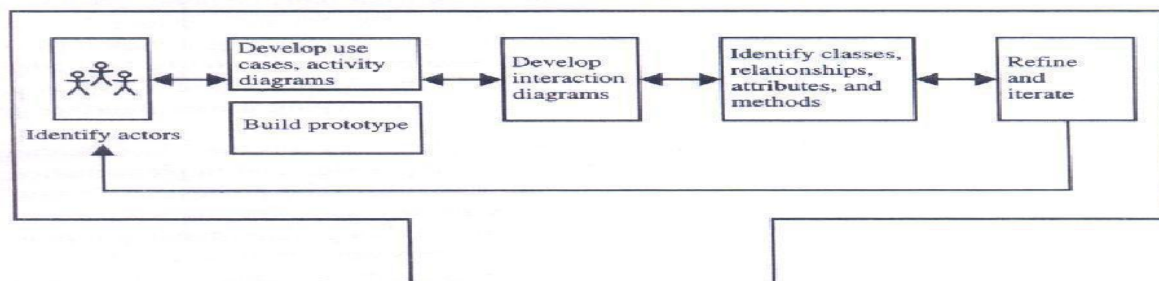
Business object analysis is a process of understanding the system's requirements and establishing the goals of an application. To understand the users' requirements, we need to find out how they "use" the system. This can be accomplished by developing use cases. Defer unimportant details until later. State *what* must be done, not *how* it should be done. This, of course, is easier said than done. Yet another tool that can be very useful for understanding users' requirements is preparing a prototype of the user interface.

USE CASE DRIVEN OBJECT ORIENTED ANALYSIS

The object-oriented analysis (OOA) phase of the unified approach uses actors and use cases to describe the system from the users' perspective. The *actors* are external factors that interact with the system; *use cases* are scenarios that describe how actors use the system. The use cases identified here will be involved throughout the development process. The OOA process consists of the following steps:

1. *Identify the actors:* Who is using the system? Or, in the case of a new system, who will be using the system?
2. *Develop a simple business process model using UML activity diagram.*
3. *Develop the use case:* What are the users doing with the system? Or, in case of the new system, what will users be doing with the system? .Use cases provide us with comprehensive documentation of the system under study.
4. *Prepare interaction diagrams:* Determine the sequence. .Develop collaboration diagrams.
5. *Classification-develop a static UML class diagram:* Identify classes. . Identify relationships. Identify attributes. Identify methods.
6. *Iterate and refine:* If needed, repeat the preceding steps.

The object-oriented analysis process in the Unified Approach (UA).



BUSINESS PROCESS MODELING

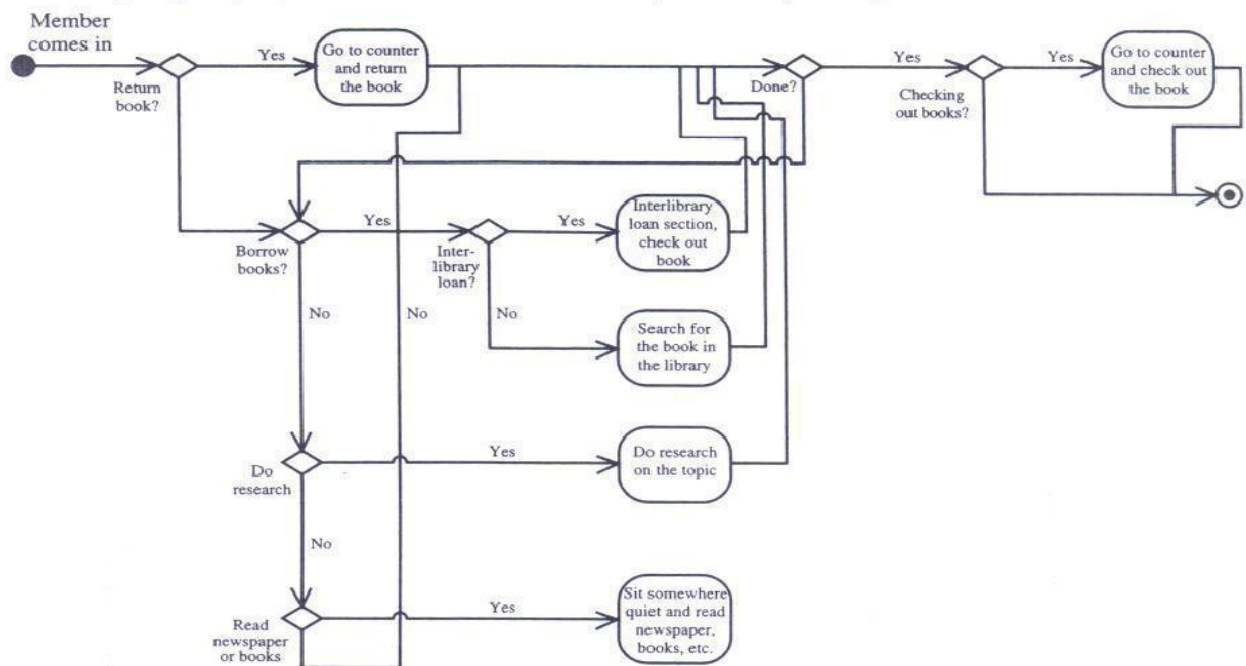
Business process modeling can be very time consuming, so the main idea should be to get a basic model without spending too much time on the process.

The advantage of developing a business process model is that it makes you more familiar with the system and therefore the user requirements and also aids in developing use cases.

For example, let us define the steps or activities involved in using your school library. These activities can be represented with an activity diagram.

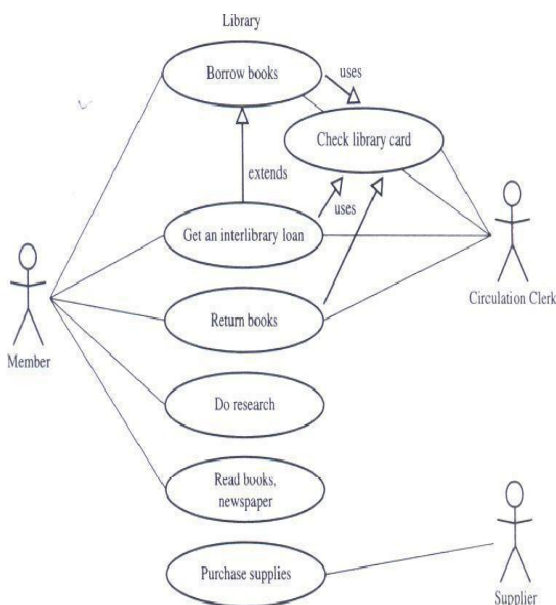
Developing an activity diagram of the business process can give us better understandings of what sort of activities are performed in a library by a library member.

This activity diagram (AD) shows some activities that can be performed by a library member.

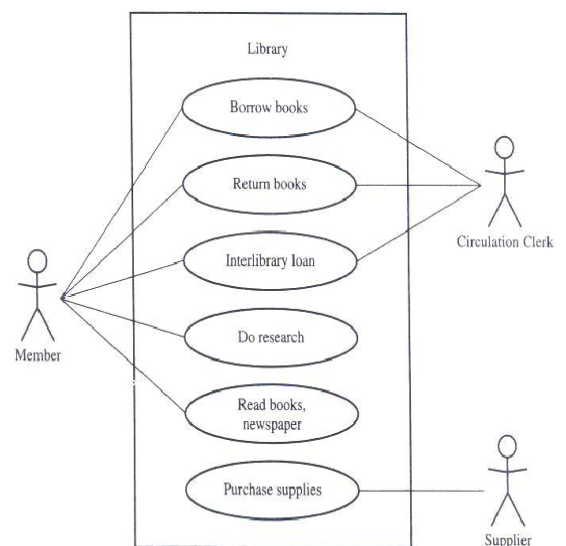


USE-CASE MODEL

Use cases are scenarios for understanding system requirements. A use-case model can be instrumental in project development, planning, and documentation of systems requirements.



A use case is an interaction between users and a system; it captures the goal of the users and the responsibility of the system to its users



The use-case model describes the uses of the

system and shows the courses of events that can be performed. A use-case model also can discover classes and the relationships among subsystems of the systems. Each use or scenario represents what the user wants to do.

Each use case must have a name and short textual description, no more than a few paragraphs.

Since the use-case model provides an external view of a system or application, it is directed primarily toward the users or the "actors" of the systems, not its implementers. As you can see, these are uses of external views of the library system by an actor such as a member, circulation clerk, or supplier instead of a developer of the library system. The simpler the use-case model, the more effective it will be. It is not wise to capture all the details right at the start.

The UML class diagram, also called an object model, represents the static relationships between objects, inheritance, association, and the like. The object model represents an internal view of the system, as opposed to the use-case model, which represents the external view of the system.

The use-case diagram depicts the **extends** and uses relationships where the interlibrary loan is a special case of checking out books. Entering into the system is common to get an interlibrary loan, borrow books, and return books use cases, so it is being "used" by all these use cases.

Transaction: A transaction is an atomic set of activities that are performed either fully or not at all. A transaction is triggered by a stimulus from an actor to the system or by a point in time being reached in the system.

IDENTIFYING CLASSES The four alternative approaches for identifying classes are :

1. noun phrase approach
2. common class patterns approach
3. use case driven, sequence/collaboration modeling approach
4. Classes, Responsibilities, and Collaborators (CRC) approach

The first two approaches have been included to increase your understanding of the subject; the unified approach uses the use-case driven approach for identifying classes and understanding the behavior of objects. However, you always can combine these approaches to identify classes for a given problem. Another approach that can be used for identifying classes is Classes, Responsibilities, and Collaborators (CRC) developed by Cunningham, Wilkerson, and Beck. Classes, Responsibilities, and Collaborators, more technique than method, is used for identifying classes responsibilities and therefore their attributes and methods.

Noun Phrase Approach

The noun phrase approach was proposed by Rebecca Wirfs-Brock, Brian Wilkerson, and Lauren Wiener. In this method, reading through the requirements or use cases and looking for noun phrases. Nouns in the textual description are considered to be classes and verbs.



Using the noun phrase strategy, candidate classes can be divided into three categories:

1. Relevant Classes,

- 2. Fuzzy Area or Fuzzy Classes and
- 3. Irrelevant Classes. Database standards and even fourth-generation languages."

Selecting Classes from the Relevant and Fuzzy Categories

The following guidelines help in selecting candidate classes from the relevant and fuzzy categories of classes in the problem domain.

1. **Redundant classes.** Do not keep two classes that express the same information. If more than one word is being used to describe the same idea, select the one that is the most meaningful in the context of the system. This is part of building a common vocabulary for the system as a whole. Choose your vocabulary carefully; use the word that is being used by the user of the system.
2. **Adjectives classes.** Adjectives can be used in many ways. An adjective can suggest a different kind of object, different use of the same object, or it could be utterly irrelevant. If the use of the adjective signals that the behavior of the object is different, and then makes a new class". For example, Adult Members behave differently than Youth Members; so, the two should be classified as different classes.
3. **Attribute classes.** Tentative objects that are used only as values should be defined or restated as attributes and not as a class. For example, Client Status and Demographic of Client are not classes but attributes of the Client class.
4. **Irrelevant classes.** Each class must have a purpose and every class should be clearly defined and necessary. You must formulate a statement of purpose for each candidate class. If you cannot come up with a statement of purpose, simply eliminate the candidate class.

The process of eliminating the redundant classes and refining the remaining classes is not sequential. The process of identifying relevant classes and eliminating irrelevant classes is an incremental process. Each iteration often uncovers some classes that have been overlooked.

The ViaNet Bank ATM System: Identifying Classes by Using Noun Phrase Approach

To better understand the noun phrase method, we will go through a case and apply the noun phrase strategy for identifying the classes. We must start by reading the use cases and applying the principles discussed in this chapter for identifying classes.

Initial List of Noun Phrases: Candidate Classes The initial study of the use cases of the bank system produces the following noun phrases (candidate classes-maybe).

Account	ATM Card	Card
Account Balance	ATM Machine	Cash
Amount Approval	Bank	Check
Process	Bank Client	Checking

It is safe to eliminate the irrelevant classes. The candidate classes must be selected from relevant and fuzzy classes. The following irrelevant classes can be eliminated because they do not belong to the problem statement: Envelope, Four Digits, and Step. Strikeouts indicate eliminated classes.

Account	Account	Approval	Process	ATM Machine
Balance	Amount	ATM Card		Bank . Bank Client

Reviewing the Redundant Classes and Building a Common Vocabulary

We need to review the candidate list to see which classes are redundant. If different words are being used to describe the same idea, we must select the one that is the most meaningful in the context of the system and eliminate the others. The following are the different class names that are being used to refer to the same concept:

Client, Bank Client Account, Client's Account PIN, PIN Code

Checking, Checking Account = Bank Client (the term chosen)

Checking Account = Account

Checking Account = PIN

Checking Account = Checking Account

Savings, Savings Account = Savings

Reviewing the Classes Containing Adjectives

By again review the remaining list, now with an eye on classes with adjectives. The main question is this: Does the object represented by the noun behave differently when the adjective is applied to it?

However (it is a different use of the same object or the class is irrelevant, we must eliminate it)

Reviewing the Possible Attributes

The next review focuses on identifying the noun phrases that are attributes, not classes. The noun phrases used only as values should be restated as attributes. This process also will help us identify the attributes of the classes in the system.

Balance: An attribute of the Account class. *Invalid PIN*: It is only a value, not a class. *Password*: An attribute, possibly of the Bank Client class.

Transaction History: An attribute, possibly of the Transaction class. *PIN*: An attribute, possibly of the BankClientclass

Reviewing the Class Purpose

Identifying the classes that play role in achieving system goals and requirements is a major activity of object-oriented analysis) each class must have a purpose. Every class should be clearly defined and necessary in the context of achieving the system's goals The candidate classes are these:

ATM Machine class: Provides an interface to the ViaNet bank.

ATMCard class: Provides a client with a key to an account.

Bank Client class: A client is an individual that has a checking account and, possibly, a savings account.

Bank class: Bank clients belong to the Bank. It is a repository of accounts and processes the accounts' transactions.

The major problem with the noun phrase approach is that it depends on the completeness and correctness of the available document, which is rare in real life. On the other hand, large volumes of text on system documentation might lead to too many candidate classes.

The process of creating sequence or collaboration diagrams is a systematic way to think about how a use case (scenario) can take place; and by doing so, it forces you to think about objects involved in your application

Implementation of Scenarios

This process helps us to understand the behavior of the system's objects. When you have arrived at the lowest use-case level, you may create a child sequence diagram or accompanying collaboration diagram for the use case. With the sequence and collaboration diagrams, you can model the implementation of the scenario.

The Vianet Bank ATM System: Decomposing

Scenario with a Sequence Diagram: Object Behavior Analysis A sequence diagram represents the sequence and interactions of a given use case or scenario. The event line represents a message sent from one object to another, in which the "from" object is requesting an operation be performed by the "to" object. The "to" object performs the operation using a method that its class contains.

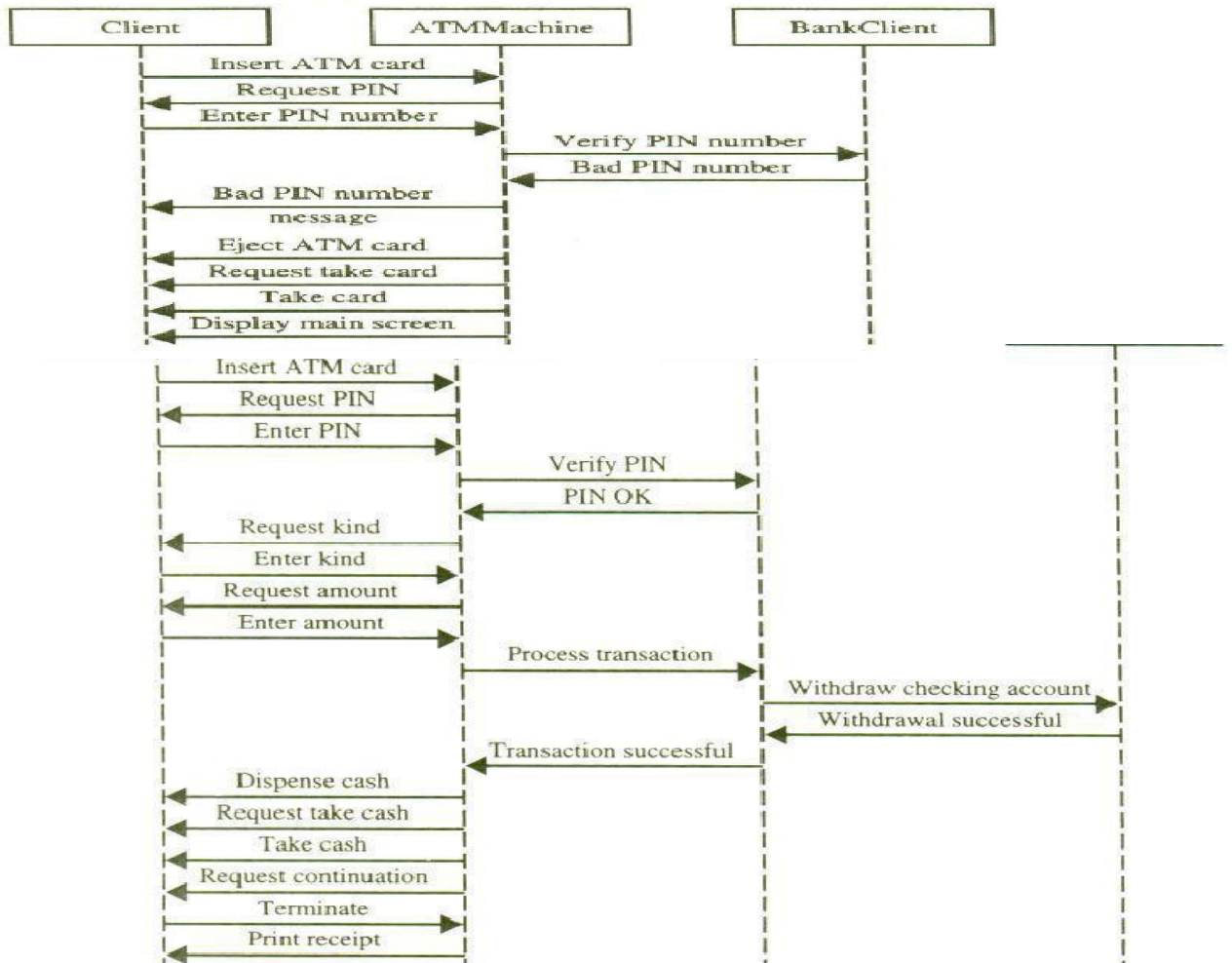
We identified the use cases for the bank system. The following are the low level (executable) use cases:

- | | | |
|-------------------|--------------------|---------------------|
| Deposit Checking | Withdraw More from | Denied Checking |
| Deposit Savings | Checking Withdraw | Transaction History |
| Invalid PIN | Savings | Savings Transaction |
| Withdraw Checking | Withdraw Savings | History |

Point of caution: you should defer the interfaces classes to the design phase and concentrate on the identifying business classes here. Consider how we would prepare a sequence diagram for the Invalid PIN use case.

The client in this case is whoever tries to access an account through the ATM, and major may not have an account. The Bank Client on the other hand has an account.

The sequence diagram for the Invalid PIN use case.



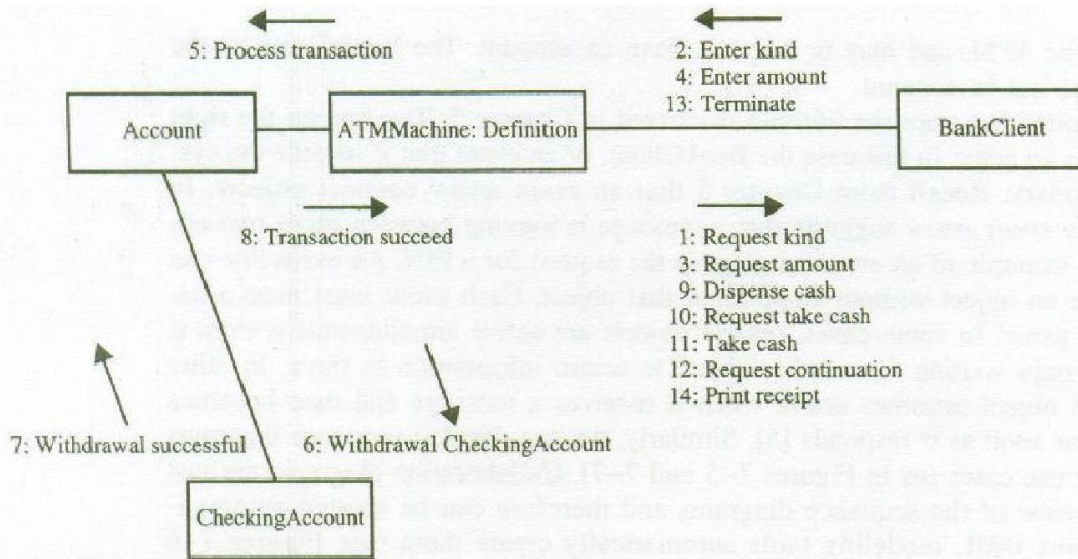
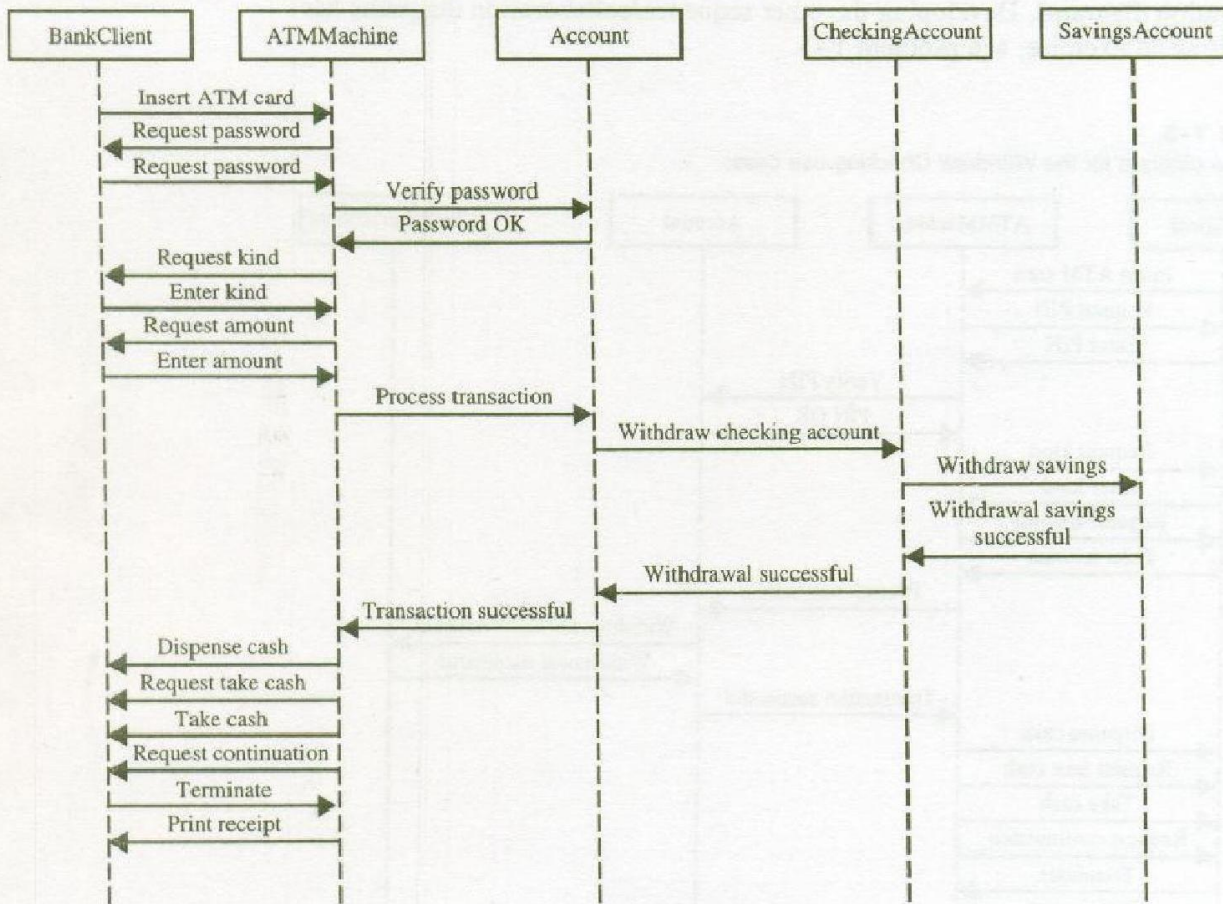
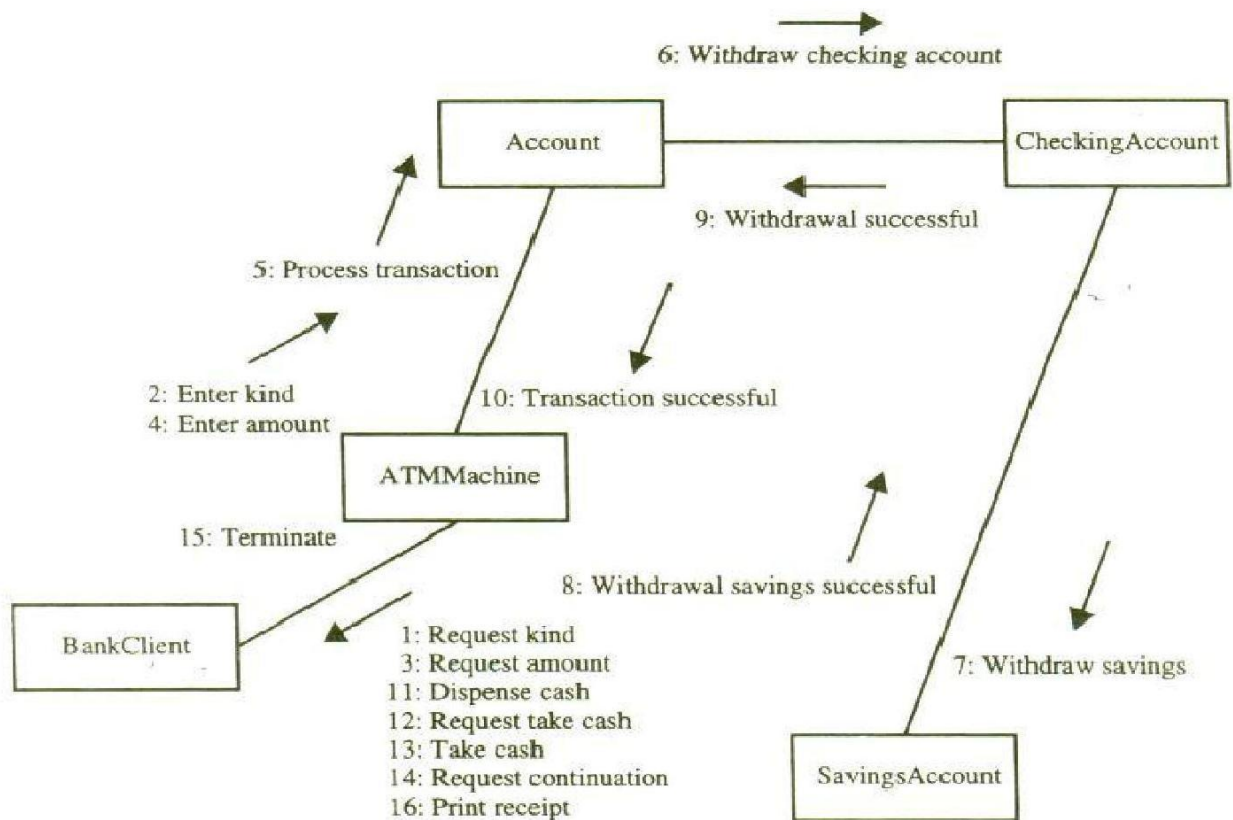


FIGURE 7-6
The collaboration diagram for the Withdraw Checking use case.

FIGURE 7-7
The sequence diagram for the Withdraw More from Checking use case.





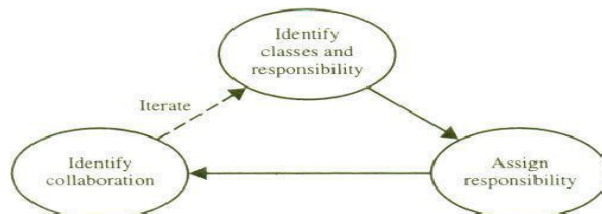
CLASSES, RESPONSIBILITIES, AND COLLABORATORS (CRC)

Classes, Responsibilities, and Collaborators is a technique used for identifying classes' responsibilities and therefore their attributes and methods. By identifying an object's responsibilities and collaborators (cooperative objects with which it works) you can identify its attributes and methods. CRC cards are 4" X 6" index cards.



Fig. A Classes, Responsibilities, and Collaborators (CRC) index card. **CRC PROCESS** The Classes, Responsibilities, and Collaborators process consists of three steps

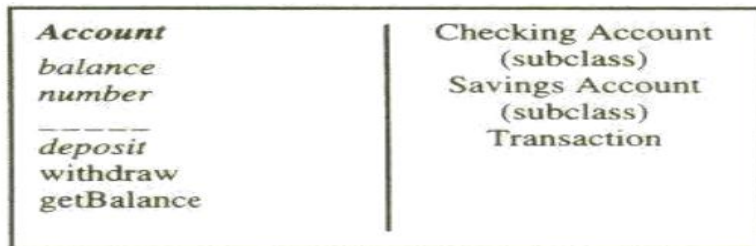
1. Identify classes' responsibilities (and identify classes).
2. Assign responsibilities.
3. Identify collaborators.



The Classes, Responsibilities, and Collaborators process.

As cards are written for familiar objects, all participants pick up the same context and ready themselves for decision making. Then, by waving cards and pointing fingers and yelling statements like, "no, this guy should do that," decisions are made. Finally, the group starts to relax as consensus has been reached and the issue becomes simply finding the right words to record a decision as a responsibility on a card.

Classes, Responsibilities, and Collaborators for the Account object.



This process is iterative. Start with few cards (classes) then proceed to play "what if." If the situation calls for a responsibility not already covered by one of the objects, either add the responsibility to an object or create a new object to address that responsibility. If one of the objects becomes too cluttered during this process, copy the information on

Analyzing relationships among classes.

Identifying association.

Association patterns.

Identifying super- and subclass hierarchies.

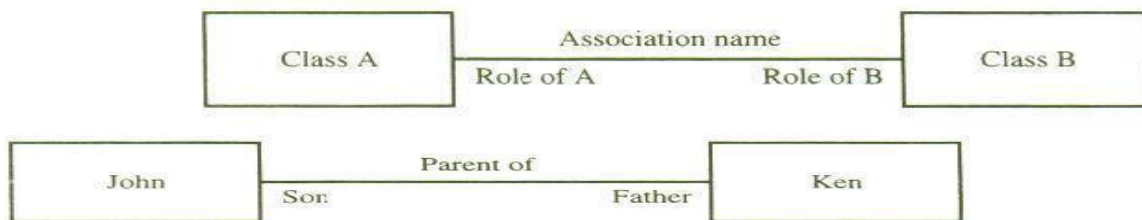
Identifying aggregation or a-part-of compositions.

Class responsibilities.

Identifying attributes and methods by analyzing use cases and other UML diagrams.

ASSOCIATIONS Association represents a physical or conceptual connection between two or more objects) For example, if an object has the responsibility for telling another object that a credit card number is valid or invalid, the two classes have an association.

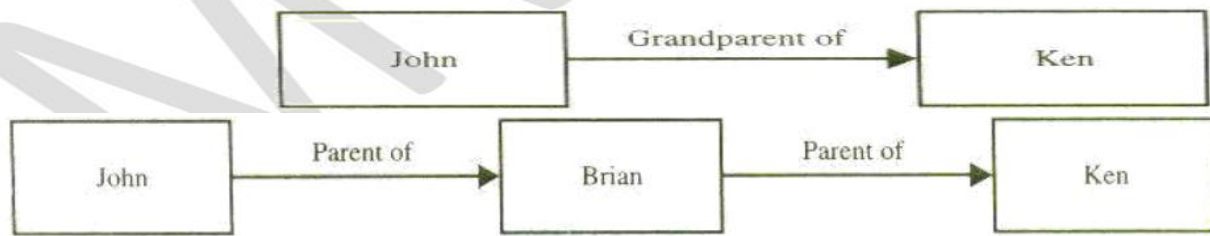
Basic association. See Chapter 5 for a detailed discussion of association.



Guideline for Identifying Association A dependency between two or more classes may be an association. Association often corresponds to a verb or prepositional phrase, such as part of, next to, works for, or contained in. A reference from one class to another is an association. Some associations are implicit or taken from general knowledge.

Common Association Patterns Communication association talk to, order to. For example, a customer places an order (communication association)with an operator person

Grandparent of Ken can be defined in terms of the parent association.



These association patterns and similar ones can be stored in the repository and added to as more patterns are discovered

Ternary associations. Ternary or n-ary association is an association among more than two classes . Ternary associations complicate the representation. When possible, restate ternary associations as binary associations **Directed actions (or derived) association.** Directed actions (derived) associations can be defined in terms of other associations. Since they are redundant, avoid these types of association. For example, Grandparent of can be defined in terms of the parent of association (see Figure). Choose association names carefully

DESIGN AXIOMS

By definition, an *axiom* is a fundamental truth that always is observed to be valid and for which there is no counterexample or exception. A *theorem* is a proposition that may not be self-evident but can be proven from accepted axioms.

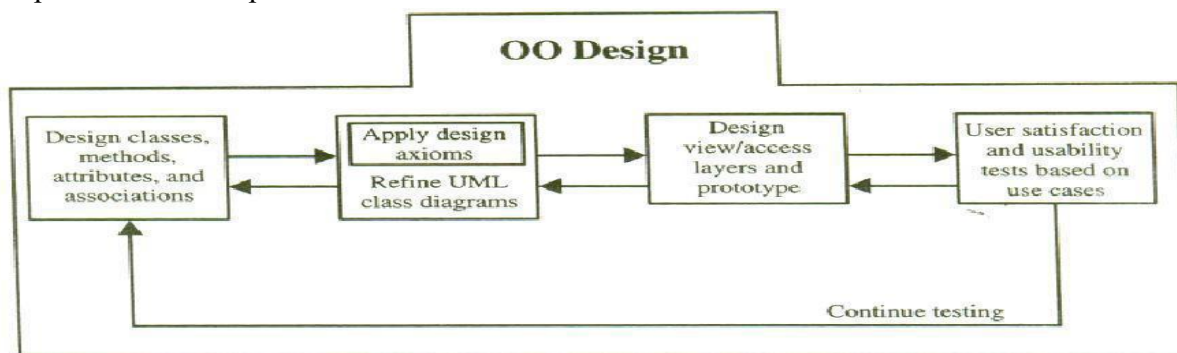
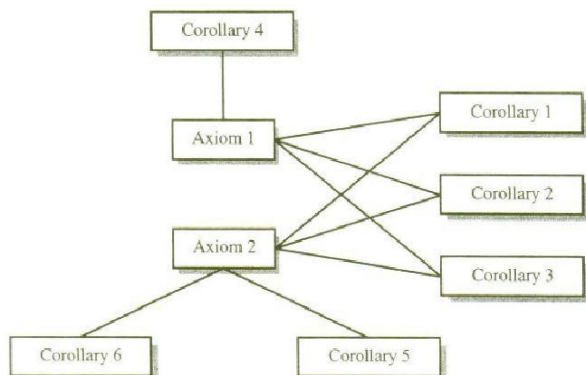


Fig:The object-oriented design process in the unified approach. Axiom 1 deals with relationships between system components (such as classes, requirements, and software components), and Axiom 2 deals with the complexity of design. Axiom 1. *The independence axiom.* Maintain the independence of components. Axiom 2. *The information axiom.* Minimize the information content of the design. Axiom 1 states that, during the design process, as we go from requirement and use case to a system component, each component must satisfy that requirement without affecting other requirements. Axiom 2 is concerned with simplicity. Occam's razor says that, "The best theory explains the known facts with a minimum amount of complexity and maximum simplicity and straightforwardness."

COROLLARIES AND ITS RELATIONSHIP WITH THE TWO AXIOMS



From the two design axioms, many corollaries may be derived as a direct consequence of the axioms. These corollaries may be more useful in making specific design decisions, since they can be applied to actual situations more easily than the original axioms. They even may be called *design rules*, and all are derived from the two basic axioms

The origin of corollaries. Corollaries 1, 2, and 3 are from both axioms, whereas corollary4 is from axiom 1 and corollaries 5 and 6 are from axiom 2. **Corollary 1. *Uncoupled design with less information content.*** Highly cohesive objects can improve coupling because only a minimal amount of essential information need be passed between objects.

Corollary 2. *Single purpose.* Each class must have a single, clearly defined purpose. When you document, you should be able to easily describe the purpose of a class in a few sentences.

Corollary 3. *Large number of simple classes.* Keeping the classes simple allows reusability.

Corollary 4. *Strong mapping.* There must be a strong association between the physical system (analysis's object) and logical design (design's object).

Corollary 5. *Standardization.* Promote standardization by designing interchangeable components and reusing existing classes or components.

Corollary 6. *Design with inheritance.* Common behavior (methods) must be moved to super classes. The super class-subclass structure must make logical sense.

Corollary 1. *Uncoupled Design with Less Information Content* The main goal here is to maximize objects cohesiveness among objects and software components in order to improve coupling because only a minimal amount of essential information need be passed between components. *Coupling* is a measure of the strength of association established by a connection from one object or software component to another.

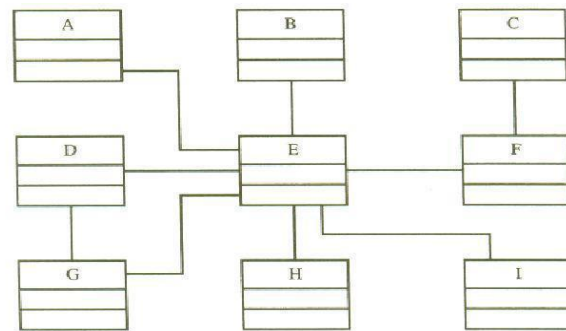
Coupling is a binary relationship: A is coupled with B. The degree of coupling is a function of

1. How complicated the connection is.
2. Whether the connection refers to the object itself or something inside it.
3. What is being sent or received.

Coupling increases (becomes stronger) with increasing complexity or obscurity of the interface.

Coupling decreases (becomes lower) when the connection is to the component interface rather than to an internal component

Table contains different types of interaction couplings. Inheritance is a form of coupling between super- and subclasses. A subclass is coupled to its superclass in terms of attributes and methods. Unlike interaction coupling, high inheritance coupling is desirable



TYPES OF COUPLING AMONG OBJECTS OR COMPONENTS (shown from highest to lowest)

Degree of coupling	Name	Description
Very high	Content coupling	The connection involves direct reference to attributes or methods of another object.
High	Common coupling	The connection involves two objects accessing a "global data space," for both to read and write.
Medium	Control coupling	The connection involves explicit control of the processing logic of one object by another.
Low	Stamp coupling	The connection involves passing an aggregate data structure to another object, which uses only a portion of the components of the data structure.
Very low	Data coupling	The connection involves either simple data items or aggregate structures all of whose elements are used by the receiving object. This should be the goal of an architectural design.

Cohesion Coupling deals with interactions between objects or software components. We also need to consider interactions within a single object or software component, called *cohesion*. Cohesion reflects the "single-purposeness" of an object. Highly cohesive components can lower coupling because only a minimum of essential information need be passed between components. Cohesion also helps in designing classes that have very specific goals and clearly defined purposes. Method cohesion, like function cohesion, means that a method should carry only one function. **Corollary 2. Single Purpose** Every class should be clearly defined and necessary in the context of achieving the system's goals. When you document a class, you should be able to easily explain its purpose in a sentence or two. If you cannot, then rethink the class and try to subdivide it into more independent pieces. In summary, keep it simple; to be more precise, each method must provide only one service. Each method should be of moderate size, no more than a page; half a page is better.

Corollary 3. Large Number of Simpler Classes, Reusability The less specialized the classes are, the more likely future problems can be solved by a recombination of existing classes, adding a minimal number of subclasses. A class that easily can be understood and reused (or inherited) contributes to the overall system, while a complex, poorly designed class is just so much dead weight and usually cannot be reused. Coad and Yourdon describe four reasons why people are not utilizing this concept:

1. Software engineering textbooks teach new practitioners to build systems from "first principles"; reusability is not promoted or even discussed.
2. The "not invented here" syndrome and the intellectual challenge of solving an interesting software problem in one's own unique way mitigates against reusing someone else's software component.
3. Unsuccessful experiences with software reusability in the past have convinced many practitioners and development managers that the concept is not practical.
4. Most organizations provide no reward for reusability; sometimes productivity is measured in terms of new lines of code written plus a discounted credit (e.g., 50 percent less credit) for reused lines of code.

Griss argues that, although reuse is widely desired and often the benefit of utilizing object technology, many object-oriented reuse efforts fail because of too narrow a focus on technology and not on the policies set forth by an organization. He recommended an institutionalized approach to software development, in which software assets intentionally are created or acquired to be reusable.

Corollary 4. Strong Mapping During the design phase, we need to design this class design its methods, its association with other objects, and its view and access classes. A strong mapping links classes identified during analysis and classes designed during the design phase (e.g., view and access classes). With OO techniques, the same paradigm is used for analysis, design, and implementation. The analyst identifies objects' types and inheritance, and thinks about events that change the state of objects. The designer adds detail to this model perhaps designing screens, user interaction, and client-server interaction. The thought process flows so naturally from analyst to design that it may be difficult to tell where analysis ends and design begins.

Corollary 5. Standardization Similarly, object-oriented systems are like organic systems, meaning that they grow as you create new applications. The knowledge of existing classes will help you determine what new classes are needed to accomplish the tasks and where you might inherit useful behavior rather than reinvent the wheel. Furthermore, class libraries must be easily searched, based

on users' criteria. For example, users should be able to search the class repository with commands like "show me all Facet classes. " The concept of design patterns might provide a way to capture the design knowledge, document it, and store it in a repository that can be shared and reused in different applications.

Corollary 6. Designing with Inheritance When you implement a class, you have to determine its ancestor, what attributes it will have, and what messages it will understand. Then, you have to construct its methods and protocols. Ideally, you will choose inheritance to minimize the amount of program instructions This is a simple, easy to understand design, although somewhat limited in the reusability of the classes. For example, if in another project you must build a system that models a vehicle assembly plant, the classes from the licensing application are not appropriate, since these classes have instructions and data that deal with the legal requirements of motor vehicle license acquisition and renewal.

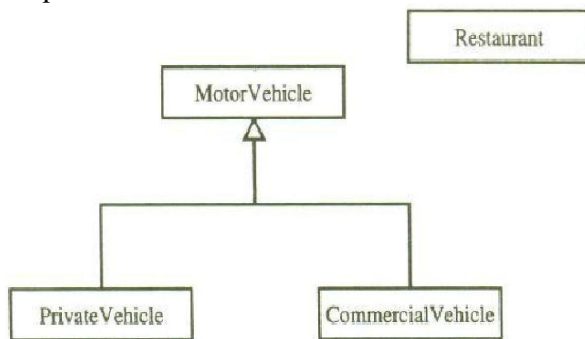
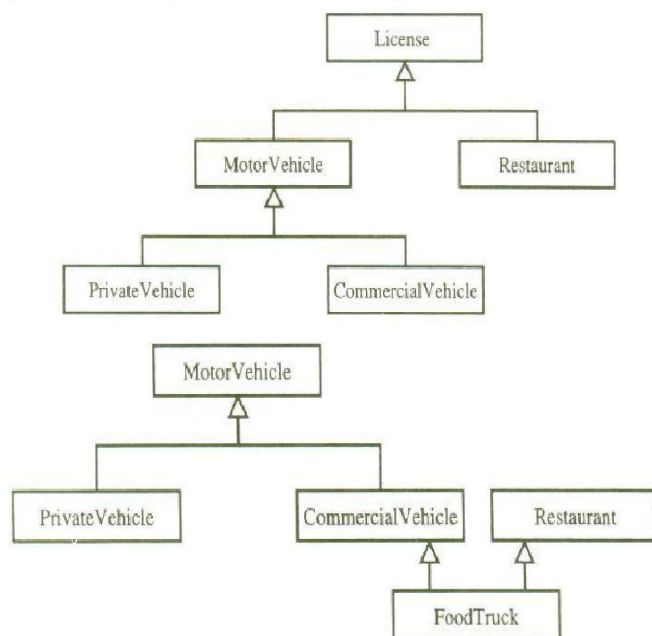


Fig: The initial single inheritance design.

Achieving Multiple Inheritance in a Single Inheritance System *Single inheritance* means that each class has only a single superclass. This technique is used in Smalltalk and several other object-oriented systems. One result of using a single inheritance hierarchy is the absence of ambiguity as to how an object will respond to a given method.

Explain relationship analysis for the ATM banking system. To better gain experience in object relationship analysis, we use the familiar bank system case and apply the concepts for identifying associations, super sub relationships, and a-part-of relationships for the classes identified. Furthermore, object-oriented analysis and design are performed in an iterative process using class diagrams This iterative process is unlike the traditional waterfall technique, in which all analysis is completed before design begins

The single inheritance design modified to allow licensing food trucks.

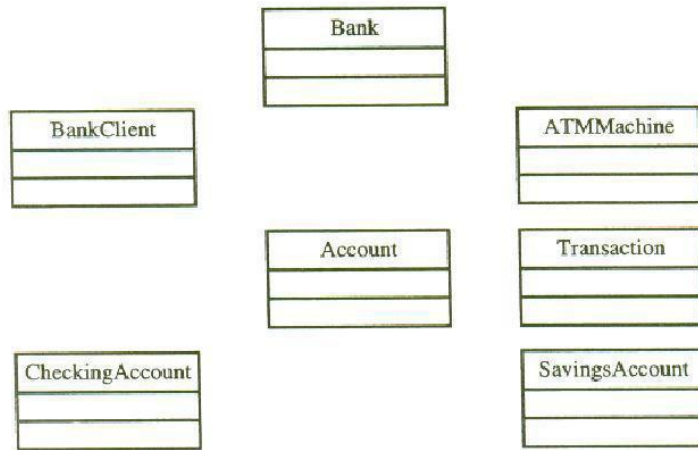


Identifying Classes' Relationships

One of the strengths of object-oriented analysis is the ability to model objects as they exist in the real world. To accurately do this, you must be able to model more than just an object's internal workings. You also must be able to model how objects relate to each other. Several different relationships exist in the ViaNet bank ATM system, so we need to define them

Developing a UML Class Diagram Based on the Use-Case Analysis

The UML class diagram is the main static analysis and design diagram of a system. The analysis generally consists of the following class diagrams. One class diagram for the system, which shows the identity and definition of classes in the system, their interrelationships, and various packages containing groupings of classes.

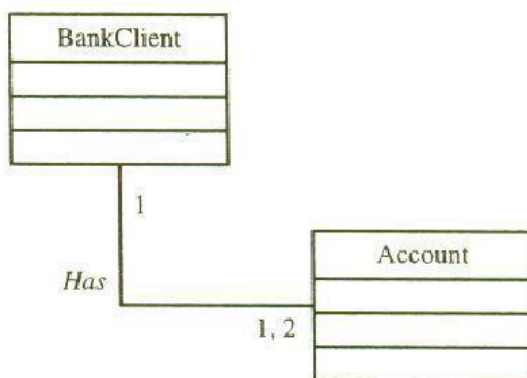


UML class diagram for the ViaNet bank ATM system. Some CASE tools such as the SA/Object Architect can automatically define classes and draw them from use cases or collaboration/sequence diagrams. However, presently, it cannot identify all the classes. For this example, S/A Object was able to identify only the BankClient class. Multiple class diagrams that represent various pieces, or views, of the system class diagram. Multiple class diagrams,

that show the specific static relationships between various classes.

Defining Association Relationships

Identifying association begins by analyzing the interactions of each class. Remember that any dependency between two or more classes is an association. The following are general guidelines for



identifying the tentative associations, as explained in this chapter: .Association often corresponds to verb or prepositional phrases, such as part of, next to, works for, or contained in. A reference from one class to another is an association. Some associations are implicit or taken from general knowledge.

Other associations and their cardinalities are defined in Table 8-1 and demonstrated in Figure

SOME ASSOCIATIONS AND THEIR CARDINALITIES IN THE BANK SYSTEM

Class	Related class	Association name	Cardinality
Account	BankClient	Has	One
BankClient	Account		One or two
SavingsAccount	CheckingAccount	Savings-Checking	One
CheckingAccount	SavingsAccount		Zero or one
Account	Transaction	Account-Transaction	Zero or more
Transaction	Account		One

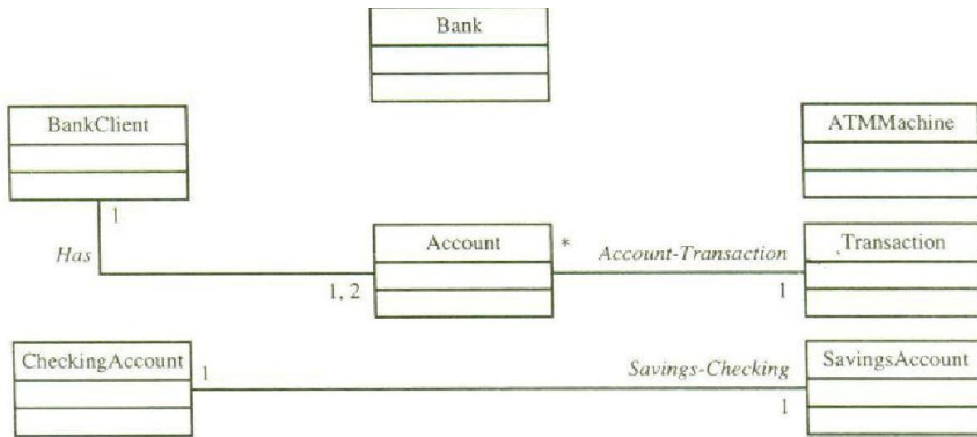
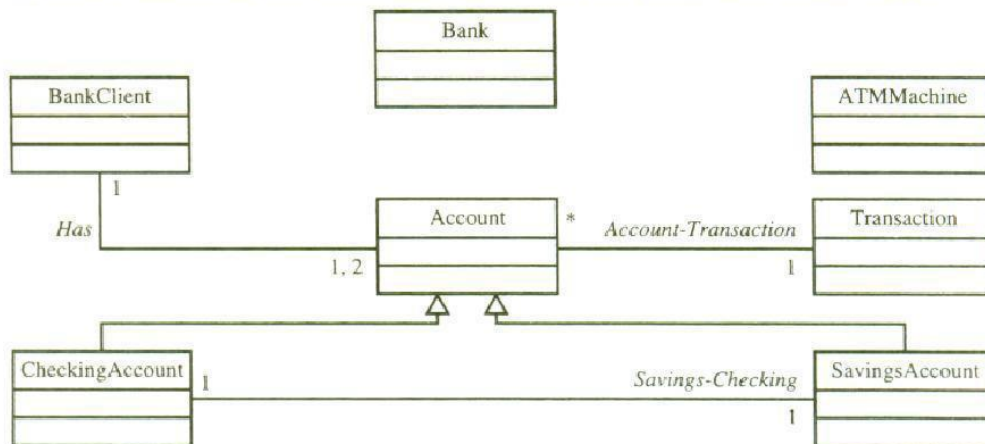


FIG: Associations among the ViaNet bank ATM system classes.

Identifying the Aggregation/a-Part-of Relationship To identify a-part-of structures, we look for the following clues: . Assembly. A physical whole is constructed from physical parts. . Container. A physical whole encompasses but is not constructed from physical parts. . Collection-Member. A conceptual whole encompasses parts that may be physical or conceptual.

Super-sub relationships among the Account, SavingsAccount, and CheckingAccount classes.



Association, generalization, and aggregation among the ViaNet bank classes. Notice that the super-sub arrows for CheckingAccount and SavingsAccount have merged. The relationship between BankClient and ATMMachine is an interface.

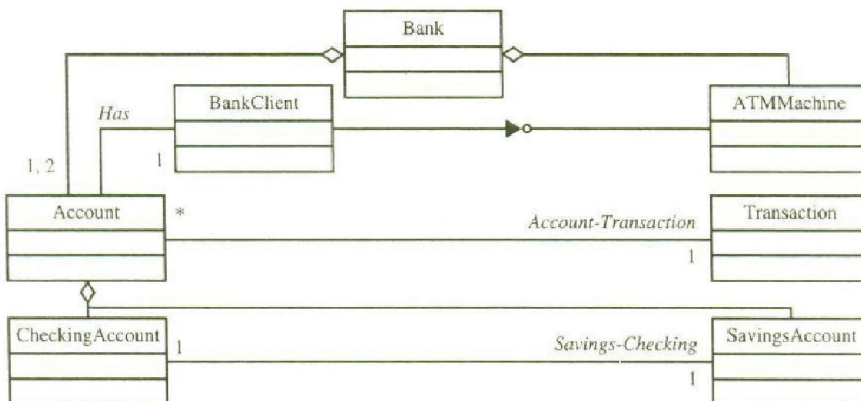


Figure depicts the association, generalization, and aggregation among the bank systems classes. If you are wondering what the relationship between the Bank Client and ATMMachine is, it is an interface. Identifying a class interface is a design activity of object-oriented system development.

UNIT – V

Design Patterns: Design Patterns – Describing design patterns - catalog of design patterns – organizing the catalog – How design patterns solve design problems – How to select a design pattern – How to use a design pattern –
creational pattern: Abstract factory – structural pattern: Adapter – behavioral pattern: chain of responsibility.

5. Briefly discuss about the scope criteria in design patterns.

The second criterion in design patterns, called Scope, which specifies whether the pattern applies primarily to classes or to objects:

- Class patterns deal with relationships between classes and their subclasses. These relationships are established through inheritance, so they are static.
- Object patterns deal with object relationships, which can be changed at run-time and are more dynamic.

6. What is meant by abstract class?

An **abstract class** is one whose main purpose is to define a common interface for its subclasses. Classes that are not abstract are called concrete classes.

7. What is meant by delegation?

Delegation is a way of making composition as powerful for reuse as inheritance. In delegation, *two* objects are involved in handling a request: a receiving object delegates operation to its **delegate**. The receiver passes itself to the delegate to let the delegated operation refer to the receiver.

8. Define toolkit.

A toolkit is a set of related and reusable classes designed to provide useful, general-purpose functionality. An example of a toolkit is a set of collection classes for lists, associative tables, stacks and etc.

9. Define framework

A framework is a set of cooperating classes that make up a reusable design for a specific class of software. For example, a framework can be geared toward building graphical editors for different domains like artistic drawing, music composition, and mechanical CAD.

10. What is meant by creational design pattern

Creational design patterns are all about class instantiation. It deals with system independence from object composition, creation and representation. Creational design patterns can be further divided into class-creational patterns and object-creational patterns.

- **Class Creational Pattern** uses inheritance to vary the class that's instantiated. Factory Method is an example of class-creational Pattern.
- **Object-creational pattern** will delegate instantiation to another object. Abstract Factory, Builder, Prototype and Singleton belong to this category.

11. Describe Abstract Factory

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

12. Define Adapter

Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

13. Define Chain of Responsibility

Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.

14. Discuss the consequences of abstract factory pattern

The Abstract Factory pattern has the following consequences:

- Isolates concrete classes by helping programmers to control the classes of objects that an application creates.

- Makes exchanging product families easy because the class of a concrete factory appears only once in an application, at the place of its instantiation. This makes it easy to change the concrete factory that an application uses.
- Promotes consistency among products by enforcing to use objects from the same family of objects.
- Supporting new kinds (in each family) of products is difficult since we need not only to define new product objects but also to extend all factories to be able to create those new product objects.

15. When to use abstract factory pattern

The Abstract Factory pattern can be used when

- A system should be independent of how its products are created, composed, and represented.
- A system should be configured with one of multiple families of products.
- A family of related product objects is designed to be used together, and you need to enforce this constraint.
- you want to provide a class library of products, and you want to reveal just their interfaces, not their implementations.

16. Describe briefly about structural patterns

Structural patterns are concerned with how classes and objects are composed to form larger structures. Structural class patterns use inheritance to compose interfaces or implementations. As a simple example, consider how multiple inheritances mix two or more classes into one. The result is a class that combines the properties of its parent classes.

17. What is meant by behavioral patterns

Behavioral patterns are concerned with algorithms and the assignment of responsibilities between objects. Behavioral patterns describe not just patterns of objects or classes but also the patterns of communication between them. These patterns characterize complex control flow that's difficult to follow at run-time. Behavioral class patterns use inheritance to distribute behavior between classes.

18. What are the benefits of chain of responsibility

Chain of Responsibility has the following benefits

1. *Reduced coupling.* The pattern frees an object from knowing which other object handles a request. An object only has to know that a request will be handled "appropriately." Both the receiver and the sender have no explicit knowledge of each other, and an object in the chain doesn't have to know about the chain's structure.
2. *Added flexibility in assigning responsibilities to objects.* Chain of Responsibility gives you added flexibility in distributing responsibilities among objects. You can add or change responsibilities for handling a request by adding to or otherwise changing the chain at run-time. You can combine this with subclassing to specialize handlers statically.

19. What is the disadvantage of chain of responsibility

Since a request has no explicit receiver, there's no guarantee it'll be handled—the request can fall off the end of the chain without ever being handled. A request can also go unhandled when the chain is not configured properly.

20. How will you Select a Design Pattern

1. Consider how design patterns solve design problems.
2. Scan Intent sections.
3. Study how patterns interrelate.
4. Study patterns of like purpose.
5. Examine a cause of redesign.
6. Consider what should be variable in your design.

21. How to Use a Design Pattern

1. Read the pattern once through for an overview.
2. Go back and study the Structure, Participants, and Collaborations sections.
3. Look at the Sample Code section to see a concrete example of the pattern in code.
4. Choose names for pattern participants that are meaningful in the application context.
5. Define the classes.
6. Define application-specific names for operations in the pattern.
7. Implement the operations to carry out the responsibilities and collaborations in the pattern.

22. What are the different techniques used for implementing abstract factory pattern

Some useful techniques for implementing the Abstract Factory pattern are

- Factories as singletons.
- Creating the products.
- Defining extensible factories.

23. When can the adapter pattern be used

The Adapter pattern can be used when

- We want to use an existing class, and its interface does not match the one you need.
- We want to create a reusable class that cooperates with unrelated or unforeseen classes, that is, classes that don't necessarily have compatible interfaces.
- (object adapter only) We need to use several existing subclasses, but it's impractical to adapt their interface by subclassing every one. An object adapter can adapt the interface of its parent class.

24. What is a class adapter

A class adapter

- Adapts Adaptee to Target by committing to a concrete Adapter class. As a consequence, a class adapter won't work when we want to adapt a class and all its subclasses.
- Lets Adapter override some of Adaptee's behavior, since Adapter is a subclass of Adaptee.
- Introduces only one object, and no additional pointer indirection is needed to get to the adaptee.

25. What is an object adapter

An object adapter

- Lets a single Adapter work with many Adaptees—that is, the Adaptee itself and all of its subclasses (if any). The Adapter can also add functionality to all Adaptees at once.
- Makes it harder to override Adaptee behavior. It will require subclassing Adaptee and making Adapter refer to the subclass rather than the Adaptee itself.

26. What are the issues in implementing adapter.

Some of the issues in implementing adapter are

1. Implementing class adapters in C++. In a C++ implementation of a class adapter, Adapter would inherit publicly from Target and privately from Adaptee. Thus Adapter would be a subtype of Target but not of Adaptee.
2. Pluggable adapters. Let's look at three ways to implement pluggable adapters for the TreeDisplay widget described earlier, which can lay out and display a hierarchical structure automatically.

27. What are the implementation approaches of narrow interface

The narrow interface leads to three implementation approaches:

- Using abstract operations.
- Using delegate objects.
- Parameterized adapters.

28. When to Use Chain of Responsibility

Use Chain of Responsibility when

- More than one object may handle a request, and the handler isn't known a priori. The handler should be ascertained automatically.
- You want to issue a request to one of several objects without specifying the receiver explicitly.
- The set of objects that can handle a request should be specified dynamically.

29. What are the implementation issues in Chain of Responsibility

Here are implementation issues to consider in Chain of Responsibility:

1. Implementing the successor chain.
2. Connecting successors.
3. Representing requests.
4. Automatic forwarding in Smalltalk.

30. What is meant by Pluggable adapters.

A class is more reusable when you minimize the assumptions other classes must make to use it. By building interface adaptation into a class, you eliminate the assumption that other classes see the same interface.

11 Marks

1. What is Design Pattern? Explain its elements.

Design patterns make it easier to reuse successful designs and architectures. Expressing proven techniques as design patterns makes them more accessible to developers of new systems.

Design patterns help you choose design alternatives that make a system reusable and avoid alternatives that compromise reusability.

Design patterns can even improve the documentation and maintenance of existing systems by furnishing an explicit specification of class and object interactions and their underlying intent. Put simply, design patterns help a designer get a design –right|| faster.

Design Patterns are the best solutions for the re-occurring problems in the application programming environment. The solutions are expressed in terms of objects and interfaces.

A pattern has four essential elements:

1. The **pattern name** is a handle we can use to describe a design problem, its solutions, and consequences in a word or two. Naming a pattern immediately increases our design vocabulary. It lets us design at a higher level of abstraction. It makes it easier to think about designs and to communicate them and their trade-offs to others. Finding good names has been one of the hardest parts of developing our catalog.

2. The **problem** describes when to apply the pattern. It explains the problem and its context. It might describe specific design problems such as how to represent algorithms as objects. It might describe class or object structures that are symptomatic of an inflexible design. Sometimes the problem will include a list of conditions that must be met before it makes sense to apply the pattern.

3. The **solution** describes the elements that make up the design, their relationships, responsibilities, and collaborations. The solution doesn't describe a particular concrete design or implementation, because a pattern is like a template that can be applied in many different situations. Instead, the pattern provides an abstract description of a design problem and how a general arrangement of elements (classes and objects in our case) solves it.

4. The **consequences** are the results and trade-offs of applying the pattern. Though consequences are often unvoiced when we describe design decisions, they are critical for evaluating design alternatives and

for understanding the costs and benefits of applying the pattern. The consequences for software often concern space and time trade-offs. They may address language and implementation issues as well. Since reuse is often a factor in object-oriented design, the consequences of a pattern include its impact on a system's flexibility, extensibility, or portability. Listing these consequences explicitly helps you understand and evaluate them.

The design patterns are *descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context.*

A design pattern names, abstracts, and identifies the key aspects of a common design structure that make it useful for creating a reusable object-oriented design.

The design pattern identifies the participating classes and their instances, their roles and collaborations, and the distribution of responsibilities.

Each design pattern focuses on a particular object-oriented design problem or issue. It describes when it applies, whether or not it can be applied in view of other design constraints, and the consequences and trade-offs of its use.

2. Explain briefly Describing Design patterns.

Design Patterns are described in Graphical notations, while important and useful, aren't sufficient. They simply capture the end product of the design process as relationships between classes and objects. To reuse the design, we must also record the decisions, alternatives, and trade-offs that led to it. Concrete examples are important too, because they help you see the design in action. The Gang of Four (GoF) used a consistent format to describe patterns. They developed a template for describing a design pattern. The template lent a uniform structure to the information and made design patterns easier to learn, compare and use. This template describes a design pattern with:

Pattern Name and Classification

It conveys the essence of the pattern succinctly good name is vital, because it will become part of design vocabulary.

Intent

- What does the design pattern do?
- What is its rationale and intent?
- What particular design issue or problem does it address?

Also Known As

Other well-known names for the pattern, if any.

Motivation

A scenario that illustrates a design problem and how the class and object structures in the pattern solve the problem. The scenario will help you to understand the more abstract description of the pattern.

Applicability

- What are the situations in which the design patterns can be applied?
- What are examples of the poor designs that the pattern can address?
- How can recognize situations?

Structure

Graphical representation of the classes in the pattern using a notation based on the object Modeling Technique(OMT).

Participants

The classes and/or objects participating in the design pattern and their responsibilities.

Collaborations

How the participants collaborate to carry out their responsibilities.

Consequences

- How does the pattern support its objectives?
- What are the trade-offs and result of using the pattern?
- What aspect of the system structure does it let vary independently?

Implementation

What pitfalls, hints, or techniques should be aware of when implementing the pattern ?

Sample Code

Code fragments that illustrate how might implement the pattern in c++ or Smalltalk.

Known Uses

Examples of the pattern found in real systems.

Related Patterns

What design patterns are closely related to this one? What are the important differences?

With Which other patterns should this one be used?

3. Explain the Catalog of Design patterns.

Design Pattern Name	Description
Abstract Factory	Provide an interface for creating families of related or dependent objects without specifying their concrete classes.
Adapter	Convert the interface of a class into another interface clients expect.
Bridge	Decouple an abstraction from its implementation so that two can vary independently.
Builder	Separate the construction of a complex object from its representation so that the same construction process can create different representations.
Chain of Responsibility	Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.
Command	Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.
Composite	Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.
Decorator	Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to sub-classing for extending functionality.
Facade	Provide a unified interface to a set of interfaces in a subsystem. Façade defines a higher-level interface that makes the subsystem easier to use.

Factory Method	Define an interface for creating an object, but let subclasses decide which class to instantiate.
Flyweight	Use sharing to support large numbers of fine-grained objects efficiently.
Interpreter	Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.
Iterator	Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.
Mediator	Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.
Memento	Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.
Observer	Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
Prototype	Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype
Proxy	Provide a surrogate or placeholder for another object to control access to it.
Singleton	Ensure a class only has one instance, and provide a global point of access to it.
State	Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.
Strategy	Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.
Template Method	Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.
Visitor	Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.

4. How to organize the catalog? Explain in detail.

Design patterns vary in their granularity and level of abstraction. Because there are many design patterns, we need a way to organize them. This section classifies design patterns so that we can refer to families of related patterns. The classification helps you learn the patterns in the catalog faster, and it can direct efforts to find new patterns as well.

We classify design patterns by two criteria as shown in the table.

The *first* criterion, called purpose, reflects what a pattern does. Patterns can have either creational, structural, or behavioral purpose.

- Creational patterns concern the process of object creation.
- Structural patterns deal with the composition of classes or objects.
- Behavioral patterns characterize the ways in which classes or objects interact and distribute responsibility.

The *second* criterion, called Scope, which specifies whether the pattern applies primarily to classes or to objects:

- Class patterns deal with relationships between classes and their subclasses. These relationships are established through inheritance, so they are static.
- Object patterns deal with object relationships, which can be changed at run-time and are more dynamic.

		Purpose		
		Creational	Structural	Behavioral
Scope	Class	Factory Method	Adapter	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Facade Proxy	Chain of Responsibility Command Iterator Mediator Memento Flyweight (195) Observer State Strategy Visitor

Table: Design pattern space

Almost all patterns use inheritance to some extent. So the only patterns labeled "class patterns" are those that focus on class relationships. Note that most patterns are in the Object scope. Creational class patterns defer some part of object creation to subclasses, while Creational object patterns defer it to another object. The Structural class patterns use inheritance to compose classes, while the Structural object patterns describe ways to assemble objects. The Behavioral class patterns use inheritance to describe algorithms and flow of control, whereas the Behavioral object patterns describe how a group of objects cooperate to perform a task that no single object can carry out alone.

There are other ways to organize the patterns. Some patterns are often used together. For example, Composite is often used with Iterator or Visitor. Some patterns are alternatives: Prototype is often an alternative to Abstract Factory. Some patterns result in similar designs even though the patterns have different intents. For example, the structure diagrams of Composite and Decorator are similar. Yet another way to organize design patterns is according to how they reference each other in their "Related Patterns" sections. Figure depicts these relationships graphically. Clearly there are many ways to organize design patterns. Having multiple ways of thinking about patterns will deepen your insight into what they do, how they compare, and when to apply them.

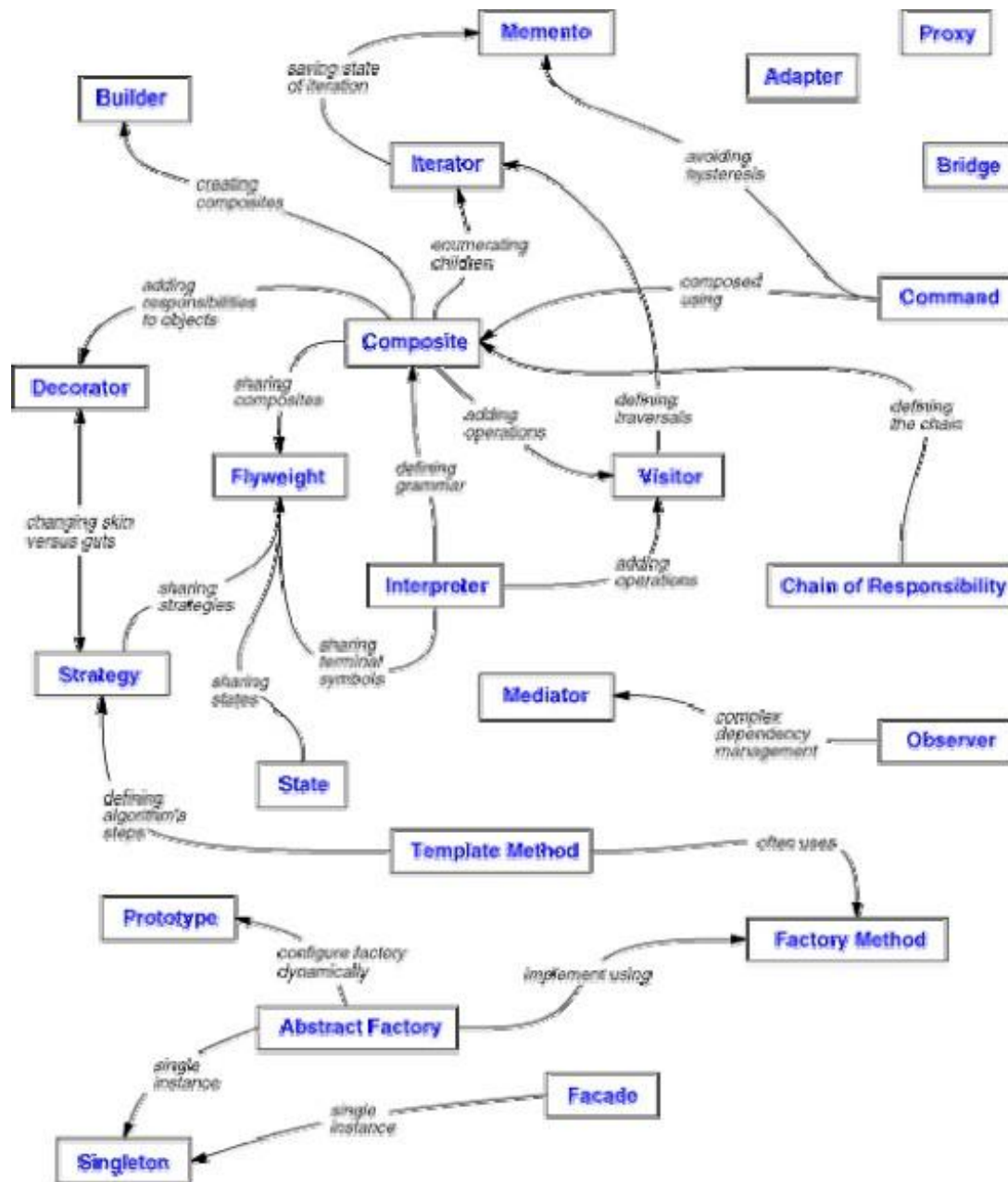


Figure 1.1: Design pattern relationships

5. How Design Patterns solve design problems?

1. Finding Appropriate Objects

- Design patterns help you identify less-obvious abstractions and the objects that can capture them.

2. Determining Object Granularity

- **Facade** pattern describes how to represent complete subsystems as objects.
- **Flyweight** pattern describes how to support huge numbers of objects at the finest granularities.
- **Abstract Factory** and **Builder** yield objects whose only responsibilities are creating other objects.
- **Visitor** and **Command** yield objects whose only responsibilities are to implement a request on another object or group of objects.

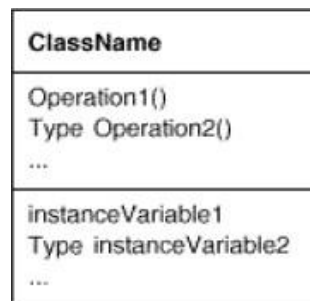
3. Specifying Object Interfaces

- The set of all signatures defined by an object's operations is called the **interface** to the object.

- A **type** is a name used to denote a particular interface.
- An object may have many types, and widely different objects can share a type.
- A type is a **subtype** of another if its interface contains the interface of its **supertype**, or a subtype *inheriting* the interface of its supertype.
- Objects are known only through their interfaces. An object's interface says nothing about its implementation
- When a request is sent to an object, the particular operation that's performed depends on *both* the request *and* the receiving object.
- **Dynamic binding**: the run-time association of a request to an object and one of its operations.
- **Polymorphism**: dynamic binding can substitute objects that have identical interfaces for each other at run-time
- Design patterns help you define interfaces by identifying their key elements and the kinds of data that get sent across an interface.
- Design patterns also specify relationships between interfaces.

4. Specifying Object Implementations

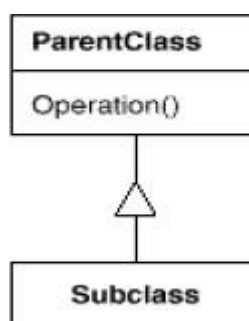
- An object's implementation is defined by its **class**.



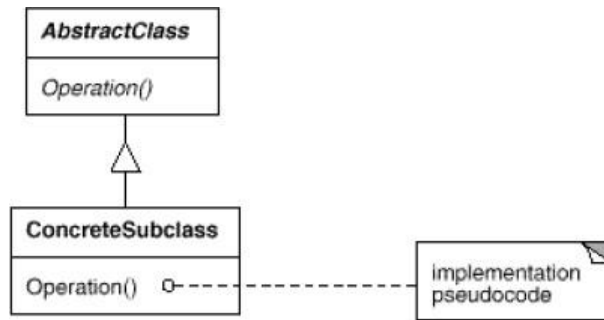
- Objects are created by instantiating a class.
- A dashed arrowhead line indicates a class that instantiates objects of another class. The arrow points to the class of the instantiated objects.



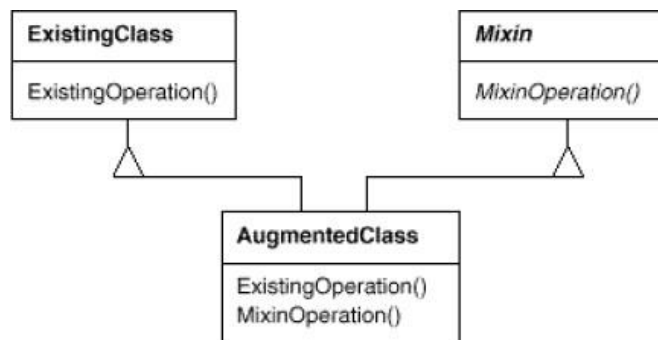
New classes can be defined in terms of existing classes using class inheritance. When a subclass inherits from a parent class, it includes the definitions of all the data and operations that the parent class defines. Objects that are instances of the subclass will contain all data defined by the subclass and its parent classes, and they'll be able to perform all operations defined by this subclass and its parents.



- An **abstract class** is one whose main purpose is to define a common interface for its subclasses.



- A **mixin class** is a class that's intended to provide an optional interface or functionality to other classes.



Class versus Interface Inheritance

- An object's class defines how the object is implemented.
- An object's type only refers to its interface.
- An object can have many types.
- Objects of different classes can have the same type.
- - Relationship between class and type: class as type (C++) vs. interface as type (Java).
- Class inheritance: Sub-typing + Implementation inheritance.
- Interface inheritance: Sub-typing only (Polymorphism).
- Pure abstract classes as interfaces.
- Many of the design patterns depend on the distinction between class and interface inheritances

First Principle of reusable object-oriented design:

Programming to an Interface, not an Implementation

- o Class inheritance-based implementation reuse is only half the story. Inheritance's ability to define families of objects with **identical interfaces** is also important, because polymorphism depends on it.
- o Two benefits to manipulating objects solely in terms of the interface defined by abstract classes:
 1. Clients remain unaware of the specific types of objects they use, as long as the objects adhere to the interface that clients expect.
 2. Clients remain unaware of the classes that implement these objects. Clients only know about the abstract class(es) defining the interface.
- o Don't declare variables to be instances of particular concrete classes.
- o Creational patterns ensure that your system is written in terms of interfaces, not implementations.

5. Putting Reuse Mechanisms to Work

Inheritance versus Composition

- **White-box reuse:** class inheritance.
- **Black-box reuse:** object composition
 - o Class inheritance:

Advantages

supported by programming languages, defined statically at compile-time and is straightforward to use.

- f* make it easier to modify the implementation being reused, when a subclass overrides some but not all operations.

Disadvantages

- f* Cannot change the implementations/representations inherited from parent classes at run-time.
- f* Implementation dependency between a subclass and its parent class.

o Object composition

Advantages

- f* Defined dynamically at run-time by referencing interfaces of objects.
- f* Access other objects through their interfaces only, not break encapsulation.
- f* Fewer implementation dependencies.
- f* Small class hierarchies.

Disadvantages

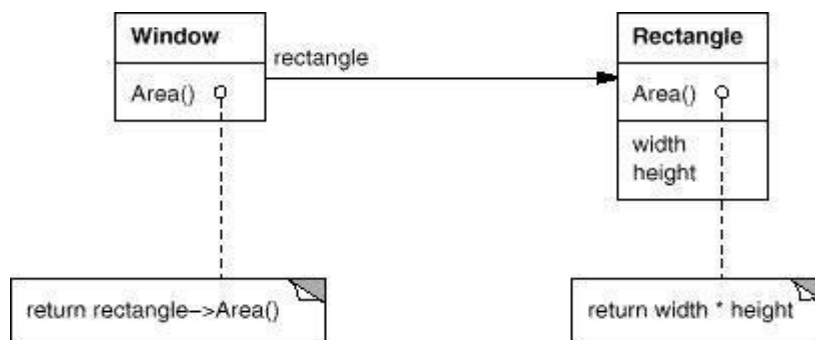
- f* More objects.
- f* The system's behavior will depend on their interrelationships instead of being defined in one class.

The second principle of object-oriented design:

Favor object composition over class inheritance.

Delegation

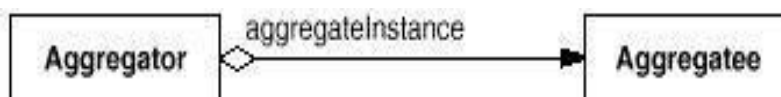
- o In delegation, *two* objects are involved in handling a request: a receiving object delegates operations to its **delegate**.
- o The receiver passes itself to the delegate to let the delegated operation refer to the receiver.



- o Advantage: it makes it easy to compose behaviors at run-time and to change the way they're composed.
- o Disadvantage: harder to understand than more static software, and run-time inefficiencies,
- o Delegation works best when it's used in standard patterns.
- o Design patterns that use delegation: State, Strategy, Visitor, Mediator, Chain of Responsibility, and Bridge patterns.

6. Relating Run-Time and Compile-Time Structures

- An object-oriented program's run-time structure often bears little resemblance to its code structure.
- **Aggregation**
 - o Manifested at run-times.
 - o One object owns (having) or is responsible for another object (being part).



- **Acquaintance**
 - o Manifested at compile-times.
 - o An object merely *knows of* another object (association, using).
 - o A weaker relationship than aggregation.
- In implementation or code, aggregation and acquaintance cannot be distinct.
- Many design patterns capture the distinction between compile-time and run-time structures explicitly.
- The run-time structures aren't clear from the code until you understand the patterns.

7. Designing for Change

- Common causes of redesign along with the design pattern(s) that address them:
 1. *Creating an object by specifying a class explicitly.*
Design patterns: Abstract Factory, Factory Method, Prototype.
 2. *Dependence on specific operations..*
Design patterns: Chain of Responsibility, Command.
 3. *Dependence on hardware and software platform..* Design patterns: Abstract Factory, Bridge.
 4. *Dependence on object representations or implementations..*
Design patterns: Abstract Factory, Bridge, Memento, Proxy.
 5. *Algorithmic dependencies.*
Design patterns: Builder, Iterator , Strategy, Template Method , Visitor.
 6. *Tight coupling.*
Design patterns: Abstract Factory, Bridge, Chain of Responsibility, Command, Facade, Mediator, Observer.
Extending functionality by subclassing.
Design patterns: Bridge, Chain of Responsibility, Composite, Decorator, Observer, Strategy.
 7. *Inability to alter classes conveniently.*
Design patterns: Adapter, Decorator, Visitor.
- The role design patterns play in the development of three broad classes of software: application programs, toolkits, and frameworks.
 - **Application Programs**
 - f* Internal reuse, maintainability, and extension are high priorities.
 - f* Design patterns that reduce dependencies can increase internal reuse.
 - f* Design patterns make an application more maintainable when they're used to limit platform dependencies and to layer a system.
 - f* Design patterns enhance extensibility.
 - **Toolkits (class/component libraries)**
 - o Code reuse
 - o Application-general design
- **Frameworks**
 - o A framework is a set of cooperating classes that make up a reusable design for a specific class of software.
 - o The framework dictates the architecture of your application.
 - o Frameworks emphasize *design reuse* over code reuse.
 - o Frameworks are implemented as class hierarchies..
 - o Reuse on framework level leads to an inversion of control between the application and the software on which it's based.
 - o Mature frameworks usually incorporate several design patterns
 - o Design patterns vs. frameworks
 1. *Design patterns are more abstract than frameworks.*
 2. *Design patterns are smaller architectural elements than frameworks.*
 3. *Design patterns are less specialized than frameworks.*

6. How to Select a Design Pattern?

With more than 20 design patterns in the catalog to choose from, it might be hard to find the one that addresses a particular design problem, especially if the catalog is new and unfamiliar to you. Here are several different approaches to finding the design pattern that's right for your problem:

1. Consider how design patterns solve design problems.

How design patterns help you find appropriate objects, determine object granularity, specify object interfaces, and several other ways in which design patterns solve design problems. Referring to these discussions can help guide your search for the right pattern.

2. Scan Intent sections.

Lists the Intent sections from all the patterns in the catalog. Read through each pattern's intent to find one or more that sound relevant to your problem.

3. Study how patterns interrelate.

Studying the relationships between design patterns graphically can help direct you to the right pattern or group of patterns.

4. Study patterns of like purpose.

The catalog has three chapters, one for creational patterns, another for structural patterns, and a third for behavioral patterns. Each chapter starts off with introductory comments on the patterns and concludes with a section that compares and contrasts them. These sections give you insight into the similarities and differences between patterns of like purpose.

5. Examine a cause of redesign.

Look at the causes of redesign starting to see if your problem involves one or more of them. Then look at the patterns that help you avoid the causes of redesign.

6. Consider what should be variable in your design.

This approach is the opposite of focusing on the causes of redesign. Instead of considering what might *force* a change to a design, consider what you want to be *able* to change without redesign. The focus here is on *encapsulating the concept that varies* a theme of many design patterns. Table lists the design aspect(s) that design patterns let you vary independently, thereby letting you change them without redesign.

Purpose	Design Pattern	Aspect(s) that can vary
Creational	Abstract Factory	families of product objects
	Builder	how a composite object gets created
	Factory Method	subclass of object that is instantiated
	Prototype	class of object that is instantiated
	Singleton	the sole instance of a class
Structural	Adapter	interface to an object
	Bridge	implementation of an object
	Composite	structure and composition of an object
	Decorator	responsibilities of an object without subclassing
	Facade	interface to a subsystem
	Flyweight	storage costs of objects
	Proxy	how an object is accessed; its location
Behavioral	Chain of Responsibility	object that can fulfill a request
	Command	when and how a request is fulfilled
	Interpreter	grammar and interpretation of a language

Iterator	how an aggregate's elements are accessed, traversed
Mediator	how and which objects interact with each other
Memento	what private information is stored outside an object, and when
Observer	number of objects that depend on another object; how the dependent objects stay up to date
State	states of an object
Strategy	an algorithm
Template Method	steps of an algorithm
Visitor	operations that can be applied to object(s) without changing their class(es)

Table 1.2: Design aspects that design patterns let you vary

7. How to Use a Design Pattern?

Once picked a design pattern, follow the step-by-step approach to applying a design pattern effectively:

1. *Read the pattern once through for an overview.*

Pay particular attention to the Applicability and Consequences sections to ensure the pattern is right for the problem.

2. *Go back and study the Structure, Participants, and Collaborations sections.*

Make sure you understand the classes and objects in the pattern and how they relate to one another.

3. *Look at the Sample Code section to see a concrete example of the pattern in code.*

Studying the code helps you learn how to implement the pattern.

4. *Choose names for pattern participants that are meaningful in the application context.*

The names for participants in design patterns are usually too abstract to appear directly in an application. Nevertheless, it's useful to incorporate the participant name into the name that appears in the application. That helps make the pattern more explicit in the implementation.

5. *Define the classes.*

Declare their interfaces, establish their inheritance relationships, and define the instance variables that represent data and object references. Identify existing classes in your application that the pattern will affect, and modify them accordingly.

6. *Define application-specific names for operations in the pattern.*

Here again, the names generally depend on the application. Use the responsibilities and collaborations associated with each operation as a guide. Also, be consistent in your naming conventions.

7. *Implement the operations to carry out the responsibilities and collaborations in the pattern.*

The Implementation section offers hints to guide you in the implementation.

These are just guidelines to get you started. Over time we develop our own way of working with design patterns. No discussion of how to use design patterns would be complete without a few words on how *not* to use them. Design patterns should not be applied indiscriminately. Often they achieve flexibility and variability by introducing additional levels of indirection, and that can complicate a design and/or cost some performance. A design pattern should only be applied when the flexibility it affords is actually needed. The Consequences sections are most helpful when evaluating a pattern's benefits and liabilities.

8. Explain Design Patterns and its uses.

In software engineering, a **design pattern** is a general repeatable solution to a commonly occurring problem in software design. A design pattern isn't a finished design that can be transformed directly into code. It is a description or template for how to solve a problem that can be used in many different situations.

Uses of Design Patterns

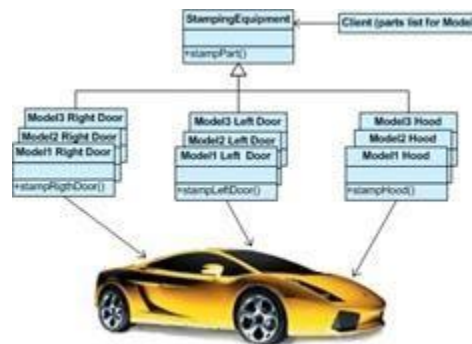
Design patterns can speed up the development process by providing tested, proven development paradigms. Effective software design requires considering issues that may not become visible until later in the implementation. Reusing design patterns helps to prevent subtle issues that can cause major problems and improves code readability for coders and architects familiar with the patterns.

Often, people only understand how to apply certain software design techniques to certain problems. These techniques are difficult to apply to a broader range of problems. Design patterns provide general solutions, documented in a format that doesn't require specifics tied to a particular problem.

In addition, patterns allow developers to communicate using well-known, well understood names for software interactions. Common design patterns can be improved over time, making them more robust than ad-hoc designs.

Creational design patterns

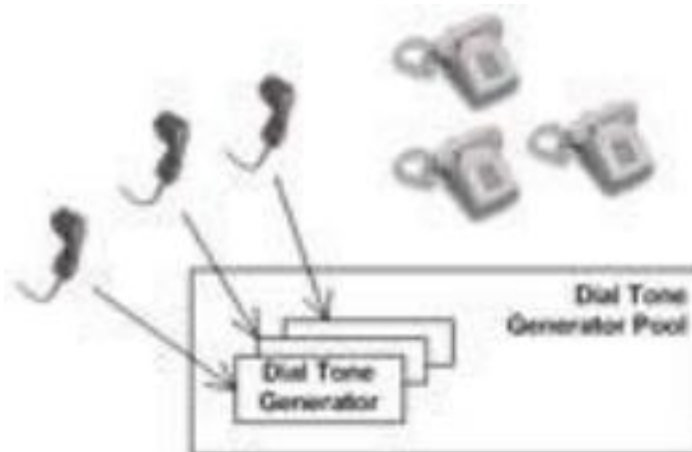
This design patterns is all about class instantiation. This pattern can be further divided into class-creation patterns and object-creational patterns. While class-creation patterns use inheritance effectively in the instantiation process, object-creation patterns use delegation effectively to get the job done.



- Abstract Factory
Creates an instance of several families of classes
- Builder
Separates object construction from its representation
- Factory Method
Creates an instance of several derived classes
- Object Pool
Avoid expensive acquisition and release of resources by recycling objects that are no longer in use
- Prototype
A fully initialized instance to be copied or cloned
- Singleton
A class of which only a single instance can exist

Structural design patterns

This design patterns is all about Class and Object composition. Structural class-creation patterns use inheritance to compose interfaces. Structural object-patterns define ways to compose objects to obtain new functionality.



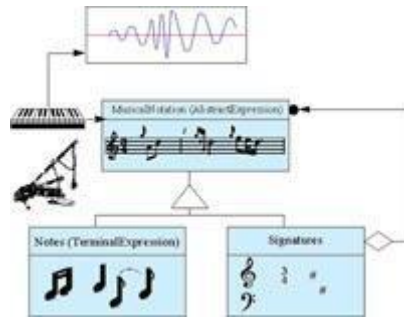
- Adapter
Match interfaces of different classes
- Bridge
Separates an object's interface from its implementation
- Composite
A tree structure of simple and composite objects
- Decorator
Add responsibilities to objects dynamically
- Facade
A single class that represents an entire subsystem
- Flyweight
A fine-grained instance used for efficient sharing



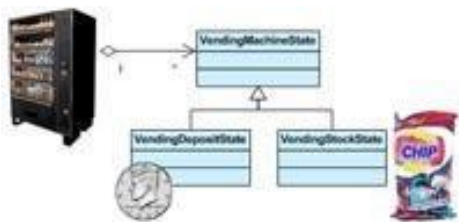
- Private Class Data
Restricts accessor/mutator access
- Proxy
An object representing another object

Behavioral design patterns

This design patterns is all about Class's objects communication. Behavioral patterns are those patterns that are most specifically concerned with communication between objects.



- Chain of responsibility
A way of passing a request between a chain of objects
- Command
Encapsulate a command request as an object
- Interpreter
A way to include language elements in a program
- Iterator
Sequentially access the elements of a collection
- Mediator
Defines simplified communication between classes
- Memento
Capture and restore an object's internal state
- Null Object
Designed to act as a default value of an object
- Observer
A way of notifying change to a number of classes



- State
Alter an object's behavior when its state changes
- Strategy
Encapsulates an algorithm inside a class
- Template method
Defer the exact steps of an algorithm to a subclass
- Visitor
Defines a new operation to a class without change

9. Explain briefly about Creational Design Patterns with examples.

Abstract Factory Design Pattern

Intent

- Provide an interface for creating families of related or dependent objects without specifying their concrete classes.
- A hierarchy that encapsulates: many possible -platforms, and the construction of a suite of -products.
- The new operator considered harmful.

Problem

If an application is to be portable, it needs to encapsulate platform dependencies. These -platforms might include: windowing system, operating system, database, etc. Too often, this encapsulation is not engineered in advance, and lots of #ifdef case statements with options for all currently supported platforms begin to procreate like rabbits throughout the code.

Discussion

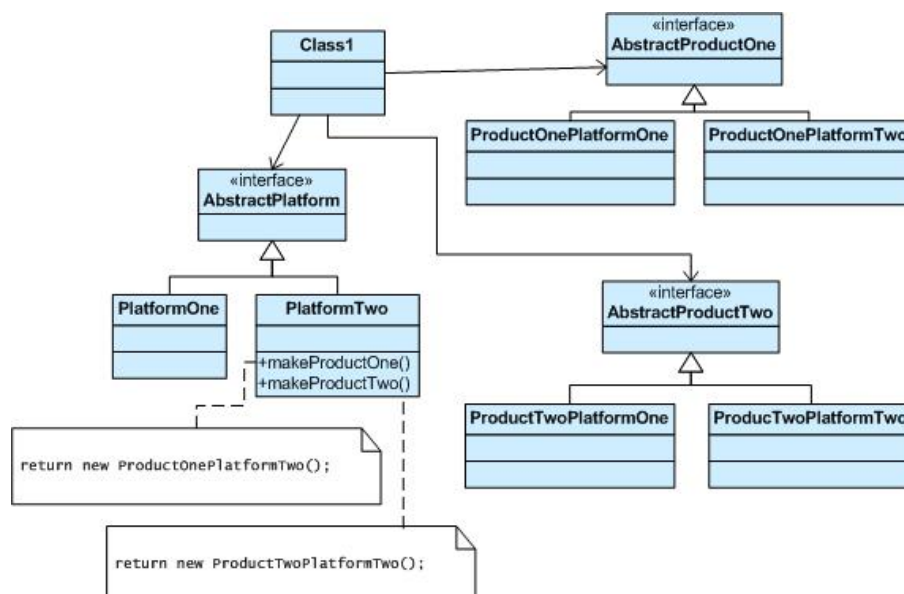
Provide a level of indirection that abstracts the creation of families of related or dependent objects without directly specifying their concrete classes. The -factory object has the responsibility for providing creation services for the entire platform family. Clients never create platform objects directly, they ask the factory to do that for them.

This mechanism makes exchanging product families easy because the specific class of the factory object appears only once in the application - where it is instantiated. The application can wholesale replace the entire family of products simply by instantiating a different concrete instance of the abstract factory.

Because the service provided by the factory object is so pervasive, it is routinely implemented as a Singleton.

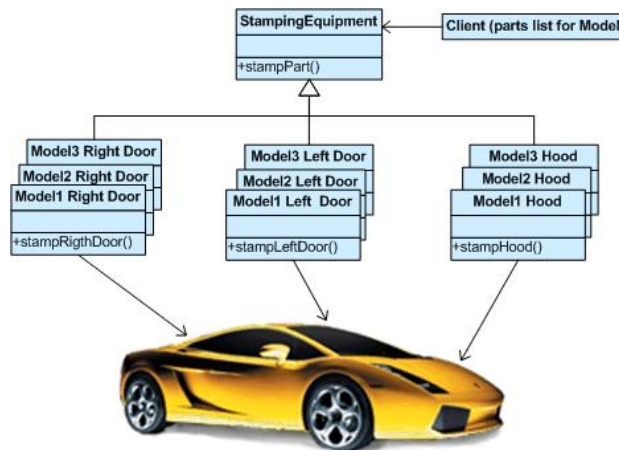
Structure

The Abstract Factory defines a Factory Method per product. Each Factory Method encapsulates the new operator and the concrete, platform-specific, product classes. Each -platform is then modeled with a Factory derived class.



Example

The purpose of the Abstract Factory is to provide an interface for creating families of related objects, without specifying concrete classes. This pattern is found in the sheet metal stamping equipment used in the manufacture of Japanese automobiles. The stamping equipment is an Abstract Factory which creates auto body parts. The same machinery is used to stamp right hand doors, left hand doors, right front fenders, left front fenders, hoods, etc. for different models of cars. Through the use of rollers to change the stamping dies, the concrete classes produced by the machinery can be changed within three minutes.



Check list

1. Decide if -platform independence and creation services are the current source of pain.
2. Map out a matrix of -platforms versus -products.
3. Define a factory interface that consists of a factory method per product.
4. Define a factory derived class for each platform that encapsulates all references to the new operator.
5. The client should retire all references to new, and use the factory methods to create the product objects.

Rules of thumb

- Sometimes creational patterns are competitors: there are cases when either Prototype or Abstract Factory could be used profitably. At other times they are complementary: Abstract Factory might store a set of Prototypes from which to clone and return product objects, Builder can use one of the other patterns to implement which components get built. Abstract Factory, Builder, and Prototype can use Singleton in their implementation.
- Abstract Factory, Builder, and Prototype define a factory object that's responsible for knowing and creating the class of product objects, and make it a parameter of the system. Abstract Factory has the factory object producing objects of several classes. Builder has the factory object building a complex product incrementally using a correspondingly complex protocol. Prototype has the factory object (aka prototype) building a product by copying a prototype object.
- Abstract Factory classes are often implemented with Factory Methods, but they can also be implemented using Prototype.
- Abstract Factory can be used as an alternative to Facade to hide platform-specific classes.
- Builder focuses on constructing a complex object step by step. Abstract Factory emphasizes a family of product objects (either simple or complex). Builder returns the product as a final step, but as far as the Abstract Factory is concerned, the product gets returned immediately.
- Often, designs start out using Factory Method and evolve toward Abstract Factory, Prototype, or Builder as the designer discovers where more flexibility is needed.

Builder Design Pattern

Intent

- Separate the construction of a complex object from its representation so that the same construction process can create different representations.
- Parse a complex representation, create one of several targets.

Problem

An application needs to create the elements of a complex aggregate. The specification for the aggregate exists on secondary storage and one of many representations needs to be built in primary storage.

Discussion

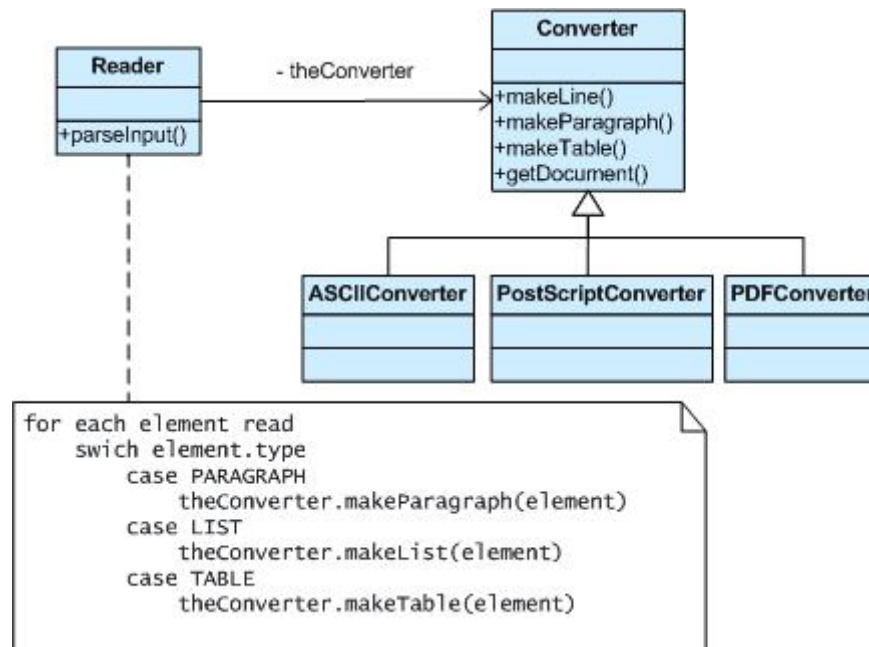
Separate the algorithm for interpreting (i.e. reading and parsing) a stored persistence mechanism (e.g. RTF files) from the algorithm for building and representing one of many target products (e.g. ASCII, TeX, text widget). The focus/distinction is on creating complex aggregates.

The `-director` invokes `-builder` services as it interprets the external format. The `-builder` creates part of the complex object each time it is called and maintains all intermediate state. When the product is finished, the client retrieves the result from the `-builder`.

Affords finer control over the construction process. Unlike creational patterns that construct products in one shot, the Builder pattern constructs the product step by step under the control of the `-director`.

Structure

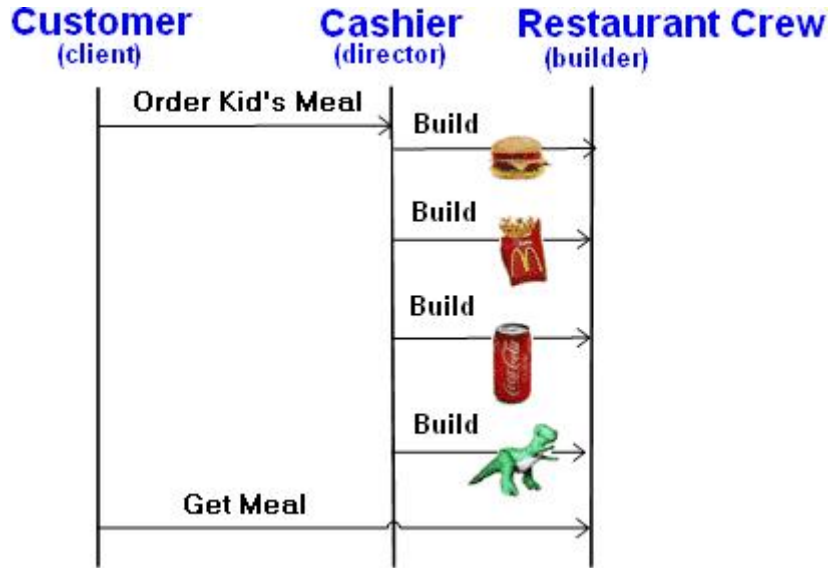
The Reader encapsulates the parsing of the common input. The Builder hierarchy makes possible the polymorphic creation of many peculiar representations or targets.



Example

The Builder pattern separates the construction of a complex object from its representation so that the same construction process can create different representations. This pattern is used by fast food restaurants to construct children's meals. Children's meals typically consist of a main item, a side item, a drink, and a toy (e.g., a hamburger, fries, Coke, and toy dinosaur). Note that there can be variation in the content of the children's meal, but the construction process is the same. Whether a customer orders a hamburger, cheeseburger, or chicken, the process is the same. The employee at the counter directs the

crew to assemble a main item, side item, and toy. These items are then placed in a bag. The drink is placed in a cup and remains outside of the bag. This same process is used at competing restaurants.



Check list

1. Decide if a common input and many possible representations (or outputs) is the problem at hand.
2. Encapsulate the parsing of the common input in a Reader class.
3. Design a standard protocol for creating all possible output representations. Capture the steps of this protocol in a Builder interface.
4. Define a Builder derived class for each target representation.
5. The client creates a Reader object and a Builder object, and registers the latter with the former.
6. The client asks the Reader to -constructl.
7. The client asks the Builder to return the result.

Rules of thumb

- Sometimes creational patterns are complementary: Builder can use one of the other patterns to implement which components get built. Abstract Factory, Builder, and Prototype can use Singleton in their implementations.
- Builder focuses on constructing a complex object step by step. Abstract Factory emphasizes a family of product objects (either simple or complex). Builder returns the product as a final step, but as far as the Abstract Factory is concerned, the product gets returned immediately.
- Builder often builds a Composite.
- Often, designs start out using Factory Method (less complicated, more customizable, subclasses proliferate) and evolve toward Abstract Factory, Prototype, or Builder (more flexible, more complex) as the designer discovers where more flexibility is needed.

Factory Method Design Pattern

Intent

- Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.
- Defining a -virtualll constructor.
- The new operator considered harmful.

Problem

A framework needs to standardize the architectural model for a range of applications, but allow for individual applications to define their own domain objects and provide for their instantiation.

Discussion

Factory Method is to creating objects as Template Method is to implementing an algorithm. A superclass specifies all standard and generic behavior (using pure virtual -placeholdersll for creation steps), and then delegates the creation details to subclasses that are supplied by the client.

Factory Method makes a design more customizable and only a little more complicated. Other design patterns require new classes, whereas Factory Method only requires a new operation.

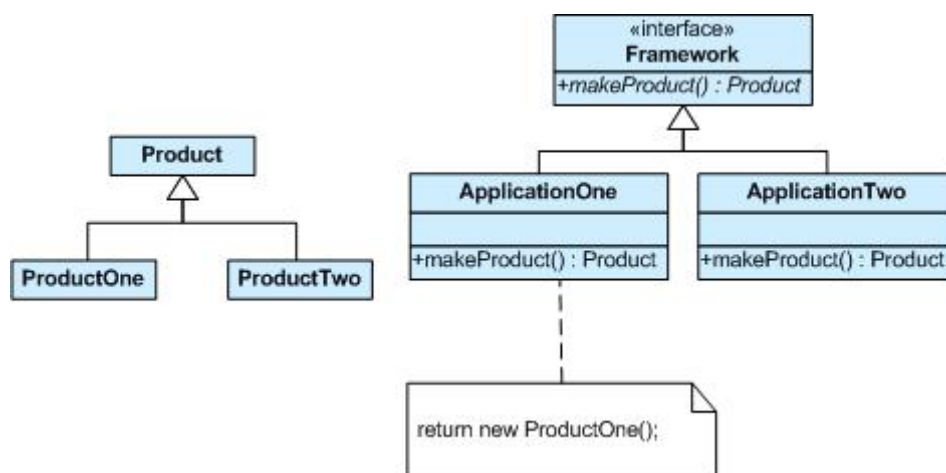
People often use Factory Method as the standard way to create objects; but it isn't necessary if: the class that's instantiated never changes, or instantiation takes place in an operation that subclasses can easily override (such as an initialization operation).

Factory Method is similar to Abstract Factory but without the emphasis on families.

Factory Methods are routinely specified by an architectural framework, and then implemented by the user of the framework.

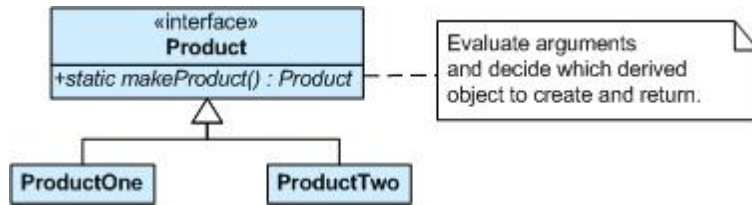
Structure

The implementation of Factory Method discussed in the Gang of Four (below) largely overlaps with that of Abstract Factory. For that reason, the presentation in this chapter focuses on the approach that has become popular since.

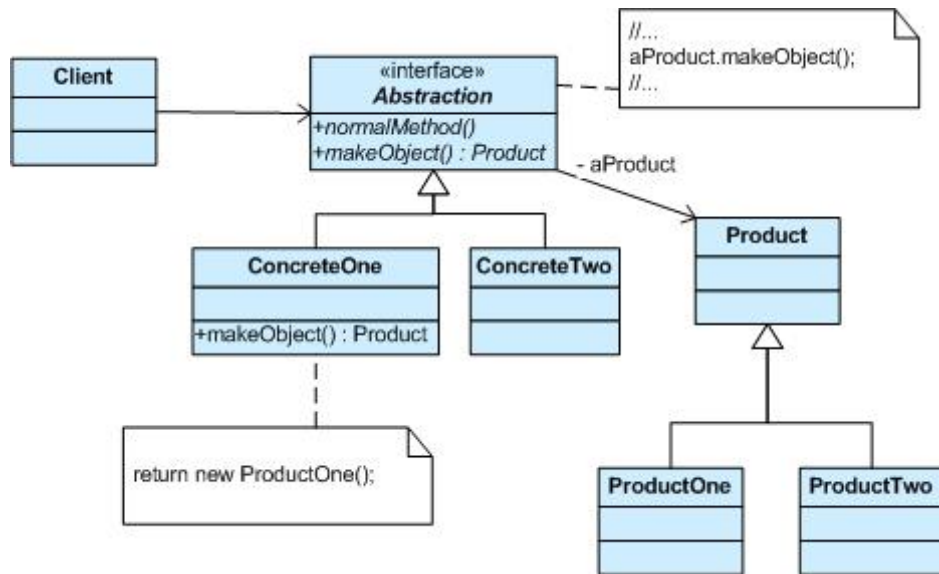


An increasingly popular definition of factory method is: a static method of a class that returns an object of that class' type. But unlike a constructor, the actual object it returns might be an instance of a subclass. Unlike a constructor, an existing object might be reused, instead of a new object created. Unlike a constructor, factory methods can have different and more descriptive names (e.g.

Color.make_RGB_color(float red, float green, float blue) and Color.make_HSB_color(float hue, float saturation, float brightness)

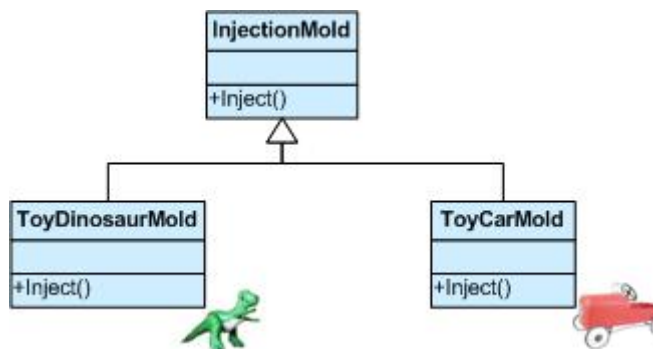


The client is totally decoupled from the implementation details of derived classes. Polymorphic creation is now possible.



Example

The Factory Method defines an interface for creating objects, but lets subclasses decide which classes to instantiate. Injection molding presses demonstrate this pattern. Manufacturers of plastic toys process plastic molding powder, and inject the plastic into molds of the desired shapes. The class of toy (car, action figure, etc.) is determined by the mold.



Check list

1. If you have an inheritance hierarchy that exercises polymorphism, consider adding a polymorphic creation capability by defining a static factory method in the base class.
2. Design the arguments to the factory method. What qualities or characteristics are necessary and sufficient to identify the correct derived class to instantiate?

3. Consider designing an internal –object pool that will allow objects to be reused instead of created from scratch.
4. Consider making all constructors private or protected.

Rules of thumb

- Abstract Factory classes are often implemented with Factory Methods, but they can be implemented using Prototype.
- Factory Methods are usually called within Template Methods.
- Factory Method: creation through inheritance. Prototype: creation through delegation.
- Often, designs start out using Factory Method (less complicated, more customizable, subclasses proliferate) and evolve toward Abstract Factory, Prototype, or Builder (more flexible, more complex) as the designer discovers where more flexibility is needed.
- Prototype doesn't require subclassing, but it does require an Initialize operation. Factory Method requires subclassing, but doesn't require Initialize.
- The advantage of a Factory Method is that it can return the same instance multiple times, or can return a subclass rather than an object of that exact type.
- Some Factory Method advocates recommend that as a matter of language design (or failing that, as a matter of style) absolutely all constructors should be private or protected. It's no one else's business whether a class manufactures a new object or recycles an old one.
- The new operator considered harmful. There is a difference between requesting an object and creating one. The new operator always creates an object, and fails to encapsulate object creation. A Factory Method enforces that encapsulation, and allows an object to be requested without inextricable coupling to the act of creation.

Prototype Design Pattern

Intent

- Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.
- Co-opt one instance of a class for use as a breeder of all future instances.
- The new operator considered harmful.

Problem

Application –hard wires the class of object to create in each –new expression.

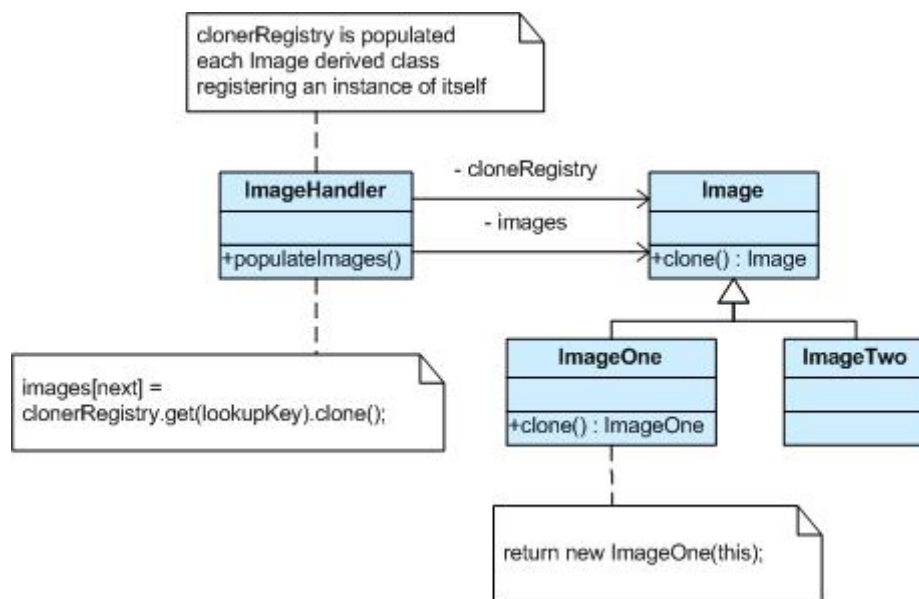
Discussion

Declare an abstract base class that specifies a pure virtual –clone method, and, maintains a dictionary of all –cloneable concrete derived classes. Any class that needs a –polymorphic constructor capability: derives itself from the abstract base class, registers its prototypical instance, and implements the clone() operation.

The client then, instead of writing code that invokes the –new operator on a hard-wired class name, calls a –clone operation on the abstract base class, supplying a string or enumerated data type that designates the particular concrete derived class desired.

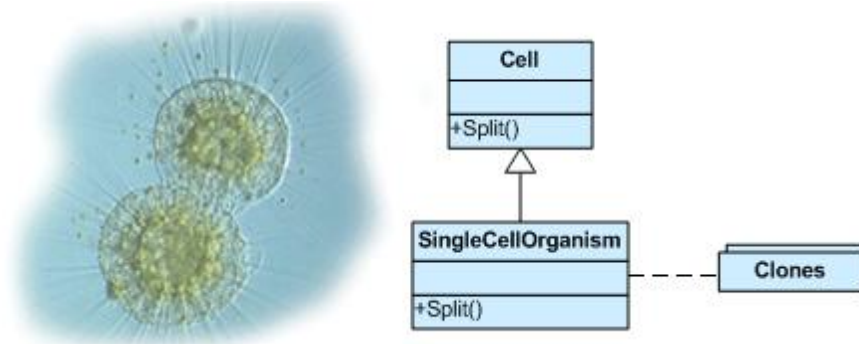
Structure

The Factory knows how to find the correct Prototype, and each Product knows how to spawn new instances of itself.



Example

The Prototype pattern specifies the kind of objects to create using a prototypical instance. Prototypes of new products are often built prior to full production, but in this example, the prototype is passive and does not participate in copying itself. The mitotic division of a cell - resulting in two identical cells - is an example of a prototype that plays an active role in copying itself and thus, demonstrates the Prototype pattern. When a cell splits, two cells of identical genotype result. In other words, the cell clones itself.



Check list

1. Add a `clone()` method to the existing `-product||` hierarchy.
2. Design a `-registry||` that maintains a cache of prototypical objects. The registry could be encapsulated in a new Factory class, or in the base class of the `-product||` hierarchy.
3. Design a factory method that: may (or may not) accept arguments, finds the correct prototype object, calls `clone()` on that object, and returns the result.
4. The client replaces all references to the new operator with calls to the factory method.

Rules of thumb

- Sometimes creational patterns are competitors: there are cases when either Prototype or Abstract Factory could be used properly. At other times they are complementary: Abstract Factory might store a set of Prototypes from which to clone and return product objects. Abstract Factory, Builder, and Prototype can use Singleton in their implementations.

- Abstract Factory classes are often implemented with Factory Methods, but they can be implemented using Prototype.
- Factory Method: creation through inheritance. Prototype: creation through delegation.
- Often, designs start out using Factory Method (less complicated, more customizable, subclasses proliferate) and evolve toward Abstract Factory, Prototype, or Builder (more flexible, more complex) as the designer discovers where more flexibility is needed.
- Prototype doesn't require subclassing, but it does require an `-initialize` operation. Factory Method requires subclassing, but doesn't require `Initialize`.
- Designs that make heavy use of the Composite and Decorator patterns often can benefit from Prototype as well.
- Prototype co-opts one instance of a class for use as a breeder of all future instances.
- Prototypes are useful when object initialization is expensive, and you anticipate few variations on the initialization parameters. In this context, Prototype can avoid expensive `-creation from scratch`, and support cheap cloning of a pre-initialized prototype.
- Prototype is unique among the other creational patterns in that it doesn't require a class – only an object. Object-oriented languages like Self and Omega that do away with classes completely rely on prototypes for creating new objects.

Singleton Design Pattern

Intent

- Ensure a class has only one instance, and provide a global point of access to it.
- Encapsulated `-just-in-time initialization` or `-initialization on first use`.

Problem

Application needs one, and only one, instance of an object. Additionally, lazy initialization and global access are necessary.

Discussion

Make the class of the single instance object responsible for creation, initialization, access, and enforcement. Declare the instance as a private static data member. Provide a public static member function that encapsulates all initialization code, and provides access to the instance.

The client calls the accessor function (using the class name and scope resolution operator) whenever a reference to the single instance is required.

Singleton should be considered only if all three of the following criteria are satisfied:

- Ownership of the single instance cannot be reasonably assigned
- Lazy initialization is desirable
- Global access is not otherwise provided for

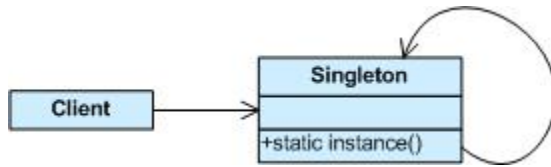
If ownership of the single instance, when and how initialization occurs, and global access are not issues, Singleton is not sufficiently interesting.

The Singleton pattern can be extended to support access to an application-specific number of instances.

The `-static` member function accessor approach will not support subclassing of the Singleton class. If subclassing is desired, refer to the discussion in the book.

Deleting a Singleton class/instance is a non-trivial design problem. See [-To Kill A Singleton](#) by John Vlissides for a discussion.

Structure



Make the class of the single instance responsible for access and `-initialization` on first use. The single instance is a private static attribute. The accessor function is a public static method.



Example

The Singleton pattern ensures that a class has only one instance and provides a global point of access to that instance. It is named after the singleton set, which is defined to be a set containing one element. The office of the President of the United States is a Singleton. The United States Constitution specifies the means by which a president is elected, limits the term of office, and defines the order of succession. As a result, there can be at most one active president at any given time. Regardless of the personal identity of the active president, the title, `-The President of the United States` is a global point of access that identifies the person in the office.



Check list

1. Define a private static attribute in the `-single instance` class.
2. Define a public static accessor function in the class.
3. Do `-lazy initialization` (creation on first use) in the accessor function.
4. Define all constructors to be protected or private.
5. Clients may only use the accessor function to manipulate the Singleton.

Rules of thumb

- Abstract Factory, Builder, and Prototype can use Singleton in their implementation.
- Facade objects are often Singletons because only one Facade object is required.

- State objects are often Singletons.
- The advantage of Singleton over global variables is that you are absolutely sure of the number of instances when you use Singleton, and, you can change your mind and manage any number of instances.
- The Singleton design pattern is one of the most inappropriately used patterns. Singletons are intended to be used when a class must have exactly one instance, no more, no less. Designers frequently use Singletons in a misguided attempt to replace global variables. A Singleton is, for intents and purposes, a global variable. The Singleton does not do away with the global; it merely renames it.
- When is Singleton unnecessary? Short answer: most of the time. Long answer: when it's simpler to pass an object resource as a reference to the objects that need it, rather than letting objects access the resource globally. The real problem with Singletons is that they give you such a good excuse not to think carefully about the appropriate visibility of an object. Finding the right balance of exposure and protection for an object is critical for maintaining flexibility.
- Our group had a bad habit of using global data, so I did a study group on Singleton. The next thing I know Singletons appeared everywhere and none of the problems related to global data went away. The answer to the global data question is not, -Make it a Singleton. The answer is, -Why in the hell are you using global data? Changing the name doesn't change the problem. In fact, it may make it worse because it gives you the opportunity to say, -Well I'm not doing that, I'm doing this - even though this and that are the same thing.

10. Explain briefly about Structural Design Pattern with examples.

Adapter Design Pattern

Intent

- Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.
- Wrap an existing class with a new interface.
- Impedance match an old component to a new system

Problem

An -off the shelf component offers compelling functionality that you would like to reuse, but its -view of the world is not compatible with the philosophy and architecture of the system currently being developed.

Discussion

Reuse has always been painful and elusive. One reason has been the tribulation of designing something new, while reusing something old. There is always something not quite right between the old and the new. It may be physical dimensions or misalignment. It may be timing or synchronization. It may be unfortunate assumptions or competing standards.

It is like the problem of inserting a new three-prong electrical plug in an old two-prong wall outlet – some kind of adapter or intermediary is necessary.

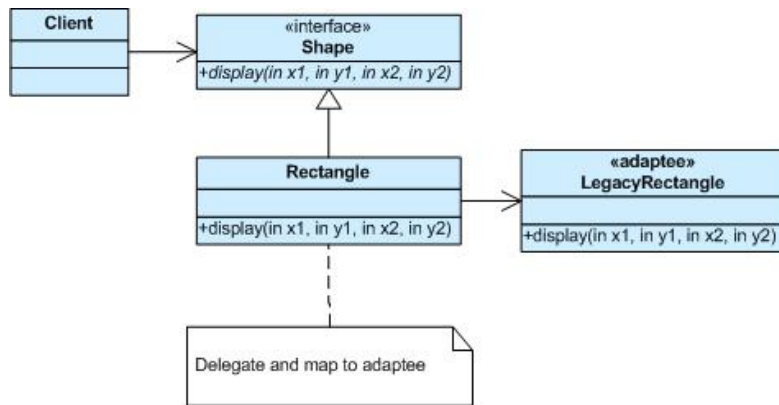


Adapter is about creating an intermediary abstraction that translates, or maps, the old component to the new system. Clients call methods on the Adapter object which redirects them into calls to the legacy component. This strategy can be implemented either with inheritance or with aggregation.

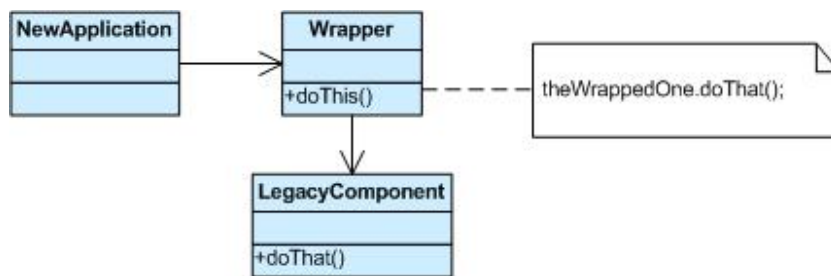
Adapter functions as a wrapper or modifier of an existing class. It provides a different or translated view of that class.

Structure

Below, a legacy Rectangle component's display() method expects to receive -x, y, w, h parameters. But the client wants to pass -upper left x and y and -lower right x and y. This incongruity can be reconciled by adding an additional level of indirection - i.e. an Adapter object.

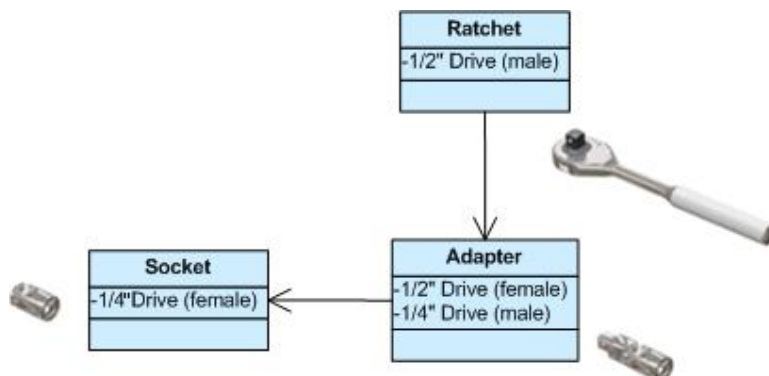


The Adapter could also be thought of as a -wrapper.



Example

The Adapter pattern allows otherwise incompatible classes to work together by converting the interface of one class into an interface expected by the clients. Socket wrenches provide an example of the Adapter. A socket attaches to a ratchet, provided that the size of the drive is the same. Typical drive sizes in the United States are 1/2" and 1/4". Obviously, a 1/2" drive ratchet will not fit into a 1/4" drive socket unless an adapter is used. A 1/2" to 1/4" adapter has a 1/2" connection to fit on the 1/2" drive ratchet, and a 1/4" connection to fit in the 1/4" drive socket.



Check list

1. Identify the players: the component(s) that want to be accommodated (i.e. the client), and the component that needs to adapt (i.e. the adaptee).
2. Identify the interface that the client requires.
3. Design a -wrapper class that can -impedance match the adaptee to the client.
4. The adapter/wrapper class -has all instance of the adaptee class.
5. The adapter/wrapper class -maps the client interface to the adaptee interface.
6. The client uses (is coupled to) the new interface

Bridge Design Pattern

Intent

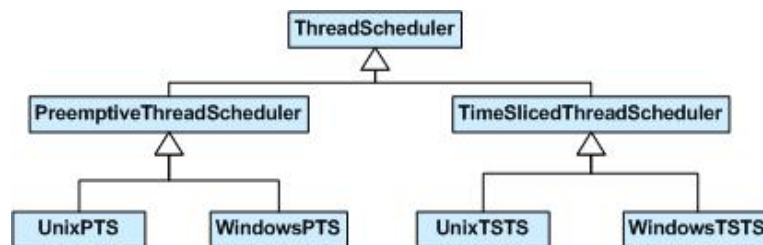
- Decouple an abstraction from its implementation so that the two can vary independently.
- Publish interface in an inheritance hierarchy, and bury implementation in its own inheritance hierarchy.
- Beyond encapsulation, to insulation

Problem

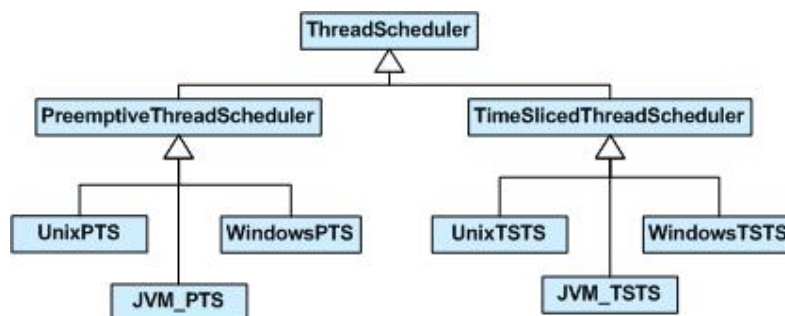
-Hardening of the software arteries has occurred by using subclassing of an abstract base class to provide alternative implementations. This locks in compile-time binding between interface and implementation. The abstraction and implementation cannot be independently extended or composed.

Motivation

Consider the domain of -thread scheduling.

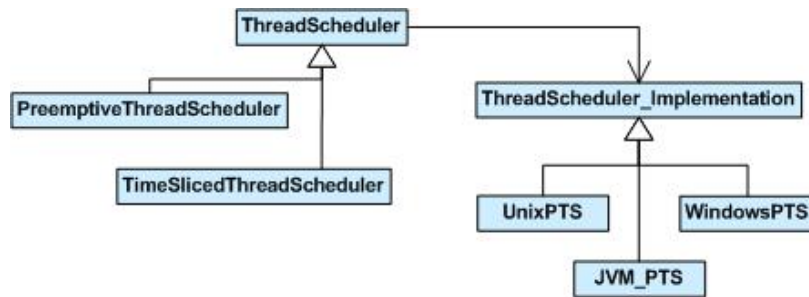


There are two types of thread schedulers, and two types of operating systems or -platforms. Given this approach to specialization, we have to define a class for each permutation of these two dimensions. If we add a new platform (say ... Java's Virtual Machine), what would our hierarchy look like?



What if we had three kinds of thread schedulers, and four kinds of platforms? What if we had five kinds of thread schedulers, and ten kinds of platforms? The number of classes we would have to define is the product of the number of scheduling schemes and the number of platforms.

The Bridge design pattern proposes refactoring this exponentially explosive inheritance hierarchy into two orthogonal hierarchies – one for platform-independent abstractions, and the other for platform-dependent implementations.



Discussion

Decompose the component's interface and implementation into orthogonal class hierarchies. The interface class contains a pointer to the abstract implementation class. This pointer is initialized with an instance of a concrete implementation class, but all subsequent interaction from the interface class to the implementation class is limited to the abstraction maintained in the implementation base class. The client interacts with the interface class, and it in turn –delegates‖ all requests to the implementation class.

The interface object is the –handle‖ known and used by the client; while the implementation object, or –body‖, is safely encapsulated to ensure that it may continue to evolve, or be entirely replaced (or shared) at run-time.

Use the Bridge pattern when:

- you want run-time binding of the implementation,
- you have a proliferation of classes resulting from a coupled interface and numerous implementations,
- you want to share an implementation among multiple objects,
- you need to map orthogonal class hierarchies.

Consequences include:

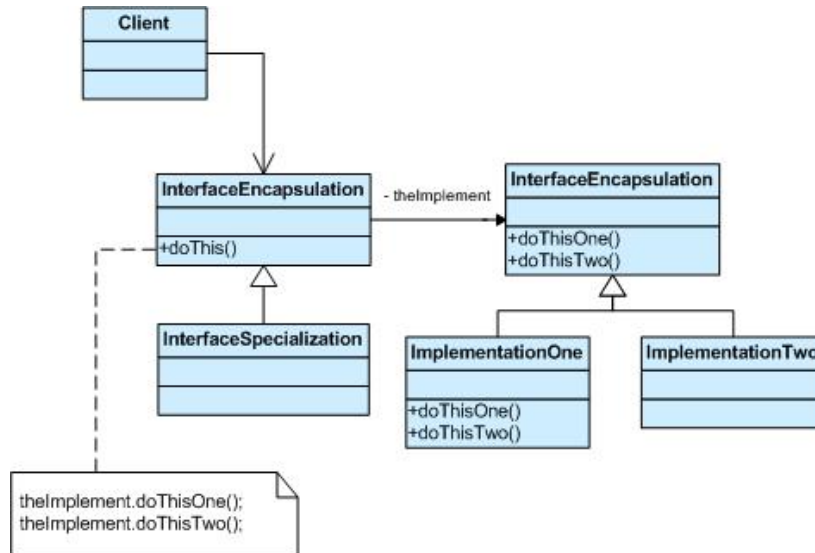
- decoupling the object's interface,
- improved extensibility (you can extend (i.e. subclass) the abstraction and implementation hierarchies independently),
- hiding details from clients.

Bridge is a synonym for the –handle/body‖ idiom. This is a design mechanism that encapsulates an implementation class inside of an interface class. The former is the body, and the latter is the handle. The handle is viewed by the user as the actual class, but the work is done in the body. –The handle/body class idiom may be used to decompose a complex abstraction into smaller, more manageable classes. The idiom may reflect the sharing of a single resource by multiple classes that control access to it (e.g. reference counting).‖

Structure

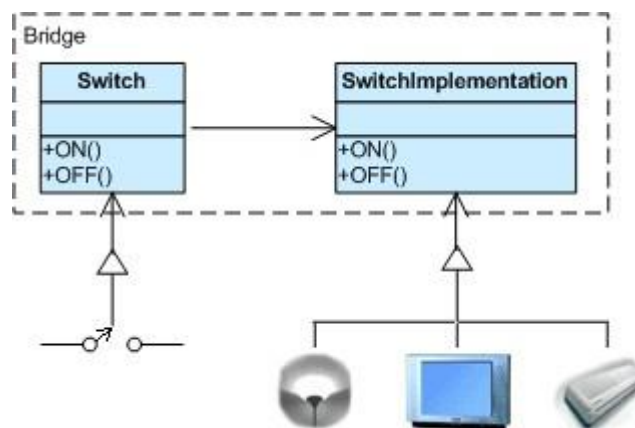
The Client doesn't want to deal with platform-dependent details. The Bridge pattern encapsulates this complexity behind an abstraction –wrapper‖.

Bridge emphasizes identifying and decoupling –interface‖ abstraction from –implementation‖ abstraction.



Example

The Bridge pattern decouples an abstraction from its implementation, so that the two can vary independently. A household switch controlling lights, ceiling fans, etc. is an example of the Bridge. The purpose of the switch is to turn a device on or off. The actual switch can be implemented as a pull chain, simple two position switch, or a variety of dimmer switches.



Check list

1. Decide if two orthogonal dimensions exist in the domain. These independent concepts could be: abstraction/platform, or domain/infrastructure, or front-end/back-end, or interface/implementation.
2. Design the separation of concerns: what does the client want, and what do the platforms provide.
3. Design a platform-oriented interface that is minimal, necessary, and sufficient. Its goal is to decouple the abstraction from the platform.
4. Define a derived class of that interface for each platform.
5. Create the abstraction base class that –has all platform object and delegates the platform-oriented functionality to it.
6. Define specializations of the abstraction class if desired.

Decorator Design Pattern

Intent

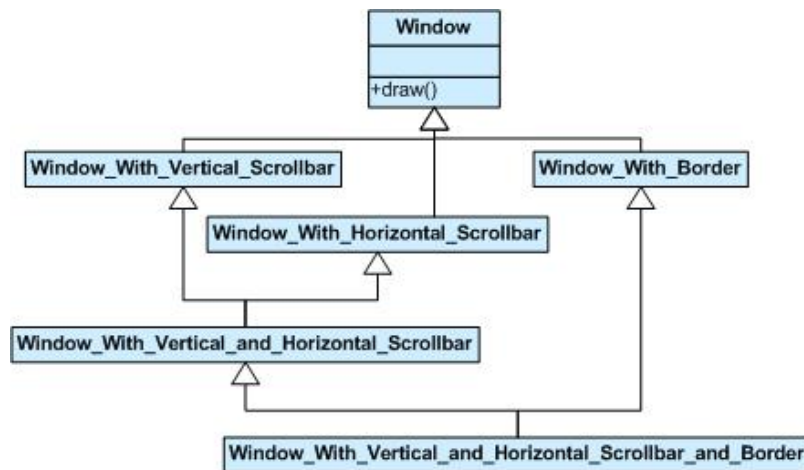
- Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.
- Client-specified embellishment of a core object by recursively wrapping it.
- Wrapping a gift, putting it in a box, and wrapping the box.

Problem

You want to add behavior or state to individual objects at run-time. Inheritance is not feasible because it is static and applies to an entire class.

Discussion

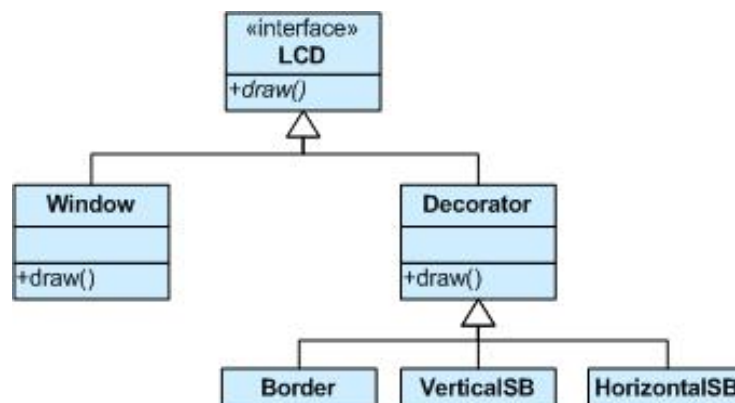
Suppose you are working on a user interface toolkit and you wish to support adding borders and scroll bars to windows. You could define an inheritance hierarchy like ...



But the Decorator pattern suggests giving the client the ability to specify whatever combination of -features|| is desired.

```
Widget* aWidget = new
BorderDecorator(newHorizontalScrollBarDec
orator( newVerticalScrollBarDecorator(
new Window( 80, 24 ))));
aWidget->draw();
```

This flexibility can be achieved with the following design



Another example of cascading (or chaining) features together to produce a custom object might look like ...

```
Stream* aStream = new
CompressingStream(new ASCII7Stream(
newFileStream( "fileName.dat" ));
aStream->putString( "Hello world" );
```

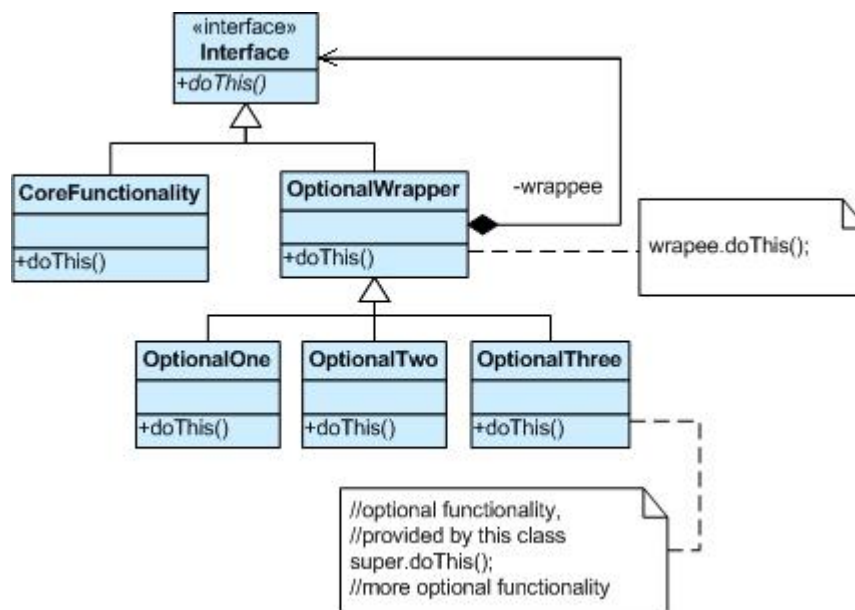
The solution to this class of problems involves encapsulating the original object inside an abstract wrapper interface. Both the decorator objects and the core object inherit from this abstract interface. The interface uses recursive composition to allow an unlimited number of decorator -layers to be added to each core object.

Note that this pattern allows responsibilities to be added to an object, not methods to an object's interface. The interface presented to the client must remain constant as successive layers are specified.

Also note that the core object's identity has now been -hidden inside of a decorator object. Trying to access the core object directly is now a problem.

Structure

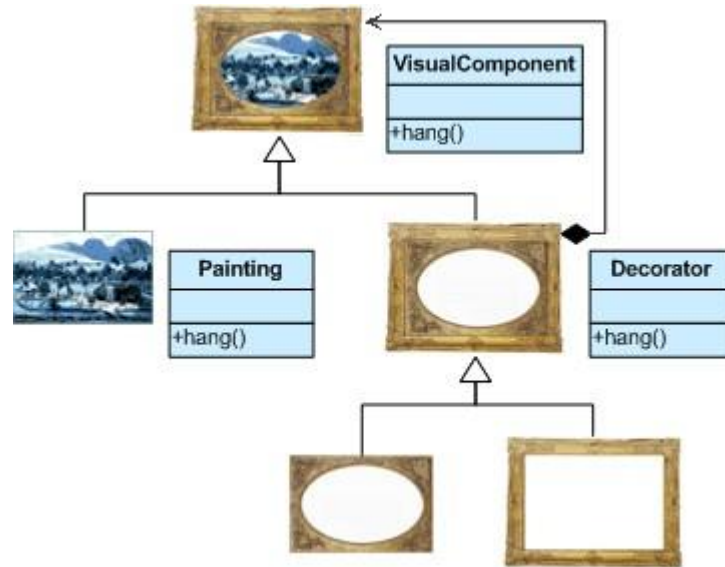
The client is always interested in CoreFunctionality.doThis(). The client may, or may not, be interested in OptionalOne.doThis() and OptionalTwo.doThis(). Each of these classes always delegate to the Decorator base class, and that class always delegates to the contained -wrappee object.



Example

The Decorator attaches additional responsibilities to an object dynamically. The ornaments that are added to pine or fir trees are examples of Decorators. Lights, garland, candy canes, glass ornaments, etc., can be added to a tree to give it a festive look. The ornaments do not change the tree itself which is recognizable as a Christmas tree regardless of particular ornaments used. As an example of additional functionality, the addition of lights allows one to -light up a Christmas tree.

Although paintings can be hung on a wall with or without frames, frames are often added, and it is the frame which is actually hung on the wall. Prior to hanging, the paintings may be matted and framed, with the painting, matting, and frame forming a single visual component.



Check list

1. Ensure the context is: a single core (or non-optional) component, several optional embellishments or wrappers, and an interface that is common to all.
2. Create a -Lowest Common Denominator interface that makes all classes interchangeable.
3. Create a second level base class (Decorator) to support the optional wrapper classes.
4. The Core class and Decorator class inherit from the LCD interface.
5. The Decorator class declares a composition relationship to the LCD interface, and this data member is initialized in its constructor.
6. The Decorator class delegates to the LCD object.
7. Define a Decorator derived class for each optional embellishment.
8. Decorator derived classes implement their wrapper functionality - and - delegate to the Decorator base class.
9. The client configures the type and ordering of Core and Decorator objects.

Composite Design Pattern

Intent

- Compose objects into tree structures to represent whole-part hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.
- Recursive composition
- -Directories contain entries, each of which could be a directory.||
- 1-to-many -has all up the -is all hierarchy

Problem

Application needs to manipulate a hierarchical collection of -primitive|| and -compositell objects. Processing of a primitive object is handled one way, and processing of a composite object is handled differently. Having to query the -typell of each object before attempting to process it is not desirable.

Discussion

Define an abstract base class (Component) that specifies the behavior that needs to be exercised uniformly across all primitive and composite objects. Subclass the Primitive and Composite classes off of

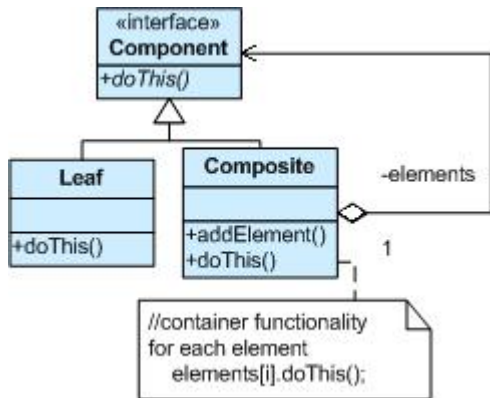
the Component class. Each Composite object –couples|| itself only to the abstract type Component as it manages its –children||.

Use this pattern whenever you have –composites that contain components, each of which could be a compositel||.

Child management methods [e.g. addChild(), removeChild()] should normally be defined in the Composite class. Unfortunately, the desire to treat Primitives and Composites uniformly requires that these methods be moved to the abstract Component class. See the –Opinions|| section below for a discussion of –safety|| versus –transparency|| issues.

Structure

Composites that contain Components, each of which could be a Composite.



Menus that contain menu items, each of which could be a menu.

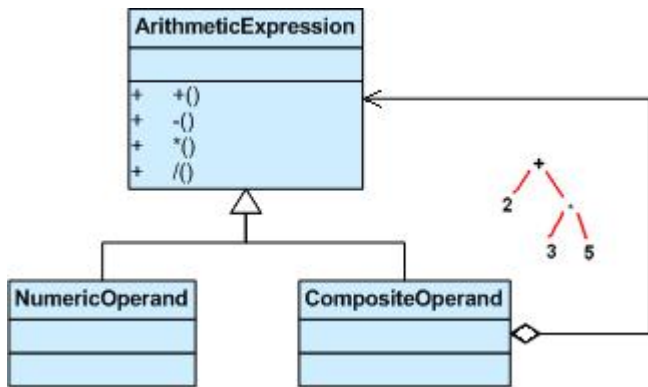
Row-column GUI layout managers that contain widgets, each of which could be a row-column GUI layout manager.

Directories that contain files, each of which could be a directory.

Containers that contain Elements, each of which could be a Container.

Example

The Composite composes objects into tree structures and lets clients treat individual objects and compositions uniformly. Although the example is abstract, arithmetic expressions are Composites. An arithmetic expression consists of an operand, an operator (+ - * /), and another operand. The operand can be a number, or another arithmetic expression. Thus, 2 + 3 and (2 + 3) + (4 * 6) are both valid expressions.



Check list

1. Ensure that your problem is about representing –whole-part‖ hierarchical relationships.
2. Consider the heuristic, –Containers that contain containees, each of which could be a container.‖ For example, –Assemblies that contain components, each of which could be an assembly.‖ Divide your domain concepts into container classes, and containee classes.
3. Create a –lowest common denominator‖ interface that makes your containers and containees interchangeable. It should specify the behavior that needs to be exercised uniformly across all containee and container objects.
4. All container and containee classes declare an –is all relationship to the interface.
5. All container classes declare a one-to-many –has all relationship to the interface.
6. Container classes leverage polymorphism to delegate to their containee objects.
7. Child management methods [e.g. addChild(), removeChild()] should normally be defined in the Composite class. Unfortunately, the desire to treat Leaf and Composite objects uniformly may require that these methods be promoted to the abstract Component class. See the Gang of Four for a discussion of these –safety‖ versus –transparency‖ trade-offs.

Rules of thumb

- Composite and Decorator have similar structure diagrams, reflecting the fact that both rely on recursive composition to organize an open-ended number of objects.
- Composite can be traversed with Iterator. Visitor can apply an operation over a Composite. Composite could use Chain of Responsibility to let components access global properties through their parent. It could also use Decorator to override these properties on parts of the composition. It could use Observer to tie one object structure to another and State to let a component change its behavior as its state changes.
- Composite can let you compose a Mediator out of smaller pieces through recursive composition.
- Decorator is designed to let you add responsibilities to objects without subclassing. Composite’s focus is not on embellishment but on representation. These intents are distinct but complementary. Consequently, Composite and Decorator are often used in concert.
- Flyweight is often combined with Composite to implement shared leaf nodes.

Opinions

The whole point of the Composite pattern is that the Composite can be treated atomically, just like a leaf. If you want to provide an Iterator protocol, fine, but I think that is outside the pattern itself. At the heart of this pattern is the ability for a client to perform operations on an object without needing to know that there are many objects inside.

Being able to treat a heterogeneous collection of objects atomically (or transparently) requires that the –child management‖ interface be defined at the root of the Composite class hierarchy (the abstract Component class). However, this choice costs you safety, because clients may try to do meaningless things like add and remove objects from leaf objects. On the other hand, if you –design for safety‖, the child management interface is declared in the Composite class, and you lose transparency because leaves and Composites now have different interfaces.

Smalltalk implementations of the Composite pattern usually do not have the interface for managing the components in the Component interface, but in the Composite interface. C++ implementations tend to put

it in the Component interface. This is an extremely interesting fact, and one that I often ponder. I can offer theories to explain it, but nobody knows for sure why it is true.

My Component classes do not know that Composites exist. They provide no help for navigating Composites, nor any help for altering the contents of a Composite. This is because I would like the base class (and all its derivatives) to be reusable in contexts that do not require Composites. When given a base class pointer, if I absolutely need to know whether or not it is a Composite, I will use `dynamic_cast` to figure this out. In those cases where `dynamic_cast` is too expensive, I will use a Visitor.

Common complaint: -if I push the Composite interface down into the Composite class, how am I going to enumerate (i.e. traverse) a complex structure? My answer is that when I have behaviors which apply to hierarchies like the one presented in the Composite pattern, I typically use Visitor, so enumeration isn't a problem - the Visitor knows in each case, exactly what kind of object it's dealing with. The Visitor doesn't need every object to provide an enumeration interface.

Composite doesn't force you to treat all Components as Composites. It merely tells you to put all operations that you want to treat -uniformly in the Component class. If add, remove, and similar operations cannot, or must not, be treated uniformly, then do not put them in the Component base class. Remember, by the way, that each pattern's structure diagram doesn't define the pattern; it merely depicts what in our experience is a common realization thereof. Just because Composite's structure diagram shows child management operations in the Component base class doesn't mean all implementations of the pattern must do the same.

Facade Design Pattern

Intent

- Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.
- Wrap a complicated subsystem with a simpler interface.

Problem

A segment of the client community needs a simplified interface to the overall functionality of a complex subsystem.

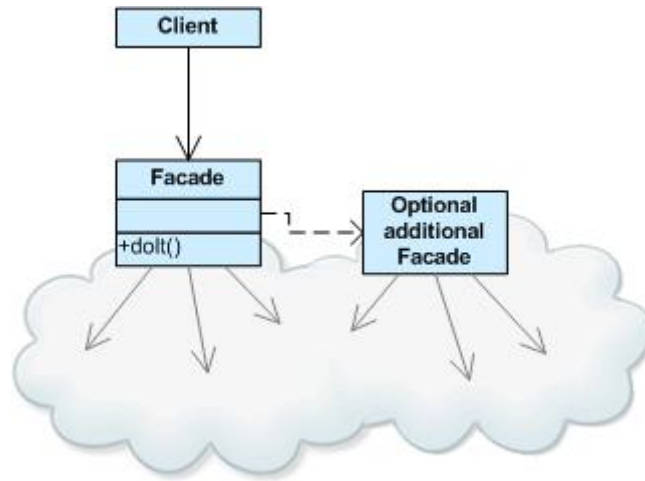
Discussion

Facade discusses encapsulating a complex subsystem within a single interface object. This reduces the learning curve necessary to successfully leverage the subsystem. It also promotes decoupling the subsystem from its potentially many clients. On the other hand, if the Facade is the only access point for the subsystem, it will limit the features and flexibility that -power users may need.

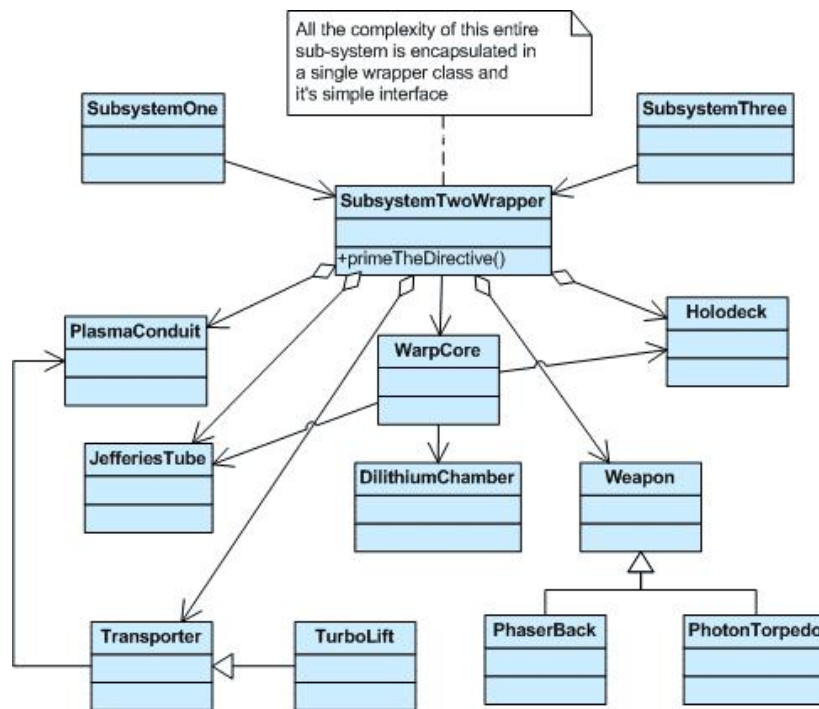
The Facade object should be a fairly simple advocate or facilitator. It should not become an all-knowing oracle or -god object.

Structure

Facade takes a -riddle wrapped in an enigma shrouded in mystery, and interjects a wrapper that tames the amorphous and inscrutable mass of software.

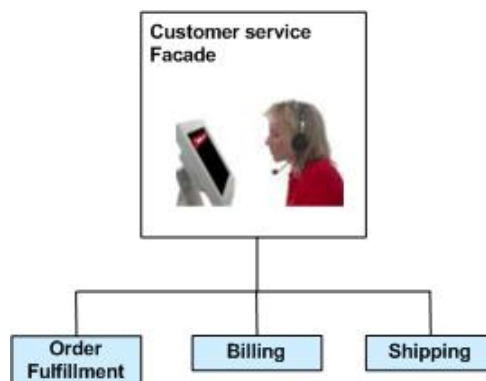


SubsystemOne and SubsystemThree do not interact with the internal components of SubsystemTwo. They use the SubsystemTwoWrapper -facade (i.e. the higher level abstraction).



Example

The Facade defines a unified, higher level interface to a subsystem that makes it easier to use. Consumers encounter a Facade when ordering from a catalog. The consumer calls one number and speaks with a customer service representative. The customer service representative acts as a Facade, providing an interface to the order fulfillment department, the billing department, and the shipping department.



Check list

1. Identify a simpler, unified interface for the subsystem or component.
2. Design a `_wrapper` class that encapsulates the subsystem.
3. The facade/wrapper captures the complexity and collaborations of the component, and delegates to the appropriate methods.
4. The client uses (is coupled to) the Facade only.
5. Consider whether additional Facades would add value.

Rules of thumb

- Facade defines a new interface, whereas Adapter uses an old interface. Remember that Adapter makes two existing interfaces work together as opposed to defining an entirely new one.
- Whereas Flyweight shows how to make lots of little objects, Facade shows how to make a single object represent an entire subsystem.
- Mediator is similar to Facade in that it abstracts functionality of existing classes. Mediator abstracts/centralizes arbitrary communications between colleague objects. It routinely –adds value, and it is known/referenced by the colleague objects. In contrast, Facade defines a simpler interface to a subsystem, it doesn't add new functionality, and it is not known by the subsystem classes.
- Abstract Factory can be used as an alternative to Facade to hide platform-specific classes.
- Facade objects are often Singletons because only one Facade object is required.
- Adapter and Facade are both wrappers; but they are different kinds of wrappers. The intent of Facade is to produce a simpler interface, and the intent of Adapter is to design to an existing interface. While Facade routinely wraps multiple objects and Adapter wraps a single object; Facade could front-end a single complex object and Adapter could wrap several legacy objects.

Question: So the way to tell the difference between the Adapter pattern and the Facade pattern is that the Adapter wraps one class and the Facade may represent many classes?

Answer: No! Remember, the Adapter pattern changes the interface of one or more classes into one interface that a client is expecting. While most textbook examples show the adapter adapting one class, you may need to adapt many classes to provide the interface a client is coded to. Likewise, a Facade may provide a simplified interface to a single class with a very complex interface. The difference between the two is not in terms of how many classes they –wrap, it is in their intent.

Flyweight Design Pattern

Intent

- Use sharing to support large numbers of fine-grained objects efficiently.
- The Motif GUI strategy of replacing heavy-weight widgets with light-weight gadgets.

Problem

Designing objects down to the lowest levels of system –granularity provides optimal flexibility, but can be unacceptably expensive in terms of performance and memory usage.

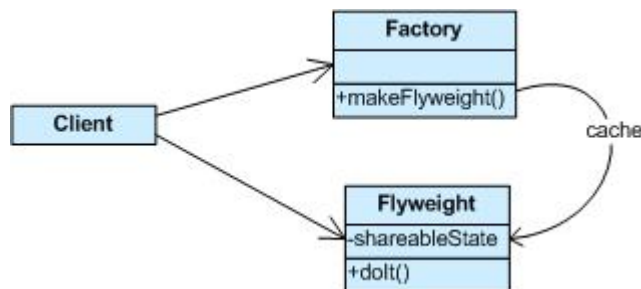
Discussion

The Flyweight pattern describes how to share objects to allow their use at fine granularities without prohibitive cost. Each flyweight object is divided into two pieces: the state-dependent (extrinsic) part, and the state-independent (intrinsic) part. Intrinsic state is stored (shared) in the Flyweight object. Extrinsic state is stored or computed by client objects, and passed to the Flyweight when its operations are invoked.

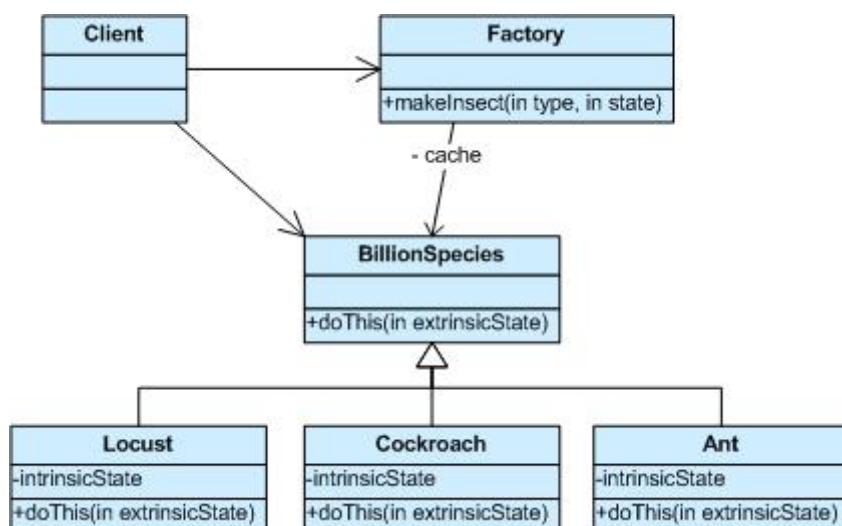
An illustration of this approach would be Motif widgets that have been re-engineered as light-weight gadgets. Whereas widgets are intelligent enough to stand on their own; gadgets exist in a dependent relationship with their parent layout manager widget. Each layout manager provides context-dependent event handling, real estate management, and resource services to its flyweight gadgets, and each gadget is only responsible for context-independent state and behavior.

Structure

Flyweights are stored in a Factory's repository. The client restrains herself from creating Flyweights directly, and requests them from the Factory. Each Flyweight cannot stand on its own. Any attributes that would make sharing impossible must be supplied by the client whenever a request is made of the Flyweight. If the context lends itself to economy of scale (i.e. the client can easily compute or look-up the necessary attributes), then the Flyweight pattern offers appropriate leverage.

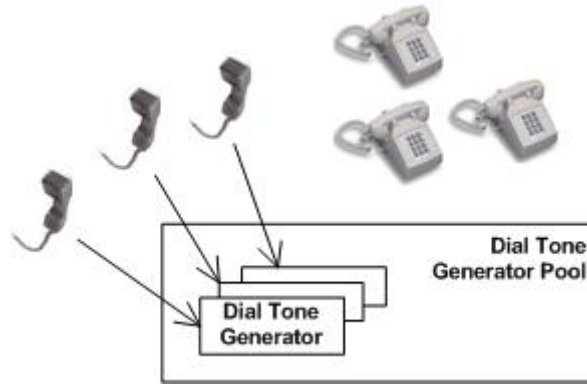


The Ant, Locust, and Cockroach classes can be light-weight because their instance-specific state has been de-encapsulated, or externalized, and must be supplied by the client.



Example

The Flyweight uses sharing to support large numbers of objects efficiently. The public switched telephone network is an example of a Flyweight. There are several resources such as dial tone generators, ringing generators, and digit receivers that must be shared between all subscribers. A subscriber is unaware of how many resources are in the pool when he or she lifts the handset to make a call. All that matters to subscribers is that a dial tone is provided, digits are received, and the call is completed.



Check list

1. Ensure that object overhead is an issue needing attention, and, the client of the class is able and willing to absorb responsibility realignment.
2. Divide the target class's state into: shareable (intrinsic) state, and non-shareable (extrinsic) state.
3. Remove the non-shareable state from the class attributes, and add it to the calling argument list of affected methods.
4. Create a Factory that can cache and reuse existing class instances.
5. The client must use the Factory instead of the new operator to request objects.
6. The client (or a third party) must look-up or compute the non-shareable state, and supply that state to class methods.

Rules of thumb

- Whereas Flyweight shows how to make lots of little objects, Facade shows how to make a single object represent an entire subsystem.
- Flyweight is often combined with Composite to implement shared leaf nodes.
- Terminal symbols within Interpreter's abstract syntax tree can be shared with Flyweight.
- Flyweight explains when and how State objects can be shared.

Private Class Data

Intent

- Control write access to class attributes
- Separate data from methods that use it
- Encapsulate class data initialization
- Providing new type of final - *final after constructor*

Problem

A class may expose its attributes (class variables) to manipulation when manipulation is no longer desirable, e.g. after construction. Using the private class data design pattern prevents that undesirable manipulation.

A class may have one-time mutable attributes that cannot be declared final. Using this design pattern allows one-time setting of those class attributes.

The motivation for this design pattern comes from the design goal of protecting class state by minimizing the visibility of its attributes (data).

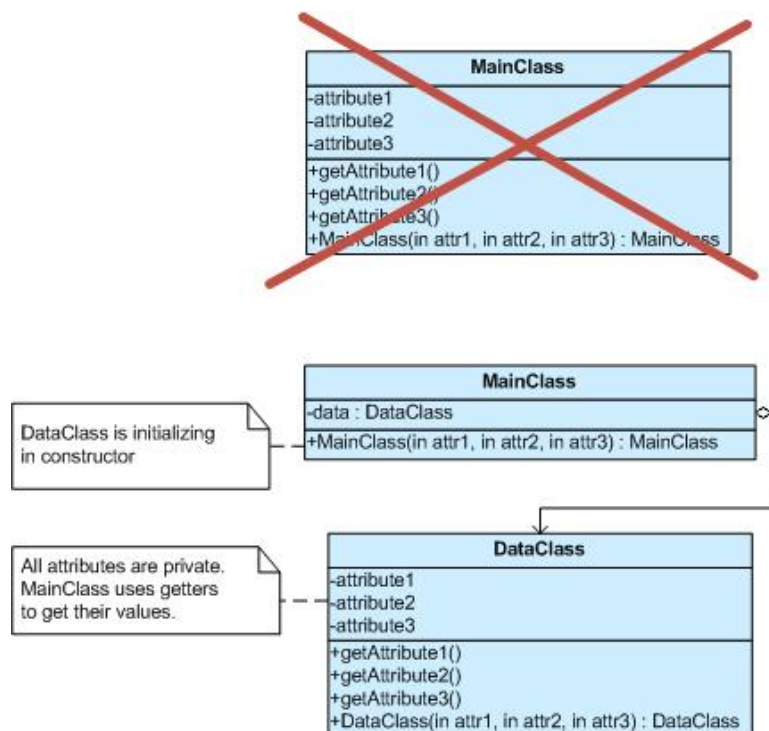
Discussion

The private class data design pattern seeks to reduce exposure of attributes by limiting their visibility.

It reduces the number of class attributes by encapsulating them in single Data object. It allows the class designer to remove write privilege of attributes that are intended to be set only during construction, even from methods of the target class.

Structure

The private class data design pattern solves the problems above by extracting a data class for the target class and giving the target class instance an instance of the extracted data class.



Check list

1. Create data class. Move to data class all attributes that need hiding.
2. Create in main class instance of data class.
3. Main class must initialize data class through the data class's constructor.
4. Expose each attribute (variable or property) of data class through a getter.
5. Expose each attribute that will change in further through a setter.

Proxy Design Pattern

Intent

- Provide a surrogate or placeholder for another object to control access to it.
- Use an extra level of indirection to support distributed, controlled, or intelligent access.
- Add a wrapper and delegation to protect the real component from undue complexity.

Problem

You need to support resource-hungry objects, and you do not want to instantiate such objects unless and until they are actually requested by the client.

Discussion

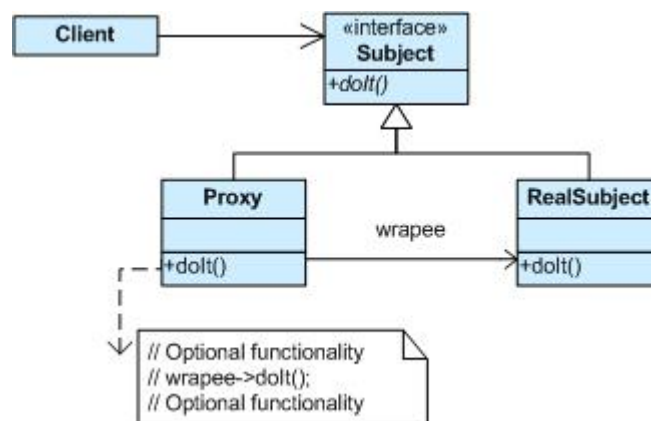
Design a surrogate, or proxy, object that: instantiates the real object the first time the client makes a request of the proxy, remembers the identity of this real object, and forwards the instigating request to this real object. Then all subsequent requests are simply forwarded directly to the encapsulated real object.

There are four common situations in which the Proxy pattern is applicable.

1. A virtual proxy is a placeholder for –expensive to create objects. The real object is only created when a client first requests/accesses the object.
2. A remote proxy provides a local representative for an object that resides in a different address space. This is what the –stub code in RPC and CORBA provides.
3. A protective proxy controls access to a sensitive master object. The –surrogate object checks that the caller has the access permissions required prior to forwarding the request.
4. A smart proxy interposes additional actions when an object is accessed. Typical uses include:
 - Counting the number of references to the real object so that it can be freed automatically when there are no more references (aka smart pointer),
 - Loading a persistent object into memory when it's first referenced,
 - Checking that the real object is locked before it is accessed to ensure that no other object can change it.

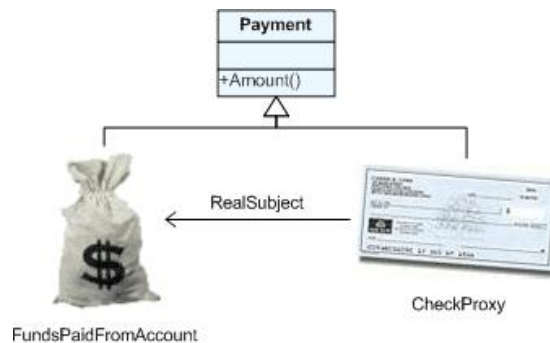
Structure

By defining a Subject interface, the presence of the Proxy object standing in place of the RealSubject is transparent to the client.



Example

The Proxy provides a surrogate or place holder to provide access to an object. A check or bank draft is a proxy for funds in an account. A check can be used in place of cash for making purchases and ultimately controls access to cash in the issuer's account.



Check list

1. Identify the leverage or -aspect|| that is best implemented as a wrapper or surrogate.
2. Define an interface that will make the proxy and the original component interchangeable.
3. Consider defining a Factory that can encapsulate the decision of whether a proxy or original object is desirable.
4. The wrapper class holds a pointer to the real class and implements the interface.
5. The pointer may be initialized at construction, or on first use.
6. Each wrapper method contributes its leverage, and delegates to the wrappee object.

Rules of thumb

- Adapter provides a different interface to its subject. Proxy provides the same interface. Decorator provides an enhanced interface.
- Decorator and Proxy have different purposes but similar structures. Both describe how to provide a level of indirection to another object, and the implementations keep a reference to the object to which they forward requests.

11. Explain briefly Behavioral Design Pattern with examples

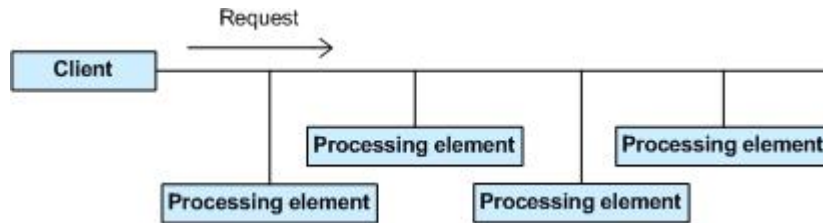
Chain of Responsibility

Intent

- Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.
- Launch-and-leave requests with a single processing pipeline that contains many possible handlers.
- An object-oriented linked list with recursive traversal.

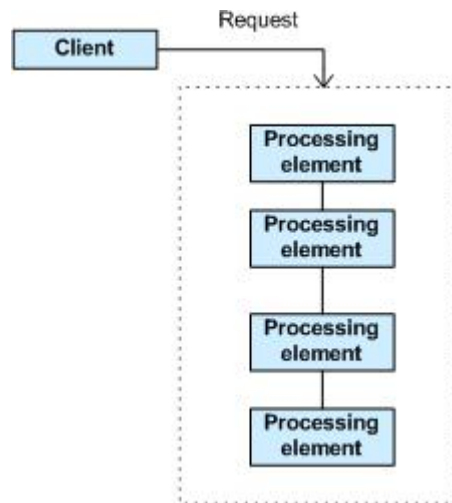
Problem

There is a potentially variable number of -handler|| or -processing element|| or -nodell objects, and a stream of requests that must be handled. Need to efficiently process the requests without hard-wiring handler relationships and precedence, or request-to-handler mappings.



Discussion

Encapsulate the processing elements inside a -pipeline abstraction; and have clients -launch and leave their requests at the entrance to the pipeline.



The pattern chains the receiving objects together, and then passes any request messages from object to object until it reaches an object capable of handling the message. The number and type of handler objects isn't known a priori, they can be configured dynamically. The chaining mechanism uses recursive composition to allow an unlimited number of handlers to be linked.

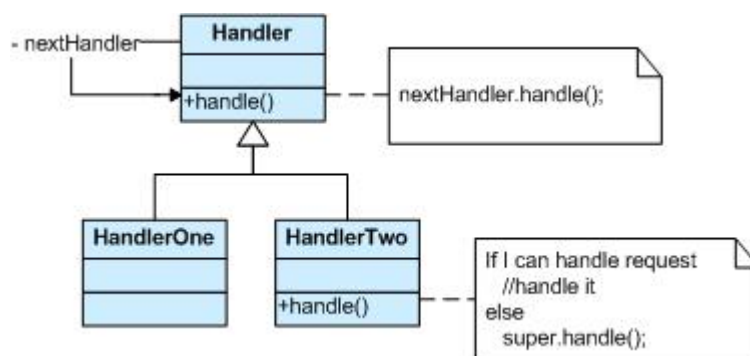
Chain of Responsibility simplifies object interconnections. Instead of senders and receivers maintaining references to all candidate receivers, each sender keeps a single reference to the head of the chain, and each receiver keeps a single reference to its immediate successor in the chain.

Make sure there exists a -safety net to -catch any requests which go unhandled.

Do not use Chain of Responsibility when each request is only handled by one handler, or, when the client object knows which service object should handle the request.

Structure

The derived classes know how to satisfy Client requests. If the -current object is not available or sufficient, then it delegates to the base class, which delegates to the -next object, and the circle of life continues.



Multiple handlers could contribute to the handling of each request. The request can be passed down the entire length of the chain, with the last link being careful not to delegate to a `-null next`.

Example

The Chain of Responsibility pattern avoids coupling the sender of a request to the receiver by giving more than one object a chance to handle the request. ATM use the Chain of Responsibility in money giving mechanism.



Check list

1. The base class maintains a `-next` pointer.
2. Each derived class implements its contribution for handling the request.
3. If the request needs to be `-passed on`, then the derived class `-calls back` to the base class, which delegates to the `-next` pointer.
4. The client (or some third party) creates and links the chain (which may include a link from the last node to the root node).
5. The client `-launches` and `leaves` each request with the root of the chain.
6. Recursive delegation produces the illusion of magic.

Rules of thumb

- Chain of Responsibility, Command, Mediator, and Observer, address how you can decouple senders and receivers, but with different trade-offs. Chain of Responsibility passes a sender request along a chain of potential receivers.
- Chain of Responsibility can use Command to represent requests as objects.
- Chain of Responsibility is often applied in conjunction with Composite. There, a component's parent can act as its successor.

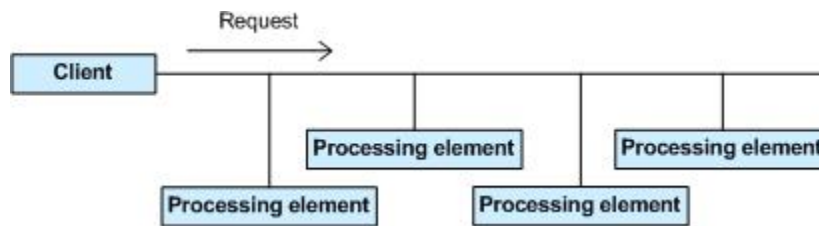
Chain of Responsibility

Intent

- Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.
- Launch-and-leave requests with a single processing pipeline that contains many possible handlers.
- An object-oriented linked list with recursive traversal.

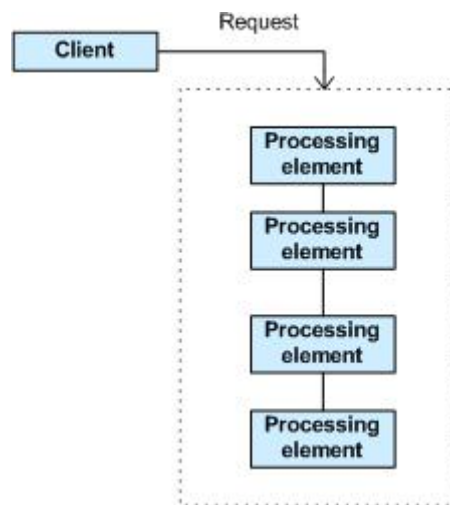
Problem

There is a potentially variable number of `-handler` or `-processing element` or `-node` objects, and a stream of requests that must be handled. Need to efficiently process the requests without hard-wiring handler relationships and precedence, or request-to-handler mappings.



Discussion

Encapsulate the processing elements inside a `-pipeline` abstraction; and have clients `-launch` and `leave` their requests at the entrance to the pipeline.



The pattern chains the receiving objects together, and then passes any request messages from object to object until it reaches an object capable of handling the message. The number and type of handler objects isn't known a priori, they can be configured dynamically. The chaining mechanism uses recursive composition to allow an unlimited number of handlers to be linked.

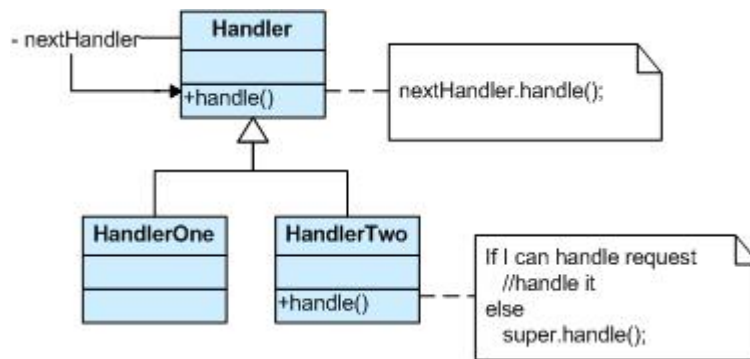
Chain of Responsibility simplifies object interconnections. Instead of senders and receivers maintaining references to all candidate receivers, each sender keeps a single reference to the head of the chain, and each receiver keeps a single reference to its immediate successor in the chain.

Make sure there exists a `-safety net` to `-catch` any requests which go unhandled.

Do not use Chain of Responsibility when each request is only handled by one handler, or, when the client object knows which service object should handle the request.

Structure

The derived classes know how to satisfy Client requests. If the `-current` object is not available or sufficient, then it delegates to the base class, which delegates to the `-next` object, and the circle of life continues.



Multiple handlers could contribute to the handling of each request. The request can be passed down the entire length of the chain, with the last link being careful not to delegate to a `-null` next.

Example

The Chain of Responsibility pattern avoids coupling the sender of a request to the receiver by giving more than one object a chance to handle the request. ATM use the Chain of Responsibility in money giving mechanism.



Check list

1. The base class maintains a `-next` pointer.
2. Each derived class implements its contribution for handling the request.
3. If the request needs to be `-passed on`, then the derived class `-calls back` to the base class, which delegates to the `-next` pointer.
4. The client (or some third party) creates and links the chain (which may include a link from the last node to the root node).
5. The client `-launches` and leaves each request with the root of the chain.
6. Recursive delegation produces the illusion of magic.

Rules of thumb

- Chain of Responsibility, Command, Mediator, and Observer, address how you can decouple senders and receivers, but with different trade-offs. Chain of Responsibility passes a sender request along a chain of potential receivers.
- Chain of Responsibility can use Command to represent requests as objects.
- Chain of Responsibility is often applied in conjunction with Composite. There, a component's parent can act as its successor.

Command Design Pattern

Intent

- Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.
- Promote –invocation of a method on an object to full object status
- An object-oriented callback

Problem

Need to issue requests to objects without knowing anything about the operation being requested or the receiver of the request.

Discussion

Command decouples the object that invokes the operation from the one that knows how to perform it. To achieve this separation, the designer creates an abstract base class that maps a receiver (an object) with an action (a pointer to a member function). The base class contains an execute() method that simply calls the action on the receiver.

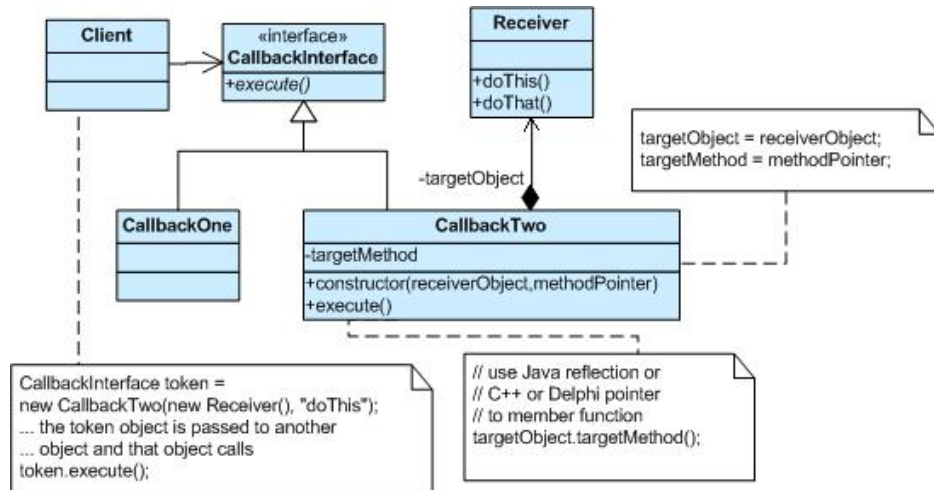
All clients of Command objects treat each object as a –black box by simply invoking the object's virtual execute() method whenever the client requires the object's –service.

A Command class holds some subset of the following: an object, a method to be applied to the object, and the arguments to be passed when the method is applied. The Command's –execute method then causes the pieces to come together.

Sequences of Command objects can be assembled into composite (or macro) commands.

Structure

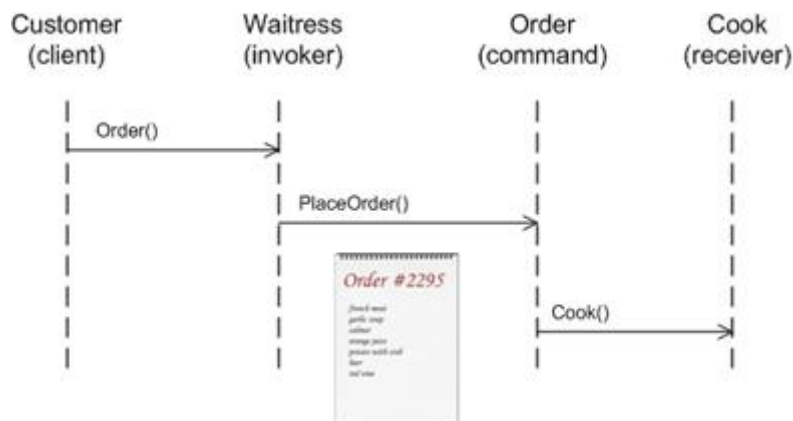
The client that creates a command is not the same client that executes it. This separation provides flexibility in the timing and sequencing of commands. Materializing commands as objects means they can be passed, staged, shared, loaded in a table, and otherwise instrumented or manipulated like any other object.



Command objects can be thought of as -tokens that are created by one client that knows what need to be done, and passed to another client that has the resources for doing it.

Example

The Command pattern allows requests to be encapsulated as objects, thereby allowing clients to be parameterized with different requests. The -check at a diner is an example of a Command pattern. The waiter or waitress takes an order or command from a customer and encapsulates that order by writing it on the check. The order is then queued for a short order cook. Note that the pad of -checks used by each waiter is not dependent on the menu, and therefore they can support commands to cook many different items.



Check list

1. Define a Command interface with a method signature like execute().
2. Create one or more derived classes that encapsulate some subset of the following: a -receiver object, the method to invoke, the arguments to pass.
3. Instantiate a Command object for each deferred execution request.
4. Pass the Command object from the creator (aka sender) to the invoker (aka receiver).
5. The invoker decides when to execute().

Rules of thumb

- Chain of Responsibility, Command, Mediator, and Observer, address how you can decouple senders and receivers, but with different trade-offs. Command normally specifies a sender-receiver connection with a subclass.

- Chain of Responsibility can use Command to represent requests as objects.
- Command and Memento act as magic tokens to be passed around and invoked at a later time. In Command, the token represents a request; in Memento, it represents the internal state of an object at a particular time. Polymorphism is important to Command, but not to Memento because its interface is so narrow that a memento can only be passed as a value.
- Command can use Memento to maintain the state required for an undo operation.
- MacroCommands can be implemented with Composite.
- A Command that must be copied before being placed on a history list acts as a Prototype.
- Two important aspects of the Command pattern: interface separation (the invoker is isolated from the receiver), time separation (stores a ready-to-go processing request that's to be stated later).

Interpreter Design Pattern

Intent

- Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.
- Map a domain to a language, the language to a grammar, and the grammar to a hierarchical object-oriented design.

Problem

A class of problems occurs repeatedly in a well-defined and well-understood domain. If the domain were characterized with a –language, then problems could be easily solved with an interpretation –engine.

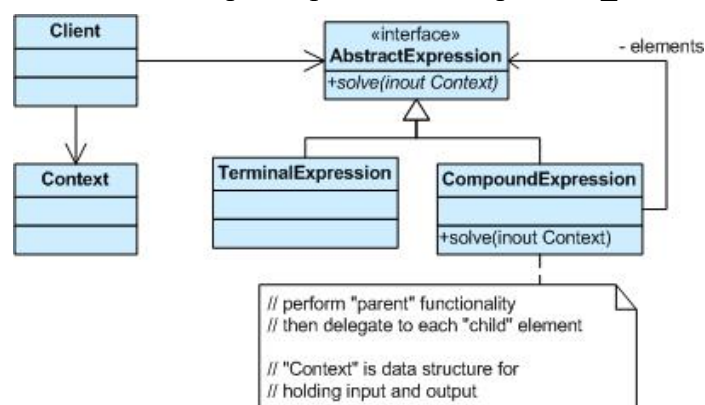
Discussion

The Interpreter pattern discusses: defining a domain language (i.e. problem characterization) as a simple language grammar, representing domain rules as language sentences, and interpreting these sentences to solve the problem. The pattern uses a class to represent each grammar rule. And since grammars are usually hierarchical in structure, an inheritance hierarchy of rule classes maps nicely.

An abstract base class specifies the method interpret(). Each concrete subclass implements interpret() by accepting (as an argument) the current state of the language stream, and adding its contribution to the problem solving process.

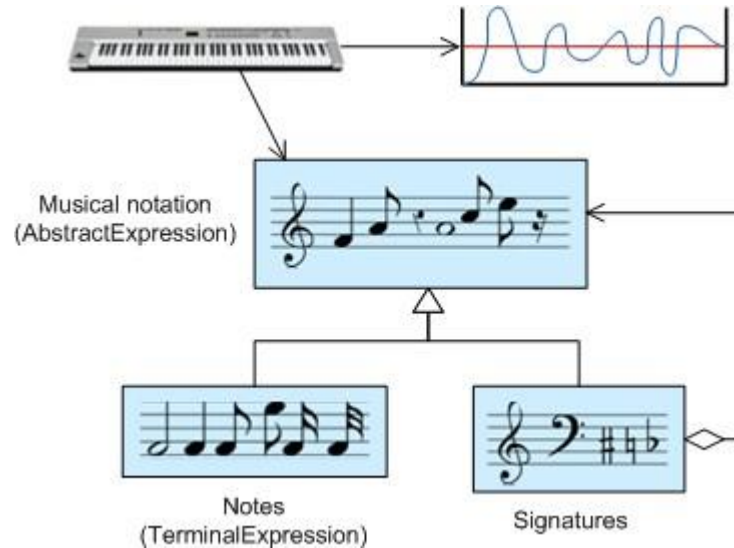
Structure

Interpreter suggests modeling the domain with a recursive grammar. Each rule in the grammar is either a `__composite` (a rule that references other rules) or a terminal (a leaf node in a tree structure). Interpreter relies on the recursive traversal of the Composite pattern to interpret the `__sentences` it is asked to process.



Example

The Interpreter pattern defines a grammatical representation for a language and an interpreter to interpret the grammar. Musicians are examples of Interpreters. The pitch of a sound and its duration can be represented in musical notation on a staff. This notation provides the language of music. Musicians playing the music from the score are able to reproduce the original pitch and duration of each sound represented.



Check list

1. Decide if a -little languagell offers a justifiable return on investment.
2. Define a grammar for the language.
3. Map each production in the grammar to a class.
4. Organize the suite of classes into the structure of the Composite pattern.
5. Define an interpret(Context) method in the Composite hierarchy.
6. The Context object encapsulates the current state of the input and output as the former is parsed and the latter is accumulated. It is manipulated by each grammar class as the -interpretingll process transforms the input into the output.

Rules of thumb

- Considered in its most general form (i.e. an operation distributed over a class hierarchy based on the Composite pattern), nearly every use of the Composite pattern will also contain the Interpreter pattern. But the Interpreter pattern should be reserved for those cases in which you want to think of this class hierarchy as defining a language.
- Interpreter can use State to define parsing contexts.
- The abstract syntax tree of Interpreter is a Composite (therefore Iterator and Visitor are also applicable).
- Terminal symbols within Interpreter's abstract syntax tree can be shared with Flyweight.
- The pattern doesn't address parsing. When the grammar is very complex, other techniques (such as a parser) are more appropriate.

Iterator Design Pattern

Intent

- Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.
- The C++ and Java standard library abstraction that makes it possible to decouple collection classes and algorithms.
- Promote to –full object status the traversal of a collection.
- Polymorphic traversal

Problem

Need to –abstract the traversal of wildly different data structures so that algorithms can be defined that are capable of interfacing with each transparently.

Discussion

–An aggregate object such as a list should give you a way to access its elements without exposing its internal structure. Moreover, you might want to traverse the list in different ways, depending on what you need to accomplish. But you probably don’t want to bloat the List interface with operations for different traversals, even if you could anticipate the ones you’ll require. You might also need to have more than one traversal pending on the same list. And, providing a uniform interface for traversing many types of aggregate objects (i.e. polymorphic iteration) might be valuable.

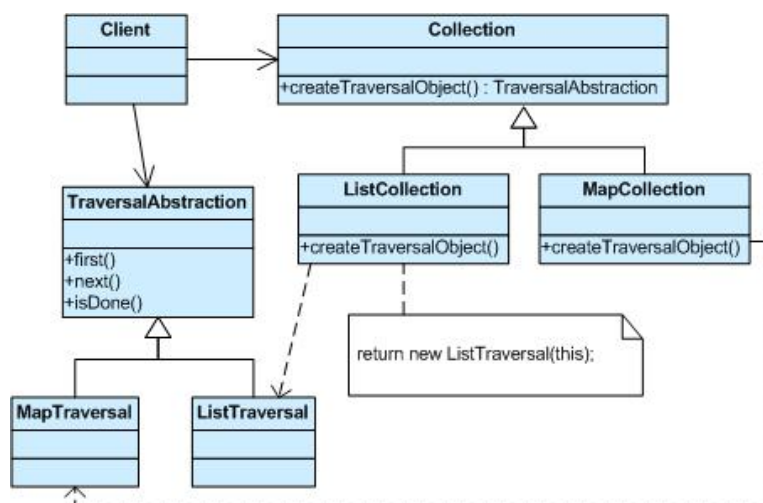
The Iterator pattern lets you do all this. The key idea is to take the responsibility for access and traversal out of the aggregate object and put it into an Iterator object that defines a standard traversal protocol.

The Iterator abstraction is fundamental to an emerging technology called –generic programming. This strategy seeks to explicitly separate the notion of –algorithm from that of –data structure. The motivation is to: promote component-based development, boost productivity, and reduce configuration management.

As an example, if you wanted to support four data structures (array, binary tree, linked list, and hash table) and three algorithms (sort, find, and merge), a traditional approach would require four times three permutations to develop and maintain. Whereas, a generic programming approach would only require four plus three configuration items.

Structure

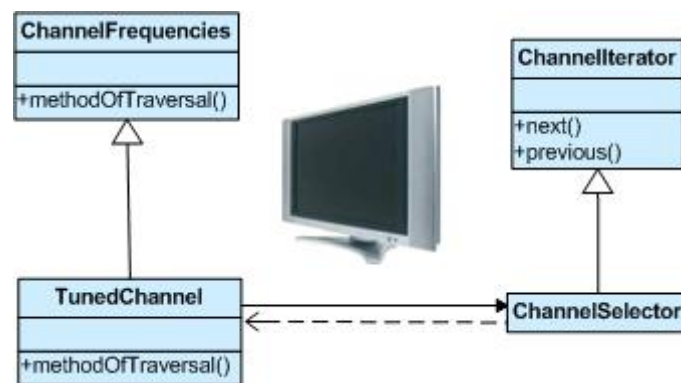
The Client uses the Collection class’ public interface directly. But access to the Collection’s elements is encapsulated behind the additional level of abstraction called Iterator. Each Collection derived class knows which Iterator derived class to create and return. After that, the Client relies on the interface defined in the Iterator base class.



Example

The Iterator provides ways to access elements of an aggregate object sequentially without exposing the underlying structure of the object. Files are aggregate objects. In office settings where access to files is made through administrative or secretarial staff, the Iterator pattern is demonstrated with the secretary acting as the Iterator. Several television comedy skits have been developed around the premise of an executive trying to understand the secretary's filing system. To the executive, the filing system is confusing and illogical, but the secretary is able to access files quickly and efficiently.

On early television sets, a dial was used to change channels. When channel surfing, the viewer was required to move the dial through each channel position, regardless of whether or not that channel had reception. On modern television sets, a next and previous button are used. When the viewer selects the -next|| button, the next tuned channel will be displayed. Consider watching television in a hotel room in a strange city. When surfing through channels, the channel number is not important, but the programming is. If the programming on one channel is not of interest, the viewer can request the next channel, without knowing its number.



Check list

1. Add a create_iterator() method to the -collection|| class, and grant the -iterator|| class privileged access.
2. Design an -iterator|| class that can encapsulate traversal of the -collection|| class.
3. Clients ask the collection object to create an iterator object.
4. Clients use the first(), is_done(), next(), and current_item() protocol to access the elements of the collection class.

Rules of thumb

- The abstract syntax tree of Interpreter is a Composite (therefore Iterator and Visitor are also applicable).
- Iterator can traverse a Composite. Visitor can apply an operation over a Composite.
- Polymorphic Iterators rely on Factory Methods to instantiate the appropriate Iterator subclass.
- Memento is often used in conjunction with Iterator. An Iterator can use a Memento to capture the state of an iteration. The Iterator stores the Memento internally.

Mediator Design Pattern

Intent

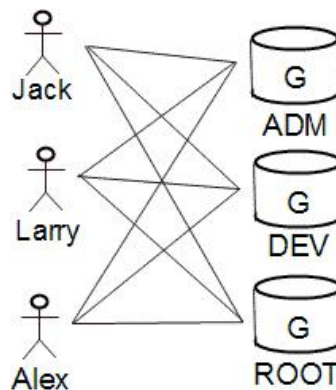
- Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.
- Design an intermediary to decouple many peers.
- Promote the many-to-many relationships between interacting peers to -full object status-.

Problem

We want to design reusable components, but dependencies between the potentially reusable pieces demonstrates the -spaghetti codell phenomenon (trying to scoop a single serving results in an -all or nothing clump-).

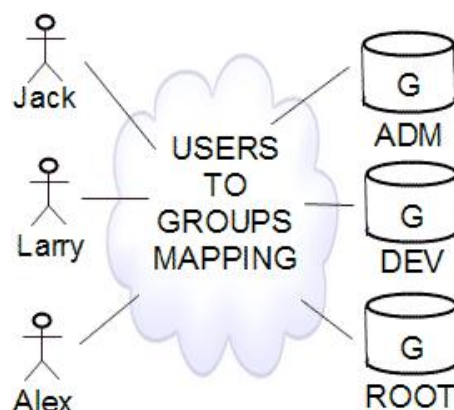
Discussion

In Unix, permission to access system resources is managed at three levels of granularity: world, group, and owner. A group is a collection of users intended to model some functional affiliation. Each user on the system can be a member of one or more groups, and each group can have zero or more users assigned to it. Next figure shows three users that are assigned to all three groups.



If we were to model this in software, we could decide to have User objects coupled to Group objects, and Group objects coupled to User objects. Then when changes occur, both classes and all their instances would be affected.

An alternate approach would be to introduce -an additional level of indirection- - take the mapping of users to groups and groups to users, and make it an abstraction unto itself. This offers several advantages: Users and Groups are decoupled from one another, many mappings can easily be maintained and manipulated simultaneously, and the mapping abstraction can be extended in the future by defining derived classes.

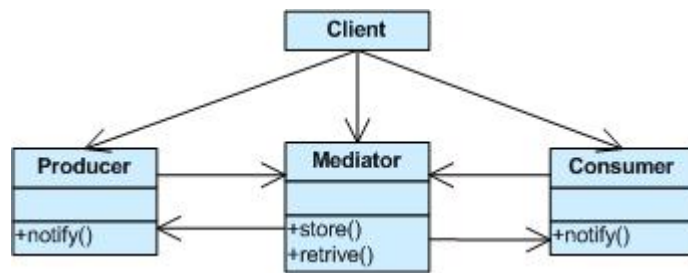


Partitioning a system into many objects generally enhances reusability, but proliferating interconnections between those objects tend to reduce it again. The mediator object: encapsulates all interconnections, acts as the hub of communication, is responsible for controlling and coordinating the interactions of its clients, and promotes loose coupling by keeping objects from referring to each other explicitly.

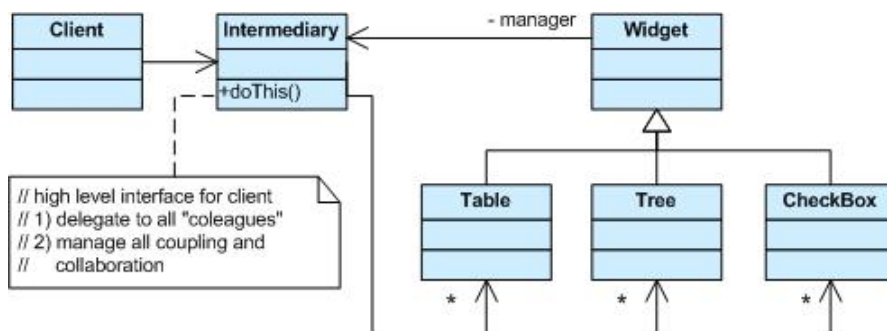
The Mediator pattern promotes a -many-to-many relationship network to -full object status. Modelling the inter-relationships with an object enhances encapsulation, and allows the behavior of those inter-relationships to be modified or extended through subclassing.

An example where Mediator is useful is the design of a user and group capability in an operating system. A group can have zero or more users, and, a user can be a member of zero or more groups. The Mediator pattern provides a flexible and non-invasive way to associate and manage users and groups.

Structure

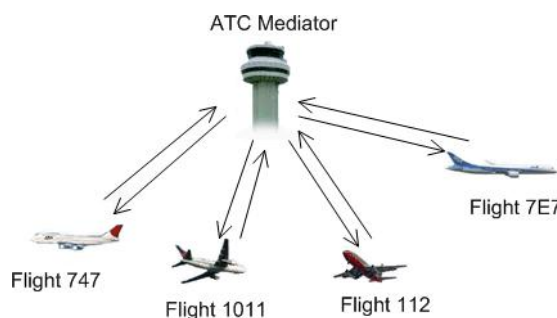


Colleagues (or peers) are not coupled to one another. Each talks to the Mediator, which in turn knows and conducts the orchestration of the others. The -many to many mapping between colleagues that would otherwise exist, has been -promoted to full object status. This new abstraction provides a locus of indirection where additional leverage can be hosted.



Example

The Mediator defines an object that controls how a set of objects interact. Loose coupling between colleague objects is achieved by having colleagues communicate with the Mediator, rather than with each other. The control tower at a controlled airport demonstrates this pattern very well. The pilots of the planes approaching or departing the terminal area communicate with the tower rather than explicitly communicating with one another. The constraints on who can take off or land are enforced by the tower. It is important to note that the tower does not control the whole flight. It exists only to enforce constraints in the terminal area.



Check list

1. Identify a collection of interacting objects that would benefit from mutual decoupling.
2. Encapsulate those interactions in the abstraction of a new class.
3. Create an instance of that new class and rework all `-peer` objects to interact with the Mediator only.
4. Balance the principle of decoupling with the principle of distributing responsibility evenly.
5. Be careful not to create a `-controller` or `-god` object.

Rules of thumb

- Chain of Responsibility, Command, Mediator, and Observer, address how you can decouple senders and receivers, but with different trade-offs. Chain of Responsibility passes a sender request along a chain of potential receivers. Command normally specifies a sender-receiver connection with a subclass. Mediator has senders and receivers reference each other indirectly. Observer defines a very decoupled interface that allows for multiple receivers to be configured at run-time.
- Mediator and Observer are competing patterns. The difference between them is that Observer distributes communication by introducing `-observer` and `-subject` objects, whereas a Mediator object encapsulates the communication between other objects. We've found it easier to make reusable Observers and Subjects than to make reusable Mediators.
- On the other hand, Mediator can leverage Observer for dynamically registering colleagues and communicating with them.
- Mediator is similar to Facade in that it abstracts functionality of existing classes. Mediator abstracts/centralizes arbitrary communication between colleague objects, it routinely `-adds value`, and it is known/referenced by the colleague objects (i.e. it defines a multidirectional protocol). In contrast, Facade defines a simpler interface to a subsystem, it doesn't add new functionality, and it is not known by the subsystem classes (i.e. it defines a unidirectional protocol where it makes requests of the subsystem classes but not vice versa).

Memento Design Pattern

Intent

- Without violating encapsulation, capture and externalize an object's internal state so that the object can be returned to this state later.
- A magic cookie that encapsulates a `-check point` capability.
- Promote undo or rollback to full object status.

Problem

Need to restore an object back to its previous state (e.g. `-undo` or `-rollback` operations).

Discussion

The client requests a Memento from the source object when it needs to checkpoint the source object's state. The source object initializes the Memento with a characterization of its state. The client is the `-caretaker` of the Memento, but only the source object can store and retrieve information from the Memento

(the Memento is –opaque to the client and all other objects). If the client subsequently needs to –rollback the source object’s state, it hands the Memento back to the source object for reinstatement.

An unlimited –undo and –redo capability can be readily implemented with a stack of Command objects and a stack of Memento objects.

The Memento design pattern defines three distinct roles:

1. *Originator* - the object that knows how to save itself.
2. *Caretaker* - the object that knows why and when the Originator needs to save and restore itself.
3. *Memento* - the lock box that is written and read by the Originator, and shepherded by the Caretaker.

Example

The Memento captures and externalizes an object’s internal state so that the object can later be restored to that state. This pattern is common among do-it-yourself mechanics repairing drum brakes on their cars. The drums are removed from both sides, exposing both the right and left brakes. Only one side is disassembled and the other serves as a Memento of how the brake parts fit together. Only after the job has been completed on one side is the other side disassembled. When the second side is disassembled, the first side acts as the Memento.

Check list

1. Identify the roles of –caretaker and –originator.
2. Create a Memento class and declare the originator a friend.
3. Caretaker knows when to –check point the originator.
4. Originator creates a Memento and copies its state to that Memento.
5. Caretaker holds on to (but cannot peek into) the Memento.
6. Caretaker knows when to –roll back the originator.
7. Originator reinstates itself using the saved state in the Memento.

Rules of thumb

- Command and Memento act as magic tokens to be passed around and invoked at a later time. In Command, the token represents a request; in Memento, it represents the internal state of an object at a particular time. Polymorphism is important to Command, but not to Memento because its interface is so narrow that a memento can only be passed as a value.
- Command can use Memento to maintain the state required for an undo operation.
- Memento is often used in conjunction with Iterator. An Iterator can use a Memento to capture the state of an iteration. The Iterator stores the Memento internally.

Observer Design Pattern

Intent

- Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
- Encapsulate the core (or common or engine) components in a Subject abstraction, and the variable (or optional or user interface) components in an Observer hierarchy.
- The -View|| part of Model-View-Controller.

Problem

A large monolithic design does not scale well as new graphing or monitoring requirements are levied.

Discussion

Define an object that is the -keeper|| of the data model and/or business logic (the Subject). Delegate all -view|| functionality to decoupled and distinct Observer objects. Observers register themselves with the Subject as they are created. Whenever the Subject changes, it broadcasts to all registered Observers that it has changed, and each Observer queries the Subject for that subset of the Subject's state that it is responsible for monitoring.

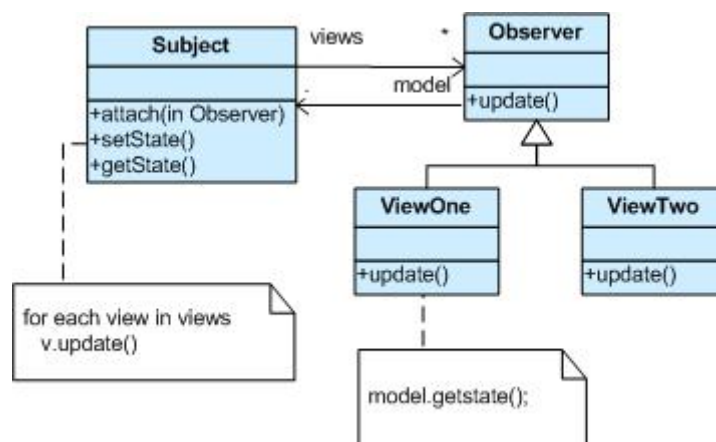
This allows the number and -type|| of -view|| objects to be configured dynamically, instead of being statically specified at compile-time.

The protocol described above specifies a -pull|| interaction model. Instead of the Subject -pushing|| what has changed to all Observers, each Observer is responsible for -pulling|| its particular -window of interest|| from the Subject. The -push|| model compromises reuse, while the -pull|| model is less efficient.

Issues that are discussed, but left to the discretion of the designer, include: implementing event compression (only sending a single change broadcast after a series of consecutive changes has occurred), having a single Observer monitoring multiple Subjects, and ensuring that a Subject notify its Observers when it is about to go away.

The Observer pattern captures the lion's share of the Model-View-Controller architecture that has been a part of the Smalltalk community for years.

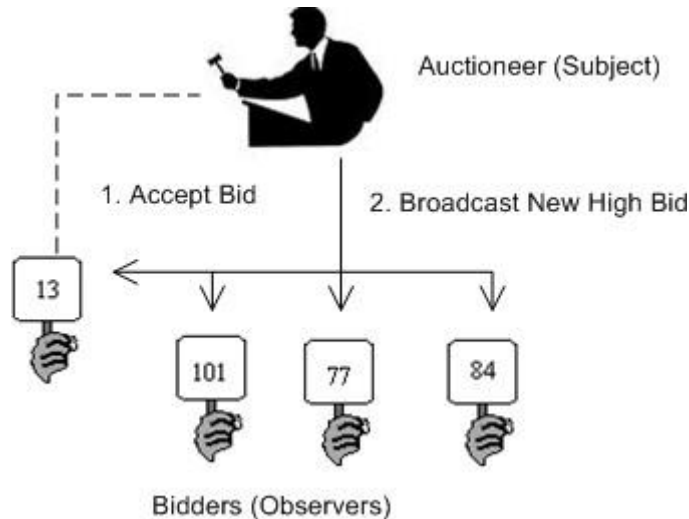
Structure



Subject represents the core (or independent or common or engine) abstraction. Observer represents the variable (or dependent or optional or user interface) abstraction. The Subject prompts the Observer objects to do their thing. Each Observer can call back to the Subject as needed.

Example

The Observer defines a one-to-many relationship so that when one object changes state, the others are notified and updated automatically. Some auctions demonstrate this pattern. Each bidder possesses a numbered paddle that is used to indicate a bid. The auctioneer starts the bidding, and –observes|| when a paddle is raised to accept the bid. The acceptance of the bid changes the bid price which is broadcast to all of the bidders in the form of a new bid.



Check list

1. Differentiate between the core (or independent) functionality and the optional (or dependent) functionality.
2. Model the independent functionality with a –subject|| abstraction.
3. Model the dependent functionality with an –observer|| hierarchy.
4. The Subject is coupled only to the Observer base class.
5. The client configures the number and type of Observers.
6. Observers register themselves with the Subject.
7. The Subject broadcasts events to all registered Observers.
8. The Subject may –push|| information at the Observers, or, the Observers may –pull|| the information they need from the Subject.

Rules of thumb

- Chain of Responsibility, Command, Mediator, and Observer, address how you can decouple senders and receivers, but with different trade-offs. Chain of Responsibility passes a sender request along a chain of potential receivers. Command normally specifies a sender-receiver connection with a subclass. Mediator has senders and receivers reference each other indirectly. Observer defines a very decoupled interface that allows for multiple receivers to be configured at run-time.
- Mediator and Observer are competing patterns. The difference between them is that Observer distributes communication by introducing –observer|| and –subject|| objects, whereas a Mediator object encapsulates the communication between other objects. We’ve found it easier to make reusable Observers and Subjects than to make reusable Mediators.

- On the other hand, Mediator can leverage Observer for dynamically registering colleagues and communicating with them.

State Design Pattern

Intent

- Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.
- An object-oriented state machine
- wrapper + polymorphic wrappee + collaboration

Problem

A monolithic object's behavior is a function of its state, and it must change its behavior at run-time depending on that state. Or, an application is characterized by large and numerous case statements that vector flow of control based on the state of the application.

Discussion

The State pattern is a solution to the problem of how to make behavior depend on state.

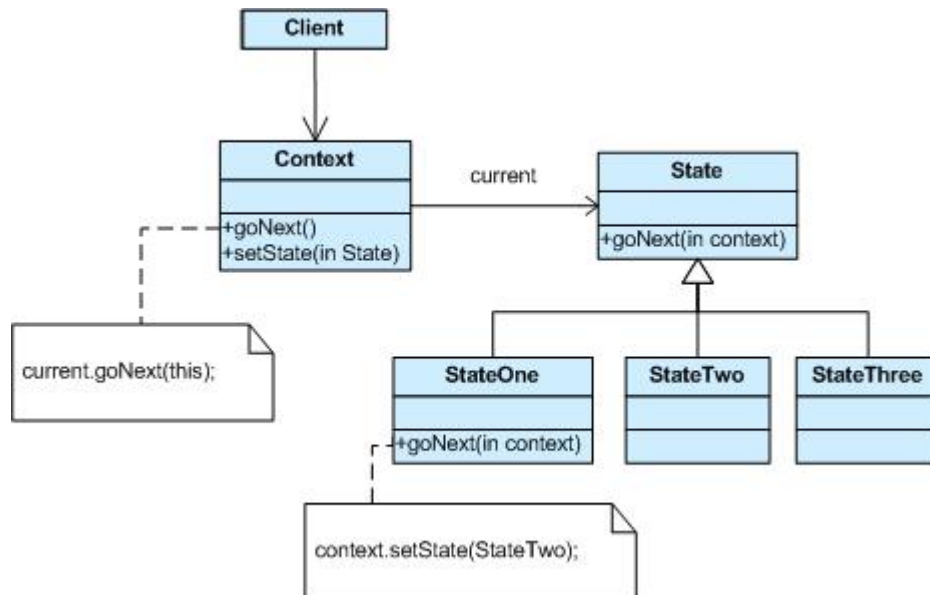
- Define a `-context` class to present a single interface to the outside world.
- Define a State abstract base class.
- Represent the different `-states` of the state machine as derived classes of the State base class.
- Define state-specific behavior in the appropriate State derived classes.
- Maintain a pointer to the current `-state` in the `-context` class.
- To change the state of the state machine, change the current `-state` pointer.

The State pattern does not specify where the state transitions will be defined. The choices are two: the `-context` object, or each individual State derived class. The advantage of the latter option is ease of adding new State derived classes. The disadvantage is each State derived class has knowledge of (coupling to) its siblings, which introduces dependencies between subclasses.

A table-driven approach to designing finite state machines does a good job of specifying state transitions, but it is difficult to add actions to accompany the state transitions. The pattern-based approach uses code (instead of data structures) to specify state transitions, but it does a good job of accommodating state transition actions.

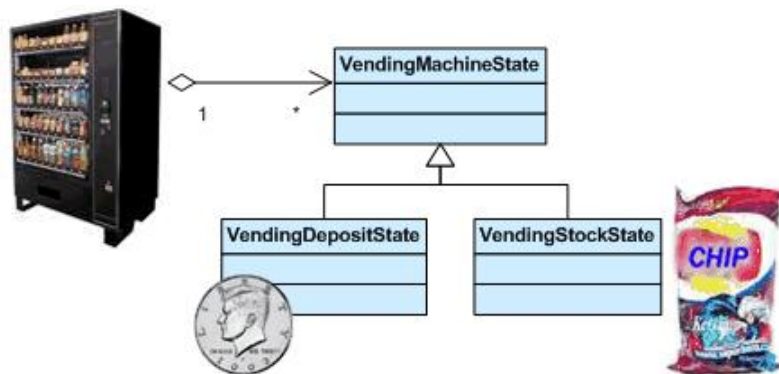
Structure

The state machine's interface is encapsulated in the `-wrapper` class. The wrappee hierarchy's interface mirrors the wrapper's interface with the exception of one additional parameter. The extra parameter allows wrappee derived classes to call back to the wrapper class as necessary. Complexity that would otherwise drag down the wrapper class is neatly compartmented and encapsulated in a polymorphic hierarchy to which the wrapper object delegates.



Example

The State pattern allows an object to change its behavior when its internal state changes. This pattern can be observed in a vending machine. Vending machines have states based on the inventory, amount of currency deposited, the ability to make change, the item selected, etc. When currency is deposited and a selection is made, a vending machine will either deliver a product and no change, deliver a product and change, deliver no product due to insufficient currency on deposit, or deliver no product due to inventory depletion.



Check list

1. Identify an existing class, or create a new class, that will serve as the `-state machine` from the client's perspective. That class is the `-wrapper` class.
2. Create a State base class that replicates the methods of the state machine interface. Each method takes one additional parameter: an instance of the wrapper class. The State base class specifies any useful `-default` behavior.
3. Create a State derived class for each domain state. These derived classes only override the methods they need to override.
4. The wrapper class maintains a `-current` State object.
5. All client requests to the wrapper class are simply delegated to the current State object, and the wrapper object's `this` pointer is passed.
6. The State methods change the `-current` state in the wrapper object as appropriate.

Rules of thumb

- State objects are often Singletons.
- Flyweight explains when and how State objects can be shared.

- Interpreter can use State to define parsing contexts.
- Strategy has 2 different implementations, the first is similar to State. The difference is in binding times (Strategy is a bind-once pattern, whereas State is more dynamic).
- The structure of State and Bridge are identical (except that Bridge admits hierarchies of envelope classes, whereas State allows only one). The two patterns use the same structure to solve different problems: State allows an object's behavior to change along with its state, while Bridge's intent is to decouple an abstraction from its implementation so that the two can vary independently.
- The implementation of the State pattern builds on the Strategy pattern. The difference between State and Strategy is in the intent. With Strategy, the choice of algorithm is fairly stable. With State, a change in the state of the -context|| object causes it to select from its -palettel|| of Strategy objects.

Strategy Design Pattern

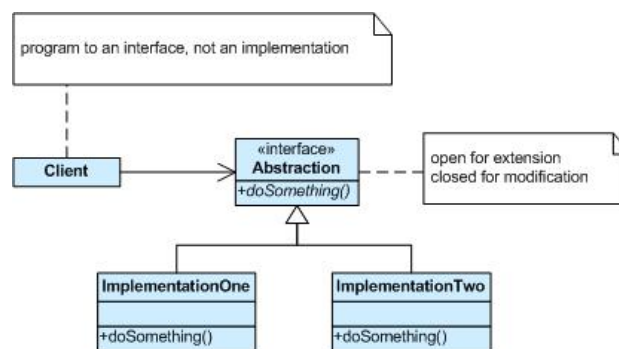
Intent

- Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from the clients that use it.
- Capture the abstraction in an interface, bury implementation details in derived classes.

Problem

One of the dominant strategies of object-oriented design is the -open-closed principle||.

Figure demonstrates how this is routinely achieved - encapsulate interface details in a base class, and bury implementation details in derived classes. Clients can then couple themselves to an interface, and not have to experience the upheaval associated with change: no impact when the number of derived classes changes, and no impact when the implementation of a derived class changes.



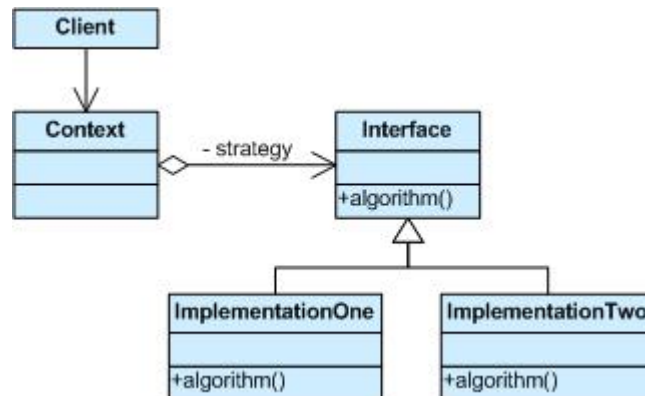
A generic value of the software community for years has been, -maximize cohesion and minimize coupling||. The object-oriented design approach shown in figure is all about minimizing coupling. Since the client is coupled only to an abstraction (i.e. a useful fiction), and not a particular realization of that abstraction, the client could be said to be practicing -abstract coupling||. an object-oriented variant of the more generic exhortation -minimize coupling||.

A more popular characterization of this -abstract coupling|| principle is -Program to an interface, not an implementation||.

Clients should prefer the -additional level of indirection|| that an interface (or an abstract base class) affords. The interface captures the abstraction (i.e. the -useful fiction||) the client wants to exercise, and the implementations of that interface are effectively hidden.

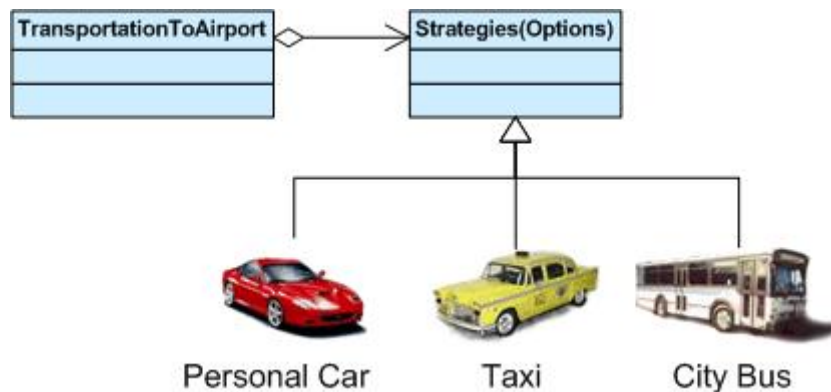
Structure

The Interface entity could represent either an abstract base class, or the method signature expectations by the client. In the former case, the inheritance hierarchy represents dynamic polymorphism. In the latter case, the Interface entity represents template code in the client and the inheritance hierarchy represents static polymorphism.



Example

A Strategy defines a set of algorithms that can be used interchangeably. Modes of transportation to an airport is an example of a Strategy. Several options exist such as driving one's own car, taking a taxi, an airport shuttle, a city bus, or a limousine service. For some airports, subways and helicopters are also available as a mode of transportation to the airport. Any of these modes of transportation will get a traveler to the airport, and they can be used interchangeably. The traveler must choose the Strategy based on tradeoffs between cost, convenience, and time.



Check list

1. Identify an algorithm (i.e. a behavior) that the client would prefer to access through a -flex point.
2. Specify the signature for that algorithm in an interface.
3. Bury the alternative implementation details in derived classes.
4. Clients of the algorithm couple themselves to the interface.

Rules of thumb

- Strategy is like Template Method except in its granularity.
- State is like Strategy except in its intent.
- Strategy lets you change the guts of an object. Decorator lets you change the skin.

- State, Strategy, Bridge (and to some degree Adapter) have similar solution structures. They all share elements of the `_handle/body` idiom. They differ in intent - that is, they solve different problems.
- Strategy has 2 different implementations, the first is similar to State. The difference is in binding times (Strategy is a bind-once pattern, whereas State is more dynamic).
- Strategy objects often make good Flyweights.

Important Questions

2 Marks

1. What is a Design Pattern?
2. What does design pattern provide?
3. Describe briefly about structural pattern
4. What is meant by behavioral pattern
5. What is meant by creational design pattern

11 Marks

1. What is Design Pattern? Explain its elements. (Ref.Pg.No 5, Qn.no.1)
2. Explain briefly Describing Design patterns. (Ref.Pg.No 6, Qn.no.2)
3. Explain the Catalog of Design patterns. (Ref.Pg.No 7, Qn.no.3)
4. How to organize the catalog? Explain in detail. (Ref.Pg.No 8, Qn.no.4)
5. How Design Patterns solve design problems? (Ref.Pg.No 10, Qn.no.5)
6. How to Select a Design Pattern? (Ref.Pg.No 15, Qn.no.6)
7. How to Use a Design Pattern? (Ref.Pg.No 16, Qn.no.7)
8. Explain Design Patterns and its uses. (Ref.Pg.No 17, Qn.no.8)
9. Explain briefly about Creational Design Patterns with examples. (Ref.Pg.No 20, Qn.no.9)
10. Explain briefly about Structural Design Pattern with examples. (Ref.Pg.No 30, Qn.no.10)
11. Explain briefly about Behavioral Design Pattern with examples(Ref.Pg.No 47, Qn.no.11)

Reference Books:

1. Ali Bahrami, Object Oriented systems development, Tata Mcgraw Hill Education Private Ltd,1999.
2. Carol Britton and Jill Doake, A student Gide to Object Oriented Development, Elsevier,Butterworth – Heinemann, Eighth Edition, 2007.
3. Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, Design Patterns – elements of reusable object oriented software, Addison Wesley, 1994.
4. Craig Larman,"Applying UML and Patterns: An Introduction to object-oriented Analysis and Design and iterative development", Third Edition, Pearson Education, 2005
5. Mike O'Docherty –Object-Oriented Analysis & design – understanding system development with UML 2.0, John Wiley, 2005.

6. Grady Booch, James Rumbaugh, Ivar Jacobson, -The UML user Guide, Pearson Education, 2005
7. Timothy C. Lethbridge, Robert Laganier- Object-Oriented Software Engineering – A practical software development using UML and Java, Tata McGraw-Hill, New Delhi, March 2003.
8. David William Brown, -An Introduction to Object Oriented Analysis Objects and UML in Plain English, 2nd Edition, Wiley, 2001.