What is modelling?

- Testing a physical entity before building it: Medieval built scale models of Gothic Cathedrals to test the forces on the structures. Engineers test scale models of airplanes, cars and boats to improve their dynamics.
- Communication with customers: Architects and product designers build models to show their customers
- Visualization: Storyboards of movies, TV shows and advertisements let writers see how their ideas flow.
- Reduction of complexity: Models reduce complexity to understand directly by separating out a small number of important things to do with at a time.

Object Oriented Thinking

Superficially the term ***object-oriented (OO)*** means that we organize software as a collection of discrete objects that incorporate both data structure and behavior. This contrasts with previous programming approaches in which data structure and behavior are only loosely connected.

There is some dispute about exactly what characteristics are required by an OO approach, but they generally include four aspects: identity, classification, inheritance, and polymorphism.

Identity means that data is quantized into discrete, distinguishable entities called ***objects***.

The *first paragraph in this chapter*, *my workstation*, and the *white queen in a chess game* are examples of objects. Figure 1.1 shows some additional objects. Objects can be concrete, such as a *file* in a file system, or conceptual, such as a *scheduling policy* in a multiprocessing operating system. Each object has its own inherent identity. In other words, two objects are distinct even if all their attribute values (such as name and size) are identical.

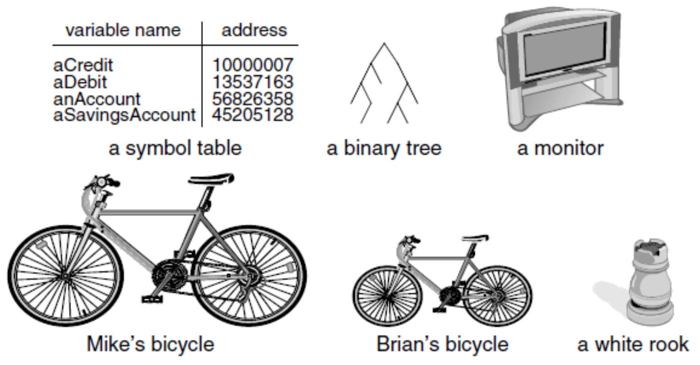


Figure 1.1 Objects. Objects lie at the heart of object-oriented technology.

Classification means that objects with the same data structure (***attributes***) and behavior (***operations***) are grouped into a class. *Paragraph*, *Monitor*, and

Chess Piece are examples of classes. A ***class*** is an abstraction that describes properties important to an application and ignores the rest. Any choice of classes is arbitrary and depends on the application.

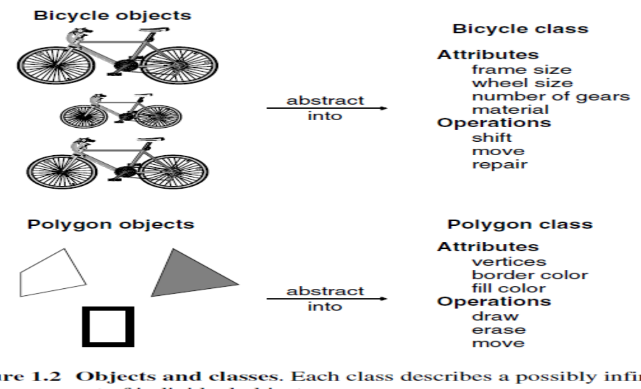
Each class describes a possibly infinite set of individual objects. Each object is said to be an ***instance*** of its class. An object has its own value for each attribute but shares the attribute names and operations with other instances of the class. Figure 1.2 shows two classes and some of their respective instances. An object contains an implicit reference to its own class; it "knows what kind of thing it is."

Inheritance is the sharing of attributes and operations (***features***) among classes based on a hierarchical relationship. A ***superclass*** has general information that ***subclasses*** refine and elaborate. Each subclass incorporates, or inherits, all the features of its superclass and adds its own unique features. Subclasses need not repeat the features of the superclass. For example, *Scrolling Window* and *Fixed Window* are subclasses of *Window.* Both subclasses inherit the features of *Window,* such as a visible region on the screen. *Scrolling Window* adds a scroll bar and an offset. The ability to factor out common features of several classes into a superclass can greatly reduce repetition within designs and programs and is one of the main

advantages of OO technology.

Polymorphism means that the same operation may behave differently for different classes. The *move* operation, for example, behaves differently for a pawn than for the queen in a chess game. An ***operation*** is a procedure or transformation that an object performs or is subject to. *Right Justify*, *display*, and *move* are examples of operations. An implementation of an operation by a specific class is called a ***method***. Because an OO operator is polymorphic,

it may have more than one method implementing it, each for a different class of object.



igure 1.2 **Objects and classes.** Each class describes a possibly infinite set of individual objects.

implement an operation based on the name of the operation and the class of the object being operated on. The user of an operation need not be aware of how many methods exist to implement a given polymorphic operation.

Developers can add new classes without changing existing code, as long as they provide methods for each applicable operation.

OO Methodology

System conception. Software development begins with business analysts or users conceiving an application and formulating tentative requirements.

■ **Analysis**. The analyst scrutinizes and rigorously restates the requirements from system conception by constructing models. The analyst must work with the requestor to understand the problem, because problem statements are rarely complete or correct. The analysis model is a concise, precise abstraction of *what* the desired system must do, not *how* it will be done. The analysis model should not contain implementation decisions. For example, a *Window* class in a workstation windowing system would be described in terms of its visible attributes and operations.

The analysis model has two parts: the ***domain model***, a description of the real-world objects reflected within the system; and the ***application model***, a description of the parts of the application system itself that are visible to the user. For example, domain objects for a stockbroker application might include stock, bond, trade, and commission. Application

objects might control the execution of trades and present the results. Application experts who are not programmers can understand and criticize a good model.

System design. The development team devise a high-level strategy—the ***system architecture***—

for solving the application problem. They also establish policies that will serve

as a default for the subsequent, more detailed portions of design. The system designer must decide what performance characteristics to optimize, choose a strategy of attacking the problem, and make tentative resource allocations. For example, the system designer might decide that changes to the workstation screen must be fast and smooth, even when windows are moved or erased, and choose an appropriate communications protocol and memory buffering strategy.

■ **Class design**. The class designer adds details to the analysis model in accordance with the system design strategy. The class designer elaborates both domain and application objects using the same OO concepts and notation, although they exist on different conceptual planes. The focus of class design is the data structures and algorithms needed to implement each class. For example, the class designer now determines data structures and algorithms for each of the operations of the *Window* class.

■ **Implementation**. Implementers translate the classes and relationships developed during class design into a particular programming language, database, or hardware. Programming should be straightforward, because all of the hard decisions should have already been made. During implementation, it is important to follow good software engineering practice so that traceability to the design is apparent and so that the system remains flexible and extensible. For example, implementers would code the *Window* class in a programming language, using calls to the underlying graphics system on the workstation.

OO Themes

Abstraction lets you focus on essential aspects of an application while ignoring details. This means focusing on what an object is and does, before deciding how to implement it. Use of abstraction preserves the freedom to make decisions as long as possible by avoiding premature commitments to details. Most modern languages provide data abstraction, but inheritance

and polymorphism add power. The ability to abstract is probably the most important

skill required for OO development.

Encapsulation *(*also ***information hiding****)* separates the external aspects of an object, that are accessible to other objects, from the internal implementation details, that are hidden from other objects. Encapsulation prevents portions of a program from becoming so interdependent that a small change has massive ripple effects. You can change an objects implementation without affecting the applications that use it. You may want to change the

implementation of an object to improve performance, fix a bug, consolidate code, or support porting. Encapsulation is not unique to OO languages, but the ability to combine data structure and behavior in a single entity makes encapsulation cleaner and more powerful than in prior languages, such as Fortran, Cobol, and C.

Combining Data and Behavior The caller of an operation need not consider how many implementations exist. Operator polymorphism shifts the burden of deciding what implementation to use from the calling code to the class hierarchy. For example, non-OO code to display the contents of a window

must distinguish the type of each figure, such as polygon, circle, or text, and call the appropriate procedure to display it. An OO program would simply invoke the *draw* operation on each figure; each object implicitly decides which procedure to use, based on its class. Maintenance is easier, because the calling code need not be modified when a new class is added.In an OO system, the data structure hierarchy matches the operation inheritance hierarchy

(Figure 1.3).

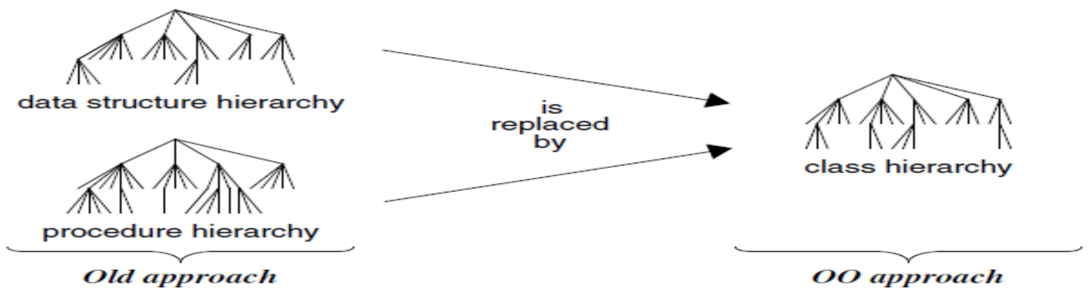


Figure 1.3 OO vs. prior approach. An OO approach has one unified hierarchy for both data and behavior.

Sharing

OO techniques promote sharing at different levels. Inheritance of both data structure and behavior lets subclasses share common code. This sharing via inheritance is one of the main advantages of OO languages. More important than the savings in code is the conceptual clarity from recognizing that different operations are all really the same thing. This reduces the number of distinct cases that you must understand and analyze.

OO development not only lets you share information within an application, but also offers the prospect of reusing designs and code on future projects. OO development provides the tools, such as abstraction, encapsulation, and inheritance, to build libraries of reusable components. Unfortunately, reuse has been overemphasized as a justification for OO technology.

Reuse does not just happen; developers must plan by thinking beyond the immediate application and investing extra effort in a more general design.

Emphasis on the Essence of an Object

OO technology stresses what an object *is,* rather than how it is *used*. The uses of an object depend on the details of the application and often change during development. As requirements evolve, the features supplied by an object are much more stable than the ways it is used, hence software systems built on object structure are more stable in the long run. OO development places a greater emphasis on data structure and a lesser emphasis on procedure structure than functional-decomposition methodologies. In this respect, OO development is similar to information modeling techniques used in database design, although OO development adds the concept of class-dependent behavior.

Synergy

Identity, classification, polymorphism, and inheritance characterize OO languages. Each of these concepts can be used in isolation, but together they complement each other synergistically.The benefits of an OO approach are greater than they might seem at first. The emphasis on the essential properties of an object forces the developer to think more carefully and deeply about what an object is and does. The resulting system tends to be cleaner, more general, and more robust than it would be if the emphasis were only on the use of data and operations.

History of UML Building Blocks of UML

The 1990s was the period of development of object-oriented languages such as C++. These object-oriented languages (OOL) were used to build complex but compelling systems.
It was developed by software engineers Grady Booch, Ivar Jacobson, and James Rumbaugh of Rational software during 1994 and 1995. It was under development till 1996.
All of UML creators, viz, Grady Booch, Ivar Jacobson, and James Rumbaugh had a fabulous idea for creating a language that will reduce the complexity.

- Booch's method was flexible to work with throughout the design and creation of objects.
- Jacobson's method contributed a great way to work on use-cases. It further has a great approach for high-level design.
- Rumbaugh's method turned out to be useful while handling sensitive systems.
- Behavioral models and state-charts were added in the UML by David Harel.

UML was acknowledged as a norm by Object Management Group (OMG) in 1997. Object Management Group is responsible for maintaining UML regularly since it was adopted as a standard.
In 2005, the International Organization for Standardization established UML as an ISO standard. It is used in many industries for designing object-oriented models.
UML's latest version is 2.5.1 which was released in December 2017.

Building Blocks of UML

As UML describes the real-time systems, it is very important to make a conceptual model and then proceed gradually. The conceptual model of UML can be mastered by learning the following three major elements

- UML building blocks
- Rules to connect the building blocks
- Common mechanisms of UML

This chapter describes all the UML building blocks. The building blocks of UML can be defined as

- Things

- Relationships
- Diagrams

Things

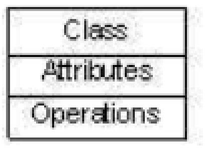
Things are the most important building blocks of UML. Things can be

- Structural
- Behavioral
- Grouping
- Annotational

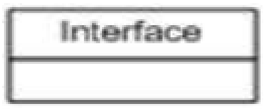
Structural Things

Structural things define the static part of the model. They represent the physical and conceptual elements. Following are the brief descriptions of the structural things.

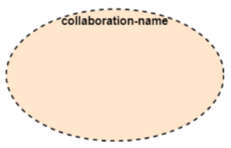
Class − Class represents a set of objects having similar responsibilities.



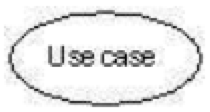
Interface − Interface defines a set of operations, which specify the responsibility of a class.



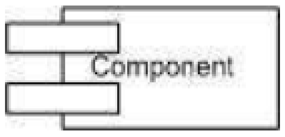
Collaboration −Collaboration defines an interaction between elements.



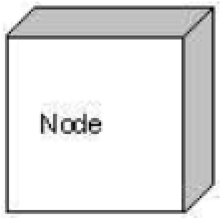
Use case −Use case represents a set of actions performed by a system for a specific goal.



Component −Component describes the physical part of a system.



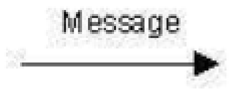
Node – A node can be defined as a physical element that exists at run time.



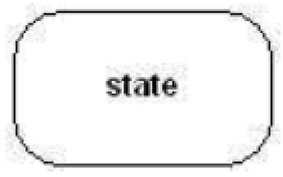
Behavioral Things

A behavioral thing consists of the dynamic parts of UML models. Following are the behavioral things

Interaction – Interaction is defined as a behavior that consists of a group of messages exchanged among elements to accomplish a specific task.



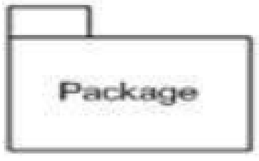
State machine – State machine is useful when the state of an object in its life cycle is important. It defines the sequence of states an object goes through in response to events. Events are external factors responsible for state change



Grouping Things

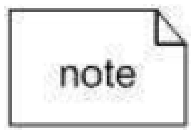
Grouping things can be defined as a mechanism to group elements of a UML model together. There is only one grouping thing available –

Package – Package is the only one grouping thing available for gathering structural and behavioral things.



Annotational Things

Annotational things can be defined as a mechanism to capture remarks, descriptions, and comments of UML model elements. **Note** - It is the only one Annotational thing available. A note is used to render comments, constraints, etc. of an UML element.



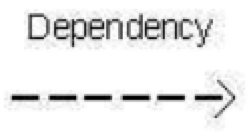
Relationship

Relationship is another most important building block of UML. It shows how the elements are associated with each other and this association describes the functionality of an application.

There are four kinds of relationships available.

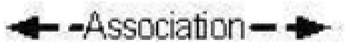
Dependency

Dependency is a relationship between two things in which change in one element also affects the other.



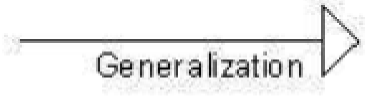
Association

Association is basically a set of links that connects the elements of a UML model. It also describes how many objects are taking part in that relationship.



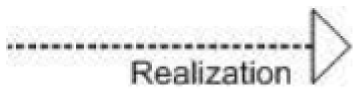
Generalization

Generalization can be defined as a relationship which connects a specialized element with a generalized element. It basically describes the inheritance relationship in the world of objects.



Realization

Realization can be defined as a relationship in which two elements are connected. One element describes some responsibility, which is not implemented and the other one implements them. This relationship exists in case of interfaces.



UML Diagrams

UML diagrams are the ultimate output of the entire discussion. All the elements, relationships are used to make a complete UML diagram and the diagram represents a system.

The visual effect of the UML diagram is the most important part of the entire process. All the other elements are used to make it complete.

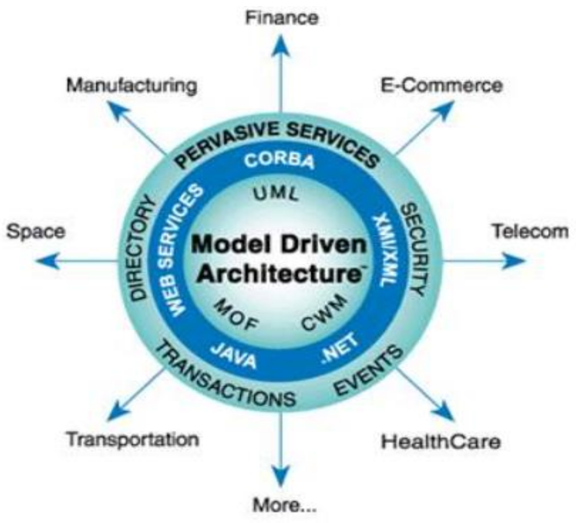
UML includes the following nine diagrams, the details of which are described in the subsequent chapters.

- Class diagram
- Object diagram
- Use case diagram
- Sequence diagram
- Collaboration diagram
- Activity diagram
- State chart diagram
- Deployment diagram
- Component diagram

Introduction to OMG Standard MDA

Model-Driven Architecture

Model Driven Architecture® (MDA®) is an approach to software design, development and implementation spearheaded by the OMG. MDA provides guidelines for structuring software specifications that are expressed as models.



MDA separates business and application logic from underlying platform technology. Platform-independent models of an application or integrated system's business functionality and behavior, built using UML and the other associated OMG modeling standards, can be realized through the MDA on virtually any platform, open or proprietary, including Web Services, .NET, CORBA R, J2EE, and others. These platform-independent models document the business functionality and behavior of an application separate from the technology-specific code that implements it, insulating the core of the application from technology and its relentless churn cycle while enabling interoperability both within and across platform boundaries. No longer tied to each other, the business and technical aspects of an application or integrated system can each evolve at its own pace - business logic responding to business need, and technology taking advantage of new developments - as the business requires.

XMI

XMI (XML Metadata Interchange) is a format specification produced by the Object Management Group (OMG). The XMI format allows the interchange of objects and models through an XMI formatted file. The XMI format is commonly used to exchange UML models between other tools or software.

In addition to XMI, Rhapsody provides additional tools for developers to examine the models, such as:

- Rhapsody ReporterPLUS reports in Word, PowerPoint, and HTML format. Reports are created without any conversion to another format.
- Model simulation capabilities show how the model components work together.
- The COM API exports a set of COM interfaces representing the metamodel objects and application operational functions.
- Write macros provide a means to examine the model within the product.

The Rhapsody XMI export and import feature facilitates the following development tasks:

- Export an entire Rhapsody model to XMI to be closely examined as a whole
- Export the whole model to XMI to be searched in an HTML browser
- Export the model to XMI in order to parse the entire model with another UML tool or a non-UML tool
- Imports XMI models or pieces of other XMI models into Rhapsody models
- Exchange models to or from the Tau system

UML 2.0

Modeling Diagrams in UML 2.0

Modeling Interactions

The interaction diagrams described in UML 2.0 is different than the earlier versions. However, the basic concept remains the same as the earlier version. The major difference is the enhancement and additional features added to the diagrams in UML 2.0.

UML 2.0 models object interaction in the following four different ways.

- **Sequence diagram** is a time dependent view of the interaction between objects to accomplish a behavioral goal of the system. The time sequence is similar to the earlier version of sequence diagram. An interaction may be designed at any level of abstraction within the system design, from subsystem interactions to instancelevel.
- **Communication diagram** is a new name added in UML 2.0. Communication diagram is a structural view of the messaging between objects, taken from the Collaboration diagram concept of UML 1.4 and earlier versions. This can be defined as a modified version of collaboration diagram.
- **Interaction Overview diagram** is also a new addition in UML 2.0. An Interaction Overview diagram describes a high-level view of a group of interactions combined into a logic sequence, including flow-control logic to navigate between the interactions.
- **Timing diagram** is also added in UML 2.0. It is an optional diagram designed to specify the time constraints on the messages sent and received in the course of an interaction.

From the above description, it is important to note that the purpose of all the diagrams are to send/receive messages. The handling of these messages are internal to the objects. Hence, the objects also have options to receive and send messages, and here comes another important aspect called interface. Now these interfaces are responsible for accepting and sending messages to one another.

It can thus be concluded that the interactions in UML 2.0 are described in a different way and that is the reason why the new diagram names have come into picture. If we analyze the new diagrams then it is clear that all the diagrams are created based upon the interaction diagrams described in the earlier versions. The only difference is the additional features added in UML 2.0 to make the diagrams more efficient and purpose oriented.

Modeling Collaborations

As we have already discussed, collaboration is used to model common interactions between objects. We can say that collaboration is an interaction where a set of messages are handled by a set of objects having pre-defined roles.

The important point to note is the difference between the collaboration diagram in the earlier version and in UML 2.0 version. To distinguish, the name of the collaboration diagram has been changed in UML 2.0. In UML 2.0, it is named as **Communication diagram.**

Consequently, collaboration is defined as a class with attributes (properties) and behavior (operations). Compartments on the collaboration class can be user defined and may be used for interactions (Sequence diagrams) and structural elements (Composite Structure diagram).

Following figure models the Observer design pattern as collaboration between an object in the role of an observable item and any number of objects as the observers.

Modeling Communication

Communication diagram is slightly different than the collaboration diagrams of the earlier versions. We can say it is a scaled back version of the earlier UML versions. The distinguishing factor of the communication diagram is the link between objects.

This is a visual link and it is missing in the sequence diagram. In the sequence diagram, only the messages passed between the objects are shown even if there is no link between them.

Communication diagram is used to prevent the modeler from making this mistake by using an Object diagram format as the basis for messaging. Each object on a Communication diagram is called an object lifeline.

The message types in a Communication diagram are the same as in a Sequence diagram. Communication diagram may model synchronous, asynchronous, return, lost, found, a object-creation messages.

Following figure shows an Object diagram with three objects and two links that form the basis for the Communication diagram. Each object on a Communication diagram is called an object lifeline.

Modeling an Interaction Overview

In practical usage, a sequence diagram is used to model a single scenario. A number of sequence diagrams are used to complete the entire application. Hence, while modeling a single scenario, it is possible to forget the total process and this can introduce errors.

To solve this issue, the new interaction overview diagram combines the flow of control from an activity diagram and messaging specification from the sequence diagram.

Activity diagram uses activities and object flows to describe a process. The Interaction Overview diagram uses interactions and interaction occurrences. The lifelines and messages found in Sequence diagrams appear only within the interactions or interaction occurrences. However, the lifelines (objects) that participate in the Interaction Overview diagram may be listed along with the diagram name.

Following figure shows an interaction overview diagram with decision diamonds, frames, and termination point.

Modeling a Timing Diagram

The name of this diagram itself describes the purpose of the diagram. It basically deals with the time of the events over its entire lifecycle.

A timing diagram can therefore be defined as a special purpose interaction diagram made to focus on the events of an object in its life time. It is basically a mixture of state machine and interaction diagram. The timing diagram uses the following timelines −

- State time line
- General value time line

A lifeline in a Timing diagram forms a rectangular space within the content area of a frame. It is typically aligned horizontally to read from the left to right. Multiple lifelines may be stacked within the same frame to model the interaction between them.

RUP

Rational Unified Process (RUP) is a framework for software engineering processes. RUP is an Iterative and incremental approach to improving problem knowledge through consecutive revisions. It is an architecture-centric and use-case-driven approach that manages risk and is flexible to change. RUP incrementally improves an effective solution through repeated iterations.

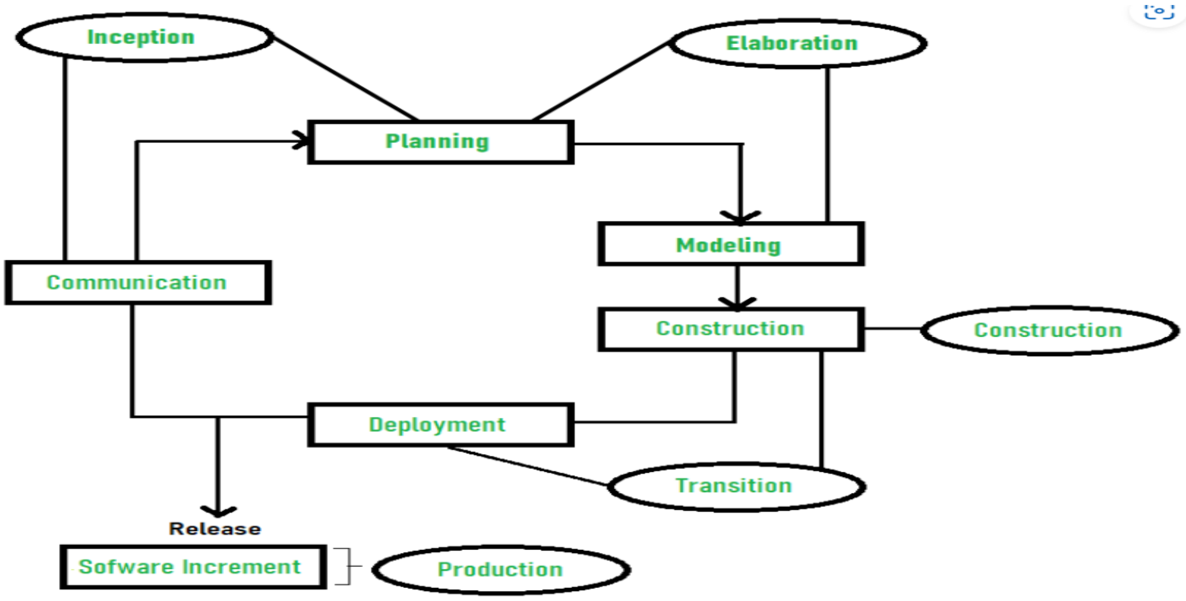
Rational Unified Process (RUP) is a software development process for object-oriented models. It is also known as the Unified Process Model. It is created by Rational Corporation and is designed and documented using UML (Unified Modeling Language). This process is included in the IBM Rational Method Composer (RMC) product. IBM (International Business Machine Corporation) allows us to customize, design, and personalize the unified process.

RUP is proposed by Ivar Jacobson, Grady Bootch, and James Rambaugh. Some characteristics of RUP include being use-case driven, Iterative (repetition of the process), incremental (increase in value) by nature, delivered online using web technology, can be customized or tailored in modular and electronic form, etc. RUP reduces unexpected development costs and prevents the wastage of resources.

Phases of RUP

There is a total of five phases of the life cycle of RUP:

1. Inception
2. Elaboration
3. Construction
4. Transition
5. Production



1. **Inception –**
   - Communication and planning are the main ones.
   - Identifies the scope of the project using a use-case model allowing managers to estimate costs and time required.

- Customers' requirements are identified and then it becomes easy to make a plan for the project.
- The project plan, Project goal, risks, use-case model, and Project description, are made.
- The project is checked against the milestone criteria and if it couldn't pass these criteria then the project can be either canceled or redesigned.

2. **Elaboration –**
    - Planning and modeling are the main ones.
    - A detailed evaluation and development plan is carried out and diminishes the risks.
    - Revise or redefine the use-case model (approx. 80%), business case, and risks.
    - Again, checked against milestone criteria and if it couldn't pass these criteria then again project can be canceled or redesigned.
    - Executable architecture baseline.
3. **Construction –**
    - The project is developed and completed.
    - System or source code is created and then testing is done.
    - Coding takes place.
4. **Transition –**
    - The final project is released to the public.
    - Transit the project from development into production.
    - Update project documentation.
    - Beta testing is conducted.
    - Defects are removed from the project based on feedback from the public.
5. **Production –**
    - The final phase of the model.
    - The project is maintained and updated accordingly.

Advantages of Rational Unified Process (RUP)

Following are the advantages of Rational Unified Process (RUP):

1. RUP provides good documentation, it completes the process in itself.

2. RUP provides risk-management support.
3. RUP reuses the components, and hence total time duration is less.
4. Good online support is available in the form of tutorials and training.

Disadvantages of Rational Unified Process (RUP)

Following are the disadvantages of Rational Unified Process (RUP):

1. Team of expert professional is required, as the process is complex.
2. Complex and not properly organized process.
3. More dependency on risk management.
4. Hard to integrate again and again.

Rational Unified Process (RUP) Best Practices

Following are the Rational Unified Process (RUP) best practices:

1. **Develop incrementally**
    - The iterative approach to development supported by the Rational Unified Process considerably lowers the risk profile of a project by addressing the greatest risk items at every stage of the lifecycle.
    - The development team remains focused on delivering results since every iteration closes with an actual release, and regular status updates make sure the project continues on track.
2. **Handle requirements**
    - In the Rational Unified Process (RUP), use cases and scenarios are essential tools for capturing and managing functional requirements
3. **Utilize modular architectures**
    - Rational Unified Process provides component-based software development.
    - Components are complex modules or subsystems that perform a specific function.
    - The Rational Unified Process is a systematic way to creating architecture with new and existing components.
4. **Diagram software**
    - To depict all important elements, users, and their interactions, make use of diagrams.

- Unified Modeling Language (UML), created by Rational Software, is the foundation for successful visual modeling.

5. **Ensure software quality**
    - Reviewing quality in relation to requirements based on functionality, application performance, system performance, and dependability is important.
    - You can get help with these test kinds' planning, designing, implementing, carrying out, and evaluating them with the Rational Unified Process.
6. **Manage software changes**
    - Numerous teams work on numerous projects, sometimes in separate places, on various platforms, etc. Therefore, it is necessary to ensure that any modifications made to a system are consistently synchronized and validated.

4+1 Architecture

Architecture View Model

A model is a complete, basic, and simplified description of software architecture which is composed of multiple views from a particular perspective or viewpoint.

A view is a representation of an entire system from the perspective of a related set of concerns. It is used to describe the system from the viewpoint of different stakeholders such as end-users, developers, project managers, and testers.

4+1 View Model

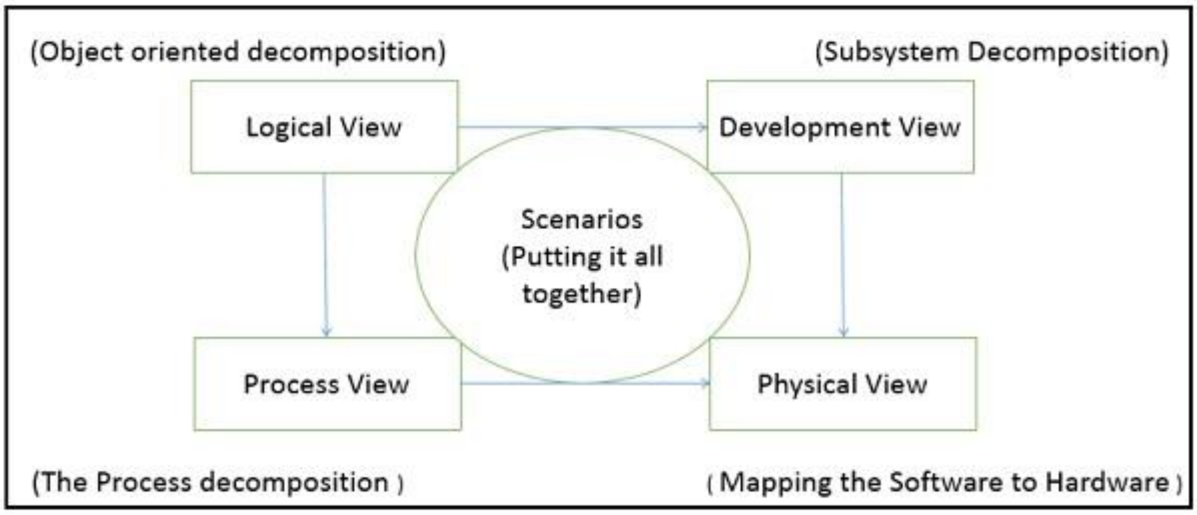
The 4+1 View Model was designed by Philippe Kruchten to describe the architecture of a software–intensive system based on the use of multiple and concurrent views. It is a **multiple view** model that addresses different features and concerns of the system. It standardizes the software design documents and makes the design easy to understand by all stakeholders.

It is an architecture verification method for studying and documenting software architecture design and covers all the aspects of software architecture for all stakeholders. It provides four essential views −

- **The logical view or conceptual view** − It describes the object model of the design.
- **The process view** − It describes the activities of the system, captures the concurrency and synchronization aspects of the design.

- **The physical view** – It describes the mapping of software onto hardware and reflects its distributed aspect.
- **The development view** – It describes the static organization or structure of the software in its development of environment.

This view model can be extended by adding one more view called **scenario view** or **use case view** for end-users or customers of software systems. It is coherent with other four views and are utilized to illustrate the architecture serving as "plus one" view, (4+1) view model. The following figure describes the software architecture using five concurrent views (4+1) model.

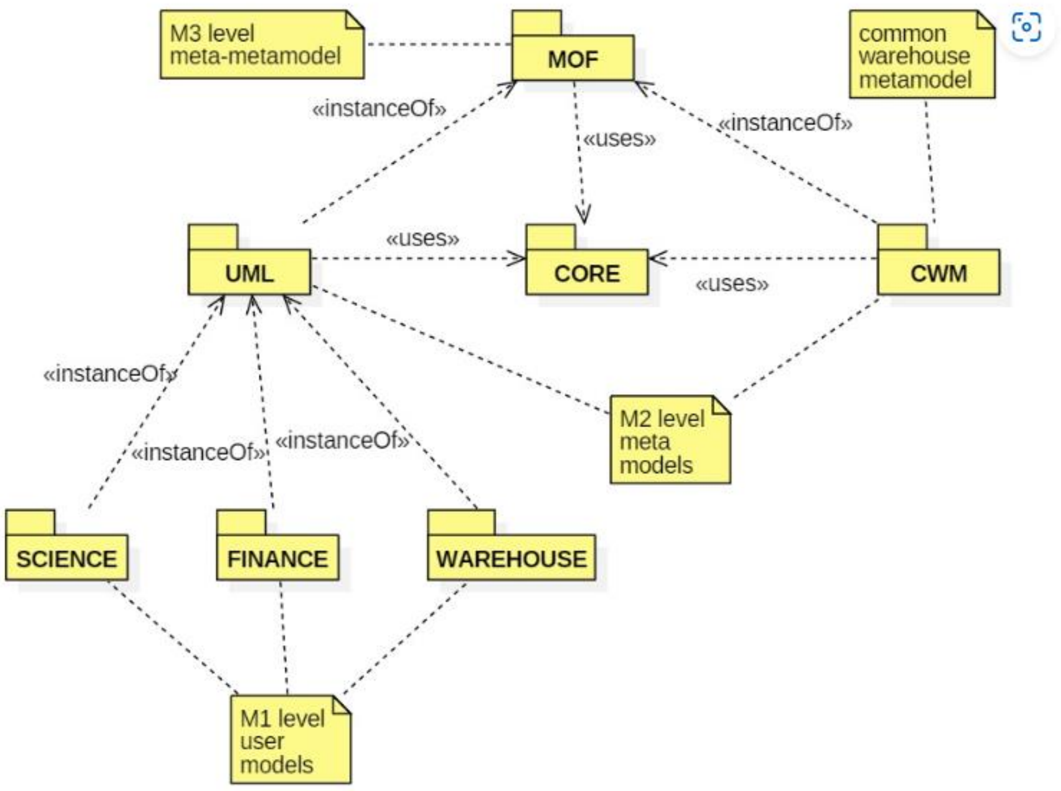

(Object oriented decomposition)
(Subsystem Decomposition)
Logical View
Development View
Scenarios (Putting it all together)
Process View
Physical View
(The Process decomposition)
(Mapping the Software to Hardware)

UML Meta Model

he UML specification [UML-OMG] is insanely complex and filled with inconsistencies. Basically, the specification uses UML to specify UML. Thus, the specification is sometimes referred to as the UML Meta-Model

Meta-models

The Object Management Group (OMG) is in the business of specifying standards for object-oriented developers—modeling standards (like UML), middleware standards (like CORBA), data warehousing standards (like SAS), etc. Here's my simplified summary of the OMG specifications:



here are three levels of abstraction in the OMG specifications:

M1 = Models (i.e., models created by UML users)

M2 = Meta-Models, M1 models are instances of M2 models = {UML, CWM}

M3 = Meta-Meta-Models, M2 models are instances of M3 models = {MOF}

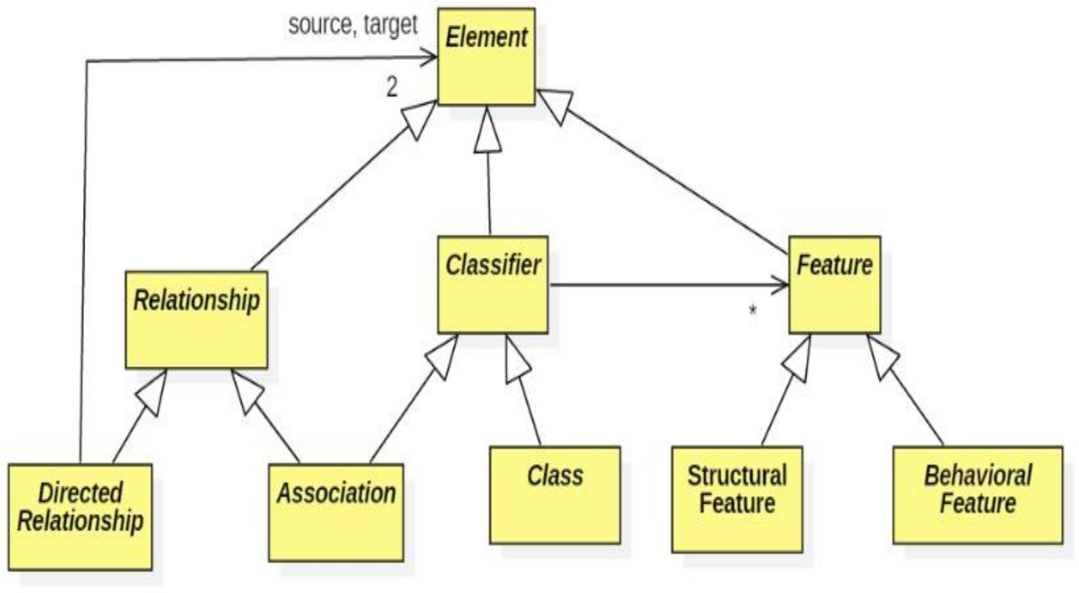
The Meta Object Facility (MOF) was originally a CORBA type system. It is used for defining meta-models. MOF is to domain models what EBNF is to grammars. MOF could be used to define web services as well as OO concepts.

UML and CWM are M2 meta-models that instantiate MOF.

Core + UML

All of the OMG specifications depend on a tiny core of modeling concepts. [UML-KF] describes these here:

Here's my simplified version:



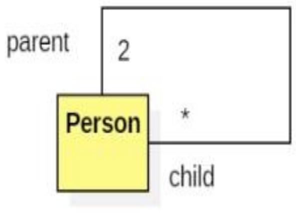
Everything that appears in a UML diagram is an element. The two most important types of elements are relationships (arrows) and classifiers (nodes).

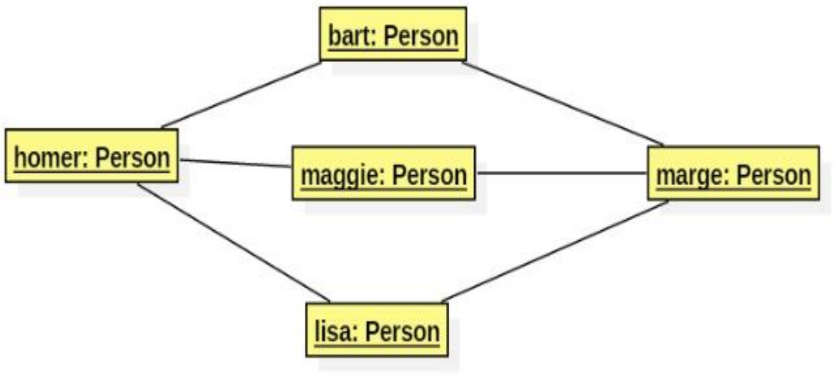
A classifier represents a set of instances. Examples of classifiers include classes, interfaces, packages, components, use cases, actors, data types (primitive types and enumerations), associations, and collaborations. An instance of a class or interface is an object, an instance of an association is a link. A classifier can have structural and behavioral features. For a class these would correspond to attributes and operations.

A directed relationship has a source element and a target element. Dependencies, generalizations, and realizations are types of directed relationships.

An (binary) association represents a semantic relationship between (two) classes. A semantic relationship is a relationship is a domain-significant relationship. For example, there could be many relationships between two people—teacher/student, parent/child, employee/dependent, etc. But in a genealogy model, for example, we are probably only interested in the parent/child relationship.

An instance of a parent/child association would be a link connecting an instance of person representing a parent to an instance of person representing one of the parent's children. For example:





Object

The purpose of class modeling is to describe objects. For example, *Joe Smith*, *Simplex company*, *process number 7648*, and *the top window* are objects.

An ***object*** is a concept, abstraction, or thing with identity that has meaning for an application. Objects often appear as proper nouns or specific references in problem descriptions and discussions with users. Some objects have real-world counterparts (Albert Einstein and the General Electric company), while others are conceptual entities (simulation run 1234 and the formula for solving a quadratic equation). Still others (binary tree 634 and the array

bound to variable *a*) are introduced for implementation reasons and have no correspondence to physical reality. The choice of objects depends on judgment and the nature of a problem; there can be many correct representations.

All objects have identity and are distinguishable. Two apples with the same color, shape, and texture are still individual apples; a person can eat one and then eat the other. Similarly, identical twins are two distinct persons, even though they may look the same. The term ***identity*** means that objects are distinguished by their inherent existence and not by descriptive

properties that they may have.

Class

An object is an instance—or occurrence—of a class. A class describes a group of objects with the same properties (attributes), behavior (operations), kinds of relationships, and semantics.

Person, company, process, and window are all classes. Each person has name and birthdate and may work at a job. Each process has an owner, priority, and list of required resources.

Classes often appear as common nouns and noun phrases in problem descriptions and discussions with users.

Objects in a class have the same attributes and forms of behavior. Most objects derive their individuality from differences in their attribute values and specific relationships to other objects. However, objects with identical attribute values and relationships are possible. The choice of classes depends on the nature and scope of an application and is a matter of judgment.

The objects in a class share a common semantic purpose, above and beyond the requirement of common attributes and behavior. For example, a barn and a horse may both have a cost and an age. If barn and horse were regarded as purely financial assets, they could belong to the same class. If the developer took into consideration that a person paints a barn and feeds a horse, they would be modeled as distinct classes. The interpretation of semantics depends

on the purpose of each application and is a matter of judgment.

Each object "knows" its class. Most OO programming languages can determine

an object's class at run time. An object's class is an implicit property of the

object. If objects are the focus of modeling, why bother with classes? The notion of abstraction is at the heart of the matter. By grouping objects into classes, we abstract a problem. Abstraction

gives modeling its power and ability to generalize from a few specific cases to a host of similar cases. Common definitions (such as class name and attribute names) are stored once per class rather than once per instance. You can write operations once for each class, so that all the objects in the class benefit from code reuse. For example, all ellipses share the same procedures to draw them, compute their areas, and test for intersection with a line; polygons would have a separate set of procedures. Even special cases, such as circles and squares, can use the general procedures, though more efficient procedures are possible

Class diagrams

provide a graphic notation for modeling classes and their relationships,thereby describing possible objects. Class diagrams are useful both for abstract modelling and for designing actual programs. They are concise, easy to understand, and work well in practice. We will use class diagrams throughout this book to represent the structure of applications.

We will also occasionally use object diagrams. An ***object diagram*** shows individual objects and their relationships. Object diagrams are helpful for documenting test cases and discussing examples. A class diagram corresponds to an infinite set of object diagrams.

Figure 3.1 shows a class (left) and instances (right) described by it. Objects *JoeSmith*, *MarySharp*, and an anonymous person are instances of class *Person*. The UML symbol for an object is a box with an object name followed by a colon and the class name. The object name and class name are both underlined. Our convention is to list the object name and class name in boldface.

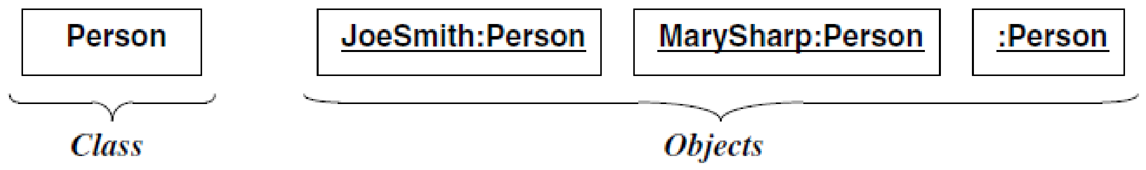


Figure 3.1 A class and objects. Objects and classes are the focus of class modeling.

The UML symbol for a class also is a box. Our convention is to list the class name in boldface, center the name in the box, and capitalize the first letter. We use singular nouns for the names of classes.Note how we run together multiword names, such as *JoeSmith*, separating the words with intervening capital letters. This is the convention we use for referring to objects, classes,

and other constructs. Alternative conventions would be to use intervening spaces (Joe Smith) or underscores (Joe_Smith). The mixed capitalization convention is popular in the OO literature

but is not a UML requirement.

Need &Application of UML

Need?

- **Improve communication**: UML diagrams help developers communicate about a system and its requirements.
- **Standardize systems**: UML diagrams help developers create successful and standardized systems.
- **Understand potential outcomes**: UML diagrams help developers understand potential outcomes or errors in programs.
- **Plan software development**: UML diagrams help developers plan software development and prioritization.

- **Analyze existing software**: UML diagrams help developers analyze existing software.

Application

- **Understand software systems**: UML diagrams help software engineers understand the design, code, and implementation of software systems.
- **Model business processes**: UML diagrams can be used to model business processes and workflows.
- **Improve communication**: UML diagrams can help improve communication between software developers.
- **Enhance collaboration**: UML diagrams can help enhance collaboration between software developers.
- **Unify design**: UML diagrams can help unify design.
- **Simplify complex systems**: UML diagrams can help simplify complex systems.
- **Aid in documentation**: UML diagrams can help aid in documentation.

Links and Associations

A ***link*** is a physical or conceptual connection among objects. For example, Joe Smith *Works- For* Simplex company. Most links relate two objects, but some links relate three or more objects.

This chapter discusses only binary associations; Chapter 4 discusses n-ary

associations. Mathematically, we define a link as a tuple—that is, a list of objects. A link is an instance of an association.

An ***association*** is a description of a group of links with common structure and common semantics. For example, a person *WorksFor* a company. The links of an association connect objects from the same classes. An association describes a set of potential links in the same way that a class describes a set of potential objects. Links and associations often appear as verbs in problem statements.

Figure 3.7 is an excerpt of a model for a financial application. Stock brokerage firms need to perform tasks such as recording ownership of various stocks, tracking dividends, alerting customers to changes in the market, and computing margin requirements. The top portion of the figure shows a class diagram and the bottom shows an object diagram.

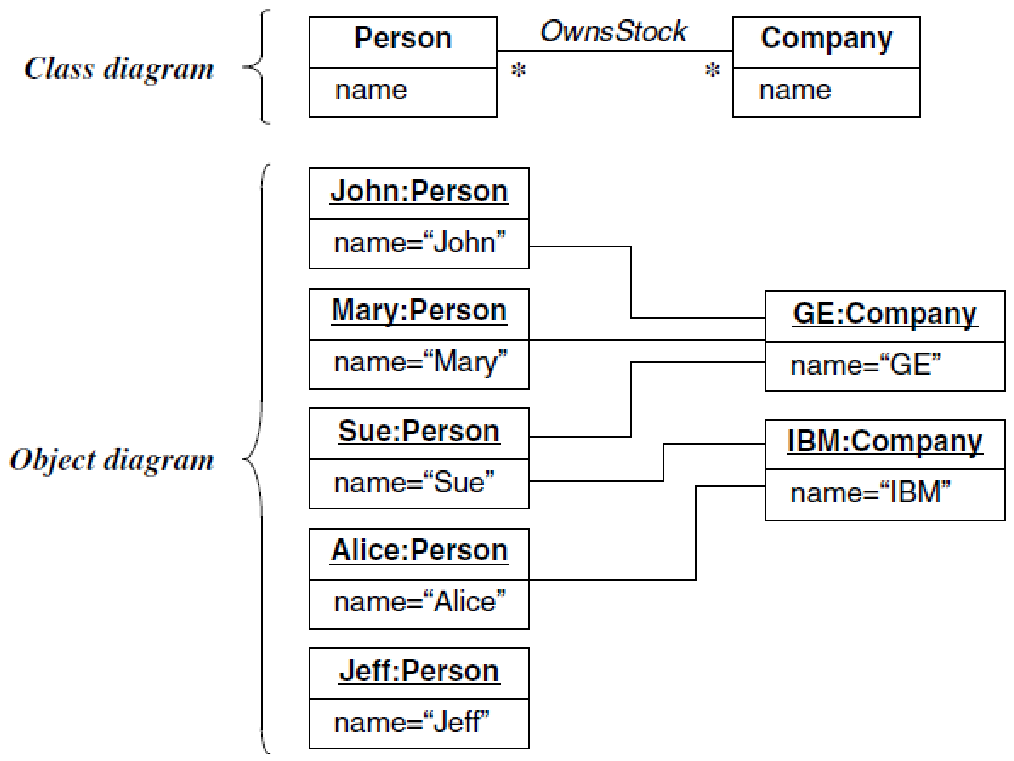


Figure 3.7 Many-to-many association. An association describes a set of potential links in the same way that a class describes a set of potential objects.

A link is a relationship among objects. Modeling a link as a reference disguises the fact that the link is not part of either object by itself, but depends on both of them together. A company is not part of a person, and a person is not part of a company. Furthermore, using a pair of matched references, such as the reference from *Person* to *Company* and the reference from *Company* to a set of *Persons,* hides the fact that the forward and inverse references depend

on each other. Therefore, you should model all connections among classes as associations, even in designs for programs.

Generalization & Inheritance

Generalization is the relationship between a class (the ***superclass***) and one or more variations of the class (the ***subclasses***). Generalization organizes classes by their similarities and differences, structuring the description of objects. The superclass holds common attributes, operations, and associations; the subclasses add specific attributes, operations, and associations.

Each subclass is said to ***inherit*** the features of its superclass. Generalization is sometimes called the "is-a" relationship, because each instance of a subclass is an instance of the superclass as well. Simple generalization organizes classes into a hierarchy; each subclass has a single immediate superclass. (Chapter 4 discusses a more complex form of generalization in which a subclass may have multiple immediate superclasses.) There can be multiple levels of generalizations. Figure 3.24 shows several examples of generalization for equipment. Each piece of equipment is a pump, heat exchanger, or tank. There are several kinds of pumps: centrifugal, diaphragm, and plunger. There are several kinds of tanks: spherical, pressurized, and floating roof. The fact that the tank generalization symbol is drawn below the pump generalization symbol is not significant. Several objects are displayed at the bottom of the figure. Each object inherits features from one class at each level of the generalization. Thus *P101* embodies the features of equipment, pump, and diaphragm pump. *E302* has the properties of equipment and heat exchanger. A large hollow arrowhead denotes generalization. The arrowhead points to the superclass.

You may directly connect the superclass to each subclass, but we normally prefer to group subclasses as a tree. For convenience, you can rotate the triangle and place it on any side, but if possible you should draw the superclass on top and the subclasses on the bottom. The curly braces denote a UML comment, indicating that there are additional subclasses that

the diagram does not show.

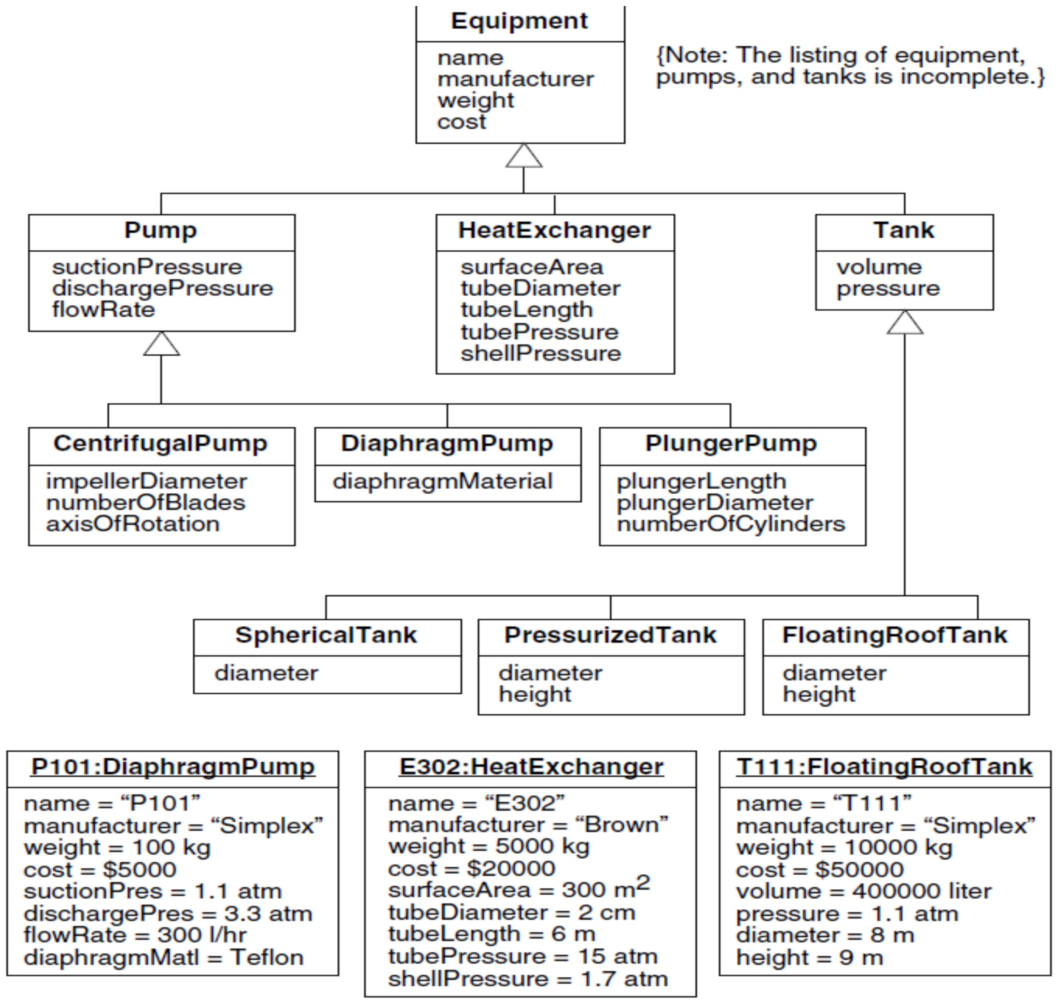


Figure 3.24 **A multilevel inheritance hierarchy with instances.** Generalization organizes classes by their similarities and differences, structuring the description of objects.

Generalization is transitive across an arbitrary number of levels. The terms ***ancestor*** and ***descendant*** refer to generalization of classes across multiple levels. An instance of a subclass is simultaneously an instance of all its ancestor classes. An instance includes a value for every attribute of every ancestor class. An instance can invoke any operation on any ancestor class. Each subclass not only inherits all the features of its ancestors but adds its own specific features as well. For example, *Pump* adds attributes *suctionPressure*, *dischargePressure*, and *flowRate*, which other kinds of equipment do not share.

Navigation of class models

So far we have shown how class models can express the structure of an application. Now we show how they can also express the behavior of navigating among classes. Navigation is important because it lets you exercise a model and uncover hidden flaws and omissions so that you can repair them. You can perform navigation manually (an informal technique) or write

navigation expressions (as we will explain).

Consider the simple model for credit card accounts in Figure 3.27. An institution may issue many credit card accounts, each identified by an account number. Each account has a maximum credit limit, a current balance, and a mailing address. The account serves one or more customers who reside at the mailing address. The institution periodically issues a statement for each account. The statement lists a payment due date, finance charge, and minimum payment. The statement itemizes various transactions that have occurred throughout the billing interval: cash advances, interest charges, purchases, fees, and adjustments to the account. The name of the merchant is printed for each purchase.

We can pose a variety of questions against the model.

■ What transactions occurred for a credit card account within a time interval?

■ What volume of transactions were handled by an institution in the last year?

■ What customers patronized a merchant in the last year by any kind of credit card?

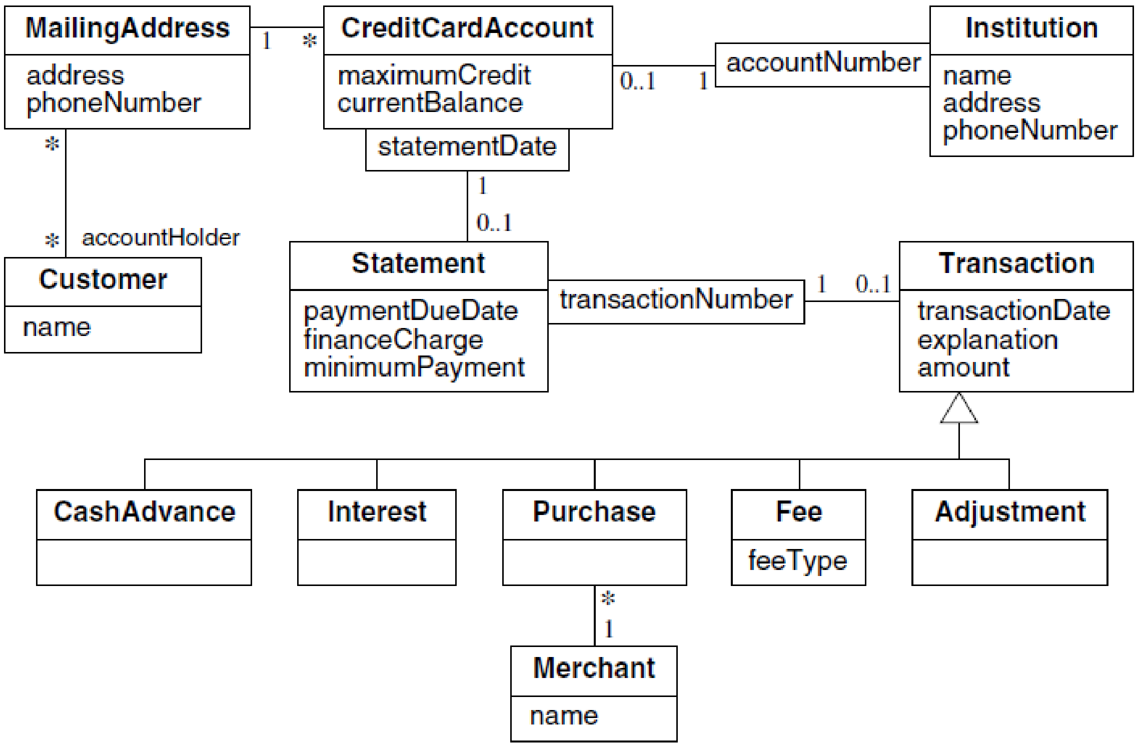


Figure 3.27 Class model for managing credit card accounts

How many credit card accounts does a customer currently have?

■ What is the total maximum credit for a customer, for all accounts?

The UML incorporates a language that can express these kinds of questions—the ***Object***

Constraint Language (OCL) [Warmer-99]. The next two sections discuss the OCL, and Section 3.5.3 then expresses the credit card questions using the OCL. By no means do we cover the complete OCL; we just cover the portions relevant to traversing class models.