Unified Modeling Language Guide Version 0.2.1 May 13, 2001

Introduction

UML

UML is a notation (graphical language with rules for creating analysis and design methods). UML is a supporting tool for the project.

The Design Process

The UML design process involves the creation of various graphical or text based documents. In UML, these documents are called **artifacts** and they describe the output of a step in the process. The UML design process has two main parts which are:

- Analysis - What is the problem?
- Design - How should the problem be solved?

The reason for this analysis and design process is to allow the project to be broken down into component parts which provide the following project characteristics:

- Detail is hidden
- The system is modular
- Components are connected and interact
- Layer complexity
- Components may be reusable in other products.
- Variations on a theme.

Iterations

The traditional design process involves a step by step process. This process basically includes:

1. Requirements Definition
2. Analysis
3. Design
4. Installation
5. Testing

This process does not allow for changes to the overall design once a particular phase is reached. Changes to a design once the testing phase is reached can be very difficult. This causes projects to have a high maintenance cost long after initial completion. UML and the iterative design process save money by allowing the design of systems that have low maintenance cost. The UML process allows for designs to have **iterative** processes. Essentially this means that the each iteration add some function (based on a use case) to the system. Additional features or functions are added in succeeding iterations. This allows a project to become active sooner and as feedback is given, additions or modifications can be made with less effort. Each iteration will have a time limit and will have specific goals. This is called "time boxing".

Design steps are broken into iterations which may be performed over and over. Each iteration includes:

1. Analysis
2. Design
3. Code + Design +Test + Integration

It is important to note that test and integration is done during each iteration. This means code is tested to the extent possible during the iteration to be sure of its quality. Also integration should be done during every iteration (Except the first) to get the code from each iteration working together. If integration is left to the end of the project, it may become very difficult. Iterations have a length of two to six weeks. Iterations are **"time boxed"** with each iteration having a fixed length with specific goals. Use case requirements are used to determine the development to be done during each iteration. Therefore an iteration may include one or more of the following:

- A complete use case.
- A simplified use case.
- The remainder of the use case after the simplified use case was completed in a previous iteration.
- More than one use case.

The idea is to schedule just enough work in each iteration to be able to complete the work by the end of the scheduled iteration.

Workflows

Workflows include:

1. Business modeling
2. Requirements
3. Analysis and Design
4. Implementation and Test
5. Deployment
6. Configuration and change management
7. Project Management
8. Environment - Required tools and software for the project work are made available to the team.

During each project phase and each iteration, there is a certain amount of work being done in each workflow category. Obviously during the inception and elaboration phases business modeling and requirements definitions are done more, however even during the successive phases such as construction, some business modeling or requirements definitions work may be necessary. This will be especially true if the plan changes, however even if the plan does not change some reassessment of requirements definitions should be done along the way.

Artifacts

During each iteration the below artifacts (documents) are created. They are normally created in the order shown.

1. Use Case Diagram
2. High Level Use Case Diagram
3. Expanded Use Case Diagram
4. System Sequence Diagram
5. Domain Model (Formally called conceptual model)
6. Operation contract
7. Collaboration Diagram (Also called interaction diagrams)
8. Design Class Diagrams

The first six steps are part of the analysis phase and the last two steps are part of the design phase.

The Unified Modeling Language

Project Start

Much of the content of the Project Process section of this document is purely connotative and is merely my opinion about what is helpful when breaking down a large project into smaller solvable problems. I believe one weakness of the Rational Unified Process is that it does not provide for well ordered methodologies to address how to logically break a large process into smaller sections. I will attempt to address that issue here.

Real World Projects

One thing to keep in mind about UML and the Rational Unified Process is that the real world is complex and project requirements will change. Also there will be unexpected events during development. Unexpected events tend to show up more often and with greater consequences in high risk parts of the project. There are not always exact rules to go by when creating artifacts when using UML and the Rational Unified Process. This is because project situations vary and your situation will depend on many factors including your current systems and staff. This document will give some general ideas and possibly some rules of thumb.

System Requirements

Determining system requirements is a very important part of the system design process. Although this document is not intended to describe this process, due to its importance, I will briefly touch on it here. Without proper determination of system requirements, the project is most likely doomed to fail. The steps in this process are generally as follows:

1. Determine the current capabilities of the present system from the user's point of view.
2. Describe (or reference documentation about) the equipment to be part of the system or that the system must interface with along with protocols and physical media used. Consider some typical calculations the system will do.
3. Get all input possible from user's and technical staff to determine additional capabilities and features to add. Also determine how strongly these features and capabilities are desired.
4. Determine the priorities of proposed features and capabilities.
5. Determine expected system performance including quality features ad discussed in the Management Guide in the management section.

6. Perform a preliminary risk assessment of all proposed features and system capabilities. Factors affecting risk include:
   - Difficulty and cost of the feature or capability.
   - How easily the feature can be supported by current technology.
   - Political implications of including the feature.
   - The risk of placing the wrong requirements or wrong emphasis of requirements on the system.
   - Whether the organization has staff with the technical skills to build and maintain the system.
7. Determine which system features and capabilities to include in the first iteration by using priority and risk assessment. As the risk rises and the priority of the feature drops, it is less desirable to include the feature in an early iteration. However, if a required feature of the project has a high degree of risk, the risk must be properly assessed through testing or whatever means possible early in the project cycle to determine if the project is actually feasable.

System characteristics that may be important to consider when developing requirements are:

- Scalability - Involves capacity management so increasing user demand may be met and managed efficiently. The system may supply performance information allowing administrators to change configuration to enhance performance for increased demand
- Database connectivity - Efficient data flow - The system can manage the requests to the database (if required) along with caching requests when appropriate, thereby relieving and managing some of the load on the database server (if the system interfaces to one).
- Security - There may be requirements to secure or encrypt information between different parts of the system or between the system and other systems.
- Integration - Provides support to integrate with other or older systems.
- Failure management.
- Stability of the system.
- How easy will the system be to use and learn?

When determining system requirements it is helpful to consider and outline the following:

- Goals of the system.
- System functions
- Categorize functions as essential, hidden and optional. Hidden functions are those functions that the user does not see.
- Determine system characteristics (otherwise known as attributes). These are performance considerations such as system boundary constraints, how fast the system operates, and how easy it is to use.

All system function characteristics must be characterized as required or desired.

Functional System Breakdown

One helpful way to simplify a system is to break it down into levels. Generally, an interface may be developed between these levels to keep changes in one level from effecting another level. Most networking protocols in use today use this method and it works quite well. Some levels may include:

- Display interface for the user.
- Data storage interface.
- Controller interface.

Another way to break a system down is to determine its areas of functionality and attempt to conceptually separate these parts of the system.

1. Draw the overall system use case diagram including initialization. Only consider how the main system will interact with actors on the outside of the system boundary.
2. Consider the main functional steps that the system must perform in order to make the use case happen. Below is listed some example functional areas systems must deal with.
    - User authentication - Be sure the user is authorized to use the system and determine their security access level. This will be compared to resources and resource lists to allow the user to see and list resources at their security level.
    - Resource posting - Determining resources or information provided by the system that is available to actors outside the system. How the actors will get this information? Also consider security issues.
    - Resource determination - Determine resources that are available to the system and information that the system can get from these resources.
    - Data storage
    - Data display
3. Consider the type of data that is saved or passed around the system. In light of the functional areas above, what kind of data should each functional area deal with or what parts of the data should those functional areas have access to?

Once the above steps/considerations have been made, it may be possible to break the system down into subsystems dependent on functionality with regard to dataflow. This can help simplify the system by breaking one large problem into a series of smaller problems. At this point it should be possible to create use case diagrams for each subsystem and begin the UML process.

Categories of UML Documents

	Static	Behaviorial
Requirements		Use Cases
		High Level Use Cases
Analysis	Domain Model (conceptual Diagram)	
		System Sequence Diagrams
		Operation Contracts
Design		Collaboration Diagrams
	Design Class Diagrams	

UML Use Case Diagram

A use case describes event sequences for an actor to use the system. It is a narrative description of the process. A use case is normally actor or event based. An actor will begin a process or an event will happen that the system must respond to.

Elements of a Use Case Diagram

- Boundary - System boundary can be a computer system, organization boundary, or department boundary. The system functions and actor may change depending on the system boundary location.
- Actors - An external entity (person or machine) that interacts with or uses the system.
- Sequence of events description - This describes a high level process of what an actor will do with a system. An actor may perform an event to start the system. This description does not represent individual steps in the process but represents the high level process itself.

To create a use case:

1. Define the system boundary
2. Identify actors

The actor must be able to walk away happy.

Use Case Categorization

This describes the importance of the function to the system.

- Primary - These functions are required and are common main processes.
- Secondary - These functions are secondary to the system or rarely occur. Don't need these functions in this iteration. This type of use case is rarely done.

Use Case Description Level (Abstraction)

- Essential - A general description of the business process. Do not include technology information. Use the 100 year rule where the information would be understood 100 years in the past and the future.
- Real - Design oriented, shows reports, examples. Uses technological descriptions. Real use cases are undesirable during analysis and should only be used during analysis for specific reasons. Real use cases are handy for requirements gathering.

Normally high level essential use cases and expanded essential use cases are done during the analysis phase of a project. A high level real use case is rarely done and an expanded real use case is done during the design phase only if necessary.

Use case detail level

- High Level - Brief with no detail
- Expanded - More detailed with information about every step in the process. Don't describe how the system responds.

General guidelines

When writing use cases, consider:

- Audience

- Purpose

Iteration

When making a use case diagrams the following two questions should be asked:

- What is the purpose of the system?
- What does a person using the system hope to accomplish?

UML High Level Use Case

A UML high level essential use case is a brief description of the main processes used to accomplish the system function. A high level use case each of the major processes in the system and therefore there is one high level use case for each of the major processes. The high level use case verbally describes the following:

- Name of the process.
- The actors involved with the initiating actor defined.
- The type of use case.
- The description of the process.

Syntax

Name: SystemAction

Actors: Person1 (Initiator), Person2, OtherSystem

Type: Primary

Description: The use case begins when Person1 arrives at the system with items and...

Example

In the case of a customer using a carwash to wash their car an example High Level Primary Use Case would be as follows:

Name: WashCar

Actors: Customer (Initiator)

Type: Primary

Description: The use case begins when the Customer arrives at the car wash with their car, pays for a car wash, washes their car, and the customer leaves with a clean car.

UML Expanded Use Case

A UML expanded essential use case is a more detailed description of the processes used to accomplish the system function. An expanded use case is built upon a high level use case. There are two sections to the high level use case which are a **heading** and a **body**. The heading describes the name, actors, description, type of use case, and more. The body describes typical events and alternatives to the typical events. This includes two or more columns with an actor action in one column and the system response in the other column. Typical events will happen 80% or more of the time and alternatives will happen only 20% or less of the time.

- Name of the process.
- The actors involved with the initiating actor defined.
- The description of the process from the high level use case.
- Type of use case such as primary/secondary, essential/real. Normally this is "primary, essential".
- Cross references - Reference any related system functions or use cases.
- Preconditions - Conditions that must be true before the use case can happen.

Syntax

Name: SystemAction

Actors: Person1 (Initiator), Person2, OtherSystem

Description: The use case begins when Person1 arrives at the system with items and...

Type: primary, essential

Cross references: System function1

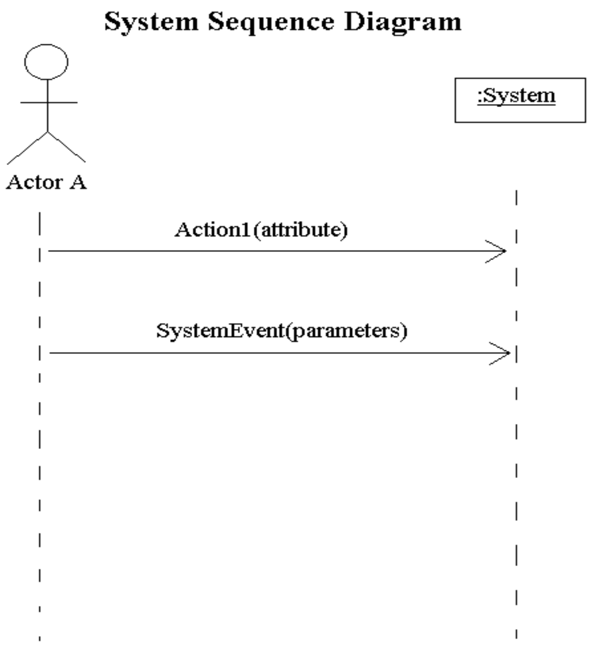
Preconditions: Resources must be available.

Actor Action	**System Response**
Typical Events:	
1.	
2.	
3.	
Alternatives:	
1.	

One step in the course of events is not normally categorized as a use case.

UML System Sequence Diagram

The UML system sequence diagram (SSD) illustrates events sequentially input from an external source to the system. The SSD will define the system events and operations. System sequence diagrams are a timeline drawing of an expanded use case. Events are related by time with the top events occurring first. System events are the important items. These are events that cause a system response.



Use case text may be placed on the left side of the system sequence diagram if desired. If this is done it is best if the use case information lines up with the events in the system sequence diagram.

There may be more than one actor to the system. An actor may be an external automated system that the system may communicate with. Automated actors or robots are shown as actors with a line horizontally through the head.

UML Domain Model

A UML domain model will relate objects in the system domain to each other. It will define concepts and terms. Objects in the domain model can be:

- Physical objects
- Abstract concepts

List Objects (Concepts)

To help the development of a domain model, it is important to identify nouns and noun phrases. Concepts that may not ultimately become objects may be listed for completeness and for discussion. The following types of concepts should be listed:

- Actor roles
- Events
- Transactions
    - Transaction line items
- Objects (physical)

    - Containers
        - Items (in container)
    - Other systems
    - Organizations
- Specification concepts should be used when redundant information is reduced through their use or deleting instances of what the specification describes can result in information loss.

Nouns can be taken from the requirements definitions and use case drawings. This means at this point all your use case drawings should be done. Actors should not be emphasized in the domain model.

If information from an object is derived from another object, that is reason to exclude it. However, if the object is required or is important to the use case, it should be included.

Domain Model Syntax

After the list of concepts is complete a domain model should be made. Consider which simple items should be attributes of objects. **The domain model is a static model.** Time flow, with sequence of events or information flow are **not** shown in the domain model. **Avoid showing procedural relationships.** This model does not include software. The objects in the domain model are candidates for programming objects.

UML Operation Contract

A UML Operation contract identifies system state changes when an operation happens. Effectively, it will define what each system operation does. An operation is taken from a system sequence diagram. It is a single event from that diagram. A domain model can be used to help generate an operation contract. The domain model can be marked as follows to help with the operation contract:

- Green - Pre existing concepts and associations.
- Blue - Created associations and concepts.
- Red - Destroyed concepts and associations.

Operation Contract Syntax

Name: appropriateName

Responsibilities: Perform a function

Cross References: System functions and Use Cases

Exceptions: none

Preconditions: Something or some relationship exists

Postconditions: An association was formed

When making an operation contract, think of the state of the system before the action (snapshot) and the state of the system after the action (a second snapshot). The conditions both before and after the action should be described in the operation contract. Do not describe how the action or state changes were done. The pre and post conditions describe state, not actions.

Typical postcondion changes:

- Object attributes were changed.
- An instance of an object was created.
- An association was formed or broken.

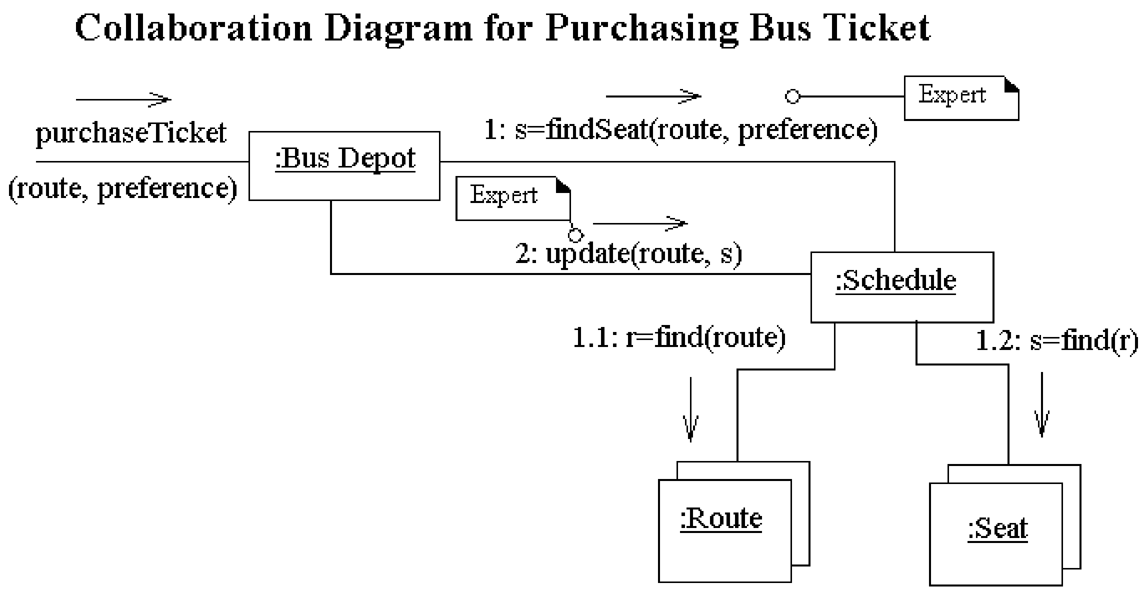
Postconditions are described in the past tense. They declare state changes to the system. Fill in the name, then responsibilities, then postconditions.

UML Collaboration Diagram

UML Collaboration diagrams (interaction diagrams) illustrate the relationship and interaction between software objects. They require use cases, system operation contracts, and domain model to already exist. The collaboration diagram illustrates messages being sent between classes and objects (instances). A diagram is created for each system operation that relates to the current development cycle (iteration).

When creating collaboration diagrams, patterns are used to justify relationships. Patterns are best principles for assigning responsibilities to objects and are described further in the section on patterns. There are two main types of patterns used for assigning responsibilities which are **evaluative** patterns and **driving** patterns.

Each system operation initiates a collaboration diagram. Therefore, there is a collaboration diagram for every system operation. An example diagram for purchasing a bus ticket.



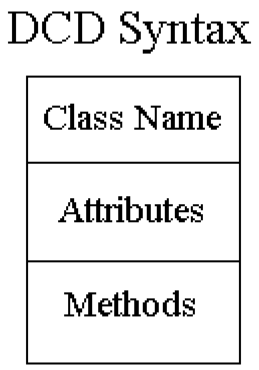
The route and seat objects are multi objects which means they are a collection of objects. The message, "purchaseTicket(route, preference) is the initializing message which is generated by the initializing actor. All other messages are generated by the system between objects. The initializing message is not numbered. The first message after the initializing message is numbered. Messages that are dependent on previous messages are numbered based on the number of the message they are dependent on. Therefore the message, "r=findRoute(route)" is numbered "1.1" since it is dependent on the message "s=findSeat(route, preference)". Any message path that is mutually exclusive is numbered with an "a" or "b". In finding route and seat messages, if finding a route or a seat were mutually exclusive, the numbering would be 1.1a and 1.1b. Patterns used for the association are associated with the message using a note.

- Creation messages - "create(parameter)"
- Iteration - Designated with a message line like: "1* [i:=1..5]: message3()"
    - Grouped - Although on two separate lines connecting objects both messages use the same variable such as the following two messages:
        - 1* [i=1..5]: message3()
        - 2* [i=1..5]: message3()
    - Separate - Written on two separate lines connecting objects each message uses a different variable name:
        - 1* [i=1..5]: message3()
        - 2* [j=1..5]: message3()
- Messages to self.
- Messages to multiobjects - The message is to the container, not the object in the container. (add, find, remove, next, size, contains)

The diagram should be split if it gets too large.

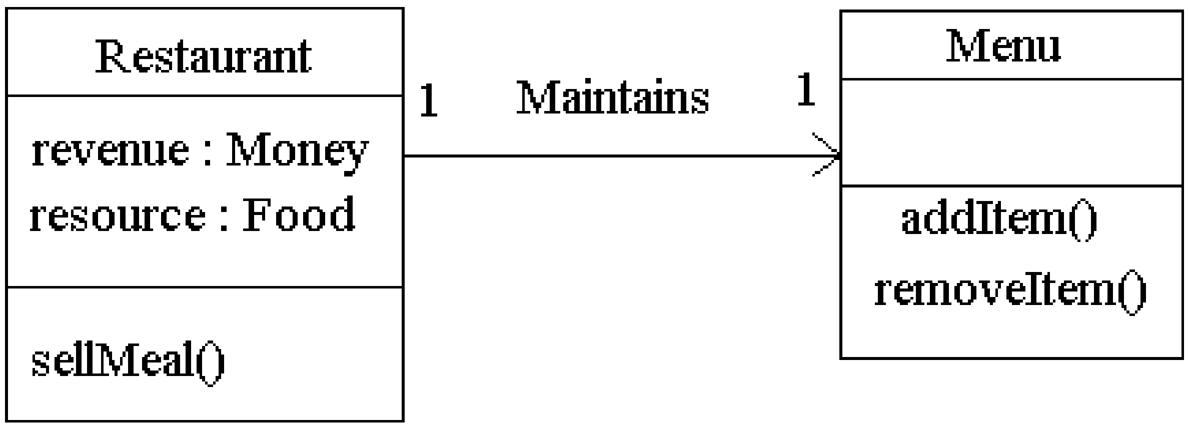
UML Design Class Diagram

UML design class diagrams (DCD) show software class definitions. They are based on the collaboration diagram. Attribute visibility is shown for permanent connections. Classes are shown with their simple attributes and methods listed.



Some attributes are depicted using associations (relationships) rather than actually being listed in the class block. These associated attributes refer to complex objects which should also be shown in the diagram. The collaboration diagram indicates methods to be contained in a class with methods posted as relationships. For instance the Schedule class has a findSeat(route, preference) method.

Partial Design Class Diagram



Temporary visibility between classes is depicted using dashed lines. Methods to be included in the class will include:

- Object creation methods with or without parameters
- A message being sent to an object.

Notation

The following optional characters in front of class attributes or methods depict meaning as shown below:

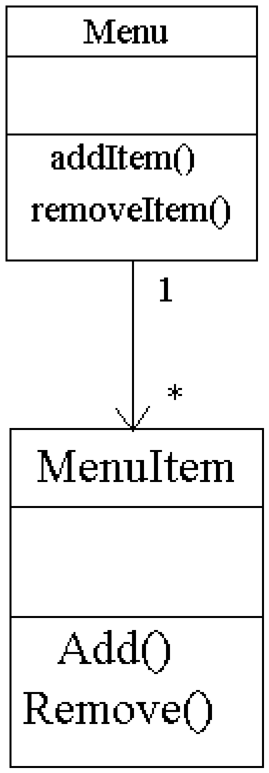
- - Private
- # Protected
- + Public

If the method or attribute is listed in italics text, it is **abstract**. If text is underlined, the method or attribute is **static**.

Multiplicity Example

Multiplicity is shown the same as in the Domain Model. The diagram below illustrates that the * next to the menuItem indicates multiplicity of the menuItem object.

Multiplicity Illustration



UML Package Diagram

UML Packages are a grouping of objects into sets of objects that provide related services. The package has responsibilities that are strongly related. The package has low coupling and low cohesion with respect to interfacing with other packages in the system.

UML Patterns

UML patterns are used to determine responsibility assignment for objects to interact in a collaboration diagram. Patterns are:

- Identified as a solution.
- A solution to a specific problem type. The solution has a name.
- Solution ideas from experts.

There are several different sets of patterns. The patterns used for collaboration diagram responsibility assognment are General Responsibility Assignment Software Patterns (GRASP). There are:

- Low coupling
- High cohesion
- Expert

- Creator
- Controller
- Pure fabrication
- Indirection
- Don't talk to strangers
- Polymorphism

Use Patterns:

- Evaluative - Patterns that indicate the degree of flexability of the design.
  - Low coupling - Coupling measures how much one class relies on another class or is connected to another class. If there are many lines between objects, coupling is high.
  - High cohesion - Keep the class as uncomplicated as possible. Don't perform functions not necessary for the respective class.
- Driving Patterns - Patterns used for problem solving.
  - Expert - The responsibility is assigned to the information expert. Who has the required information? Who is the expert with the requested information?
  - Creator - The one who creates something. Use the creator pattern when one or more of the following are true:
    - The creating class aggregates the class to be created.
    - The creating class contains the class to be created
    - The creating class closely uses the class to be created
    - The creating class records the class to be created
  - Controller - An interface between two layers such as the user and domain layer. This pattern supports low coupling and high cohesion. It handles event messages. Serves as a wrapper for a lower or domain layer. Events are received and passed through the controller.
  - Polymorphism - A class that is a subclass of a superclass performs the operation itself. This way an operation sent to the subclass can be the same operation but it is specific for the needs of that subclass. Same message, different method.
  - Pure fabrication - Make a class that performs some of the work in order to keep one class less complicated.
  - Indirection - Assign responsibility to an intermediate object.
  - Don't talk to strangers - Promote the interface. Talking to strangers causes high coupling since the object must interface with many objects. Messages cannot be sent to any other than the following from the method receiving the message:
    - A parameter of the method called by the message.
    - The self object which is the receiver.
    - A receiver attribute that references another object or an element in the referenced collection object.
    - An object that the method created.

    This pattern is being replaced by one called "Protected Variation" which is a synonym to the open/close principles. The open close principles mean the design is open to extension but it is closed to modification. Many times Generalization is used to support this. Additional generalizations (or subclasses) can be added to objects without changing surrounding objects.

- Design Patterns
    - Singleton (GoF) - Ensure a class has one instance and provide a global path to it. How to implement using java.
    - State (GoF) - The state pattern passes a reference to the state it wants to set.

      ```
      Class Depot
      {
        public static getInstance()  //Get instance of the single depot
        {
          return instance;
        }
        private static Depot instance = new Depot();
        private Depot()
        {
        }
      }
      ```

      Java creates the depot the first time getInstance() is called.

    - Prototype - Used to make a copy of an object so the copy can be modified without changing the original.
    - Flyweight - Don't make the object copy until you change it. References the template directly then apply prototype when a change is made. This pattern uses less memory by not actually making the copy until required.
    - Facade (GoF) - An object represents the system as a controller.
    - Command - A class for each message. The message is a command. The class has an execute method.
    - Forwarder-Receiver (Siemens) - For message handling.
    - Layers (Siemens) - Place login in the domain layer, not presentation layer.
    - Model-View Separation (Domain-Presentation Separation) - Domain objects do not directly send messages to presentation objects. View objects send messages to domain objects to get information to display.
    - Publish-Subscribe (Observer) - A means of allowing the presentation layer to get events to display from the domain layer without polling. An object in the presentation layer would subscribe to be notified of events in the domain layer.

Proxy/Remote Proxy (GoF) - Make a class that represents the actual class involved and let the local class communicate with the actual class.

The power of patterns comes with combining them. Design patterns are for specific coding solutions.

UML Visibility

One object must be able to see another object in order to use it. Ways to establish visibility of one object from another are:

- Parameter visibility - Pass a parameter reference to a method in the object. While that method is active, the object specified by the parameter is visible. (temporary visibility)

- Attribute visibility - Use an object attribute as a reference to the other object. (permanent visibility)
- Global - Have the object reference be globally visible. (permanent visibility)
- Local - Have the object reference be locally visible. Acquire an object reference through another object. Active only while the method is being run. (temporary visibility)

UML Layered Architecture

1. Presentation (user)
2. Application (Record transactions, authorize, etc.)
   - Application coordination
   - Domain
   - Services
3. Storage

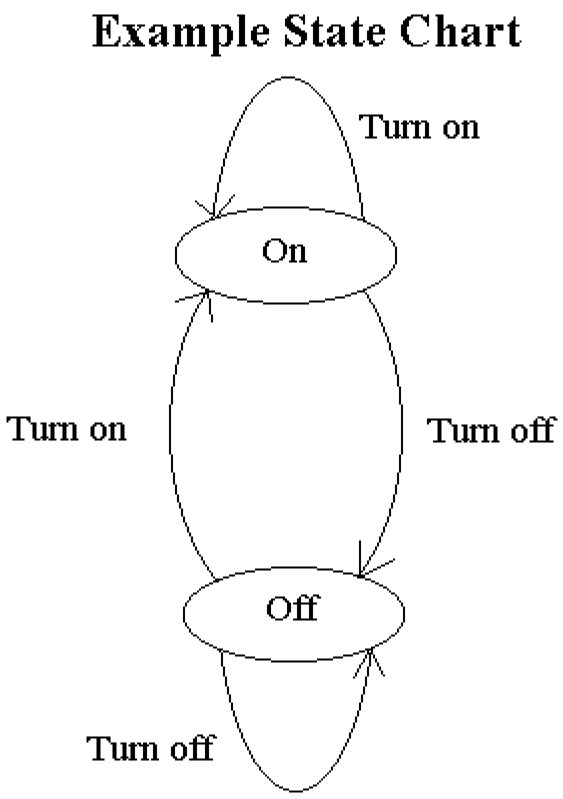
UML Use Case Relationships

This section describes how to relate use cases to each other. A dashed line between use cases is used to indicate these relationships.

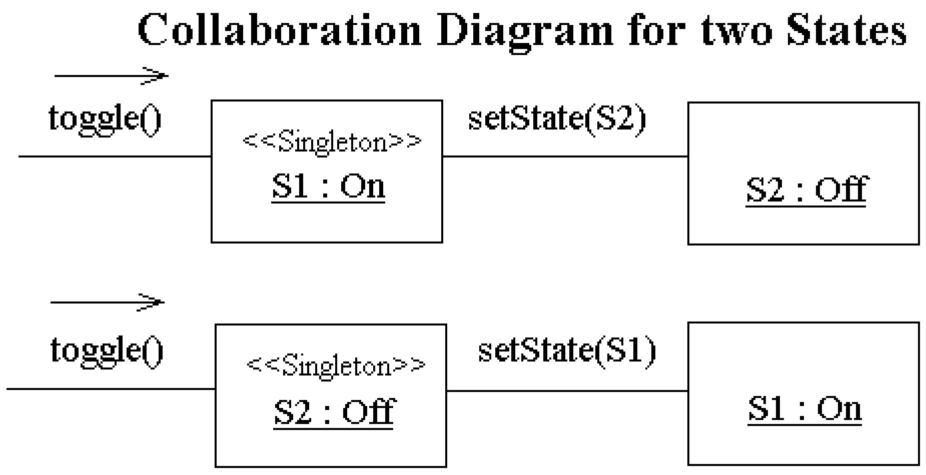
- Include - Subroutine - Factors out and organizes common subtasks. Extra behavior is added into a base use case. This behavior describes the insertion explicitly. The included use case is not a complete process. Use "include" when multiple use cases have a common function that can be used by all. Dashed line with arrow points to subroutine use case.
- Extend - Rarely used - Must perform a pre-task (Used only for critical order). The base and extended use cases are complete processes on their own. The base use case does not know about the extended use case. Arrow points to event that comes first.
- Generalization-specialization (Gen-Spec) - The gen-spec use case adds features to a generic use case. The gen-spec use case inherits features of the base use case. The gen spec can be used for use cases and actors since both can be specialized.

UML State Chart Diagram

UML State charts are not normally needed. They are needed when an object has a different reaction dependent on its state. The state design pattern uses polymorphism to define behavior.



The state pattern passes a reference to the state it wants to set. It normally will use the singleton pattern to pass this reference meaning the references to the states will be global. There will be a collaboration diagram for messages passed to each state.



UML Terms

- **Abstract class** - A class that will never be instantiated. An instance of this class will never exist.

- **Actor** - An object or person that initiates events the system is involved with.
- **Aggregation** - Is a part of another class. Shown with a hollow diamond next to the containing class in diagrams.
- **Artifacts** - Documents describing the output of a step in the design process. The description is graphic, textual, or some combination.
- **Association** - Describe important relationships between concepts or objects and may be bidirectional
- **Attributes** - Characteristics of an object which may be used to reference other objects or save object state information.
- **Class diagram** - Shows the system classes and relationships between them.
- **Collaboration diagram** - A diagram that shows how operations are done while emphasizing the roles of objects.
- **Concept** - A noun or abstract idea to be included in a domain model.
- **Construction phase** - The third phase of the Rational Unified Process during which several iterations of functionality are built into the system under construction. This is where the main work is done.
- **Domain** -The part of the universe that the system is involved with.
- **Elaboration phase** - The second phase of the Rational Unified Process that allows for additional project planning including the iterations of the construction phase.
- **Encapsulation** - Data in objects is private.
- **Generalization** - Indicates that one class is a subclass on another class (superclass). A hollow arrow points to the superclass.
- **GoF** - Gang of Four set of design patterns.
- **High cohesion** - A GRASP evaluative pattern which makes sure the class is not too complex, doing unrelated functions.
- **Low coupling** - A GRASP evaluative pattern which measures how much one class relies on another class or is connected to another class.
- **Inception phase** - The first phase of the Rational Unified Process that deals with the original conceptualization and beginning of the project.
- **Inheritance** - Subclasses inherit the attributes or characterics of their parent (superclass) class. These attributes can be overridden in the subclass.
- **Instance** - A class is used like a template to create an object. This object is called an instance of the class. Any number of instances of the class may be created.
- **Iteration** - A mini project section during which some small piece of functionality is added to the project. Includes the development loop of analysis, design and coding.
- **Message** - A request from one object to another asking the object receiving the message to do something. This is basically a call to a method in the receiving object.
- **Method** - A function or procedure in an object.
- Model
- **Multiplicity** - Shown in a domain model and indicated outside concept boxes, it indicates object quantity relationships to quantities of other objects.
- **Notation** - Graphical document with rules for creating analysis and design methods.
- **Object** - An instantiation of a class which includes attributes (variables) and methods (functions).
- **Package** - A group of UML elements that logically should be grouped together.
- **Pattern** - Solutions used to determine responsibility assignment for objects to interact. It is a name for a successful solution to a well known common problem.
- **Polymorphism** - Same message, different method. Also used as a pattern.
- **Reading Direction arrow** - Indicates the direction of a relationship in a domain model.

- **Role** - Used in a domain model, it is an optional description about the role of an actor.
- **Sequence diagram** - A diagram that shows how operations are done.
- **Statechart diagram** - A diagram that shows all possible object states.
- **Time boxing** - Each iteration will have a time limit with specific goals.
- **Transition phase** - The last phase of the Rational Unified Process during which users are trained on using the new system and the system is made available to users.
- **UML** - Unified Modeling Language utilizes text and graphic documents to enhance the analysis and design of software projects by allowing more cohesive relationships between objects.
- **Use case** - Describes event sequences for an actor to use the system. It is a narrative description of the process.
- **Workflow** - A set of activities that produces some specific result.

Recommended Reading

Title	**Author**	**Publisher**	**ISBN**
Applying UML and Patterns	Craig Larman	Prentice Hall	0137488807
Design Patterns	Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides	Addison-Wesley	0201633612
UML Distilled	Martin Fowler	Addison-Wesley	020165783x
Object Solutions	Grady Booch	Addison-Wesley	0805305947
Exploring Requirements: Quality Before Design	Donald Gause, Gerald Weinberg	Doiset House	0932633137
The Unified Modeling Language User Guide	Grady Booch, Ivar Jacobson, James Rumbaugh, Jim Rimbaugh	Addison-Wesley	0201571684
UML Toolkit	Hans-Erik Eriksson, Magnus Penker	John Wiley and sons	0471191612