6.1 Object-oriented (OO) approach

Modem corporations are faced with a profound dilemma. Increasingly, they are becoming information-based organizations, dependent on a continuous flow of data for virtually every aspect of their operations. Yet their ability to handle that data is breaking down because the volume of information is expanding faster than the capacity to process it. The result: corporations are drowning in their own data. The problem doesn't lie in hardware - computers continue to increase in speed and power at a phenomenal rate. The failure lies in software. Developing software to tap the potential of computers turns out to be a far greater challenge than building faster machines.

> Corporations are drowning in data. The failure lies in software. Most software is delivered late and over budget. We need better software and we need it faster. This is known as the software crisis.

Clearly something is wrong here. It's not that we haven't tried to improve our techniques for building software. Rather, it's taken us years to understand just how hard it is to build good software. Developing robust, large-scale software systems that can evolve to meet changing needs turns out to be one of the most demanding challenges in modem technology.

Traditional Approach

Building Program

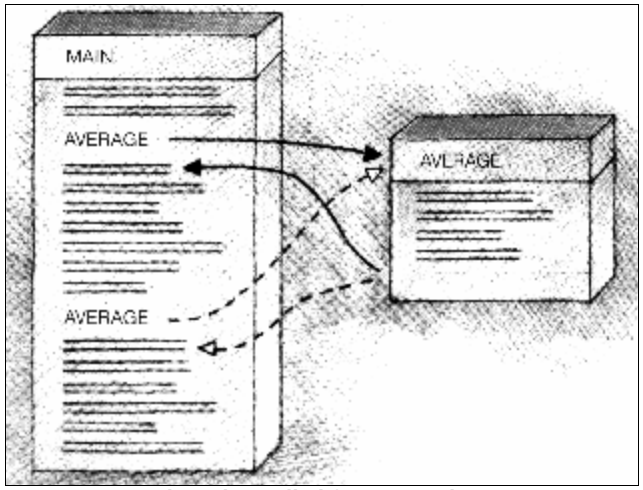
Single-procedure programs are usually written by a single programmer, who can maintain a mental image of the entire procedure, move instructions from place to place, and make design decisions freely as the program unfolds. Small groups of programmers can work in a similar style so long as all the members have free and open communication with each other. Larger programs can't be constructed as a single procedure like this. As the size of a program grows, so does the number of programmers required to build it. When a development group numbers in the tens or hundreds, the amount of communication required among the programmers becomes overwhelming. So many people are negotiating so many interacting decisions that no one has time to do the actual programming!

> Small programs can be built as a single procedure. This approach doesn't work for larger systems. Larger systems require modular programming. Subroutines support modular programming, but requires discipline.

Modular Programming

With this approach, large-scale programs will be broken down into smaller components that can be constructed independently, then combine them to form the complete system. The most elementary support for modular programming came with the invention of the subroutine in the early 1950s. A subroutine is created by pulling

a sequence of instructions out of the main routine and giving it a separate name; once defined, the subroutine can be executed simply by including its name in the program wherever it is required. Subroutines provide a natural division of labor: different programmers write the various subroutines, then assemble the completed subroutines into a working program.



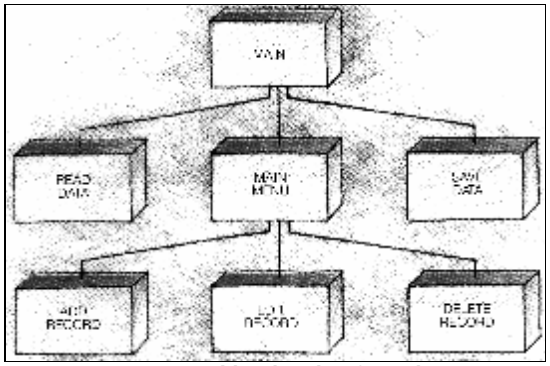
Subroutine called from two places

While subroutines provide the basic mechanism for modular programming, a lot of discipline is necessary to create well-structured software. Without that discipline, it is all too easy to write tortuously complicated programs that are resistant to change, difficult to understand, and nearly impossible to maintain. And that's what happened far too often during the early years of the industry.

> Structured programming provides functional decomposition approach that enforces discipline of building subroutines with the help of CASE tools.

Structured Programming

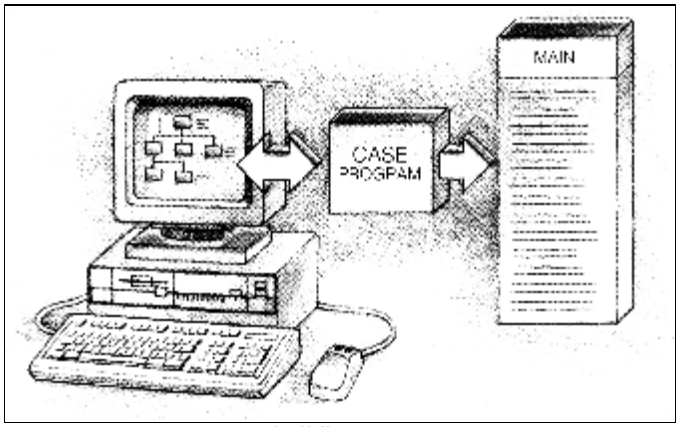
In the late 1960s, the generally poor state of software sparked a concerted effort among computer scientists to develop a more disciplined, consistent style of programming. The result of that effort was the refinement of modular programming into the approach known as structured programming. This approach relies on functional decomposition, a top-down approach to program design in which a program is systematically broken down into components, each of which is decomposed into subcomponents, and so on, down to the level of individual subroutines. Separate teams of programmers write the various components, which are later assembled into the complete program.



Program with 2 levels of nesting

Structured programming has produced significant improvements in the quality of software over the last twenty years, but its limitations are now painfully apparent. One of the more serious problems is that it's rarely possible to anticipate the design of a completed system before it's actually implemented.

Computer-Aided Software Engineering (CASE)



CASE building a program

The latest innovation in structured programming is computer-aided software engineering (CASE). With CASE, computers manage the process of functional decomposition, graphically defining subroutines in nested diagrams and verifying that all interactions between subroutines follow a correctly specified form. Advanced CASE systems can actually build complete, working programs from these diagrams once all the design information has been entered. Proponents of CASE herald the

automatic generation of programs from designs as a major breakthrough in software development. However, the process is not nearly as automatic as it first appears. In fact, a CASE tool doesn't create software at all - it simply translates the design for a system from graphical to textual form. Experience to date has shown that developing a complete graphical design for a program can be just as demanding and time-consuming as writing the program in the first place.

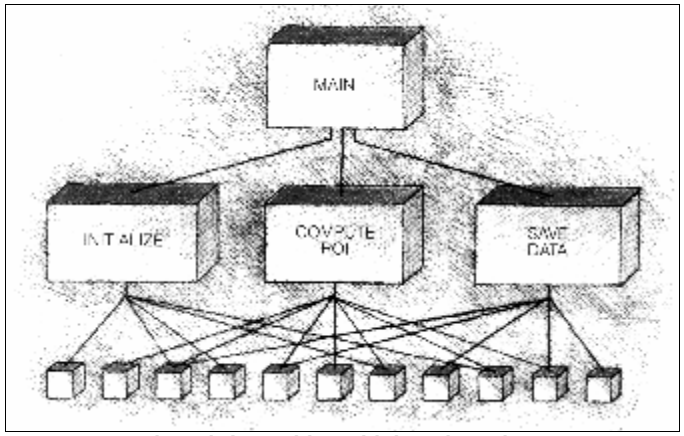
Managing Information

Most efforts to improve software development have focused on the modularization of procedures. But there is another component to software which, while less obvious, is no less important. That is the data, the collection of information operated on by the procedures. As the techniques of modular programming have evolved over the years, it has become apparent that data, too, must be modularized.

> Subroutines can share small amounts of data, but sharing too much data leads to problems. The solution lies in hiding information.

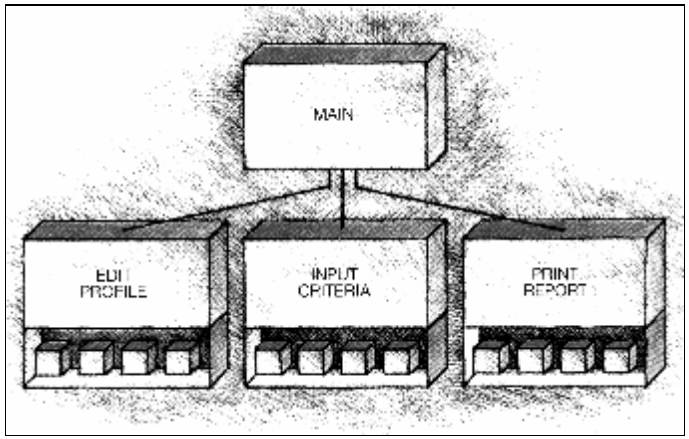
Data Within Programs

If a program requires only a few pieces of data to do its work, these pieces can safely be made available to all the different subroutines that make up the program. This arrangement is very convenient for programmers because the shared collection of data provides a communal "blackboard" on which the various subroutines can exchange information whenever they need to communicate.



Shared data with multiple subroutines

When the pieces of data number in the hundreds or thousands, however, this simple solution usually leads to mysterious errors and unpredictable behavior. The problem is that sharing data is a violation of modular programming, which requires that modules be as independent as possible.

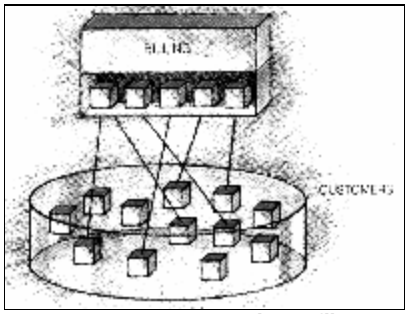


Local data within subroutines

Allowing modules to interact freely through shared data makes the actions of any one module directly dependent on the behavior of all the others. The solution to this problem is to modularize the data right along with the procedures. This is typically done by giving each subroutine its own local store of data which it alone can read and write. The strategy of **information hiding** minimize unwanted interactions between subroutines and allows them to be designed and maintained more independently.

Data Outside of Programs

Small programs often require only a few inputs and generate output that is meant to be consumed immediately. A program to calculate amortization tables, for example, might accept a base value and an amortization period from the keyboard, then print out a page of calculations. Programs of this sort don't need to store any data because they work with fresh information every time they are run. Larger programs, however, usually work with the same information over and over again. Inventory control programs, accounting systems, and engineering design tools couldn't function if they didn't have a way of preserving information from one run to the next. The simplest solution to the problem of keeping data around is to have a program store its data in an external file. When you finish running the program, it sends the data to the external file. When you start up the program again, it retrieves the data from the file. The use of a file also allows the program to work with more information than it could hold internally by reading and writing only a small portion of the file at any one time. External data files provide an adequate solution for information storage so long as data is accessed only by a single person using a single program. When data has to be shared, new problems arise.



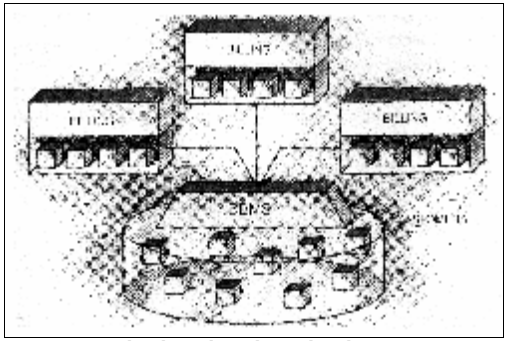
A program accessing a file

Some programs don't need to preserve data, but most large programs have to reuse data. Data can be preserved in files, but that doesn't work when data must be shared. Shared data requires a database management system.

Sharing Data

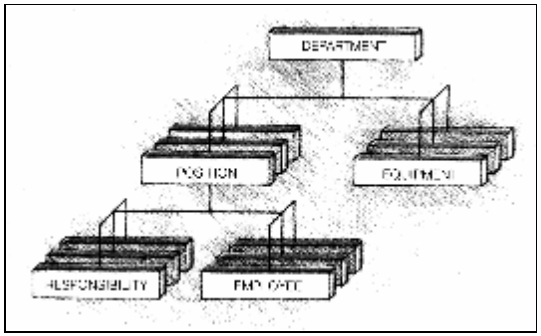
When different people can access the same file, there's always the possibility of one person changing information that others are currently using. Preventing this confusion turns out to be a fairly difficult technical problem that is not easily solved within a simple file system. Although some older programs still use files to store shared information, most multi-user systems are now built on top of special programs, called **database management systems** (**DBMSs**), that are designed to manage simultaneous access to shared data.

Databases contain structure as well as data. The network model extended the hierarchical model. Fixed data structures reduce flexibility. Relational Model removes most of the structure, but costly. OO is the new approach. Software objects combine procedures and data.



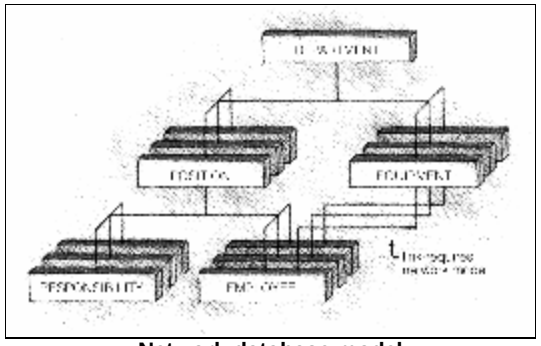
Sharing data in a database

Database management programs do more than just control access to data stored in files; they also store relationships among the various data elements. The earliest form of database manager, known as the **hierarchic model**, represented data items (called records) in tree structures. For example, a department could include records for the positions it contained and the equipment checked out to it. Each position, in turn, could be associated with a list of responsibilities and a list of employees in the department holding that position.



Hierarchic database model

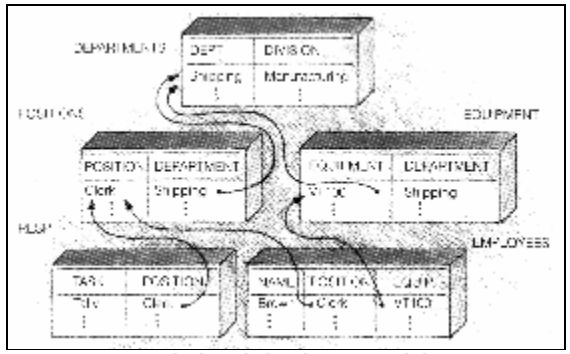
A more recent kind of database, the **network model**, allowed data to be interconnected freely, with no requirement that it fit into a tree structure. In the previous example, each piece of equipment could be associated with both a department and a fist of employees who were authorized to use it. This kind of association would not be permitted in the hierarchic model.



Network database model

The hierarchic and network database models made it easy to represent complex relationships among data elements, but there was a cost: accessing the data by the predefined relationship was slow and inefficient. Worse yet, the data structures were

hard to modify; changing these structures required system administrators to shut down the database and rebuild it.



Relational database model

A newer form of database manager, the **relational model**, addresses these problems by removing the information about complex relationships from the database. All data is stored in simple tables, with basic relationships among data items being expressed as references to values in other tables. For example, each entry in the equipment table would contain a value indicating which department it belonged to. Although the relational model is much more flexible than its predecessors, it pays a price for this flexibility. The information about complex relationships that was removed from the database must be expressed as procedures in every program that accesses the database, a dear violation of the independence required for modularity. There is also a performance penalty because the original data structures must be reassembled every time the data is accessed.

The Object-oriented Approach

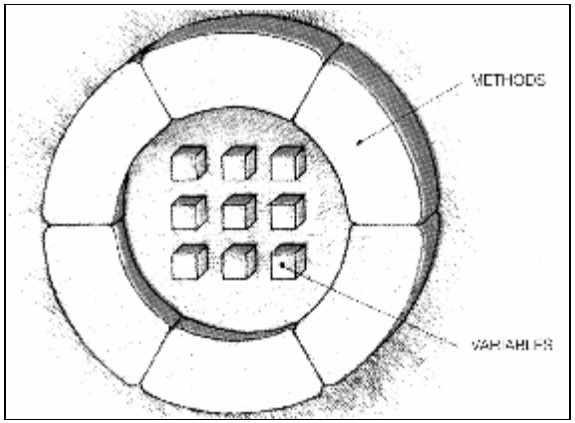
Despite all efforts to find better ways to build programs, the software crisis is growing worse with each passing year. Forty years after the invention of the subroutine, we are still building systems by hand, one instruction at a time. We've developed better methods for this construction process, but these methods don't work well in large systems. In addition, these methods usually produce defect-ridden software that's hard to modify and maintain. We need a new approach to building software, one that leaves behind the bricks and mortar of conventional programming and offers a truly better way to construct systems. This new approach must be able to handle large systems as well as small, and it must create reliable systems that are flexible, maintainable, and capable of evolving to meet changing needs. Object-oriented technology can meet these challenges and more. The remainder of this guide explains how this technology works and illustrates its potential to succeed where other methods have failed.

Objects make excellent software modules. It can interact in flexible way. These interactions are expressed as messages.

6.2 Introducing Object

This section introduces the three keys to understanding object-oriented technology - objects, messages, and classes. In fact, it's possible to apply OO approach with using no more than ten basic terms: object, method, message, class, subclass, instance, inheritance, encapsulation, abs-traction, and polymorphism.

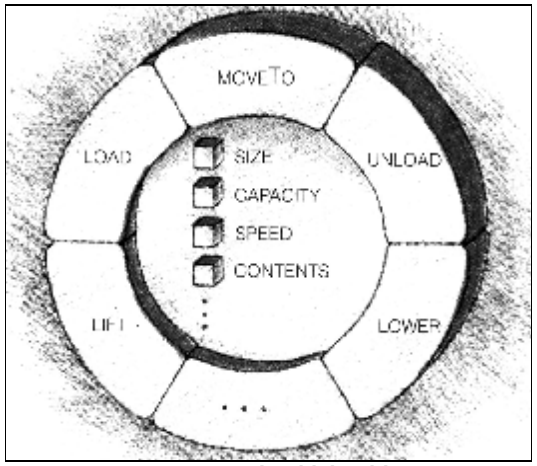
Although object-oriented technology has become popular recently, it's actually more than twenty years old. Virtually all the basic concepts of the object-oriented approach were introduced in the Simula programming language developed in Norway during the late 1960s.



An object

Inside Objects

The concept of software objects arose out of the need to model real-world objects in computer simulations. An **object** is software "package" that contains collection of related procedures and data. In the object-oriented approach, procedures go by a special name; they are called **methods**. In keeping with traditional programming terminology, the data elements are referred to as **variables** because their values can vary over time. For example, consider how you might represent an automated guided vehicle (AGV) in the simulation of a factory. The vehicle can exhibit a variety of behaviors, such as moving from one location to another or loading and unloading its contents. It must also maintain information about both its inherent characteristics (pallet size, lifting capacity, maximum speed, and so on) and its current state (contents, location, orientation, and velocity). To represent the vehicle as an object, you would describe its behaviors as methods and its characteristics as variables. During the simulation, the object would cant' out its various methods, changing its variables as needed to reflect the effects of its actions.



An automated vehicle object

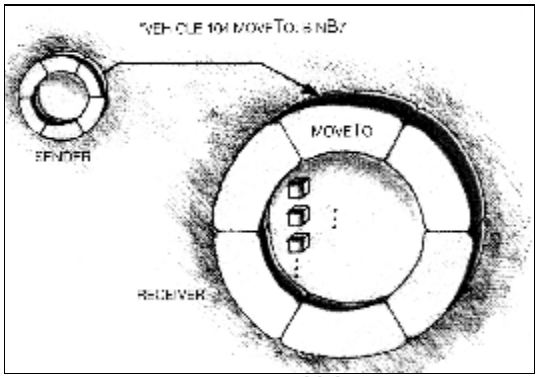
The concept of an object is simple yet powerful. Objects make ideal software modules because they can be defined and maintained independently of one another, with each object forming a neat, self-contained universe. Everything an object "knows" is expressed in its variables. Everything it can do is expressed in its methods.

Introducing Messages

Real-world objects can exhibit an infinite variety of effects on each other creating, destroying, lifting, attaching, buying, bending, sending, and so on. This tremendous variety raises an interesting problem - how can all these different kinds of interactions be represented in software?

Message support all possible interactions. There may be many objects of any given type.

The way objects interact with each other is to send each other messages asking them to carry out their methods. A **message** is simply the name of an object followed by the name of a method the object knows how to execute. If a method requires any additional information in order to know precisely what to do, the message includes that information as a collection of data elements called **parameters**. The object that initiates a message is called the **sender** and the object that receives the message is called the **receiver**.

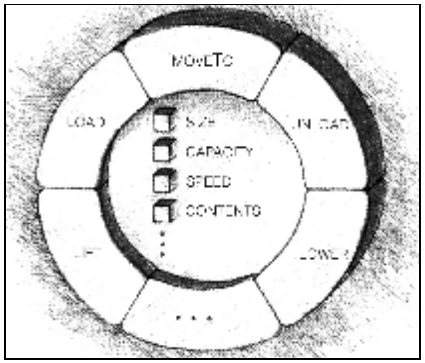


Message to vehicle104

To make an automated vehicle move to a new location, for example, some other object might send it the message:

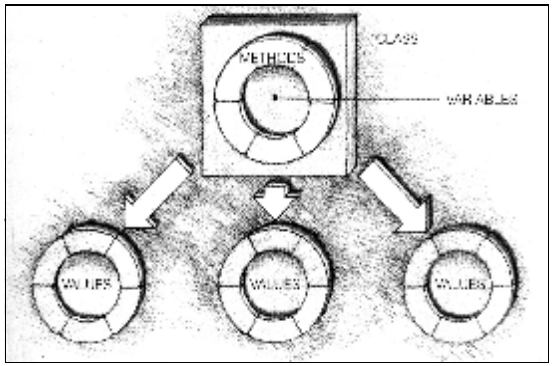
Vehicle104 **moveTo** ***binB7***

In this example, vehicle104 is the name of the receiver, moveTo is the method it is being asked to execute, and binB7 is a parameter telling the receiver where to go.



An automated vehicle object

An object-oriented simulation, then, consists of some number of objects interacting with each other by sending messages to one another. Since everything an object can do is expressed by its methods, this simple mechanism supports all possible interactions between objects.



A class and its instances

Introducing Classes

Sometimes a simulation involves only a single example of a particular kind of object. It is much more common, however, to need more than one object of each kind. An automated factory, for example, might have any number of guided vehicles. This possibility raises another concern: it would be extremely inefficient to redefine the same methods in every single occurrence of that object.

Classes defined groups of similar objects. Objects are instances of classes.

Creating Templates with Classes

A **class** is a template that defines the methods and variables to be included in a particular type of object. The descriptions of the methods and variables that support them are included only once, in the definition of the class. The objects that belong to a class, called instances of the class, contain only their particular values for the variables.

To continue the previous example, a simulated factory might contain many automated vehicles, each of which carried out the same actions and maintained the same kinds of information. The entire collection of vehicles could be represented by a class called *AutomatedUehicle*, and that class would contain the definitions of its methods and variables. The actual vehicles would be represented by instances of this class, each with its own unique name (vehicle101, vehicle102, vehide103...). Each instance would contain data values represented its own particular contents and location. When a vehicle received a message to carry out a method, it would turn to the class for the definition of that method and then apply the method to its own local data values.

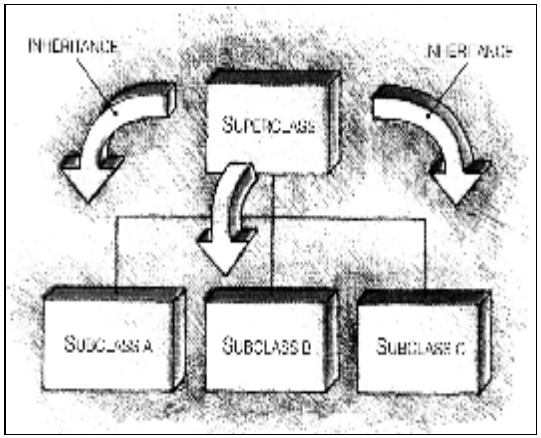
An object, then, is an instance of a particular class. Its methods and variables are defined in the class, and its values are defined in the instance. To keep my explanations simple, I usually talk about objects wherever possible, referring to classes and instances only when it's important to point out where the object's information is actually stored. For example, if I say that the object vehicle104 has a

method called *moveTo*, this is simply a more convenient way of saying that vehicle104 is an instance of a class that defines a method called *moveTo*.

> Classes can be defined in terms of each other. Inheritance is the mechanism that allow this. Complete hierarchies of classes can be built up.

Inheriting Class Information

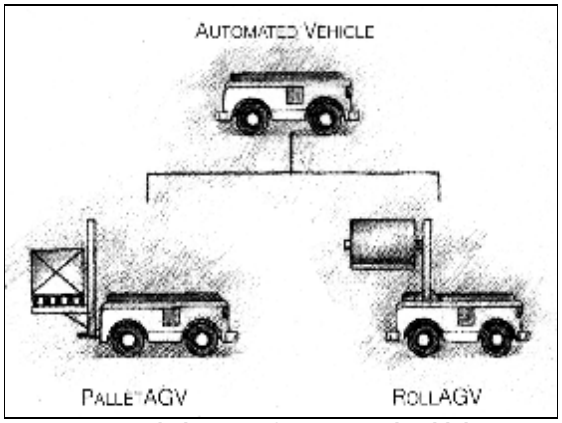
Inheritance is a mechanism whereby one class of objects can be defined as a special case of a more general class, automatically including the method and variable definitions of the general class. Special cases of a class are known as **subclasses** of that class; the more general class, in turn, is known as the **superclass** of its special cases. In addition to the methods and variables they inherit, subclasses may define their own methods and variables and may override any of the inherited characteristics. For example, the class *AutomatedVehicle* could be broken down into two subclasses, *PalletAGV* and *RollAGV* each of which inherited the general characteristics of the parent class. Either subclass could establish its own special characteristics by adding to the parent's definition or by overriding its behavior.



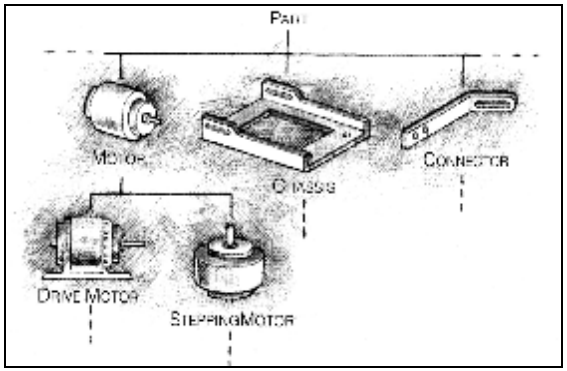
Subclasses of a superclass

Hierarchies of Classes

Classes can be nested to any degree, and inheritance will automatically accumulate down through all the levels. The resulting treelike structure is known as a **class hierarchy**. A class called *Part*, for example, could be broken down into special kinds of parts such as Motor, Chassis Connector, and so on. Motor, in turn, could be divided into *DriveMotor* and *SteppingMotor*, each of which could be broken down further as needed. An instance of, say, *VariableSpeedDriveMotor* would inherit all the characteristics of the Part class, as well as those of Motor and *DriveMotor*.



Two subclasses of automated vehicle



A class hierarchy for parts

he invention of the class hierarchy is the true genius of object-oriented technology. Human knowledge is structured in just this manner, relying on generic concepts and their refinement into increasingly specialized cases. Object-oriented technology uses the same conceptual mechanisms we employ in everyday life to build complex yet understandable software systems.

> Class hierarchies reflect human understanding. There are now many object-oriented languages, which reflect very different strategies.

Programming with Objects

Objects, messages, and classes are the central mechanisms of object-oriented technology. Traditionally, software has been viewed as a way to make a computer perform a particular task. This view is reflected in the overall progression of software

development projects: they begin with a specification of the problem to be solved, followed by a design for a system that produces the required behavior, and so on. There is a different mindset underlying object-oriented technology. Although the technology has spread far beyond its origins as a simulation language, programming with objects still retains the spirit of real-world simulation. The design of an object-oriented system begins not with the task to be performed, but rather with the aspects of the real world that need to be modeled in order to perform that task. Once these are correctly represented, the model can be used to solve a wide variety of tasks, including the original one. If you have a good model of your customers and your interactions with them, you can use this model equally well for billings, mailings, and ticklers.

> Object-oriented software models a system. Using models has many advantages. Conventional software us usually built from scratch.

The object-oriented approach to building software systems has many other advantages besides flexibility. Because the structure of the software reflects the real world, programmers can more easily understand and modify it in the future even if they aren't the same people who built the software in the first place. More importantly, the basic operations of a company tend to change much more slowly than the information needs of specific groups or individuals. This means that software based on corporate models will have a much longer life span than programs written to solve specific, immediate problems.

> Object-oriented systems are built by assembly. The approach has many important advantages. All are essential for modern system development.

Programming as Object Assembly

The process by which software is constructed is very different in the object-oriented approach. Most conventional software is still written from scratch, with very little reuse of procedures from earlier programs. Because these programs are written to solve very specific problems, it's usually easier to write new procedures than to convert existing ones.

Objects, by contrast, are general-purpose building blocks that model real-world entities rather than performing specific tasks. This makes them easy to reuse in subsequent projects, even if the objectives of the new projects are quite different. As more and more classes are accumulated, the software development effort begins to shift from creating new classes of objects to assembling existing ones in new ways. A mature object-oriented development team may devote as little as twenty percent of its time to creating new classes. The majority of its time is spent assembling proven components into new systems.

The Promise of the Approach

There is much more to the object-oriented approach than I have covered in this brief introduction, but some of the promise of this new way of thinking should now be apparent. Object-oriented technology offers some powerful techniques for creating flexible, natural software modules. Moreover, the focus on building general-purpose

models produces systems that are much easier to adapt to new demands. Finally, the extensive reuse of existing, proven components not only shortens development time, it also leads to more robust, error-free systems. Each of these benefits will play a crucial role in resolving the software crisis we now face.

> Placing data with behavior is called encapsulation that promotes information hiding.

6.3 UML

The Unified Modeling Language (UML) has been designed to help the participants in software development efforts build models will enable the team to visualize the system, specify the structure and behavior of that system, construct the system, and document the decisions made along the way.

Visualization

Models help a software development project team visualize the system they need to build that will satisfy the various requirements imposed by the project's stakeholders. The UML is specifically designed facilitate communication among the participants in a project. By offering a set of well defined diagrams, and precise notation to use on those diagrams, the UML gives everyone on the team the ability to understand what's going on with the system at any point in time with minimal risk of misinterpretation.

Specification

To specify a model, in UML terms, means to build it so that it's precise, unambiguous, and complete. Various aspects of the UML address the specification of the many decisions that have to be made as a system evolves.

Construction

The ultimate goal of a development project is working code.

Documentation

The combination of UML models and the other kinds of work products that come out of a development effort generally forms a solid set of project documentation.

Where did the UML come from?

The initial seed of Unified Method came in 1994, when Rumbaugh left General Electric to join Booch at Rational. The company made the first version of the method public a year later. There followed the 0.9 Unified Method documentation, and then version 1.0 of the Unified Method documentation, and then version 1.0 of the Unified Modeling Language. Version 1.0 was what rational submitted to the Object Management Group (OMG), the body that serves to define standards across many areas of computer science. UML 1.1 became the standard object-oriented modeling language in November of 1997.

Views of a System

In a software development project, each of the various stakeholders comes to the table with a different agenda. In turn, each stakeholder looks at the system from a different angle. The UML captures these angles as a set of five interlocking views. Each view reveals a particular set of aspects of the system from a given perspective. Fig. 1 shows the five views of a system's architecture that the UML defines.

- Use case view
- Design view
- Process view
- Implementation view
- Deployment view

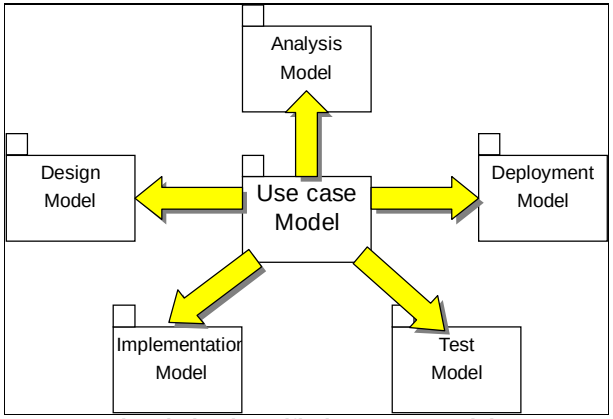
UML and Process

The Five workflows

There are five workflows within the unified process.

Requirements

The primary activities of the requirement workflow are aimed at building the use case model, which captures the functional requirements of the system being modeled. The use case model also serves as the foundation for all other development work. Diagram below shows how the use case model influences the other five UML models.



The six basic Unified Process Models

Analysis

The primary activities of the Analysis workflow are aimed at building the analysis model, which helps the developers refine and structure the functional requirements captured within the use case model.

Design

The primary activities of the Design workflow are aimed at building the design model, which describes the physical realizations of the use cases, from the use case model, and also the contents of the analysis model. The design workflow also focuses on the deployment model, which defines the physical organization of the system in terms of computational nodes.

Implementation

The primary activities of the Implementation workflow are aimed at building the implementation model, which describes how the elements of the design model are packaged into software components, such as source code file, dynamic link libraries (DLLs), and Enterprise Java Beans (EJBs).

Test

The primary activities of the Test workflow are aimed at building the test model, which describes how integration and system tests will exercise executable components from the implementation model. The test model describes how the team will perform those tests as well as unit tests.

Iterations and Increments

Each of the Unified Process's phases is divided into iterations. An iteration is simply a mini-project that's part of a workflow. Each iteration results in an increment. This is a release of the system that contains added and/or improved functionality over and above the previous release.

6.4 Basic Modeling Concepts

Objects

An object is simply a real-world thing or concept. There are three essential aspects of objects.

An object has identity – generally take the form of a human-readable name.
An object has state – names of the various properties that describe the object (its attributes).
An object has behavior – this is represented by functions, referred to as methods.

One of the fundamental principles of object-orientation (OO) is that of data hiding: an object hides its data from the rest of the world and only lets outsiders manipulate that data by way of calls to the object's methods. The formal term for this is encapsulation.

Object	Attribute	Value
Octopus card	PIN	2354
Account	ID	232-2222-3456-1000
$200 Bill	serialNumber	J2345

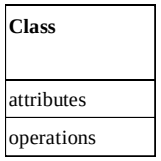
Objects, Attributes, and Values

Classes

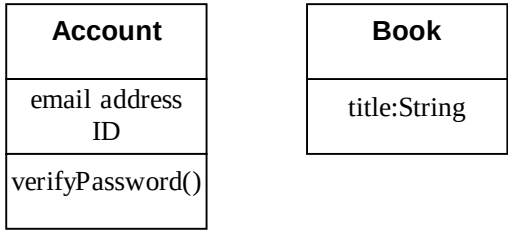
A class is a collection of objects that have the same characteristics. A class has identity in the form of a human-readable name that's unique in a particular context.

A class doesn't have state like an object does. It defines behavior in terms of operations, as opposed to methods. An operation represents a service that an object can request to affect behavior; a method is an implementation of that service.

An object that belongs to a particular class is often referred to as an instance of that class. Within the UML, the standard notation for a class is a box with three compartments, as shown in Fig. 2. The top compartment contains the name of the class, the middle compartment contains the attributes that belong to the class, and the bottom compartment contains the class's operations.



UML Class Notation



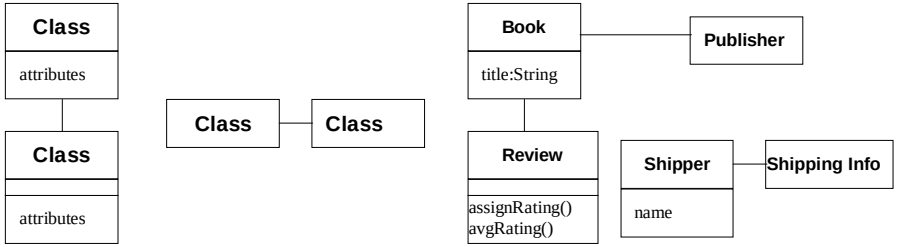
Sample Classes

Class Relationships

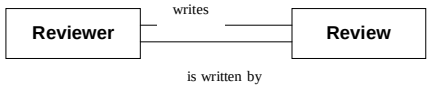
The relationships among classes provide the foundation for the structure of a new system. The followings explore how you use the UML to illustrate three kinds of class relationships.

Association

An association is a structural connection between classes. You show an association between two classes with a straight line that connects them.



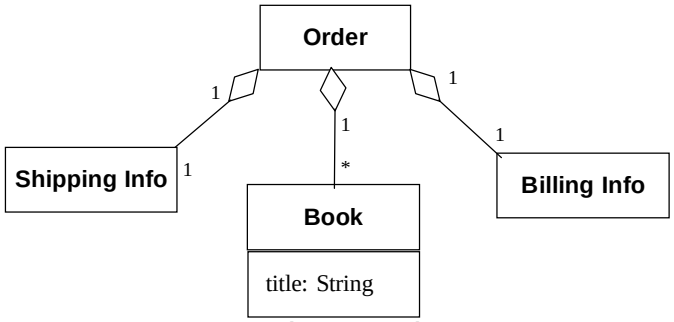
UML Association Notations and Sample Association



Association Roles

Aggregation

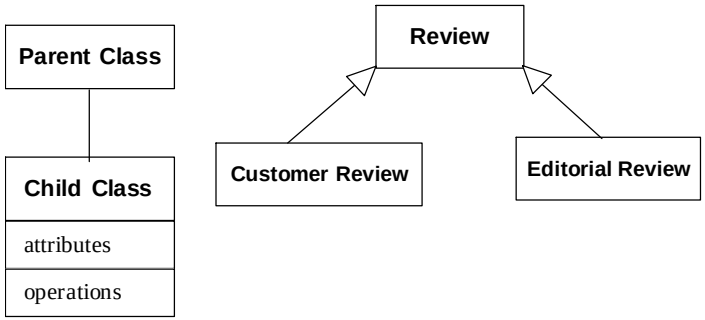
An aggregation is a special kind of association – a "whole/part" relationship within one or more smaller classes are "part" of a larger "whole". Using the UML, you show an aggregation by using a line within an open diamond at one end.



Sample Aggregation

Generalization

Generalization refers to a relationship between a general class (the super-class or parent) and a more specific version of that class (the subclass or child). A subclass inherits the attributes and operations from one super-class (single inheritance) or more than one super-class (multiple inheritance).



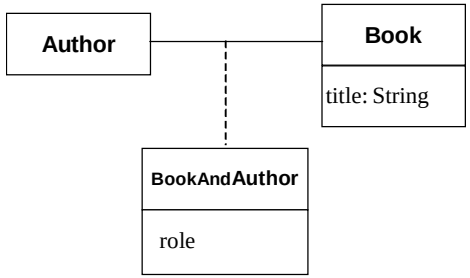
Sample Generalization

Substitutability – an object of a subclass may be substituted anywhere an object of an associated super-class is used.

Polymorphism – an object of a subclass can redefine any of the operations in inherits from its super-class(es).

Association Classes

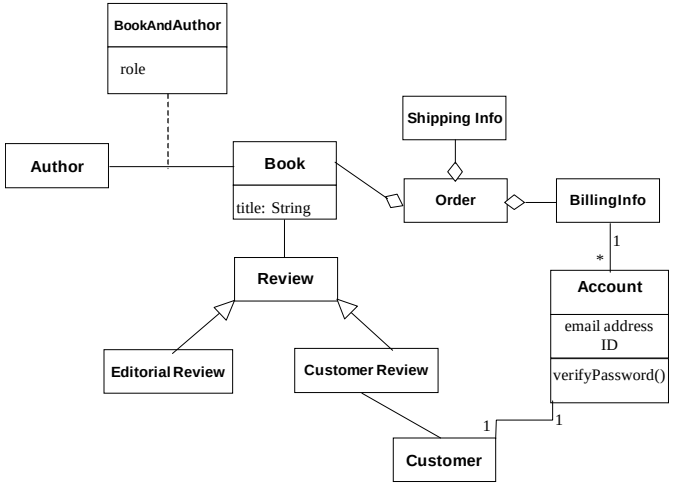
An association class is a cross between an association and a class. You use it to model an association that has interesting characteristics of its own outside the classes it connects.



Association Class

Class Diagrams

A class diagram shows classes and the various relationship in which they are involved.

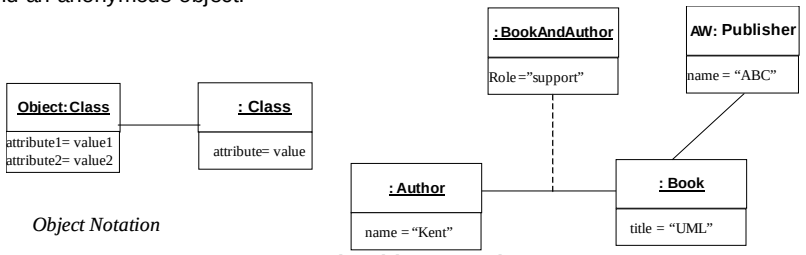


Class Diagram

Object Diagrams

The UML notation for an object takes the same basic form as that for a class. There are three differences: Within the top compartment of the class box, the name of the class to which the object belongs appears after a colon. The object may have a name, which appears before the colon, or it may be anonymous, i.e. nothing appears before the colon.

The contents of the top compartment are underlined for an object. Each attribute defined for the given class has a specific value for each object that belongs to that class. Diagram indicated below shows the UML notation for both a named object and an anonymous object.



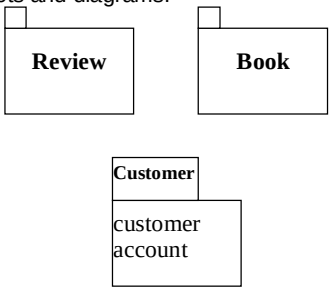
Sample Object Notation

An object diagram is basically a snapshot of part of the structure of the system being modeled. It has the same basic appearance as a class diagram, except that it shows objects, and actual values for attributes, instead of classes.

Packages

A package is a grouping of pieces of a model. Packages are very useful in managing models. They are helpful in grouping related items in order to make it easier to break work up among subteams.

A package can contain one or more kinds of model elements. You can have just classes in a package, for instance, or classes and classes diagrams, or a number of different kinds of constructs and diagrams.



Sample Packages

Capturing Requirements

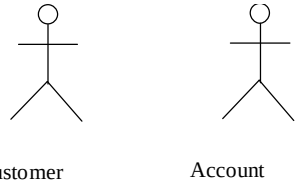
The task of capturing requirements associated with a new system is a complicated one, an one that never seems to stop. The use case model, which allows the project stakeholders to agree on what the system should do, serves as the foundation for all other development work. The elements of this model are shown as follows.

Actors and Use Cases

An actor represents one of two things:

- A role that can play with regard to a system
- An entity, such as another system or a database, that resides outside the system

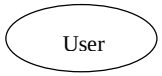
A use case is a sequence of actions that an actor performs within a system to achieve a particular goal. A good use case is expressed from the viewpoint of the actor, in present tense and active voice.



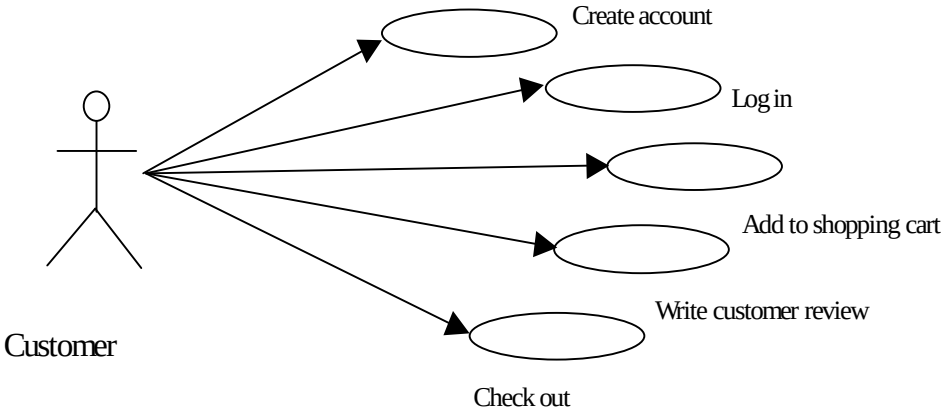
Sample Actors

Use Case Diagram

You show actors and use cases on use case diagrams.



The actor that executes a given use case usually appears on the left-hand side of the diagram.
The use case appear in the center.
Any other actors that are involved in the given use case tend to appear on the right-hand side.
Arrows show which actors are involved in which use cases.
Diagram below shows how actors and use case appear on a UML use case diagram.



Use case diagram

Flows of Events

Two kinds of events are associated with use cases.

- The main flow of events (basic course of action) is the main start-to-finish path that the actor and the system will follow under normal circumstances.
- The exceptional flow of events (alternate course of action) is a path through a use case that represents an error condition or a path that the actor and the system take less frequently.

6.5 How Things Work Together

Robustness Analysis

Robustness Analysis involves analyzing the text of a use case and identifying a first-guess set of objects that will participate in the use case, and then classifying these objects based on their characteristics.

There are three types of analysis classes: boundary classes, entity class and control class. The following subsections describe these classes in term of the objects that serve as instances of the classes.

Boundary Objects

A boundary object is an object with which an actor associated with a use case interacts.

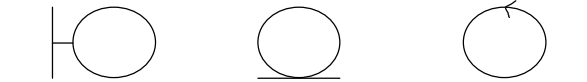
Entity Objects

An entity object is generally an object that contains long-lived information, such as that associated with databases. An entity object can also contain transient data, such as the contents of lists in windows, or search results. Entity objects also correspond with nouns in use case text.

Control Objects

A control object is an object that embodies application logic. Control objects are often used to handle things such as coordination and sequencing. They are also useful for calculations involving multiple entity objects.

Control objects serve as the connecting tissue between boundary objects and entity objects. They correspond with verbs in use case text.



Objects

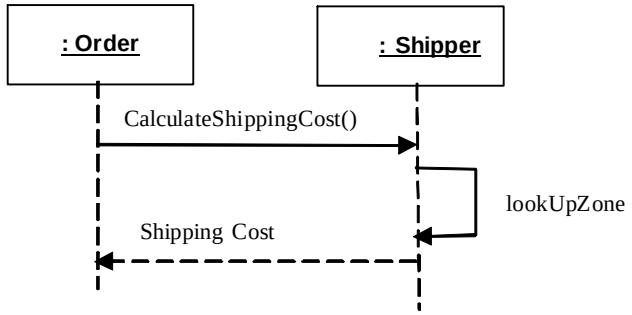
Messages and Actions

The next step in modeling the dynamic behavior of a system involves modeling the interactions among objects. These interactions take the form of set messages. A message is a communication between two objects, or within an object, is designed to result in some activity.

Call and Return

A call action is an invocation of a method on an object. A call action is synchronous, which means that the sender assumes that the receiver is ready to accept the message. An object can perform a call action on another object, or an object can perform a call action on itself. The UML represents a call action as an arrow from the calling object to the receiving object. The dashed line that appears beneath each object in the diagram is called a lifeline. A return action is the return of a value in

response to a call action. You show a return action as a dashed arrow from the object returning the value to the object receiving the value.

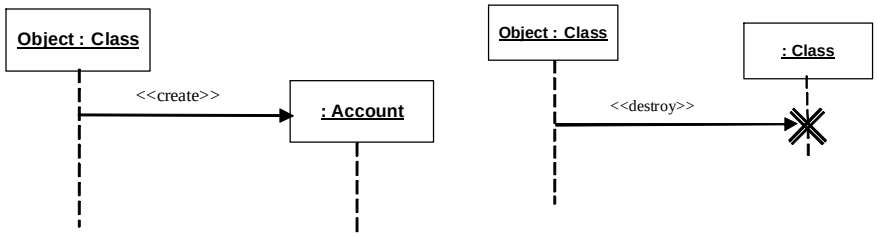


Call and return actions

Create and Destroy

A create action creates an object. (It tells a class to create an instance of itself.) Diagram indicated shows the UML notation for a create action.

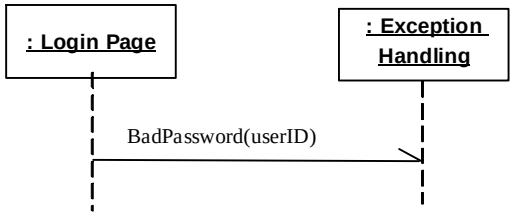
The guillemets around “create” and “destroy” indicate that these words fall into the same basic category of UML keywords as “include” and “extend”. A destroy action destroys an object.



Create Action and Destroy Action

Send

A send action sends a signal to an object. A signal is an asynchronous communication between objects: one object “throws” a signal to another object that “catches” the signal, but the sender of the signal doesn’t expect a response from the receiver unlike the sender of a call action.

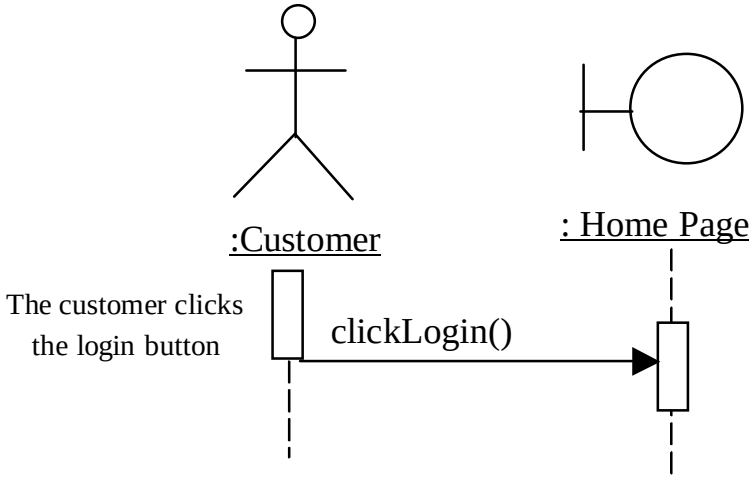


Send Action

Sequence Diagrams

The UML sequence diagram is a diagram that focuses on the time ordering of the messages that go back and forth between objects. Sequence diagrams are also associated with the Design workflow Unified Process. The development team uses sequence diagrams in deciding where to assign operations on classes. Sequence diagram notation.

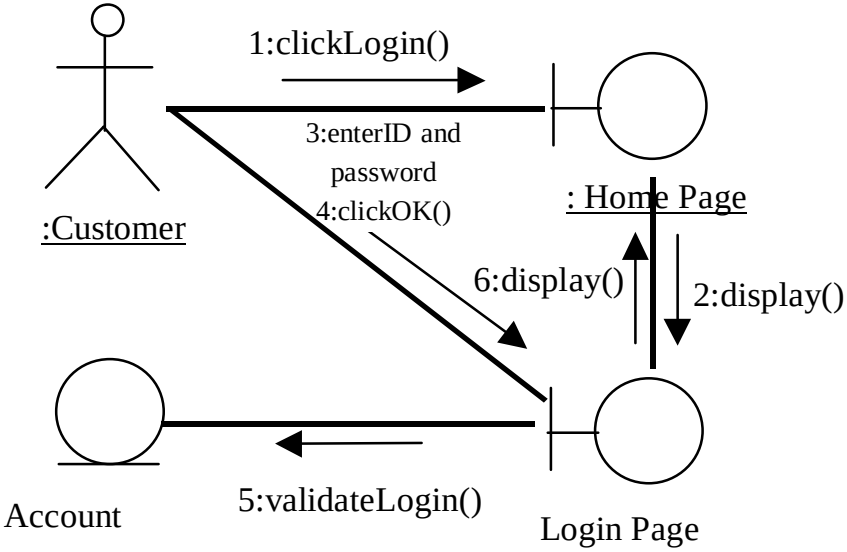
- Objects appear along the top margin.
- Each object has a lifeline, a dash line represents the life.
- A focus of control is a tall, thin rectangle that sits on top of an object's lifeline. The rectangle shows the period of time during which an object is in control of the flow.
- Messages show the actions that objects perform on each other and on themselves.



Log in Sequence diagram, part 1

Collaboration Diagrams

The UML collaboration diagram is a diagram that focuses on the organization of the objects that participate in a given set of messages.



Collaboration diagram