

Chapter 3

Applying UML

In earlier chapters we explained the concepts and principles of component software and took a look at the design process we follow in this book. We also began identifying the key model artifacts that we use in the component modeling workflows. In this chapter we look at how best to use UML to represent these component concepts and model artifacts. Subsequent chapters deal with when and how to use the various UML techniques.

This book isn't a UML primer. If you don't know the basics of UML you should first read an introductory text, such as Fowler's *UML Distilled* [Fowler99].

3.1 Why Do We Need This Chapter at All?

It may seem odd to you that we need this chapter. After all, the UML has been standardized and includes the notion of "component." Why isn't that enough?

We need to make sure that we all have a common understanding of the way we use UML in this book. There are several reasons for this:

- UML was originally designed as a language for object-oriented analysis and design. Further, it is based on the assumption of an object-oriented implementation. As we explained in Chapter 1, our focus in this book is on the external aspects of software components and not

their internals, irrespective of whether they are coded using an object-oriented language. In particular, we want to separate the specification aspects of design from the pure implementation choices. This special focus means that some UML features become less important, while others are stressed.

- Constructing systems with components places extra emphasis on the ability to plug software pieces together. In turn, this emphasizes the need for clear and precise interface definitions. Although our views on interface specification certainly fit within UML, they cause us to use it in new ways, which we need to explain.
- We are interested in techniques rather than diagrams: The same UML diagram can be used for a variety of purposes, and we need to explain how we use UML to support the techniques described later in the book.
- UML is designed to be extended. An important principle throughout the development of UML has been that it provides a base language that everyone can agree on, while accepting that its application to different contexts may need these base concepts either to be interpreted in different ways or to be extended to add the desired semantics.

Perhaps in the future UML will be extended to support component modeling concepts, but this book is for practitioners who need to design component systems now, using UML as it stands today (UML 1.3) and today's UML tools. Therefore we have done our utmost to stay within the current UML standard, and not include extensions and alternatives that, while arguably beneficial, are not supported by today's UML tools.

3.1.1 Tools

In some cases current versions of popular tools do support a particular concept in exactly the way we need, either because they are already thinking ahead, or because they have not implemented all the UML constraints to the letter, and therefore allow it to be flexed a little. This book does not take a position on, or recommend, any software tools. However, we have applied our techniques using a variety of current tools to ensure that what we say is practical today, and is easily achieved with those tools.

3.2 Extending UML with Stereotypes

UML has a number of extension mechanisms, but probably the most useful, at least in theory, is **Stereotypes**. Pretty much any UML element can have one stereotype attached to it; the stereotype's name then normally appears on the element enclosed by « ». Stereotypes are a way of letting users create new kinds of UML modeling elements, albeit ones that closely resemble some built-in type. For example, you might decide that you wish to distinguish between utility classes and regular classes. A simple way of doing that would be to define a «utility» stereotype for class and mark utility classes with that stereotype.

But there's more to it than that. Whenever you define a stereotype, UML allows you also to define a set of constraints that must hold for all elements marked with that stereotype. These constraints operate at the model level, so you could say that «utility» classes can have only static features, for example. At least, that's the theory. In practice, current UML tools almost never allow you to add such constraints, even in natural language, let alone something that can be checked automatically, such as object constraint language (OCL) [Warmer99]. So while it is easy to add UML extensions using stereotypes (and we'll be doing that), getting tool support for the constraints we'd like associated with those extensions is much harder.

3.3 Precision, Accuracy, and Completeness

OCL is a textual language for creating logical expressions. Despite being a full part of UML 1.3 and being used extensively in the definition of the UML itself, OCL is almost completely unsupported by current mainstream modeling tools, which is a shame because OCL allows us to be much more precise, especially when specifying component behavior.

Anyway, we use some OCL in this book, although you could manage without it, either by being less precise or by using natural language alone (which amounts to the same thing).

Using OCL improves precision but implies nothing about the completeness of a model. Many people confuse precision and completeness, but they aren't related. You can be very precise in the things you choose to say but leave many other things unsaid. That's the way it is (or should be) with UML models. For example, if you asked someone for a specification of their age they might say "over 30 years." This is precise, but not complete. Importantly, it's a condition that can be tested. That's what precision gives you. Contrast this with saying "I'm old." This is imprecise, because you don't know what "old" means, or your definition may not be the same as someone else's. When we're building large systems, partial models are fine, even essential.

Unfortunately, precision also doesn't imply accuracy. You can specify a software component in excruciating detail but still end up with something that totally fails to meet the need. It's always worth bearing that in mind when you are up to your elbows in OCL.

However, precision in models is useful because it allows you to test for accuracy. Once you know exactly what a model says (and what it doesn't), you're almost obliged to ask yourself whether it is right. Precise models are very good at generating test conditions.

3.4 UML Modeling Techniques

Most of the rest of this chapter is devoted to explaining how we use UML to model the various artifacts needed in component modeling. The techniques involved are explained in the workflow chapters. We have defined a number of diagrams corresponding to these artifacts that support these techniques, and these are shown in Figure 3.1 against the simple package structure we introduced in Chapter 2. These aren't new diagram types in the sense that they introduce any new notation—we work within the standard UML notation—but they are specific and focused usages of standard UML diagrams for particular purposes.

Perhaps the first thing to say is that when we use the term *model* for an artifact, as in "business type model," we are using it in a general-purpose way simply to mean a self-contained set of model elements. We do not mean it carries the specific semantics of a UML model, which is a complete abstraction of a system.

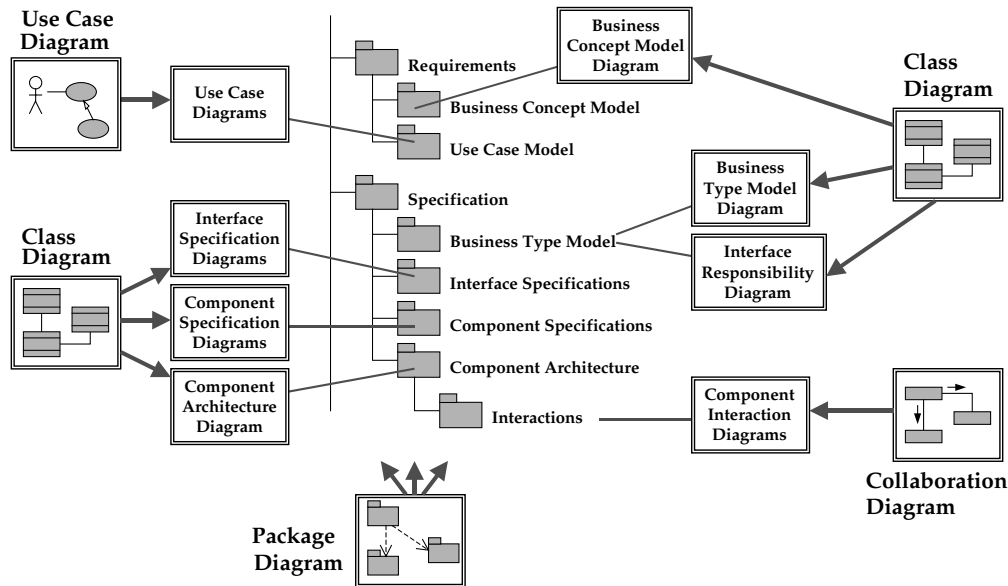


Figure 3.1 Component modeling diagrams

For the most part, the diagrams we draw bear the same names as the artifacts they visualize. For example, the **Business Concept Model Diagram** is a class diagram depicting the business concept model. An **Interface Specification Diagram** depicts an interface specification. And so it continues, with the **Business Type Model Diagram**, the **Component Specification Diagrams**, and the **Component Architecture Diagram** each depicting their corresponding artifact(s).

The only variations from the natural naming correspondence are the **Interface Responsibility Diagram** and the **Component Interaction Diagram**. The interface responsibility diagram is a class diagram depicting the business type model augmented with business interfaces and applying some rules for allocating information responsibility to those interfaces. The component interaction diagram is a collaboration diagram applied to the interaction between component objects. Each of these is explained in more detail as we go through the workflows.

As you can see from Figure 3.1, we make extensive use of the class diagram. We also use the collaboration diagram for interaction modeling, the

use case diagram for use case modeling, and the package diagram to organize things.

We could use the sequence diagram as an alternative to the collaboration diagram to depict interactions, but we like the structural emphasis the collaboration diagram gives by showing the relationships between objects, and, generally, we are not modeling complicated message sequences. Remember, this is specification, not implementation. But anywhere we've used the collaboration diagram you could use the sequence diagram instead if you prefer to, or if your tool provides better support for that interaction form.

You'll notice we don't use the component diagram or deployment diagram. Again, this is because we're focusing on specification, not implementation. UML components are implementation packaging and deployment constructs, whereas we're in the business of creating component specifications, which are specification constructs. We elaborate on this a lot more as we get into the details.

We also don't use the activity diagram or the state diagram. It's not that they're not useful, it's just that we don't need them in our workflows. In certain places we discuss where such diagrams could be used to provide additional support to the techniques we're describing, or where they might be used in related workflows to the ones we're describing. In our experience, UML activity diagrams have two main uses: to describe business processes and to describe the algorithms of operations. Neither of these uses strictly fall within the scope of this book (although you'll see an activity diagram in Chapter 4 as a lead in to use case modeling and business concept modeling).

State diagrams have a worthy history in computing. They are very useful for describing sequencing constraints, and we could have used them as part of our interface specifications. However, it is possible—but not as elegant in complex cases—to use attributes and pre- and postconditions to achieve the same effect, and that's what we've done.

As a general theme, we use the class diagram in a variety of ways to capture the structural or static aspects of our specification models, and the interaction diagrams to capture the behavioral or dynamic aspects of the specification models. At the requirements level we similarly use the class diagram to capture the business concepts in the domain being studied, and the use case diagram (which can be thought of as a specialized form

of interaction diagram) to understand the interaction between the users and the system.

We now go through the content of each of the models (business concept, use case, business type, interface specification, component specification, component architecture) and show how we apply UML in that context. Again, our purpose is not to explain UML but to show how we apply UML concepts to these particular artifacts. Throughout this chapter we illustrate the approach with small examples. We have kept them small and simple so that the use of UML is clear. We then develop a comprehensive case study through the more detailed workflow chapters.

3.5 Business Concept Model

The business concept model is a conceptual model. It is not a model of software, but a model of the information that exists in the problem domain. The main purpose of the business concept model diagram is to capture concepts and identify relationships. We use a UML class diagram to represent it, but it's important to note that it is a software-independent model. If you wish you could use a «concept» stereotype for every class in this model, but since, from a packaging point of view, it's kept under Requirements, and is quite separate from the Specification part of the model, we find in practice that we don't need to use a stereotype.

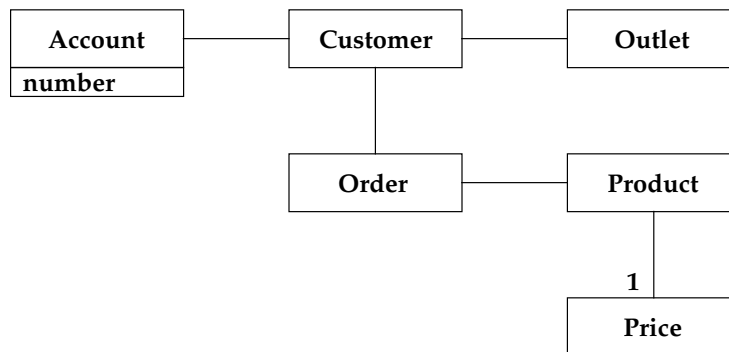


Figure 3.2 A simple business concept model

Business concept models typically capture conceptual classes and their associations. Association roles may or may not have their multiplicities specified. The model may contain attributes, if they are significant, but they need not be typed, and operations would not be used. Since the emphasis of the model is to capture domain knowledge, not to synthesize it or normalize it, you would rarely use generalization in this model. Similarly, dependency relationships would typically not be used.

3.6 Use Case Model

Love them or hate them, Use Case Models are what the UML provides for semiformal modeling of user-system interaction. Use cases are not tightly specified in UML, so you can create pretty much anything you like and describe it as a use case. Crucially, the UML does not specify the way in which use case content is described; we make that up for ourselves.

We employ use cases purely as a technique for describing the desired user-system interactions that occur at the system boundary. We recognize that use cases can validly fulfill other needs, such as modeling business operations, that are beyond the scope of this book. Use cases, the way we use them, are a projection of the requirements of a system, expressed in terms of the interactions that must occur across the system boundary.

The participants in a use case are the **Actors** and the system. An actor is an entity that interacts with the system, typically a person playing a role. It's possible for an actor to be another system, but, if it is, the details of that system are hidden from us—we see it simply as a dull and predictable person. One actor is always identified as the actor who initiates the use case; the other actors, if any, are used by the system (and sometimes the initiating actor) to meet the initiator's goal.

In a use case the actors interact with the system as a whole, not some specific part of it. The system is viewed as a homogenous black box that accepts stimuli from actors and generates responses.

It is important to distinguish between the actor and the mechanism the actor uses to communicate with the system, such as a GUI application running on a PC, a panel of buttons in an elevator, or—in the case where another system is the user—a TCP/IP connection. A use case shouldn't be

concerned with the communication mechanism, only with the business meaning of the stimuli and responses.

A perennial problem with use cases is deciding their scope and size. There seems to be no consensus on this issue, but here's our view: To a first approximation we can say that a use case is smaller than a business process but larger than a single operation on a single component. The purpose of a use case is to meet the immediate goal of an actor, such as placing an order or checking a bank account balance. It includes everything that can be done now or nearly now by the system to meet the goal. For example, if it is necessary to perform an electronic credit check with an agency before accepting an order, we would expect the use case to perform the check and proceed. On the other hand, if goods need to be ordered from a supplier to fulfill the order being placed, the use case would end when the goods are ordered; it wouldn't wait for them to arrive. The subsequent arrival of the goods would stimulate another use case. We examine this issue again in Chapter 4.

To be more accurate, in UML terms our granularity characterization really only applies to base use cases. Use cases can also be extended by other use cases and include other use cases, and these others will necessarily be of finer granularity than their base. In UML, the base element, the extensions, and the inclusions are model elements of type use case.

3.6.1 Use Case Diagrams

Use Case Diagrams, with their familiar stick figures and ellipses, are useful only as a catalogue or map. Use cases can be defined with a number of extension points to which extension use cases refer. This approach is useful for defining the broad structure of the use case in a diagram (see Figure 3.3) but still leaves the real detail of the use case to be captured in textual use case descriptions (see Figure 3.4).

3.6.2 Use Case Descriptions

The textual structure we prefer is based on that of Alistair Cockburn (see [CockburnWeb]). We present a simplified form here. A **Use Case Description** contains at least

- an identifying name and/or number
- the name of the initiating actor

- a short description of the goal of the use case
- a single numbered sequence of steps that describe the main success scenario

Except in the case of an inclusion step, each step takes the form “A does X,” where A is an actor or the system. The first step must indicate the

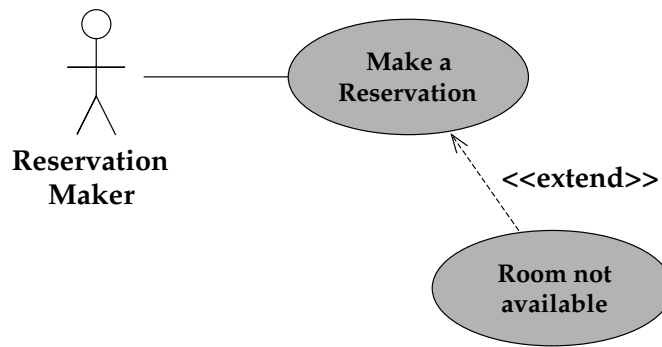


Figure 3.3 Simple use case diagram

Name	Make a Reservation
Initiator	Reservation Maker
Goal	Reserve a room at a hotel

- Steps**
or
Extension Points
- Main success scenario**
1. Reservation Maker requests to make a reservation.
 2. Reservation Maker selects hotel, dates, and room type.
 3. System provides availability and price.
 4. Reservation Maker agrees to proceed.
 5. Reservation Maker provides identification and contact details.
 6. Reservation Maker provides payment details.
 7. System makes reservation and gives it a tag.
 8. System reveals tag to Reservation Maker.
 9. System creates and sends reservation confirmation.
- Extensions**
3. Room Not Available
 - a) System offers alternative dates and room types.
 - b) Reservation Maker selects from alternatives.

Figure 3.4 Textual description of a use case

stimulus that initiates the use case (i.e., what the initiating actor does to indicate to the system that it wants this goal met). The combination of initiating actor and stimulus must be unique across all use cases.

The main success scenario describes what happens in the most common case and when nothing goes wrong. It is broken into a number of separate **Use Case Steps**. The assumption is that the steps are performed strictly sequentially in the order given—unlike most business process languages there is no way of describing parallelism, which is one reason why a use case is typically smaller than a business process. Each use case step acts as a UML use case extension point. It is the anchor point from which an extend relationship to an extension use case may be defined.

Use case steps are always written in natural language. Use cases are a semiformal technique and must be understandable by anyone familiar with the problem domain.

3.6.3 Use Case Instances

A use case description is a template for behavior that is instantiated in the environment of a deployed system each time an actor generates a stimulus. A **Use Case Instance** either succeeds or fails in meeting the goal. A simple use case consisting only of a main success scenario is assumed always to succeed.

3.6.4 Inclusions, Extensions, and Variations

One use case can **Include** another by naming it in a step. This means that when an instance of the including use case reaches that step, it invokes the included use case and then proceeds to the next step.

Extensions are a mechanism for semiformal specification of alternatives or additions to the main success scenario.¹ Each extension is described separately, after the main success scenario. An extension comprises the following:

- The step number (i.e., extension point) in the main success scenario at which the extension applies.

1. Strictly speaking, each extension describes a new use case that participates in an **Extend** relationship with the main use case.

- A condition that must be tested before that step. If the condition is true the extension is activated; if false the extension is ignored and the main success scenario continues as usual.
- A numbered sequence of steps that constitute the extension.

The steps in an extension can take any of the two forms found in the main success scenario: “A does X” or “useCaseReference” (to include the use case with that name or number). In addition, the last step in an extension can take one of the following forms:

- **Fail**—to indicate the use case is terminated with the goal unsatisfied
- **Stop**—to indicate that the use case is terminated with the goal satisfied
- **Resume N**—to indicate the next step to be performed in the main success scenario

If the last step of the extension is not one of these, the main success scenario continues with its normal next step (i.e., no step is skipped—the extension does not replace a step in the main success scenario).

Sometimes we know that there will be some significant variations in the flow of a use case but we don’t want to specify them all now as extensions. So we can include at the end of the use case description a list of **Variations**. A variation is just a free text note saying that the variation can occur.

Please see [Cockburn00] for a much more detailed template for use cases.

3.7 Business Type Model

The **Business Type Model** is a specification model. It is modeled using a UML class diagram but its purpose is quite different to that of the concept model. The business type model is concerned with modeling precisely the business information that is relevant to the scope of the envisaged system. Naturally, many of its classes bear the same names as classes in the concept model, but they do not represent the same thing. For example, the business concept model may contain a class called “customer,” which represents the concept of customer in the business domain. By contrast, the

class “customer” in the business type model represents the system’s understanding of customer, which is typically scoped down and more precisely defined.

3.7.1 Types

To indicate that the classes in the business type model are specification-level, we use the built-in stereotype «type» for them. This makes it very clear that it is not a model of programming language classes. Note, though, that in our business type models the types can’t have operations because they describe only information, not software, whereas the built-in stereotype «type» does allow operations.

The business type model is intended to represent precisely the information about the business with which the system will need to be concerned. This means it’s useful to remove redundancy and make use of generalization where it aids understanding. But it is not a database design. Don’t start thinking about normal forms or worrying about one-to-one associations. Remember it’s a specification model—it’s a set of rules.

Attributes

Attributes must be typed, and the type must be a data type. Visibility is irrelevant in a specification model, since nothing is internal or private, so it’s best set to “public.” As a matter of style we prefer all our attribute names to begin with a lowercase letter.

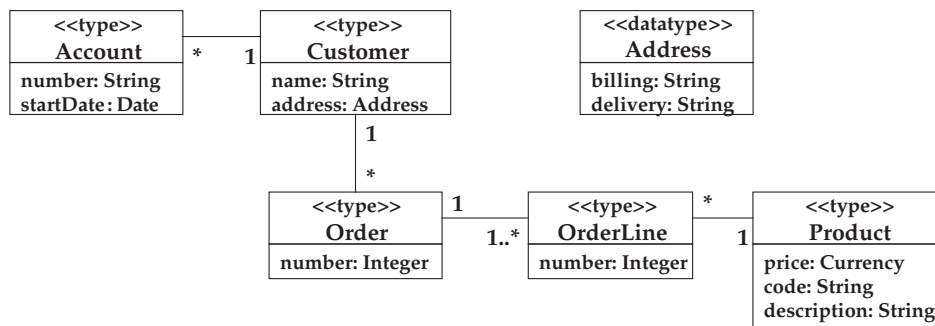


Figure 3.5 A simple business type model

Associations

As with attributes, association roles should have public visibility, and the role names, if explicitly stated, should begin with a lowercase letter. In the business type model we reserve the use of navigability (arrows on associations) to indicate responsibility for holding inter-interface associations, as explained in Chapter 5.

We don't find we need to use qualified associations, and there aren't any in the case study.

Although using composition is fine, we recommend against using aggregation unless you're certain you know what it means (which can only mean you've defined it for yourself. See Henderson-Sellers' UML 99 conference paper [Sellers99] for a gruesome dissection of the semantics of UML aggregation).

Meaning of Attribute and Association

It's important to remember that type models are specification models. They don't represent object-oriented programming classes or relational database tables (although mappings to such implementation constructs are very valuable). In a business type model, an attribute or association role describes a piece of data related to the type named in the rectangle. It represents information that the business needs to maintain and corresponds to a question you might validly ask, "What is the order number of this order?"

Collectively, the attributes and association roles define a set of **Queries** that can be applied to a type when writing specification constraints in OCL. They are a formalized vocabulary for referring to information. They do *not* represent a shorthand for get and set operations of an interface or an implementation class. The specification of operations of components is the job of the interface, not the business type.

Parameterized Attributes

Once you're comfortable with the idea of specification attributes as a tool to help define constraints, you're ready to consider parameterized attributes. A **Parameterized Attribute** is one whose value depends on the values of its

parameters. For example, a person's contact number might be dependent on the day of the week. So rather than have an attribute

```
contactNumber: String
```

we can define

```
contactNumber(day: Day): String
```

It's not an operation—it's not specifying something that a client could call or invoke or send a message to—it's simply a specification query to support the definition of constraints.

The bad news is that UML doesn't support this concept. However, it is so useful that this is an area where we're prepared to twist the UML a little. Since our regular «type» doesn't have operations (these belong to interfaces) this leaves type "operations" available for use as parameterized attributes. They must be defined as functions, always having a return value. Depending on your tool interchange requirements, it may be useful to mark these operations as attributes using a stereotype like «att». In practice, we usually leave it off. An example of how this would look as a UML class is shown in Figure 3.6.

Note: This concept should not be confused with the UML concept of a query operation, which is a true operation but one that is read-only and has no side effects.

For further details, Catalysis [D'Souza99] gives a thorough treatment of this subject.

3.7.2 Structured Data Types

The notion of a data type is defined in UML, but we want to call out the use of structured data types explicitly. We use UML classes again to define

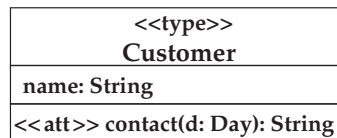


Figure 3.6 Parameterized attribute

structured data types, such as Address in Figure 3.5, and we use the «datatype» stereotype for these. This stereotype defines certain constraints: The data type may not have associations, or operations, and its attributes must also be typed as data types (simple or structured). During interface specification we also allow attributes to be references to component objects (i.e., to be of type interface).

3.7.3 Interface Type

As we develop the business type model we add interface types to it. We use a stereotyped class to represent an interface at the specification level; the stereotype is called «interface type». We have also adopted the convention of prefacing the names of interfaces with “I.”

Meaning of Attribute and Association

As for all types, the attributes and association roles of an interface are for the purpose of specification only. They do *not* imply the existence of operations that can retrieve the results of the queries; if such operations are required they must be defined explicitly. When we reach the detailed specification stage for interfaces, we define its set of attributes and association roles, and their types, as its **Interface Information Model**. The purpose of this model is to support the full specification of an interface, and we discuss that later in this chapter.

The UML «interface»

You may be aware that the UML standard includes a predefined modeling element called Interface that you can use in class diagrams: Because it is a kind of Classifier it is drawn like a class with an «interface» stereotype. Why aren't we using it?

Well, UML interfaces are really designed for modeling object-oriented implementation language constructs, notably the interface concept found in Java. They also accommodate the runtime component environment concept of interface as found in Microsoft COM and CORBA. Since UML interfaces have this implementation focus they do not have attributes or associations, and we'd like our interfaces to have both, because we have a

specification focus. Of course, you might be using a modeling tool that doesn't enforce these UML restrictions, in which case you are free to use «interface», but you would be bending the UML standard.

A useful way of thinking about the relationship between an interface type and a UML interface is to consider a UML interface as a potential realization of an interface type, in the usual "implementation realizes specification" sense. Figure 3.7 shows a customer management interface type mapping onto (or being realized by) a customer management interface in an implementation model. The implementation interface could be more specifically stereotyped as, say, «java interface» or «COM interface» to allow the constraints of those technologies to be applied.

We use the UML concept of interface in Chapter 8 when we show examples of mapping specification to implementation and discuss technology bindings.

In order to avoid verbosity in the text we generally use the term "interface" to mean "interface type," since we're mostly talking about the specification world. Where we discuss both the specification and implementation spaces together we are explicit.

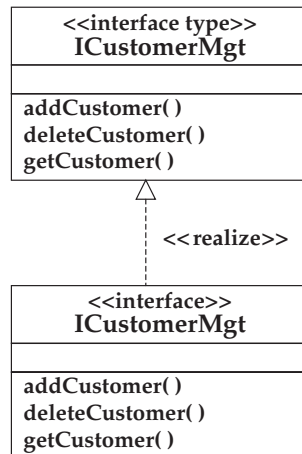


Figure 3.7 Interface types and interfaces

3.7.4 Invariants

UML defines an invariant as a stereotype of Constraint, which applies to classifiers and their relationships. For our purposes, an invariant is therefore a constraint on a type model, and we often use such constraints within business type models. We usually write invariants using the OCL.

An invariant is like a rule about the instances, so at the specification level many of the invariants you specify correspond to business rules. For this reason it's worth managing them carefully and maintaining them.

Figure 3.8 shows an IOrder interface, with an orderNo attribute and some operations.

We want to specify that order numbers are always greater than zero. We can do that with an invariant as shown: `orderNo > 0`.

An invariant is a constraint that applies to all instances of a type. This is interpreted as “for all component objects supporting the IOrder interface, at all times when the system is quiescent (i.e., not in the process of changing state) it is the case that the value of their orderNo attribute is greater than zero.” The invariant shown is actually a shorthand for

```
self.orderNo > 0
```

where `self` is an anonymous object supporting the interface. The invariant is saying that the condition must hold when `self` is any object supporting the interface. OCL allows us to omit the `self` when the meaning is clear.

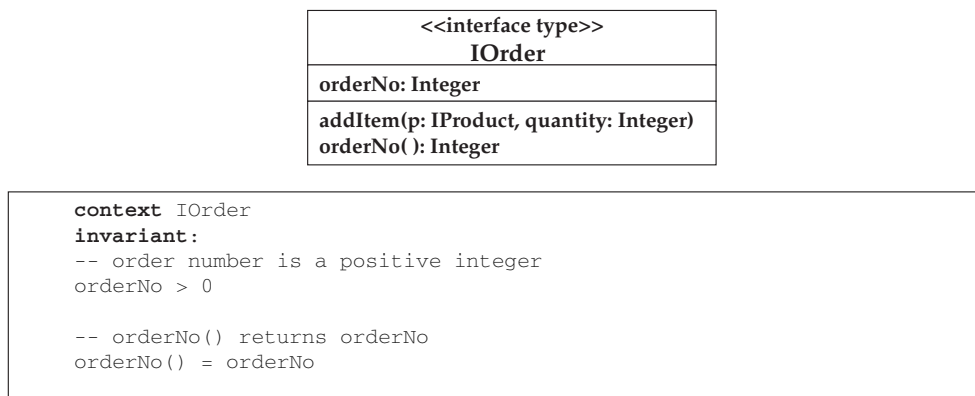


Figure 3.8 Invariants on IOrder

In addition, we want to specify that the result of the operation `orderNo()` is always the value of the `orderNo` attribute. The invariant for this is

```
orderNo() = orderNo
```

Note that we distinguish operations from attributes by always following an operation with parentheses, even if it has no parameters.

It won't have escaped your notice that much of this detailed OCL specification is done using text. In our examples we've used note boxes linked to the appropriate element in the diagram. If your tool provides specific places to hold invariant and operation specifications you can put them there. It might be useful, though, to check that your tool provides a sensible way of reporting or displaying this information together with the type model to which they apply—the value of seeing the constraints adjacent to the diagram is not to be underestimated.

3.8 Interface Specification

An **Interface Specification** is an interface together with all the other specification paraphernalia needed to define precisely what a component that offers that interface must do, and what a client of that interface can expect. An interface specification consists of these parts:

- The interface type itself
- Its information model—the attributes and association roles of the interface, and their types, transitively until closure
- Its operation specifications—operation signatures and pre- and postconditions
- Any additional invariants on the information model

3.8.1 Interface Specification Package

We group all this specification information into a single package—so each interface specification has its own package. This package may import specification information from other packages. For example, in

Figure 3.9 we have defined a specification package for the customer management interface `ICustomerMgt`. It contains the interface itself, the types it's associated with (in this case `Customer`), and any data types needed. Some data types, such as `CustDetails`, will be exclusive to the interface specification and so would normally be placed in the same package. Others, like `Address`, might be more generic and imported from shared packages of types. Still others, like common built-in types such as `String`, might be globally or implicitly available and would not need explicit import.

If you are using interface inheritance you can still retain the packaging structure and exclusive type ownership by importing the supertype's package into the subtype's package.

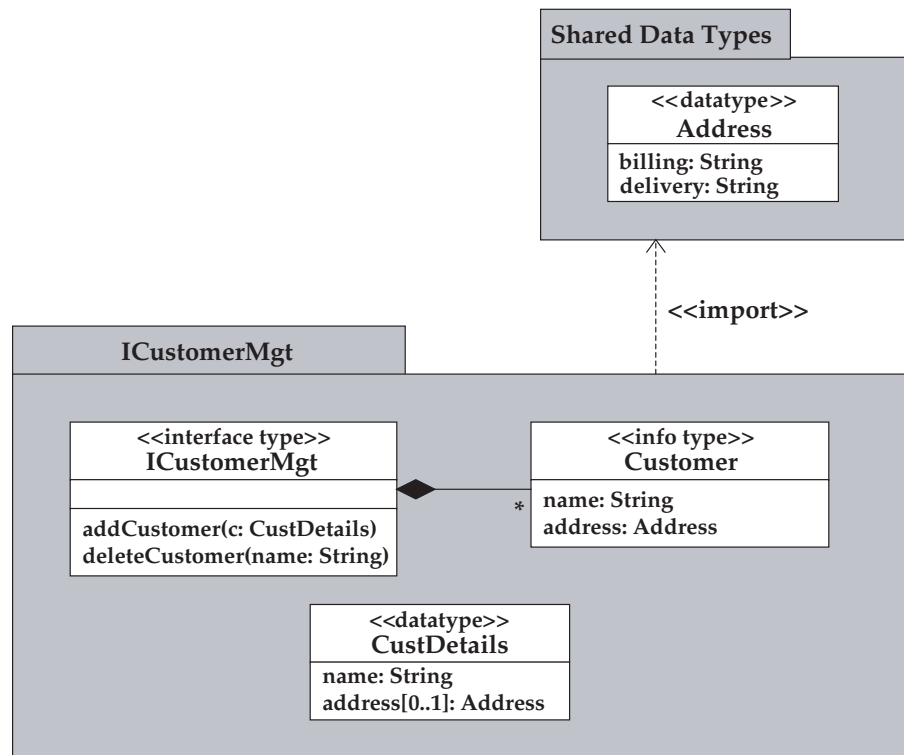


Figure 3.9 Interface specification package

3.8.2 Information Model

One of the goals of component modeling is to define relatively independent interface specifications. So whereas in the business type model we had a single integrated type model, when specifying interfaces we seek to define a number of independent type models, one for each interface. The types in these models are exclusive to their interface (that is to say, the nondata type ones are), and represent a projection or refinement of the type information in the business type model. Each interface has its own view of the information it cares about. For this reason we call these types **Information Types**, to distinguish them from the business types in the business type model, and we give them the «info type» stereotype.

We depict the information model of an interface in an interface specification diagram. This shows all the types in the information model and is a key diagram during the detailed operation specification tasks. When writing the pre- and postconditions of the interface's operations, it is essential you have this diagram in front of you.

So the business type model consists of business types (and interfaces), whereas an interface information model consists of an interface and an exclusive set of information types. Information types are the most common kind of type, so in our case study diagrams we've chosen not to display the stereotype name, to avoid clutter.

3.8.3 Operation Specification

An operation specification consists of a signature and a pre- and postcondition pair.

Signature

An operation defined on an interface has a signature comprising zero or more typed parameters. The type of a parameter can be

- a reference to a component object; the type of the parameter will typically be an interface
- a simple data type, such as an integer
- a structured data type, such as CustDetails
- a collection of any of these

Data types are always passed by value; values have no identity and cannot be shared between the operation invoker and implementer—each must have its own copy.

Pre- and Postcondition Pair

In UML, operation pre- and postconditions are defined as constraints on that operation, with the stereotypes «precondition» and «postcondition», respectively. Since they're constraints we normally write them in OCL.

A precondition is a Boolean expression that is evaluated—theoretically—before the operation executes; a postcondition is a Boolean expression that is evaluated after the operation finishes. We say *theoretically* because the pre- and postconditions are specification artifacts—they aren't embodied in the final code. The idea is that if the precondition is true, the postcondition must also be true. If the precondition is false, the behavior is undefined. We cover pre- and postconditions in more detail in Chapter 7.

Transactional Behavior

An important topic when specifying operations for distributed component systems is their transactional behavior, but this is not covered in UML. The various component environments (COM, EJB, CORBA Component Model) provide relatively rich schemes for the definition of transactional behavior. We need to specify transactional requirements in a way that readily maps onto these technologies.

In business information systems pretty much everything is transactional. The specification issue therefore boils down to whether an operation starts its own transaction or runs in an existing transaction. Since this is a binary choice we can model it simply by defining an operation stereotype for one of the cases. We choose to call out the forced new transaction case and we define a stereotype «transaction» for it. Absence of the stereotype then implies that an operation runs in an existing transaction.

3.9 Component Specification

In Chapter 1 we discussed the meaning of “component” in some detail and described a variety of different component forms. The UML concept of component is geared toward the implementation and deployment world. Within that domain it is very flexible (some would say too flexible) and can be used to accommodate a number of our more tightly defined forms, specifically Component Implementation and Installed Component. But our focus in this book is on specification. We need to represent the Component Specification form, and the UML component construct doesn’t fit the bill.

We therefore define a component specification as another stereotype of class, called «comp spec». Like the distinction in Figure 3.7 between a UML interface and an «interface type» class, a UML component realizes a «comp spec» class (see Figure 3.10).

A component specification offers a set of interface types. At the implementation level this “offers” relationship is modeled as a realization of a set of interfaces by a component. But “realization” is not the correct semantics here. We therefore define a new stereotype of UML dependency called «offers» to capture this important relationship (see Figure 3.11).

Notationally, we would like our interface types to have the option of being displayed as “lollipops,” just as UML defines for interfaces. UML

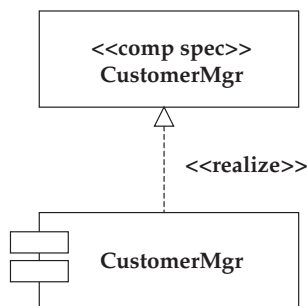


Figure 3.10 UML components realize component specifications

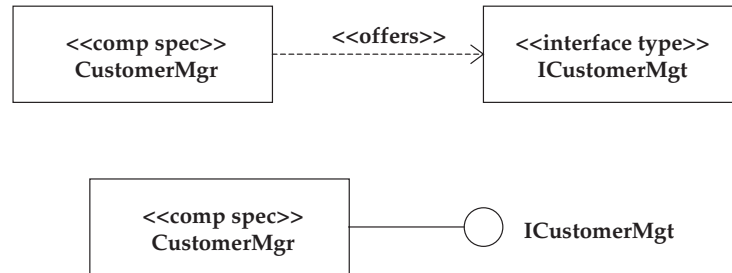


Figure 3.11 Component specifications offer interface types, with “lollipop” notation

allows you to define an icon for any stereotyped element so this fits quite nicely.²

A component specification is the building block of a component architecture. In addition to the set of interfaces it offers, it defines the set of interfaces it must use. This is an architectural constraint, not a component design choice. We model this with a simple UML usage dependency. A usage dependency between a component specification and an interface type specifies that all realizations of the component must use that interface. For example, in Figure 3.12 we specify that the customer manager component must use an address management interface. It doesn’t specify which component it must use, just which interfaces. Binding these interfaces to the actual components is an architecture task.

We call Figure 3.12 a component specification diagram. It focuses on a single component specification and details its individual dependencies.

A component specification also defines any constraints that it requires between these interfaces, and any constraints on the implementation of its operations. These operation implementation constraints can be defined using component interactions.

2. From a tool point of view, as we discussed earlier, you may choose to use the UML interface concept as if it were an interface type, but only if the tool doesn’t impose all the constraints that UML demands. You’ll then be in a position where you might be (ab)using the realize relationship between a «comp spec» class and an interface to represent the “component spec offers interface type” relationship. Ideally your tool would allow you to define that a «comp spec» class cannot be the target of a realize dependency. But if it doesn’t, and you’ve explicitly decided to use the relationship in this way, you’ll need to take care.

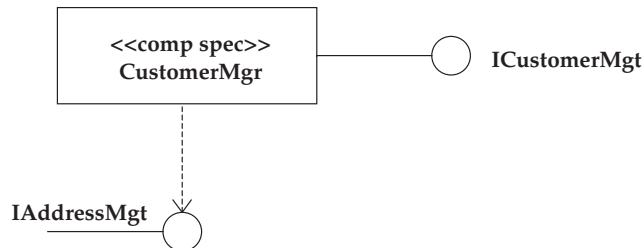


Figure 3.12 Component specification, an architecture building block

3.9.1 Component Object Interaction

We use UML collaboration diagrams to specify the desired interactions between component objects. These component interaction diagrams can focus on one particular component object and show how it uses the interfaces of other component objects. Or they can be used to depict an extended set of interacting component objects in a complete architecture.

The rectangles in the diagrams represent component objects that support the interfaces shown. Since components can support many interfaces it is quite possible that two or more rectangles represent the same object. The links must be instances of associations between interfaces that are part of the interface specification or component specification, unless they are transient (i.e., they exist no longer than the duration of the collaboration).

Figure 3.13 shows an interaction diagram for the `getAddress()` operation on a `CustomerMgr` component object supporting `ICustomerMgt`, passing in a customer `Id` and getting back some address details. In turn, it calls `getAddress()` on a component object supporting `IAddressMgt` (note the signature has changed—it's not the same operation). It passes an internal address `Id` it is holding and gets back the address details, which it can then process as needed and pass back.

Let's have a look at the object naming here. In UML, the naming rules for roles in a collaboration are `objectname/rolename:classifiername`. We're using anonymous instances so our object names are blank. Then we adjust things to add component semantics—the idea is to consider an interface as the role of a component object in an interaction. So, for the role names of the objects we use the interface names directly, and for the classifier name we identify the component specification offering that interface. If it is blank, we are leaving it unspecified.

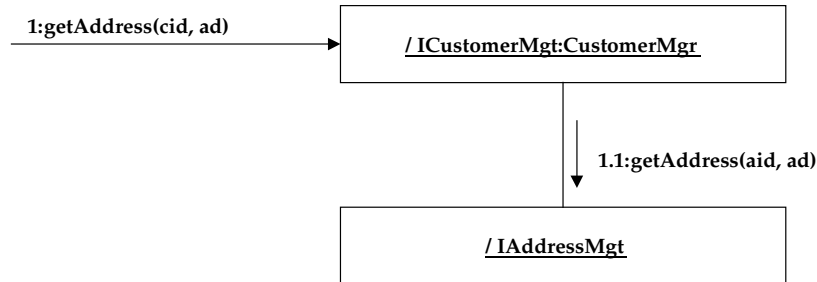


Figure 3.13 A component interaction diagram

So /ICustomerMgt:CustomerMgr specifies an instance of the `CustomerMgr` component playing an `ICustomerMgt` role. /IAddressMgt identifies an unspecified component object playing that role.

In an interaction diagram, only component objects for which we identify the component specification can send messages, because message sending is a function of the component, not an interface. The object /IAddressMgt in Figure 3.13 has no component specification indicated, so we couldn't show any outgoing messages from it.

When we use an interaction diagram to specify an individual component, we only show direct interactions between objects of the type being specified and objects supporting the interfaces it uses—we never show subsequent downstream interactions.

Even in software things don't happen without cause, so every interaction diagram should show, as an incoming arrow to one of the rectangles, the stimulus that causes the interaction. Figure 3.13 shows an interaction that begins when an object supporting the `ICustomerMgt` interface receives a `getAddress()` message. This is the first message and so is labeled 1. Subsequent messages then follow a nested-numbering scheme.

3.9.2 Specification, Not Implementation

You may be wondering why we use collaboration diagrams at all. After all, didn't we say that we are interested only in specifying the externals of the components, not the details of their internal implementations? It looks as though collaboration diagrams are all about internal design decisions. In

Figure 3.13, surely what the CustomerMgr component does in its implementation of the getAddress() operation is an internal design decision?

Well, if the implementer really does have a free choice about how best to implement the operation, we shouldn't draw this collaboration (at least not as a specification level diagram). As specifiers, we draw collaborations specifically to restrict the freedom of the implementation designer. Figure 3.13 is part of the contract between the specifier and the realizer that we discussed in Chapter 1. It says "whatever else the realization of getAddress() does, it must invoke the getAddress() operation of the associated IAddressMgt object." We are reducing the options for the realizer, to create a stable architecture into which a variety of alternative realizations can be plugged. All the realizations must conform to these restrictions.

As we show in Chapter 7, it is sometimes possible to express these constraints declaratively, rather than through the procedural mechanism of a collaboration diagram.

3.10 Component Architectures

As described in Chapter 1, we define a number of different component architectures, namely component specification architectures, component implementation architectures, and component object architectures. From the point of view of applying UML, these introduce no new ideas: They are simply diagrams that depict many components rather than just one.

A component architecture defines a context within which the interface usage dependencies of standalone component specifications are bound to the «offers» dependencies of other standalone component specifications. So, instead of a number of independent component specifications we have a single component architecture. We call the type of diagram shown in Figure 3.14 a component architecture diagram. This one shows a component specification architecture (we can tell that from the «comp spec» stereotypes). The same form (using the standard UML notation of underlining for instances) can be used to show a component object architecture.

We can also draw these diagrams without the interfaces to give a coarser-grain view (see Figure 3.15). It's unnecessary for this simple example, but for

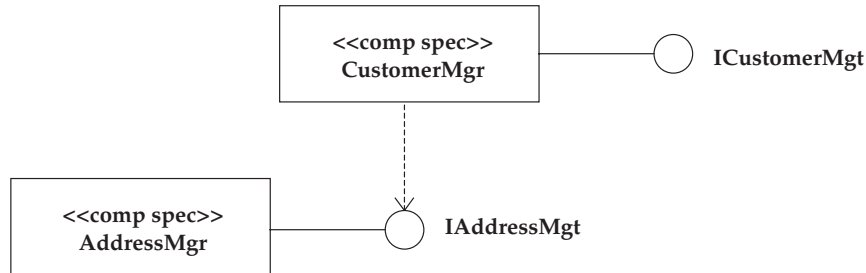


Figure 3.14 A component specification architecture

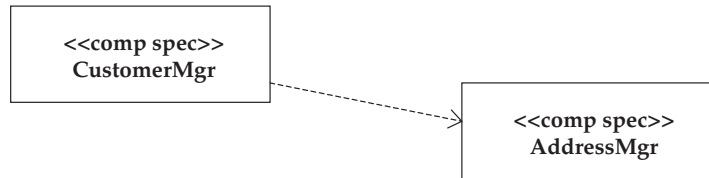


Figure 3.15 Component architecture showing dependencies at the component specification level

large architectures with many components, interfaces, and dependencies, it's a valuable abstraction.

We also draw component interaction diagrams within an architectural context, to understand how the components will work together and to discover the operations that are needed. If you know the broad shape of your architecture and your likely interfaces, the best way of figuring out the operations those interfaces need is to work through the collaborations. This process is discussed in detail in Chapter 6.

3.11 Summary

Congratulations. You've learned pretty much every bit of UML you'll need to do component specification with UML and to understand the case study. We've introduced the stereotypes we need and any constraints they have. We've described the types of diagram we need to draw. And we've

Table 3.1 Summary of UML extensions

Component Specification Concept	UML Construct	Stereotype
Component Specification	Class	«comp spec»
Interface Type	Type (Class «type»)	«interface type»
Comp Spec offers Interface Type	Dependency	«offers»
Business Concept	Class	«concept» (optional)
Business Type	Type (Class «type»)	«type»
Structured Data Type	Type (Class «type»)	«datatype»
Interface Information Type	Type (Class «type»)	«info type» (often omitted)
Parameterized Attribute	Operation	«att»
Operation requiring a new transaction	Operation	«transaction»

only had to extend UML a couple of places to fit. It's fair to say that UML has stood up pretty well to our requirements of it.

Table 3.1 summarizes the UML extensions we'll be using.

The most practical issue, of course, will be how strictly your tools support UML and what tool-interchange requirements you have. Standardizing on this set of UML extensions as a component specification profile would be a valuable step forward.