Contents

(* source Fowler, Martin, 1963-UML distilled: a brief guide to the standard object modeling language / Martin)

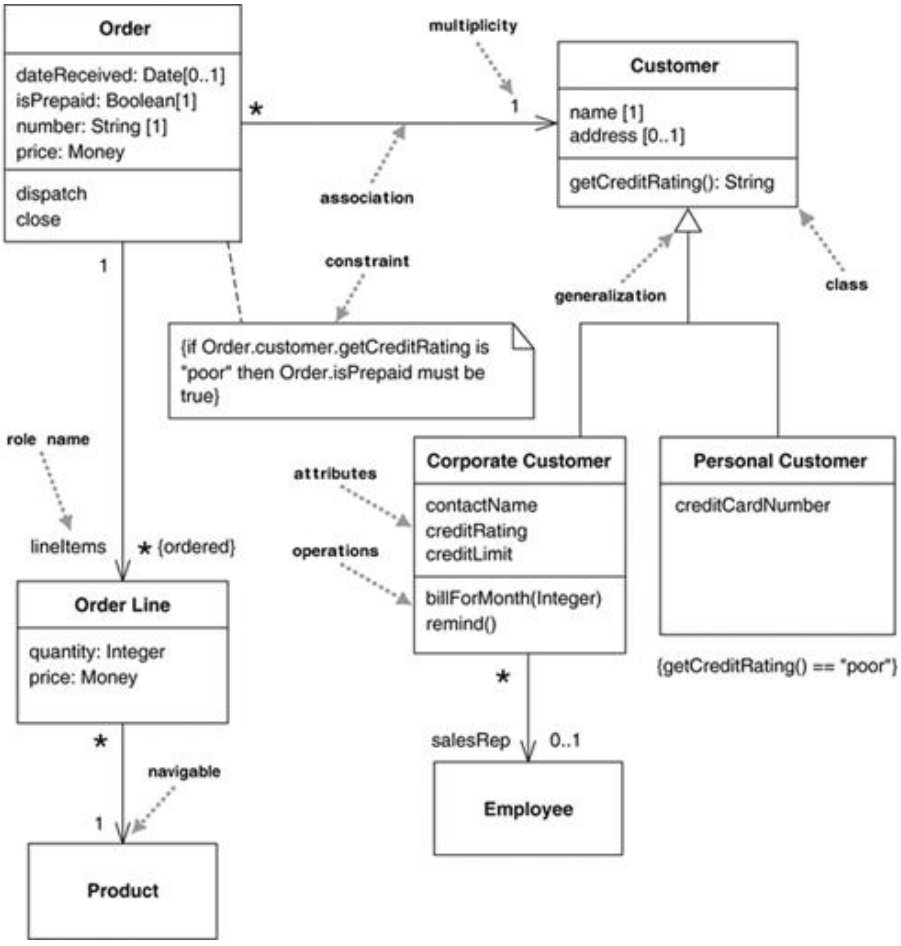
Class Diagrams: The Essentials

If someone were to come up to you in a dark alley and say, "Psst, wanna see a UML diagram?" that diagram would probably be a class diagram. The majority of UML diagrams I see are class diagrams.

The class diagram is not only widely used but also subject to the greatest range of modeling concepts. Although the basic elements are needed by everyone, the advanced concepts are used less often. *(advanced is out of scope for the assignments – Andy H)*

A class diagram describes the types of objects in the system and the various kinds of static relationships that exist among them. Class diagrams also show the properties and operations of a class and the constraints that apply to the way objects are connected. The UML uses the term feature as a general term that covers properties and operations of a class.

Figure 3.1 shows a simple class model that would not surprise anyone who has worked with order processing. The boxes in the diagram are classes, which are divided into three compartments: the name of the class (in bold), its attributes, and its operations. Figure 3.1 also shows two kinds of relationships between classes: associations and generalizations.



Properties

Properties represent structural features of a class. As a first approximation, you can think of properties as corresponding to fields in a class. The reality is rather involved, as we shall see, but that's a reasonable place to start.

Properties are a single concept, but they appear in two quite distinct notations: attributes and associations. Although they look quite different on a diagram, they are really the same thing.

Attributes

The attribute notation describes a property as a line of text within the class box itself. The full form of an attribute is:

```
visibility name: type multiplicity = default {property-string}
```

An example of this is:

```
- name: String [1] = "Untitled" {readOnly}
```

Only the `name` is necessary.

- This `visibility` marker indicates whether the attribute is public (+) or private (-);
- The `name` of the attribute—how the class refers to the attribute—roughly corresponds to the name of a field in a programming language.
- The `type` of the attribute indicates a restriction on what kind of object may be placed in the attribute. You can think of this as the type of a field in a programming language.
- See page 3 for more on `multiplicity`.
- The `default value` is the value for a newly created object if the attribute isn't specified during creation.
- The `{property-string}` allows you to indicate additional properties for the attribute. In the example, I used `{readOnly}` to indicate that clients may not modify the property. If this is missing, you can usually assume that the attribute is modifiable. I'll describe other property strings as we go.

Associations

The other way to notate a property is as an association. Much of the same information that you can show on an attribute appears on an association. Figures 3.2 and 3.3 show the same properties represented in the two different notations.

Figure 3.2. Showing properties of an order as attributes

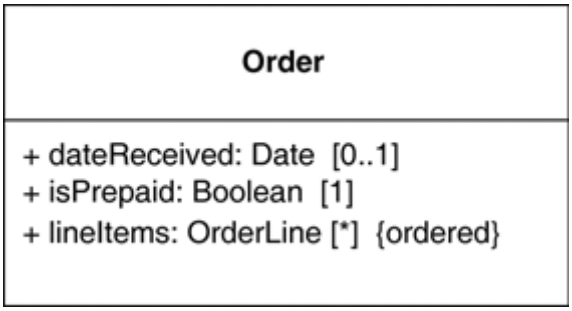
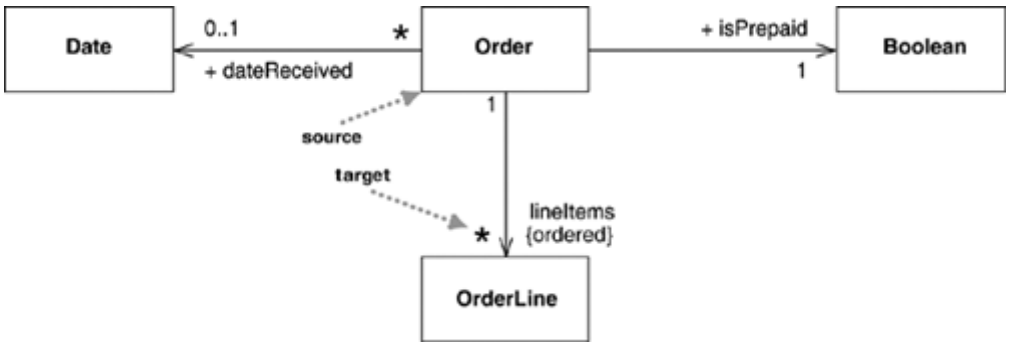


Figure 3.3. Showing properties of an order as associations



An association is a solid line between two classes, directed from the source class to the target class. The name of the property goes at the target end of the association, together with its multiplicity. The target end of the association links to the class that is the type of the property.

Although most of the same information appears in both notations, some items are different. In particular, associations can show multiplicities at both ends of the line.

With two notations for the same thing, the obvious question is, why should you use one or the other? In general, use attributes for small things, such as dates or Booleans and associations for more significant classes, such as customers and orders. I also tend to prefer to use class boxes for classes that are significant for the diagram, which leads to using associations, and attributes for things less important for that diagram. The choice is much more about emphasis than about any underlying meaning.

Multiplicity

The multiplicity of a property is an indication of how many objects may fill the property. The most common multiplicities you will see are

- **1** (An order must have exactly one customer.)
- **0..1** (A corporate customer may or may not have a single sales rep.)
- ***** (A customer need not place an Order and there is no upper limit to the number of Orders a Customer may place—zero or more orders.)

More generally, multiplicities are defined with a lower bound and an upper bound, such as 2..4 for players of a game of canasta. The lower bound may be any positive number or zero; the upper is any positive number or * (for unlimited). If the lower and upper bounds are the same, you can use one number; hence, 1 is equivalent to 1..1. Because it's a common case, * is short for 0..*.

In attributes, you come across various terms that refer to the multiplicity.

- **Optional** implies a lower bound of 0.
- **Mandatory** implies a lower bound of 1 or possibly more.
- **Single-valued** implies an upper bound of 1.
- **Multivalued** implies an upper bound of more than 1: usually *.

If I have a **multivalued** property, I prefer to use a plural form for its name.

By default, the elements in a **multivalued** multiplicity form a set, so if you ask a customer for its orders, they do not come back in any order. If the ordering of the orders in association has meaning, you need to add `{ordered}` to the association end. If you want to allow duplicates, add `{nonunique}`. (If you want to explicitly show the default, you can use `{unordered}` and `{unique}`.) You may also see collection-oriented names, such as `{bag}` for unordered, nonunique.

UML 1 allowed discontinuous multiplicities, such as 2, 4 (meaning 2 or 4, as in cars in the days before minivans). Discontinuous multiplicities weren't very common and UML 2 removed them.

The default multiplicity of an attribute is [1]. Although this is true in the meta-model, you can't assume that an attribute in a diagram that's missing a multiplicity has a value of [1], as the diagram may be suppressing the multiplicity information. As a result, I prefer to explicitly state a [1] multiplicity if it's important.

Programming Interpretation of Properties

As with anything else in the UML, there's no one way to interpret properties in code. The most common software representation is that of a field or property of your programming language. So the Order Line class from Figure 3.1 would correspond to something like the following in Java:

```
public class OrderLine...
  private int quantity;
  private Money price;
  private Order order;
  private Product product
```

In a language like C#, which has properties, it would correspond to:

```
public class OrderLine ...
  public int Quantity;
  public Money Price;
  public Order Order;
  public Product Product;
```

Note that an attribute typically corresponds to public properties in a language that supports properties but to private fields in a language that does not. In a language without properties, you may see the fields exposed through **accessor** (*getting and setting*) methods. A **read-only** attribute will have no setting method (with fields) or set action (for properties). Note that if you don't give a name for a property, it's common to use the name of the target class.

Using private fields is a very *implementation-focused* interpretation of the diagram. A more *interface-oriented* interpretation might instead concentrate on the getting methods rather than the underlying data. In this case, we might see the Order Line's attributes corresponding to the following methods:

```
public class OrderLine...
  private int quantity;
  private Product product;
  public int getQuantity() {
    return quantity;
  }
  public void setQuantity(int quantity) {
    this.quantity = quantity;
  }
  public Money getPrice() {
    return product.getPrice().multiply(quantity);
  }
```

In this case, there is no data field for price; instead, it's a computed value. But as far as clients of the Order Line class are concerned, it looks the same as a field. Clients can't tell what is a field and what is computed. This information hiding is the essence of encapsulation.

If an attribute is **multivalued**, this implies that the data concerned is a collection. So an Order class would refer to a collection of Order Lines. Because this multiplicity is ordered, that collection must be ordered, (*such as a List in Java or an IList in .NET*). If the collection is unordered, it should, strictly, have no meaningful order and thus be implemented with a set, but most people implement unordered attributes as lists as well. Some people use arrays, but the UML implies an unlimited upper bound, so I almost always use a collection for data structure.

Multivalued properties yield a different kind of interface to single-valued properties (in Java):

```
class Order {
  private Set lineItems = new HashSet();
  public Set getLineItems() {
    return Collections.unmodifiableSet(lineItems);
  }
  public void addLineItem (OrderItem arg) {
    lineItems.add (arg);
  }
  public void removeLineItem (OrderItem arg) {
    lineItems.remove(arg);
  }
```

In most cases, you don't assign to a **multivalued** property; instead, you update with add and remove methods. In order to control its Line Items property, the order must control membership of that collection; as a result, it shouldn't pass out the naked collection. In this case, I used a protection proxy to provide a read-only wrapper to the collection. You can also provide a **nonupdatable iterator** or make a copy. Its okay for clients to modify the member objects, but the clients shouldn't directly change the collection itself.

Because **multivalued** attributes imply collections, you almost never see collection classes on a class diagram. You would show them only in very low level implementation diagrams of collections themselves.

You should be very afraid of classes that are nothing but a collection of fields and their **accessors**. Object-oriented design is about providing objects that are able to do rich behavior, so they shouldn't be simply providing data to other objects. If you are making repeated calls for data by using **accessors**, that's a sign that some behavior should be moved to the object that has the data.

These examples also reinforce the fact that there is no hard-and-fast correspondence between the UML and code, yet there is a similarity. Within a project team, team conventions will lead to a closer correspondence.

Whether a property is implemented as a field or as a calculated value, it represents something an object can always provide. You shouldn't use a property to model a transient relationship, such as an object that is passed as a parameter during a method call and used only within the confines of that interaction.

Bidirectional Associations

The associations we've looked at so far are called unidirectional associations. Another common kind of association is a bidirectional association, such as Figure 3.4.

Figure 3.4. A bidirectional association

A bidirectional association is a pair of properties that are linked together as inverses. The Car class has property `owner:Person[1]`, and the Person class has a property `cars:Car[*]`. (Note how I named the `cars` property in the plural form of the property's type, a common but non-normative convention.)

The inverse link between them implies that if you follow both properties, you should get back to a set that contains your starting point. For example, if I begin with a particular MG Midget, find its owner, and then look at its owner's cars, that set should contain the Midget that I started from.

As an alternative to labeling an association by a property, many people, particularly if they have a data-modeling background, like to label an association by using a verb phrase (Figure 3.5) so that the relationship can be used in a sentence. This is legal and you can add an arrow to the association to avoid ambiguity. Most object modelers prefer to use a property name, as that corresponds better to responsibilities and operations.

Figure 3.5. Using a verb phrase to name an association

Some people name every association in some way. I choose to name an association only when doing so improves understanding. I've seen too many associations with such names as "has" or "is related to."

In Figure 3.4, the bidirectional nature of the association is made obvious by the navigability arrows at both ends of the association. Figure 3.5 has no arrows; the UML allows you to use this form either to indicate a bidirectional association or when you aren't showing navigability. My preference is to use the double-headed arrow of Figure 3.4 when you want to make it clear that you have a bidirectional association.

Implementing a bidirectional association in a programming language is often a little tricky because you have to be sure that both properties are kept synchronized. Using C#, I use code along these lines to implement a bidirectional association:

```
class Car...
```

```
  public Person Owner {
    get {return _owner;}
    set {
      if (_owner != null) _owner.friendCars().Remove(this);
      _owner = value;
      if (_owner != null) _owner.friendCars().Add(this);
    }
}
private Person _owner;
...
class Person ...
  public IList Cars {
    get {return ArrayList.ReadOnly(_cars);}
  }
  public void AddCar(Car arg) {
    arg.Owner = this;
  }
  private IList _cars = new ArrayList();
  internal IList friendCars() {
    //should only be used by Car.Owner
    return _cars;
  }
....
```

The primary thing is to let one side of the association—a single-valued side, if possible—control the relationship. For this to work, the slave end (Person) needs to leak the encapsulation of its data to the master end. This adds to the slave class an awkward method, which shouldn't really be there, unless the language has fine-grained access control. I've used the naming convention of "friend" here as a nod to C++, where the master's setter would indeed be a friend. Like much property code, this is pretty **boilerplate** stuff(1)(see ref in appendix), which is why many people prefer to use some form of code generation to produce it.

In conceptual models, navigability isn't an important issue, so I don't show any navigability arrows on conceptual models.

Operations

Operations are the actions that a class knows to carry out. Operations most obviously correspond to the methods on a class. Normally, you don't show those operations that simply manipulate properties, because they can usually be inferred.

The full UML syntax for operations is:

```
visibility name (parameter-list) : return-type {property-string}
```

- This `visibility` marker is public (+) or private (-);
- The `name` is a string.
- The `parameter-list` is the list of parameters for the operation.
- The `return-type` is the type of the returned value, if there is one.
- The `property-string` indicates property values that apply to the given operation.

The parameters in the parameter list are notated in a similar way to attributes. The form is:

```
direction name: type = default value
```

- The `name`, `type`, and `default value` are the same as for attributes.
- The `direction` indicates whether the parameter is input (`in`), output (`out`) or both (`inout`). If no direction is shown, it's assumed to be `in`.

An example operation on account might be:

```
+ balanceOn (date: Date) : Money
```

With conceptual models, you shouldn't use operations to specify the interface of a class. Instead, use them to indicate the principal responsibilities of that class, perhaps using a couple of words summarizing *a CRC responsibility*(2) (see appendix for reference if you're curious)

I often find it useful to distinguish between operations that change the state of the system and those that don't. UML defines a query as an operation that gets a value from a class without changing the system state—in other words, without side effects. You can mark such an operation with the property string `{query}`. I refer to operations that do change state as modifiers, also called commands.

Strictly, the difference between query and modifiers is whether they change the observable state [Meyer].

The observable state is what can be perceived from the outside. An operation that updates a cache would alter the internal state but would have no effect that's observable from the outside.

I find it helpful to highlight queries, as you can change the order of execution of queries and not change the system behavior.

A common convention is to try to write operations so that modifiers do not return a value; that way, you can rely on the fact that operations that return a value are queries. [Meyer]

...refers to this as the Command-Query separation principle. It's sometimes awkward to do this all the time, but you should do it as much as you can.

Other terms you sometimes see are getting methods and setting methods. A getting method returns a value from a field (and does nothing else). A setting method puts a value into a field (and does nothing else). From the outside, a client should not be able to tell whether a query is a getting method or a modifier is a setting method. Knowledge of getting and setting methods is entirely internal to the class.

Another distinction is between operation and method. An operation is something that is invoked on an object—the procedure declaration—whereas a method is the body of a procedure. The two are different when you have **polymorphism**. If you have a supertype with three subtypes, each of which overrides the supertype's `getPrice` operation, *you have one operation and four methods that implement it.*

People usually use the terms operation and method (*class's form of function – [Andy H])* interchangeably, but there are times when it is useful to be precise about the difference.

Generalization

A typical example of generalization involves the personal and corporate customers of a business. They have differences but also many similarities. The similarities can be placed in a general Customer class (the supertype), with Personal Customer and Corporate Customer as subtypes.

This phenomenon is also subject to various interpretations at the various perspectives of modeling. Conceptually, we can say that Corporate Customer is a subtype of Customer if all instances of Corporate Customer are also, by definition, instances of Customer. A Corporate Customer is then a *special kind* of Customer. The key idea is that everything we say about a Customer—associations, attributes, operations—is true also for a Corporate Customer.

With a software perspective, the obvious interpretation is *inheritance*: The Corporate Customer is a subclass of Customer. In mainstream OO languages, the **subclass** inherits all the features of the **superclass** and may override any **superclass** methods.

An important principle of using *inheritance* effectively is *substitutability*. I should be able to substitute a Corporate Customer within any code that requires a Customer, and everything should work fine. Essentially, this means that if I write code assuming I have a Customer, I can freely use any subtype of Customer. The Corporate Customer may respond to certain commands differently from another Customer, using **polymorphism**, but the caller should not need to worry about the difference. (For more on this, see the Liskov Substitution Principle (LSP) in [Martin] – *see appendix [Andy H]*.)

Although *inheritance* is a powerful mechanism, it brings in a lot of baggage that isn't always needed to achieve substitutability. A good example of this was in the early days of Java, when many people didn't like the implementation of the built-in Vector class and wanted to replace it with something lighter. However, the only way they could produce a class that was substitutable for Vector was to subclass it and that meant inheriting a lot of unwanted data and behavior.

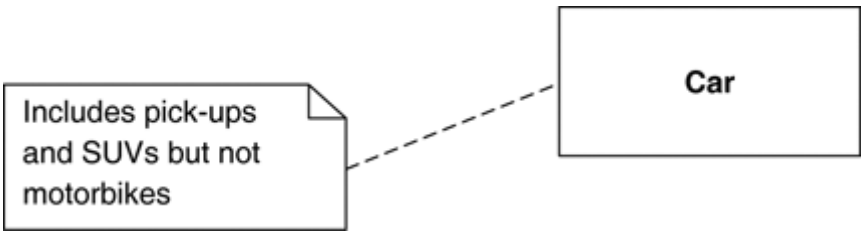
Many other mechanisms can be used to provide substitutable classes. As a result, many people like to differentiate between **subtyping**, or interface inheritance, and subclassing, or implementation inheritance. A *class* is a subtype if it is substitutable for its **supertype**, whether or not it uses inheritance. Subclassing is used as a synonym for regular inheritance.

Many other mechanisms are available that allow you to have subtyping without subclassing. Examples are implementing an interface and many of the standard design patterns [Gang of Four] (*see ref if interested [Andy H])*.

Notes and Comments

Notes are comments in the diagrams. Notes can stand on their own, or they can be linked with a dashed line to the elements they are commenting (Figure 3.6). They can appear in any kind of diagram.

Figure 3.6. A note is used as a comment on one or more diagram elements



The dashed line can sometimes be awkward because you can't position exactly where this line ends. So a common convention is to put a very small open circle at the end of the line. Sometimes, it's useful to have an in-line comment on a diagram element. You can do this by prefixing the text with two dashes: --.

Dependency

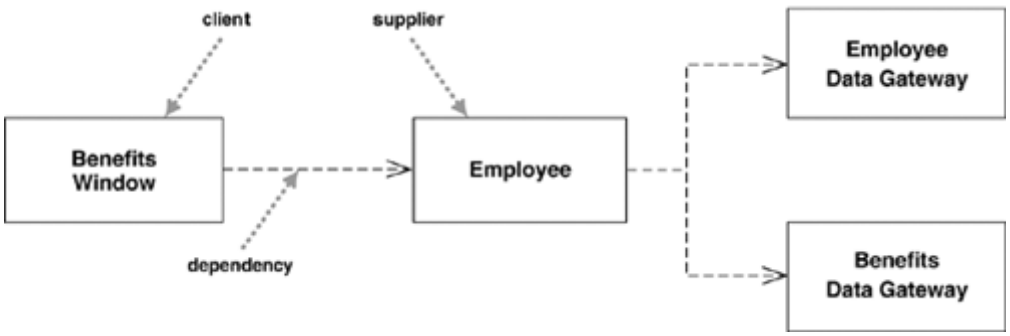
A *dependency* exists between two elements if changes to the definition of one element (the supplier) may cause changes to the other (the client). With classes, *dependencies* exist for various reasons: One class sends a message to another; one class has another as part of its data; one class mentions another as a parameter to an operation. If a class changes its interface, any message sent to that class may no longer be valid.

As computer systems grow, you have to worry more and more about controlling dependencies. If *dependencies* get out of control, each change to a system has a wide ripple effect as more and more things have to change. The bigger the ripple, the harder it is to change anything.

The UML allows you to depict *dependencies* between all sorts of elements. You use *dependencies* whenever you want to show how changes in one element might alter other elements.

Figure 3.7 shows some dependencies that you might find in a multilayered application. The Benefits Window class—a user interface, or presentation class—is dependent on the Employee class: a domain object that captures the essential behavior of the system—in this case, business rules. This means that if the employee class changes its interface, the Benefits Window may have to change.

Figure 3.7. Example dependencies



The important thing here is that the *dependency* is in only **one direction** and goes from the presentation class to the domain class. This way, we know that we can freely alter the Benefits Window without those changes having any effect on the Employee or other domain objects. I've found that a strict separation of presentation and domain logic, with the presentation depending on the domain but not vice versa, has been a valuable rule for me to follow.

A second notable thing from this diagram is that there is no direct dependency from the Benefits Window to the two Data Gateway classes. If these classes change, the Employee class may have to change. But if the change is only to the implementation of the Employee class, not its interface, the change stops there.

The UML has many varieties of dependency, each with particular semantics and keywords. The basic dependency that I've outlined here is the one I find the most useful, and I usually use it without keywords. To add more detail, you can add an appropriate keyword (Table 3.1).

The basic dependency is not a transitive relationship. An example of a transitive relationship is the "larger beard" relationship. If Jim has a larger beard than Grady, and Grady has a larger beard than Ivar, we can deduce that Jim has a larger beard than Ivar. Some kind of *dependencies*, such as substitute, are transitive, but in most cases there is a significant difference between direct and indirect *dependencies*, as there is in Figure 3.7.

Many UML relationships imply a dependency. The navigable association from Order to Customer in Figure 3.1 means that Order is dependent on Customer. A subclass is dependent on its **superclass** but not vice versa.

Table 3.1. Selected Dependency Keywords

Keyword	Meaning
«call»	The source calls an operation in the target.
«create»	The source creates instances of the target.
«derive»	The source is derived from the target.
«instantiate»	The source is an instance of the target. (Note that if the source is a class, the class itself is an instance of the class class; that is, the target class is a metaclass).
«permit»	The target allows the source to access the target's private features.
«realize»	The source is an implementation of a specification or interface defined by the target.
«refine»	Refinement indicates a relationship between different semantic levels; for example, the source might be a design class and the target the corresponding analysis class.
«substitute»	The source is substitutable for the target.
«trace»	Used to track such things as requirements to classes or how changes in one model link to changes elsewhere.

Table 3.1. Selected Dependency Keywords

Keyword	Meaning
«use»	The source requires the target for its implementation.

Your general rule should be to minimize dependencies, particularly when they cross large areas of a system. In particular, you should be wary of cycles, as they can lead to a cycle of changes. I'm not super strict on this. I don't mind mutual dependencies between closely related classes, but I do try to eliminate cycles at a broader level, particularly between packages.

Trying to show all the dependencies in a class diagram is an exercise in futility; there are too many and they change too much. Be selective and show dependencies only when they are directly relevant to the particular topic that you want to communicate. To understand and control dependencies, you are best off using them with **package diagrams** (6) *(see appendix for your own reference [Andy H]).*

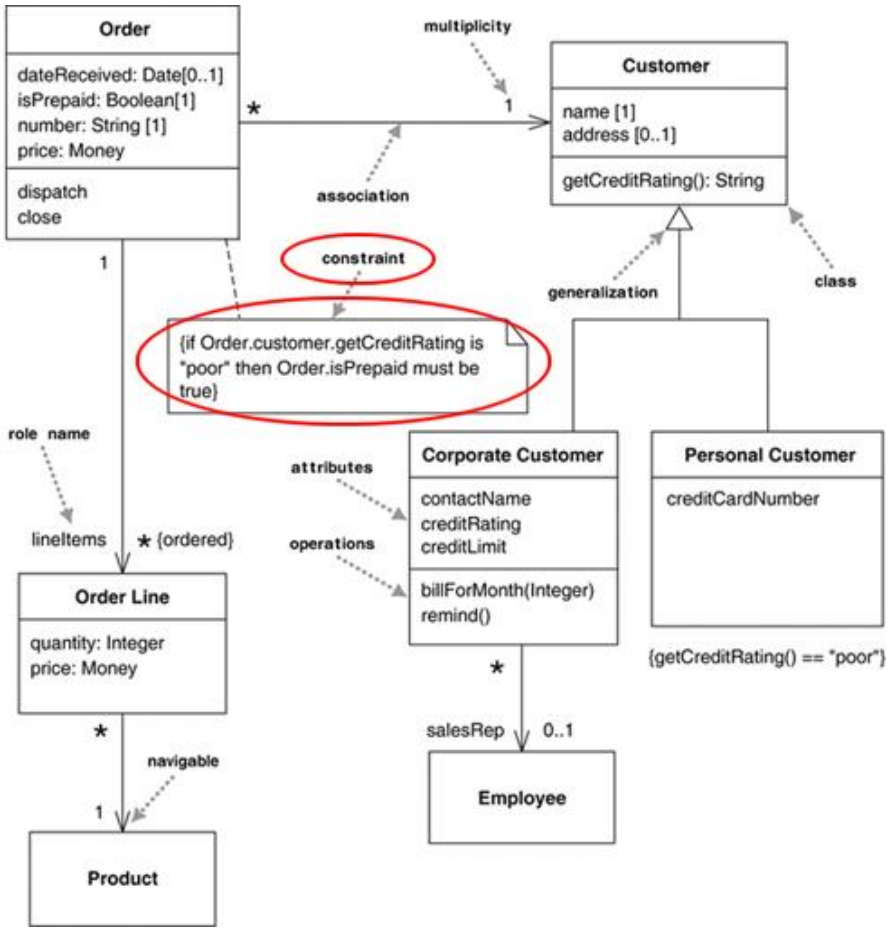
The most common case I use for *dependencies* with classes is when illustrating a transient relationship, such as when one object is passed to another as a parameter. You may see these used with keywords «parameter», «local», and «global». You may also see these keywords on associations in UML 1 models, in which case they indicate transient links, not properties. These keywords are not part of UML 2.

Dependencies can be determined by looking at code, so tools are ideal for doing dependency analysis. Getting a tool to reverse engineer pictures of dependencies is the most useful way to use this bit of the UML.

Constraint Rules

Much of what you are doing in drawing a class diagram is indicating constraints. Figure 3.1(revised below) indicates that an Order can be placed only by a single Customer. The diagram also implies that each Line Item is thought of separately: You say "40 brown widgets, 40 blue widgets, and 40 red widgets," not "120 things" on the Order. Further, the diagram says that Corporate Customers have credit limits but Personal Customers do not.

Figure 3.1 – revised



The basic constructs of association, attribute, and generalization do much to specify important constraints, but they cannot indicate every constraint. These constraints still need to be captured; the class diagram is a good place to do that.

The UML allows you to use anything to describe constraints. The only rule is that you put them inside braces ({}). You can use natural language, a programming language, or the UML's formal Object Constraint Language (OCL) [Warmer and Kleppe], which is based on predicate calculus. Using a formal notation avoids the risk of misinterpretation due to an ambiguous natural language. However, it introduces the risk of misinterpretation due to writers and readers not really understanding OCL. So unless you have readers who are comfortable with predicate calculus, I'd suggest using natural language.

Optionally, you can name a constraint by putting the name first, followed by a colon; for example, {disallow incest: husband and wife must not be siblings}.

Design by Contract

Design by Contract is a design technique developed by Bertrand Meyer [Meyer]. The technique is a central feature of the Eiffel language he developed. Design by Contract is not specific to Eiffel, however; it is a valuable technique that can be used with any programming language.

At the heart of Design by Contract is the **assertion**. An **assertion** is a Boolean statement that should never be false and, therefore, will be false only because of a bug. Typically, assertions are checked only during debug and are not checked during production execution. Indeed, a program should never assume that assertions are being checked.

Design by Contract uses three particular kinds of assertions: post-conditions, pre-conditions, and invariants. Pre-conditions and post-conditions apply to operations. A post-condition is a statement of what the world should look like after execution of an operation. For instance, if we define the operation "square root" on a number, the post-condition would take the form input = result * result, where result is the output and input is the input value. The post-condition is a useful way of saying what we do without saying how we do it—in other words, of separating interface from implementation.

A pre-condition is a statement of how we expect the world to be before we execute an operation. We might define a pre-condition for the "square root" operation of input > = 0. Such a pre-condition says that it is an error to invoke "square root" on a negative number and that the consequences of doing so are undefined.

On first glance, this seems a bad idea, because we should put some check somewhere to ensure that "square root" is invoked properly. The important question is who is responsible for doing so.

The pre-condition makes it explicit that the caller is responsible for checking. Without this explicit statement of responsibilities, we can get either too little checking—because both parties assume that the other is responsible—or too much—both parties check. Too much checking is a bad thing because it leads to a lot of duplicate checking code, which can significantly increase the complexity of a program. Being explicit about who is responsible helps to reduce this complexity. The danger that the caller forgets to check is reduced by the fact that assertions are usually checked during debugging and testing.

From these definitions of pre-condition and post-condition, we can see a strong definition of the term exception. An exception occurs when an operation is invoked with its pre-condition satisfied yet cannot return with its post-condition satisfied.

An *invariant* is an assertion about a class. For instance, an Account class may have an invariant that says that balance == sum(entries.amount()). The *invariant* is "always" true for all instances of the class. Here, "always" means "whenever the object is available to have an operation invoked on it."

In essence, this means that the invariant is added to pre-conditions and post-conditions associated with all public operations of the given class. The *invariant* may become false during execution of a method, but it should be restored to true by the time any other object can do anything to the receiver.

Assertions can play a unique role in **subclassing**. One of the dangers of inheritance is that you could redefine a subclass's operations to be inconsistent with the **superclass's** operations. Assertions reduce the chances of this. The invariants and post-conditions of a class must apply to all subclasses. The subclasses can choose to strengthen these assertions but cannot weaken them. The pre-condition, on the other hand, cannot be strengthened but may be weakened.

This looks odd at first, but it is important to allow dynamic binding. You should always be able to treat a subclass object as if it were an instance of the **superclass**, per the principle of substitutability. If a subclass strengthened its pre-condition, a **superclass** operation could fail when applied to the subclass.

When to Use Class Diagrams

Class diagrams are the backbone of the UML, so you will find yourself using them all the time. This chapter covers the basic concepts. The trouble with class diagrams is that they are so rich, they can be overwhelming to use. Here are a few tips.

- Don't try to use all the notations available to you. Start with the simple stuff in this chapter: classes, associations, attributes, generalization, and constraints.
- I've found conceptual class diagrams very useful in exploring the language of a business. For this to work, you have to work hard on keeping software out of the discussion and keeping the notation very simple.

- Don't draw models for everything; instead, concentrate on the key areas. It is better to have a few diagrams that you use and keep up to date than to have many forgotten, obsolete models.

The biggest danger with class diagrams is that you can focus exclusively on structure and ignore behavior. Therefore, when drawing class diagrams to understand software, always do them in conjunction with some form of behavioral technique. If you're going well, you'll find yourself swapping between the techniques frequently.

APPENDIX

(only for your reference if curious)

[1]

From Wikipedia, the free encyclopedia

In computer programming, **boilerplate code** or **boilerplate** is the sections of code that have to be included in many places with little or no alteration. It is more often used when referring to languages that are considered *verbose*, i.e. the programmer must write a lot of code to do minimal jobs. The need for boilerplate can be reduced through high-level mechanisms such asmetaprogramming (which has the computer automatically write the needed boilerplate text), convention over configuration (which provides good default values, reducing the need to specify program details in every project) and model-driven engineering (which uses models and model-to-code generators, eliminating the need for boilerplate manual code).

A related term is *bookkeeping code*, referring to code that is not part of the business logic but is interleaved with it in order to keep data structures updated or handle secondary aspects of the program.

[2]

CRC - When to Use Sequence Diagrams

You should use sequence diagrams when you want to look at the behavior of several objects within a single use case. Sequence diagrams are good at showing collaborations among the objects; they are not so good at precise definition of the behavior.

If you want to look at the behavior of a single object across many use cases, use a state diagram. If you want to look at behavior across many use cases or many threads, consider an activity diagram.

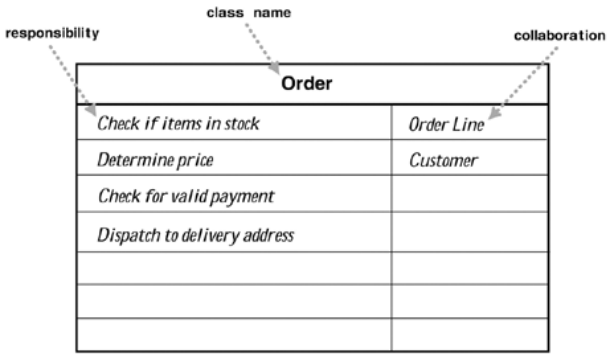
If you want to explore multiple alternative interactions quickly, you may be better off with CRC cards, as that avoids a lot of drawing and erasing. It's often handy to have a CRC card session to explore design alternatives and then use sequence diagrams to capture any interactions that you want to refer to later.

Other useful forms of interaction diagrams are communication diagrams, for showing connections; and timing diagrams, for showing timing constraints.

CRC Cards

One of the most valuable techniques in coming up with a good OO design is to explore object interactions, because it focuses on behavior rather than data. CRC (Class-Responsibility-Collaboration) diagrams, invented by Ward Cunningham in the late 1980s, have stood the test of time as a highly effective way to do this (Figure 4.6). Although they aren't part of the UML, they are a very popular technique among skilled object designers.

Figure 4.6. A sample CRC card



To use CRC cards, you and your colleagues gather around a table. Take various scenarios and act them out with the cards, picking them up in the air when they are active and moving them to suggest how they send messages to each other and pass them around. This technique is almost impossible to describe in a book yet is easily demonstrated; the best way to learn it is to have someone who has done it show it to you.

An important part of CRC thinking is identifying responsibilities. A responsibility is a short sentence that summarizes something that an object should do: an action the object performs, some knowledge the object maintains, or some important decisions the object makes. The idea is that you should be able to take any class and summarize it with a handful of responsibilities. Doing that can help you think more clearly about the design of your classes.

The second C refers to collaborators: the other classes that this class needs to work with. This gives you some idea of the links between classes—still at a high level.

One of the chief benefits of CRC cards is that they encourage animated discussion among the developers. When you are working through a use case to see how classes will implement it, the interaction diagrams in this chapter can be slow to draw. Usually, you need to consider alternatives; with diagrams, the alternatives can take too long to draw and rub out. With CRC cards, you model the interaction by picking up the cards and moving them around. This allows you to quickly consider alternatives.

As you do this, you form ideas about responsibilities and write them on the cards. Thinking about responsibilities is important, because it gets you away from the notion of classes as dumb data holders and eases the team members toward understanding the higher-level behavior of each class. A responsibility may correspond to an operation, to an attribute, or, more likely, to an undetermined clump of attributes and operations.

A common mistake I see people make is generating long lists of low-level responsibilities. But doing so misses the point. The responsibilities should easily fit on one card. Ask yourself whether the class should be split or whether the responsibilities would be better stated by rolling them up into higher-level statements.

Many people stress the importance of role playing, whereby each person on the team plays the role of one or more classes. I've never seen Ward Cunningham do that, and I find that role playing gets in the way.

Books have been written on CRC, but I've found that they never really get to the heart of the technique. The original paper on CRC, written with Kent Beck, is [Beck and Cunningham]. To learn more about both CRC cards and responsibilities in design, take a look at [Wirfs-Brock].

[3]

[Meyer] Bertrand Meyer, Object-Oriented Software Construction. Prentice-Hall, 2000.

[4]

[Martin] Robert Cecil Martin, The Principles, Patterns, and Practices of Agile Software Development, Prentice-Hall, 2003.

[5]

[Gang of Four] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995.

[6]

Packages and Dependencies

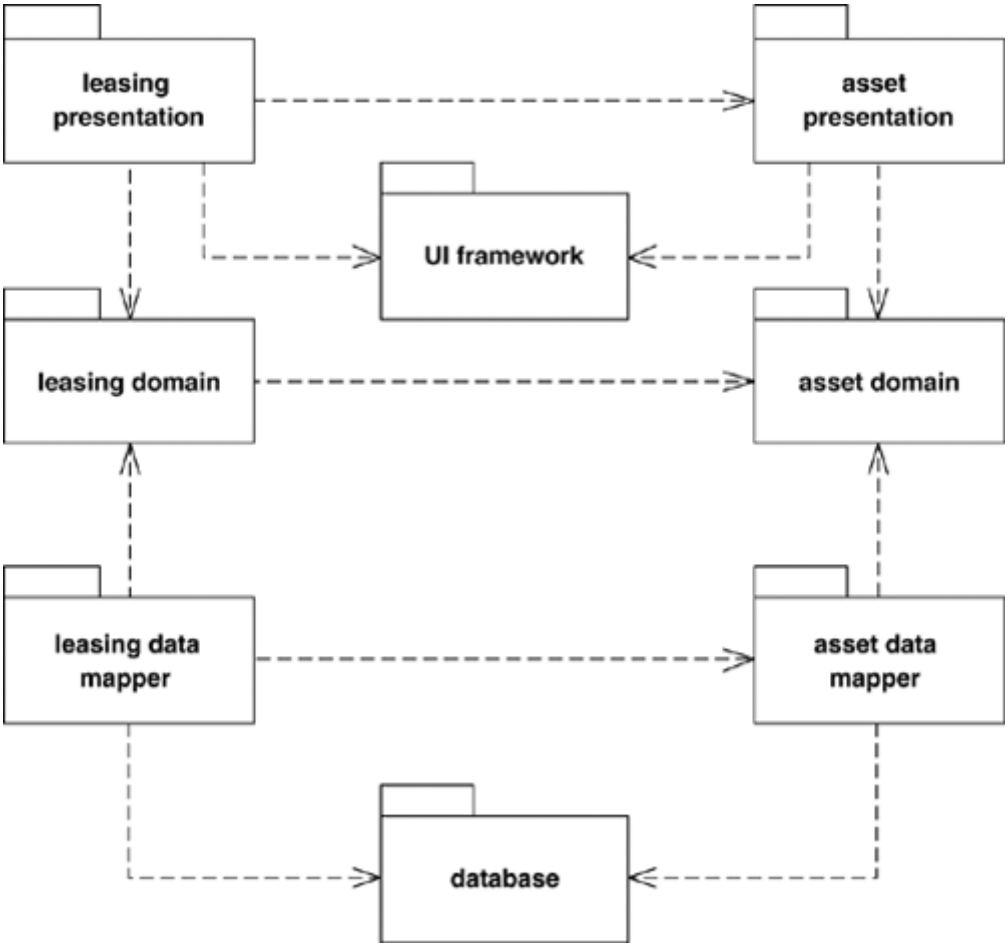
A package diagram shows packages and their dependencies. I introduced the concept of dependency on page 47. If you have packages for presentation and domain, you have a dependency from the presentation package to the domain package if any class in the presentation package has a dependency to any class in the domain package. In this way, interpackage dependencies summarize the dependencies between their contents.

The UML has many varieties of dependency, each with particular semantics and stereotype. I find it easier to begin with the unstereotyped dependency and use the more particular dependencies only if I need to, which I hardly ever do.

In a medium to large system, plotting a package diagram can be one of the most valuable things you can do to control the large-scale structure of the system. Ideally, this diagram should be generated from the code base itself, so that you can see what is really there in the system.

A good package structure has a clear flow to the dependencies, a concept that's difficult to define but often easier to recognize. Figure 7.2 shows a sample package diagram for an enterprise application, one that is well-structured and has a clear flow.

Figure 7.2. Package diagram for an enterprise application



Often, you can identify a clear flow because all the dependencies run in a single direction. Although that is a good indicator of a well-structured system, the data mapper packages of Figure 7.2 show an exception to that rule of thumb. The data mapper packages act as an insulating layer between the domain and database packages, an example of the Mapper pattern [Fowler, P of EAA].

Many authors say that there should be no cycles in the dependencies (the Acyclic Dependency Principle [Martin]). I don't treat that as an absolute rule, but I do think that cycles should be localized and that, in particular, you shouldn't have cycles that cross layers.

The more dependencies coming into a package, the more stable the package's interface needs to be, as any change in its interface will ripple into all the packages that are dependent on it (the Stable Dependencies Principle [Martin]). So in Figure 7.2, the asset domain package needs a more stable interface than the leasing data mapper package. Often, you'll find that the more stable packages tend to have a higher proportion of interfaces and abstract classes (the Stable Abstractions Principle [Martin].

The dependency relationships are not transitive. To see why this is important for dependencies, look at Figure 7.2 again. If a class in the asset domain package changes, we may have a change to classes within the leasing domain package. But this change does not necessarily ripple through to the leasing presentation. (It ripples only if the leasing domain changes its interface.)

Some packages are used in so many places that it would be a mess to draw all the dependency lines to them. In this case, a convention is to use a keyword, such as «global», on the package.

UML packages also define constructs to allow packages to import and merge classes from one package into another, using dependencies with keywords to notate this. However, rules for this kind of thing vary greatly with programming languages. On the whole, I find the general notion of dependencies to be far more useful in practice.

[7]

[Fowler, P of EAA] Martin Fowler, Patterns of Enterprise Application Architecture, Addison-Wesley, 2003.

[8]

[Warmer and Kleppe] Jos Warmer and Anneke Kleppe, The Object Constraint Language: Precise Modeling with UML, Addison-Wesley, 1998.

[9]

[Beck and Cunningham] Kent Beck and Ward Cunningham, "A Laboratory for Teaching Object-Oriented Thinking," Proceedings of OOPSLA 89, 24 (10): 1–6. http://c2.com/doc/oopsla89/paper.html

[9]

[Wirfs-Brock] Rebecca Wirfs-Brock and Alan McKean, Object Design: Roles Responsibilities and Collaborations. Prentice-Hall, 2003.