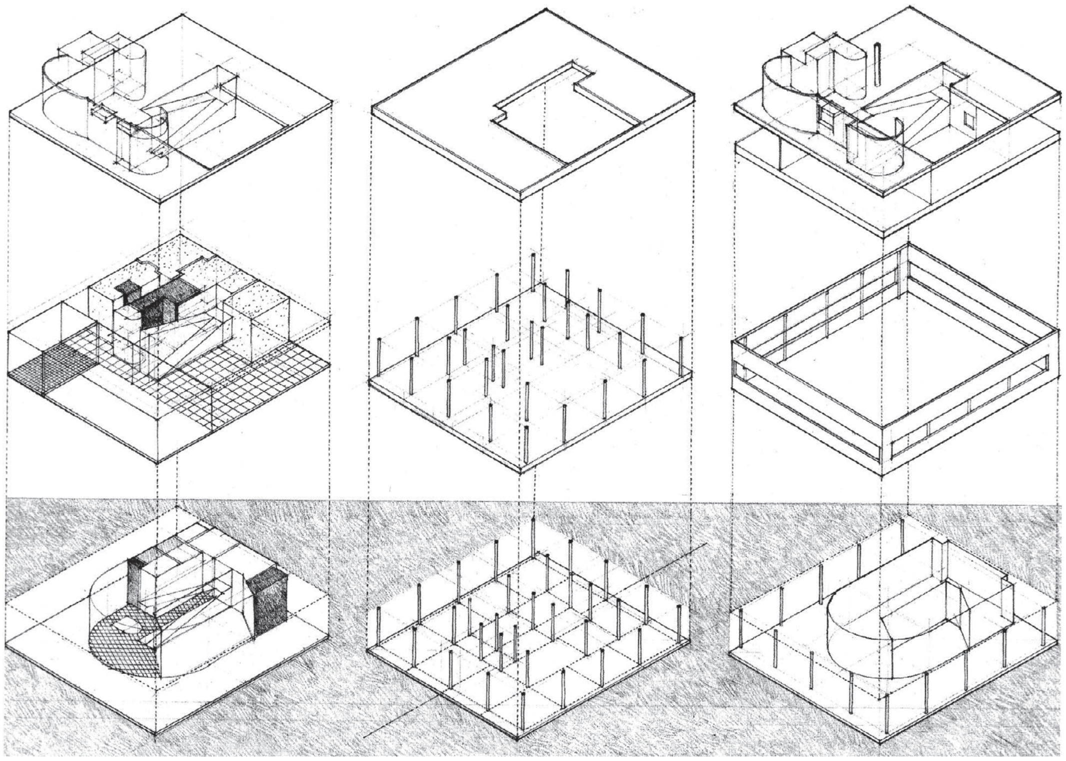
UNIFIED
MODELING
LANGUAGE

Part 2
BASIC STRUCTURAL MODELING



Chapter 4
CLASSES

In this chapter

- Classes, attributes, operations, and responsibilities
- Modeling the vocabulary of a system
- Modeling the distribution of responsibilities in a system
- Modeling nonsoftware things
- Modeling primitive types
- Making quality abstractions

Classes are the most important building block of any object-oriented system. A class is a description of a set of objects that share the same attributes, operations, relationships, and semantics. A class implements one or more interfaces.

Advanced features of classes are discussed in Chapter 9.

You use classes to capture the vocabulary of the system you are developing. These classes may include abstractions that are part of the problem domain, as well as classes that make up an implementation. You can use classes to represent software things, hardware things, and even things that are purely conceptual.

Well-structured classes have crisp boundaries and form a part of a balanced distribution of responsibilities across the system.

Getting Started

Modeling a system involves identifying the things that are important to your particular view. These things form the vocabulary of the system you are modeling. For example, if you are building a house, things like walls, doors, windows, cabinets, and lights are some of the things that will be important to you as a home owner. Each of these things can be distinguished from the other.

Each of them also has a set of properties. Walls have a height and a width and are solid. Doors also have a height and a width and are solid as well, but have the additional behavior that allows them to open in one direction. Windows are similar to doors in that both are openings that pass through walls, but windows and doors have slightly different properties. Windows are usually (but not always) designed so that you can look out of them instead of pass through them.

Individual walls, doors, and windows rarely exist in isolation, so you must also consider how specific instances of these things fit together. The things you identify and the relationships you choose to establish among them will be affected by how you expect to use the various rooms of your home, how you expect traffic to flow from room to room, and the general style and feel you want this arrangement to create.

Users will be concerned about different things. For example, the plumbers who help build your house will be interested in things like drains, traps, and vents. You, as a home owner, won't necessarily care about these things except insofar as they interact with the things in your view, such as where a drain might be placed in a floor or where a vent might intersect with the roof line.

Objects are discussed in Chapter 13.

In the UML, all of these things are modeled as classes. A class is an abstraction of the things that are a part of your vocabulary. A class is not an individual object, but rather represents a whole set of objects. Thus, you may conceptually think of "wall" as a class of objects with certain common properties, such as height, length, thickness, load-bearing or not, and so on. You may also think of individual instances of wall, such as "the wall in the southwest corner of my study."

In software, many programming languages directly support the concept of a class. That's excellent, because it means that the abstractions you create can often be mapped directly to a programming language, even if these are abstractions of nonsoftware things, such as "customer," "trade," or "conversation."

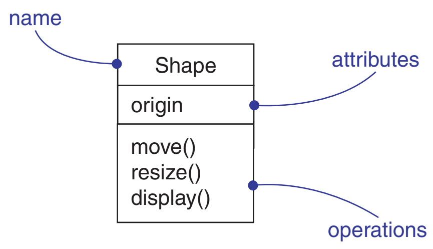


Figure 4-1: Classes

The UML provides a graphical representation of class, as well, as Figure 4-1 shows. This notation permits you to visualize an abstraction apart from any specific programming language and in a way that lets you emphasize the most important parts of an abstraction: its name, attributes, and operations.

Terms and Concepts

A *class* is a description of a set of objects that share the same attributes, operations, relationships, and semantics. Graphically, a class is rendered as a rectangle.

Names

A class name must be unique within its enclosing package, as discussed in Chapter 12.

Every class must have a name that distinguishes it from other classes. A *name* is a textual string. That name alone is known as a *simple name*; a *qualified name* is the class name prefixed by the name of the package in which that class lives. A class may be drawn showing only its name, as Figure 4-2 shows.

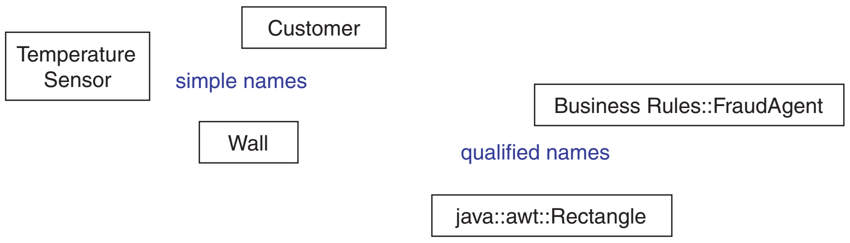


Figure 4-2: Simple and Qualified Names

Note: A class name may be text consisting of any number of letters, numbers, and certain punctuation marks (except for marks such as the double colon, which is used to separate a class name and the name of its enclosing package) and may continue over several lines. In practice, class names are short nouns or noun phrases drawn from the vocabulary of the system you are modeling. Typically, you capitalize the first letter of every word in a class name, as in `Customer` or `TemperatureSensor`.

Attributes

Attributes are related to the semantics of aggregation, as discussed in Chapter 10.

An *attribute* is a named property of a class that describes a range of values that instances of the property may hold. A class may have any number of attributes or no attributes at all. An attribute represents some property of the thing you are modeling that is shared by all objects of that class. For example, every wall has a height, width, and thickness; you might model your customers in such a way that each has a name, address, phone number, and date of birth. An attribute is therefore an abstraction of the kind of data or state an object of the class might encompass. At a given moment, an object of a class will have specific values for every one of its class's attributes. Graphically, attributes are listed in a compartment just below the class name. Attributes may be drawn showing only their names, as shown in Figure 4-3.

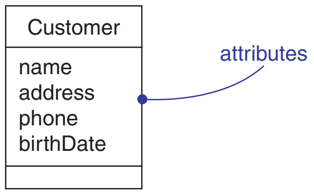


Figure 4-3: Attributes

Note: An attribute name may be text, just like a class name. In practice, an attribute name is a short noun or noun phrase that represents some property of its enclosing class. Typically, you capitalize the first letter of every word in an attribute name except the first letter, as in `name` or `loadBearing`.

You can specify other features of an attribute, such as marking it read-only or shared by all objects of the class, as discussed in Chapter 9.

You can further specify an attribute by stating its class and possibly a default initial value, as shown Figure 4-4.

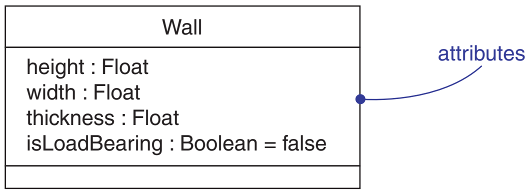


Figure 4-4: Attributes and Their Class

Operations

You can further specify the implementation of an operation by using a note, as described in Chapter 6, or by using an activity diagram, as discussed in Chapter 20.

An *operation* is the implementation of a service that can be requested from any object of the class to affect behavior. In other words, an operation is an abstraction of something you can do to an object that is shared by all objects of that class. A class may have any number of operations or no operations at all. For example, in a windowing library such as the one found in Java's `awt` package, all objects of the class `Rectangle` can be moved, resized, or queried for their properties. Often (but not always), invoking an operation on an object changes the object's data or state. Graphically, operations are listed in a compartment just below the class attributes. Operations may be drawn showing only their names, as in Figure 4-5.

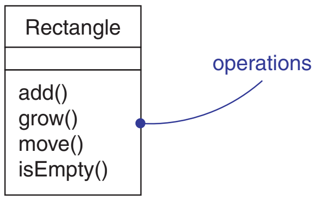


Figure 4-5: Operations

Note: An operation name may be text, just like a class name. In practice, an operation name is a short verb or verb phrase that represents some behavior of its enclosing class. Typically, you capitalize the first letter of every word in an operation name except the first letter, as in `move` or `isEmpty`.

You can specify other features of an operation, such as marking it polymorphic or constant, or specifying its visibility, as discussed in Chapter 9.

You can specify an operation by stating its signature, which includes the name, type, and default value of all parameters and (in the case of functions) a return type, as shown in Figure 4-6.

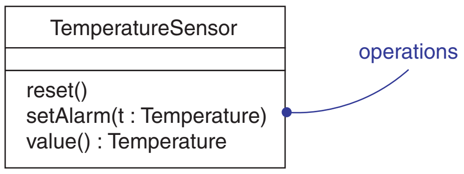


Figure 4-6: Operations and Their Signatures

Organizing Attributes and Operations

When drawing a class, you don't have to show every attribute and every operation at once. In fact, in most cases, you can't (there are too many of them to put in one figure) and you probably shouldn't (only a subset of these attributes and operations are likely to be relevant to a specific view). For these reasons, you can elide a class, meaning that you can choose to show only some or none of a class's attributes and operations. You can indicate that there are more attributes or properties than shown by ending each list with an ellipsis ("`...`"). You can also suppress the compartment entirely, in which case you can't tell if there are any attributes or operations or how many there are.

Stereotypes are discussed in Chapter 6.

To better organize long lists of attributes and operations, you can also prefix each group with a descriptive category by using stereotypes, as shown in Figure 4-7.

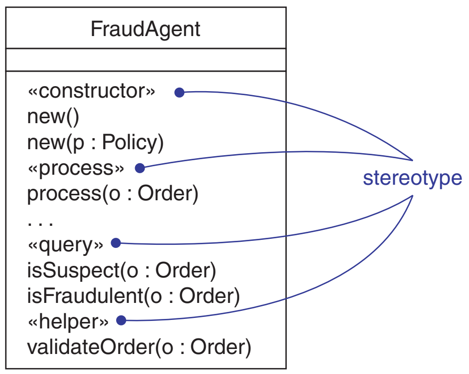


Figure 4-7: Stereotypes for Class Features

Responsibilities

Responsibilities are an example of a defined stereotype, as discussed in Chapter 6.

A *responsibility* is a contract or an obligation of a class. When you create a class, you are making a statement that all objects of that class have the same kind of state and the same kind of behavior. At a more abstract level, these corresponding attributes and operations are just the features by which the class's responsibilities are carried out. A `Wall` class is responsible for knowing about height, width, and thickness; a `FraudAgent` class, as you might find in a credit card application, is responsible for processing orders and determining if they are legitimate, suspect, or fraudulent; a `TemperatureSensor` class is responsible for measuring temperature and raising an alarm if the temperature reaches a certain point.

Modeling the semantics of a class is discussed in Chapter 9.

When you model classes, a good starting point is to specify the responsibilities of the things in your vocabulary. Techniques like CRC cards and use case-based analysis are especially helpful here. A class may have any number of responsibilities, although, in practice, every well-structured class has at least one responsibility and at most just a handful. As you refine your models, you will translate these responsibilities into a set of attributes and operations that best fulfill the class's responsibilities.

You can also draw the responsibilities of a class in a note, as discussed in Chapter 6.

Graphically, responsibilities can be drawn in a separate compartment at the bottom of the class icon, as shown in Figure 4-8.

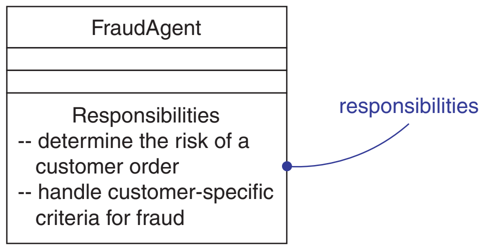

FraudAgent
Responsibilities
-- determine the risk of a customer order
-- handle customer-specific criteria for fraud
responsibilities

Figure 4-8: Responsibilities

Note: Responsibilities are just free-form text. In practice, a single responsibility is written as a phrase, a sentence, or (at most) a short paragraph.

Other Characteristics

Advanced class concepts are discussed in Chapter 9.

Attributes, operations, and responsibilities are the most common features you'll need when you create abstractions. In fact, for most models you build, the basic form of these three features will be all you need to convey the most important semantics of your classes. Sometimes, however, you'll need to visualize or specify other characteristics, such as the visibility of individual attributes and operations; language-specific features of an operation, such as whether it is polymorphic or constant; or even the exceptions that objects of the class might produce or handle. These and many other features can be expressed in the UML, but they are treated as advanced concepts.

Interfaces are discussed in Chapter 11.

When you build models, you will soon discover that almost every abstraction you create is some kind of class. Sometimes you will want to separate the implementation of a class from its specification, and this can be expressed in the UML by using interfaces.

Internal structure is discussed in Chapter 15.

When you start designing the implementation of a class, you need to model its internal structure as a set of connected parts. You can expand a top-level class through several layers of internal structure to get the eventual design.

Active classes, components, and nodes are discussed in Chapters 23, 25, and 27, and artifacts are discussed in Chapter 26.

When you start building more complex models, you will also find yourself encountering the same kinds of entities over and over again, such as classes that represent concurrent processes and threads, or classifiers that represent physical things, such as applets, Java Beans, files, Web pages, and hardware. Because these kinds of entities are so common and because they represent important architectural abstractions, the UML provides active classes (representing processes and threads) and classifiers, such as artifacts (representing physical software components) and nodes (representing hardware devices).

Class diagrams are discussed in Chapter 8.

Finally, classes rarely stand alone. Rather, when you build models, you will typically focus on groups of classes that interact with one another. In the UML, these societies of classes form collaborations and are usually visualized in class diagrams.

Common Modeling Techniques

Modeling the Vocabulary of a System

You'll use classes most commonly to model abstractions that are drawn from the problem you are trying to solve or from the technology you are using to implement a solution to that problem. Each of these abstractions is a part of the vocabulary of your system, meaning that, together, they represent the things that are important to users and to implementers.

Use cases are discussed in Chapter 17.

For users, most abstractions are not that hard to identify because, typically, they are drawn from the things that users already use to describe their system. Techniques such as CRC cards and use case-based analysis are excellent ways to help users find these abstractions. For implementers, these abstractions are typically just the things in the technology that are parts of the solution.

To model the vocabulary of a system,

- Identify those things that users or implementers use to describe the problem or solution. Use CRC cards and use case-based analysis to help find these abstractions.

- For each abstraction, identify a set of responsibilities. Make sure that each class is crisply defined and that there is a good balance of responsibilities among all your classes.
- Provide the attributes and operations that are needed to carry out these responsibilities for each class.

Figure 4-9 shows a set of classes drawn from a retail system, including `Customer`, `Order`, and `Product`. This figure includes a few other related abstractions drawn from the vocabulary of the problem, such as `Shipment` (used to track orders), `Invoice` (used to bill orders), and `Warehouse` (where products are located prior to shipment). There is also one solution-related abstraction, `Transaction`, which applies to orders and shipments.

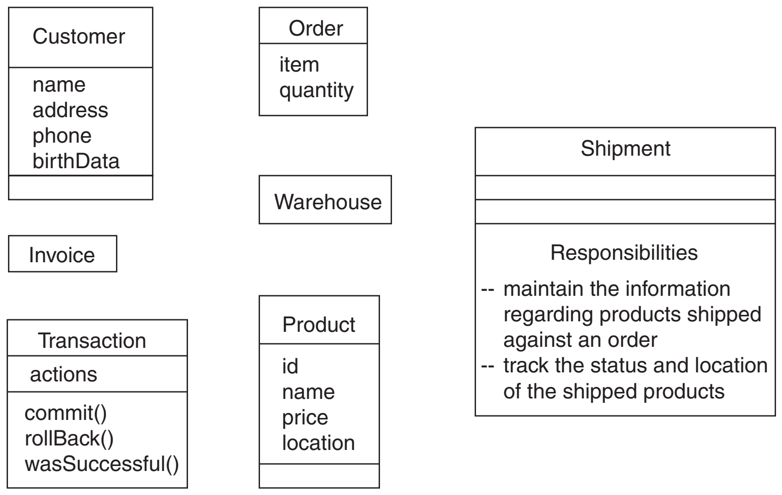


Figure 4-9: Modeling the Vocabulary of a System

Packages are discussed in Chapter 12.

As your models get larger, many of the classes you find will tend to cluster together in groups that are conceptually and semantically related. In the UML, you can use packages to model these clusters of classes.

Modeling behavior is discussed in Parts 4 and 5.

Your models will rarely be completely static. Instead, most abstractions in your system's vocabulary will interact with one another in dynamic ways. In the UML, there are a number of ways to model this dynamic behavior.

Modeling the Distribution of Responsibilities in a System

Once you start modeling more than just a handful of classes, you will want to be sure that your abstractions provide a balanced set of responsibilities. What this means is that you don't want any one class to be too big or too small. Each class should do one thing well. If you abstract classes that are too big, you'll find that your models are hard to change and are not very reusable. If you abstract classes that are too small, you'll end up with many more abstractions than you can reasonably manage or understand. You can use the UML to help you visualize and specify this balance of responsibilities.

To model the distribution of responsibilities in a system,

- Identify a set of classes that work together closely to carry out some behavior.
- Identify a set of responsibilities for each of these classes.
- Look at this set of classes as a whole, split classes that have too many responsibilities into smaller abstractions, collapse tiny classes that have trivial responsibilities into larger ones, and reallocate responsibilities so that each abstraction reasonably stands on its own.
- Consider the ways in which those classes collaborate with one another, and redistribute their responsibilities accordingly so that no class within a collaboration does too much or too little.

Collaborations are discussed in Chapter 28.

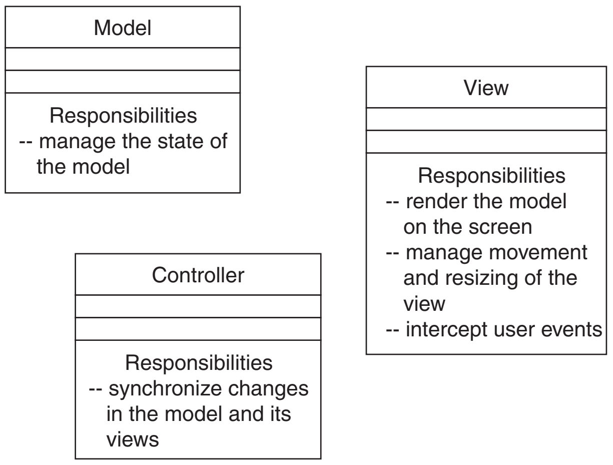

Model
Responsibilities
-- manage the state of the model
View
Responsibilities
-- render the model on the screen
-- manage movement and resizing of the view
-- intercept user events
Controller
Responsibilities
-- synchronize changes in the model and its views

Figure 4-10: Modeling the Distribution of Responsibilities in a System

user.book Page 57 Friday, April 8, 2005 10:05 AM

This set of classes forms a pattern, as discussed in Chapter 29.

For example, Figure 4-10 shows a set of classes drawn from Smalltalk, showing the distribution of responsibilities among `Model`, `View`, and `Controller` classes. Notice how all these classes work together such that no one class does too much or too little.

Modeling Nonsoftware Things

Sometimes, the things you model may never have an analog in software. For example, the people who send invoices and the robots that automatically package orders for shipping from a warehouse might be a part of the workflow you model in a retail system. Your application might not have any software that represents them (unlike customers in the example above, since your system will probably want to maintain information about them).

To model nonsoftware things,

Stereotypes are discussed in Chapter 6.

Nodes are discussed in Chapter 27.

- Model the thing you are abstracting as a class.
- If you want to distinguish these things from the UML's defined building blocks, create a new building block by using stereotypes to specify these new semantics and to give a distinctive visual cue.
- If the thing you are modeling is some kind of hardware that itself contains software, consider modeling it as a kind of node as well, so that you can further expand on its structure.

Note: The UML is mainly intended for modeling software-intensive systems, although, in conjunction with textual hardware modeling languages, such as VHDL, the UML can be quite expressive for modeling hardware systems. The OMG has also produced a UML extension called SysML intended for systems modeling.

External things are often modeled as actors, as discussed in Chapter 17.

As Figure 4-11 shows, it's perfectly normal to abstract humans (like `AccountsReceivableAgent`) and hardware (like `Robot`) as classes, because each represents a set of objects with a common structure and a common behavior.

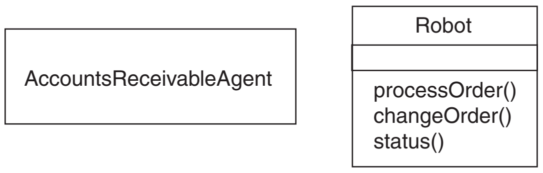


Figure 4-11: Modeling Nonsoftware Things

Modeling Primitive Types

Types are discussed in Chapter 11.

At the other extreme, the things you model may be drawn directly from the programming language you are using to implement a solution. Typically, these abstractions involve primitive types, such as integers, characters, strings, and even enumeration types, that you might create yourself.

To model primitive types,

- Model the thing you are abstracting as a class or an enumeration, which is rendered using class notation with the appropriate stereotype.
- If you need to specify the range of values associated with this type, use constraints.

Constraints are described in Chapter 6.

Types are discussed in Chapter 11.

As Figure 4-12 shows, these things can be modeled in the UML as types or enumerations, which are rendered just like classes but are explicitly marked via stereotypes. Primitive types such as integers (represented by the class `Int`) are modeled as types, and you can explicitly indicate the range of values these things can take on by using a constraint; the semantics of primitive types must be defined externally to UML. Enumeration types, such as `Boolean` and `Status`, can be modeled as enumerations, with their individual literals listed within the attribute compartment (note that they are not attributes). Enumeration types may also define operations.

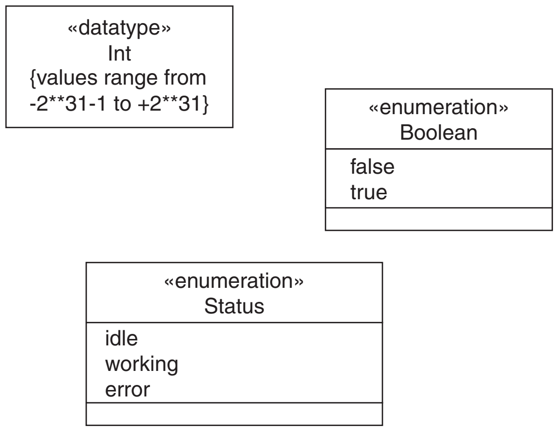


Figure 4-12: Modeling Primitive Types

Note: Some languages, such as C and C++, let you set an integer value for an enumeration literal. You can model this in the UML by attaching a note to an enumeration literal as implementation guidance. Integer values are not needed for logical modeling.

Hints and Tips

When you model classes in the UML, remember that every class should map to some tangible or conceptual abstraction in the domain of the end user or the implementer. A well-structured class

- Provides a crisp abstraction of something drawn from the vocabulary of the problem domain or the solution domain.
- Embodies a small, well-defined set of responsibilities and carries them all out very well.
- Provides a clear separation of the abstraction's specification and its implementation.
- Is understandable and simple, yet extensible and adaptable.

When you draw a class in the UML,

- Show only those properties of the class that are important to understanding the abstraction in its context.
- Organize long lists of attributes and operations by grouping them according to their category.
- Show related classes in the same class diagrams.