In discussing the structural concepts of ROOM in Chapter 6, we encountered two different types of class specifications: *actor classes* and *protocol classes*. A third type of class is the data class, which is discussed in Chapter 10. Since these class specifications describe very different types of objects, each has its own class inheritance hierarchy independent of the others. Moreover, the inheritance rules across the different hierarchies are not uniform, but are customized to the specific needs of the different object types. This, too, is different from what is found in current object-oriented languages, all of which are based on the notion of a single comprehensive class hierarchy. This not only leads to large unwieldy class hierarchies, but also forces common rules and representation for all object types.

In the remainder of this chapter, we examine how inheritance works for actor and protocol classes. In the case of actor classes, we discuss separately how inheritance works in the structural dimension and how it works in the behavioral dimension.

9.2.1 Structural Inheritance

We first look at how ROOM applies inheritance in the structural dimension.

9.2.1.1 Structural Attributes

Structure at the Schematic level is captured through actors and their decompositions. When we refer to structural inheritance, we mean the structural attributes of actor classes. The main structural attributes are

- Ports
- SAPs
- SPPs
- Component actors
- Bindings
- Layer connections
- Equivalences

In addition, some of these have further ("nested") attributes of their own. For instance, component actors have attributes that specify their replication factor, class, dynamic properties, and so forth. In principle, the same inheritance rules apply uniformly to all structural attributes.

9.2.1.2 Actor Class Inheritance Rules

The inheritance rules for actor classes are

- An actor class can have at most one parent class.
- A class automatically inherits all of the attributes of its parent class.

- Any inherited attribute can be excluded.
- Any inherited attribute can be overridden.

The first rule means that only single inheritance is supported for actor classes. While this may seem restrictive at first, as we shall see later in this section, there are ways to achieve the same effect as multiple inheritance without incurring its drawbacks. Both exclusion and overriding of structural attributes (including nested attributes) are allowed as a pragmatic measure for dealing with large and complex class hierarchies. (Note that this means that an actor class may not necessarily be a subtype of its superclasses.) The general guidelines described previously on how to use these capabilities should be applied in this case. Exclusion should be avoided except in situations where that would result in a major overhaul of the inheritance hierarchy, and overriding should be used only for refinement purposes.

Attributes not inherited from a superclass are called *local* attributes. All overridden attributes are considered local attributes.

9.2.1.3 Notational Conventions

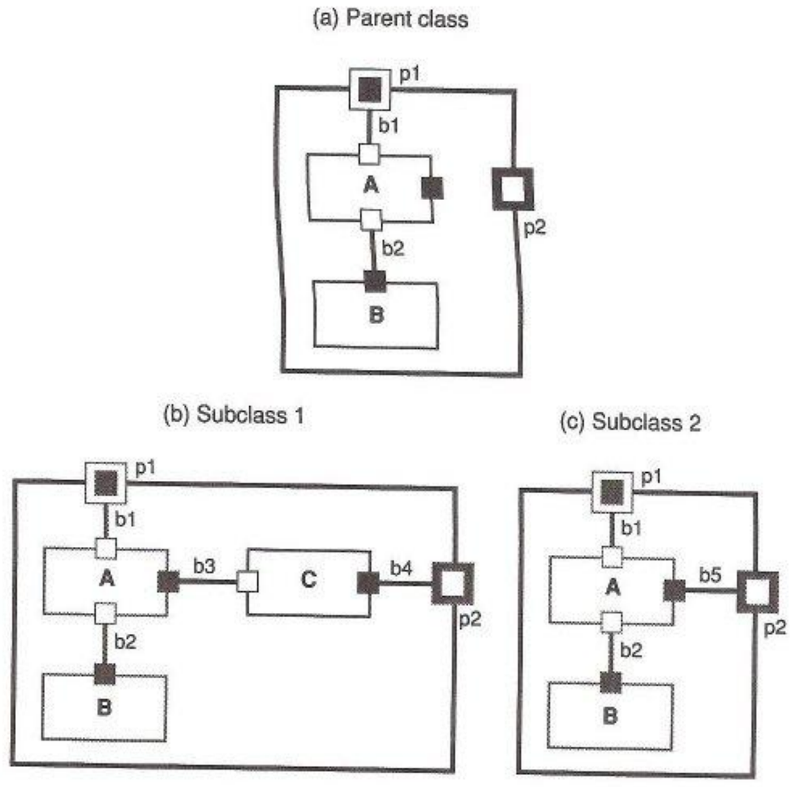
When examining an actor class specification in its graphical form, it is most useful to see the complete specification, including all inherited attributes, in a single diagram. In order to distinguish inherited attributes from local ones, we will use a convention in which inherited attributes are drawn using a lighter line style (grey), whereas local attributes are drawn with black lines. This is illustrated in Figure 9.5, which shows an example of a parent class and two of its subclasses. Both subclasses inherit all of the attributes of their parent (component actors A and B, bindings b1 and b2, and relay ports p1 and p2). To these they add their individual local attributes.

Clearly, this notational convention works best if it is supported by a computer-based graphical editing tool, since it is difficult to achieve variations in line shading by using conventional pencils or markers. This fits in with one of the basic principles of ROOM, that is oriented toward taking maximum advantage of the power nascent in computer-based tools. Nevertheless, if suitable tools are not available, some other notational convention may be used (for instance, inherited attributes may be drawn using dashed lines or lines of a different color).

9.2.1.4 Abstract Structures

What does it mean for a structure to be abstract? The most obvious way is for it to contain component actors or interface components that are themselves abstractions of their corresponding full-fledged counterparts. For example, a port may be defined in terms of an abstract protocol class that omits low-level detail (such as acknowledgment messages), or a component actor may be specified by an instance of an abstract class that only provides a high-level simulation of a concrete class. In this case, the structural specification is *com-*

Figure 9.5 Notation for Inherited and Local Attributes



plete in the sense that all the major functional components and relationships are present in the specification, even though some detail pertaining to this functionality is omitted.[4]

A second type of abstract structure is one in which major functional components, relationships, or interface components are missing. This is usually done because the missing

[4] The distinction between major functionality and detail is a relative one, and depends on the level of abstraction being considered.

functionality may be implemented differently in different subclasses. An example of this type of abstract structure can be seen in the parent class in Figure 9.5a. This class does not show the relationship between relay port p2 and the rest of the structure. The missing relationship is defined, in two different ways, by each of the two subclasses (Figure 9.5b and Figure 9.5c).

Previously in this chapter we noted that we should strive to make all classes in the class hierarchy meaningful. The difficulty with *incomplete* abstract classes is that this is not always possible. For the abstract class in Figure 9.5a, we can only guess at the possible relationship between the interface port p2 and the rest of the structure. As it stands, any messages arriving on that interface component will be discarded, making one wonder why it was defined in the first place. To put this problem in perspective, imagine the difficulty of trying to describe how a jet airplane works while avoiding any mention of the engines.

One technique for avoiding incomplete specifications is to insert an abstract "placeholder" component actor to represent the missing functionality. The various realizations of this functionality then can be defined as subclasses of the abstract placeholder class.

Another problem with incomplete abstract classes is that, with major functionality absent, they either are not executable, or, if they are, then their behavior is not representative of the behavior of their corresponding concrete class. This means that they cannot be used as high-level lightweight simulators of their corresponding system. The ability to execute an abstract architecture and thus obtain early feedback on its potential is one of the underpinnings of the ROOM methodology. For this reason, it is recommended to provide abstract classes that are both complete and executable. We will refer to such abstract classes as *simulation classes*, since they can be used in architectural simulations. In the ideal case, all abstract classes should be simulation classes. However, if pragmatic issues force us to rely on incomplete abstract classes, then we should consider specifying at least one subclass that can serve as a simulation class.

In addition to their role in architectural evaluation, simulation classes, being executable specifications, can also be used as a basis for discussing system requirements with customers and for conveying design intent to members of the development team who need a high-level understanding of the class. It is possible to have multiple simulation classes of a concrete class, each at a different level of abstraction. These successions of simulation classes are often the normal by-products of top-down development, and it is usually beneficial to retain them in the class hierarchy even after the abstract class has been refined.

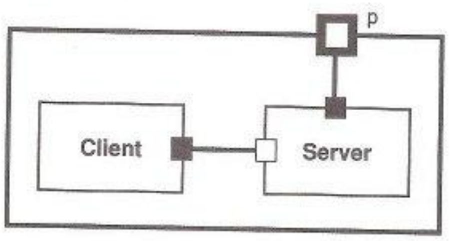
Finally, the third way in which structures can be abstract is if some of the component actors are defined as generic or substitutable. In this case, refinement takes place by mechanisms other than inheritance (either at the time the system is configured in preparation for execution or, dynamically, as the system is executing). This topic was discussed in detail in Section 6.4.3.

9.2.1.5 Subclassing

We now turn our attention to techniques for subclassing abstract structures. If we adhere to the guidelines described in Section 9.1, this involves either adding more attributes or replacing (overriding) existing ones with more refined versions. In certain special and infrequent cases, we may also need to delete an attribute.

To better understand all the issues, we use the example of a simple abstract client-server system depicted in Figure 9.6.

Figure 9.6 A Simple Client-Server Structure



The most common and simplest refinement involves substituting the class of either component by a more refined subclass. For instance, the server component in the initial abstract class may be just a simulation class and we may want to replace it with the actual concrete class. The first dilemma facing the designer in this situation is whether to modify the existing abstract class or to create a new subclass. To determine the right course of action, it is necessary to ask the following questions:

- Does the abstract class already have subclasses?
- If the class has subclasses, are there subclasses to which the modification should not apply?
- Is the abstract class a simulation class that may be useful in the future?

If we answer "yes" to any of these questions, then it is better to leave the abstract class unchanged and create a new subclass to which the modification can be made.

Another common type of refinement during subclassing involves replacing an abstract protocol (such as the one between the client and server actors) with a more detailed one. Since the protocol is a part of the type specification of the component actors, this means that they, too, must be replaced by more detailed classes that are capable of handling the more refined protocol.

Changing the inherited interface components in a subclass means that we are changing the type of a class. We must be aware that, unless the new interface components are compatible with the old ones, we cannot place instances of the new subclass in positions previously occupied by its abstract class, and may be forced into a cascade of similar refinements in other classes that incorporate this one.

Another common refinement of abstract structures is changing an attribute (such as a component or an interface) from being nonreplicated to replicated. This type of change usually entails changes in other structural parameters. For example, if we make the client actor in Figure 9.6 replicated, then we must change the interface of the server class to include a replicated port.

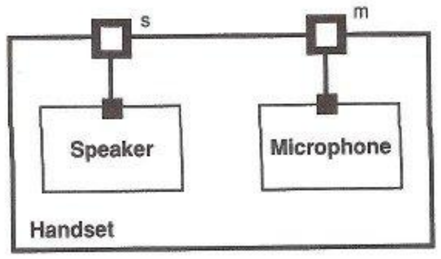
In most cases, we would prefer to refine an abstract class into a subclass without changing its semantics, perhaps by some formal means. That would ensure that any properties established for the abstract class are propagated automatically to the subclasses. Unfortunately, the current state of the art is far from this ideal, so the responsibility of preserving the desired semantics across subclassing operations is mostly dependent on the skill of the designers. The danger of corruption is greatest with overriding, since we can never be sure that an abstract attribute and its refined version are indeed compatible.

9.2.1.6 Alternatives to Structural Inheritance

We have already noted that, in some cases, it may be possible to avoid subclassing by directly modifying a class. Another common way of avoiding inappropriate subclassing is to use inclusion. For example, although one might be misled into defining a telephone handset class as a subclass of microphone, it is much more appropriate to include a microphone as a component of the handset actor structure (see Figure 9.7). The concept of relay ports, as defined in ROOM, makes this technique quite appealing since the only overhead associated with inclusion is to add a binding from a relay port to the corresponding interface component of the included component. In most implementations, any run-time overhead caused by the presence of relay ports can be optimized out.

Inclusion, in combination with relay ports and bindings, also practically eliminates the need for multiple inheritance for actor classes. In our telephone handset example, in order to make a handset behave like combination of a microphone and a speaker, all that must be done is to incorporate the necessary components into a new composite class.

Figure 9.7 Inclusion as an Alternative to Inheritance


s
m
Speaker
Microphone
Handset