Chapter 11 Reuse and Pluggable Design Frameworks in Code

Code reuse is a sought-after goal, but it does not happen automatically. It costs money, and it requires explicit attention at the level of design and in the structure of the development process within and across projects.

Reuse requires that components be built in a manner that is both generic—not overly tied to a specific application—and customizable so it can be adapted to specific needs.

Reuse comes in many flavors, from cut-and-paste to building libraries of low-level utility routines and classes to creating skeletons of entire applications with plug-points that can be customized. The latter requires a particular mind-set to extract commonality while deferring only those aspects that are variable.

There are two keys to systematic use of frameworks in code: The first is to make problem descriptions more generic. Second, you must have code techniques for implementing generic or incomplete problem specifications and then specializing and composing them.

11.1 *Reuse and the Development Process*

One of the most compelling reasons for adopting component-based approaches to development, with or without objects, is the promise of reuse. The idea is to build software from existing components primarily by assembling and replacing interoperable parts. These components range from user-interface controls such as listboxes and HTML browsers to components for networking or communication to full-blown business components. The implications for reduced development time and improved product quality make this approach very attractive.

11.1.1 What Is Reuse?

Reuse is a variety of techniques aimed at getting the most from the design and implementation work you do. We prefer not to reinvent the same ideas every time we do a new project but rather to capitalize on that work and deploy it immediately in new contexts. In that way, we can deliver more products in shorter times. Our maintenance costs are also reduced because an improvement to one piece of design work will enhance all the projects in which it is used. And quality should improve, because reused components have been well tested.

11.1.1.1 Import Beats Cut-and-Paste

Something like 70% of work on the average software design is done after its first installation. This means that an approach, such as reuse, aimed at reducing costs must be effective in that maintenance phase and not just in the initial design (see Figure 11.1).

People sometimes think of reuse as meaning cutting chunks from an existing implementation or design and pasting them into a new one. Although this accelerates the initial design process, there is no benefit later. Any improvements or fixes made to the original component will not propagate to the adapted versions. And if you're going to adapt a component by cut-and-paste, you must first look inside it and understand its entire implementation thoroughly—a fine source of bugs.

Good reuse therefore implies using the same unaltered component in many contexts, a technique much like the idea of importing packages described in Chapter 7, Using Packages. Texts on measuring reuse don't count cutting and pasting as proper reuse.

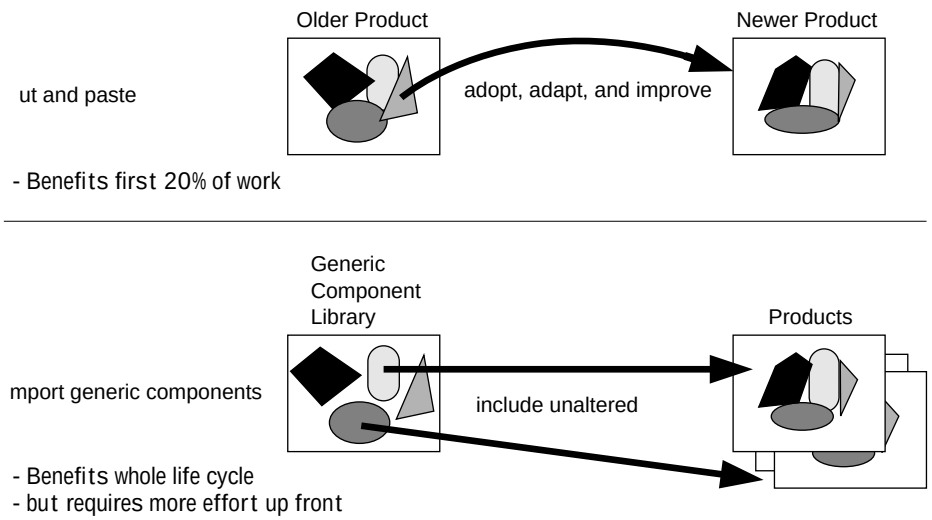


Figure 11.1 Cut-and-paste versus Import.

11.1.1.2 The Open-Closed Principle and Reuse Economics

But there is a difficulty. If alterations are not allowed, a component can be useful in many contexts only if it is designed with a good set of parameters and plug-points. It must follow the well-known Open-Closed Principle:

> *Every component should be open to extension but closed to modification.*

It takes effort to work out how to make a component more generic, and the result might run slower when deployed. The investment will be repaid in savings every time the component is used in a design and every time maintenance must be done on only one component rather than many slightly different copies. But it is not always certain exactly when or how a component will be reused in the future. So, as with all investments, generalizing a component is a calculated risk.

11.1.2 What Are the Reusable Artifacts?

A reusable artifact is any coherent chunk of work that can be used in more than one design. Following are examples.

- Compiled code; executable objects
- Source code; classes, methods
- Test fixtures and harnesses
- Designs and models: collaborations, frameworks, patterns
- User-interface "look and feel"
- Plans, strategies, and architectural rules

This list includes all kinds of development work. We have already discussed model frameworks and template packages, which can be valuable to an enterprise. They are *white-box* assets: What you see is what is they offer. By contrast, an executable piece of software, delivered without source code, can perform a useful and well-defined function and yet not be open to internal inspection. Software vendors prefer such *black-box* components.

11.1.3 Reuse Truths

What should you reuse? The executable? Source code? Interface specifications? Problem domain models?

© ***Reuse Law 1*** Don't reuse implementation code unless you intend to reuse the specification as well. Otherwise, a revised version of the implementation will break your code.

© ***Reuse Lemmas*** (1) If you reuse a specification, try a component-based approach: Implement against the interface and defer binding to the implementation.

(2) Reuse of specifications leads to reuse of implementations. In particular, whenever you can implement standardized interfaces, whether domain-specific or for infrastructure services, you enable the reuse of all other implementations that follow those standards.

(3) Successful reuse needs decent specifications.

(4) If you can componentize your problem domain descriptions themselves and reuse domain models, you greatly enhance your position to reuse interface specifications and implementations downstream.

11.1.4 A Reuse Culture

In a reuse culture, an organization focuses on building and enhancing its capital of reusable assets, which would include a mixture of all these kinds of artifacts. Like any investment, this capital must be managed and cultivated. It requires investment in building those assets, a suitable development process and roles, and training and incentives that are appropriate for reuse.

But designs must be generalized to be reusable. Generalizing a component can't be justified in terms of its original intended purpose. If you write a collision avoidance routine for airplanes, there's no reason you should do the extra work to make it usable by your maritime colleagues: You have your own deadlines to meet. And if your product deals only with air traffic, you have no reason to separate out those pieces of the airplane class that could apply to vehicles in general. All these generalizations require broadening of requirements beyond your immediate needs.

On the other hand, if you notice three lines of code that crop up in six different places in your own product, then you will easily see the point of generalizing them and calling a single routine from each place. That's because you're controlling the resources for all the places it could be used; and the problem is small enough that you can easily get a handle on what's required.

On the larger scale, reuse of components between individuals, between design teams, and across and outside organizations takes more coordination. It's usually someone who holds responsibility for all the usage sites who can assign the resources to get the generalization done. To make it happen, reuse needs an organization and a budget.

A significant part of identifying large-grained reuse comes from careful modeling of the problem domain and of the supporting domains—UI, communications, and so on—that would be a part of many applications. This activity also crosses specific project boundaries and so needs organization support.

Should a component be made sufficiently general for reuse? How generic and reusable should it be? These are decisions that must be made consciously and carefully. We have all made such generalizations when deciding that a few lines of code could be moved to a subroutine of their own. Parameterizing a whole class or group of classes follows the same principle but employs a wider variety of patterns and should be more carefully thought through.

Some components can be reused more widely than others. Some objects, routines, or patterns might be useful only in several parts of the same software product; but if it is a big product or a product family, several teams may need coordinating.

"Galloping generalization" is the syndrome wherein a group spends months producing something that runs like a snail on dope and has hundreds of interesting features, most of which will never be used. The best strategy seems to be to generalize a component only after

it is earmarked for use in more than one context; and then generalize it only as much as is necessary for the envisaged applications.

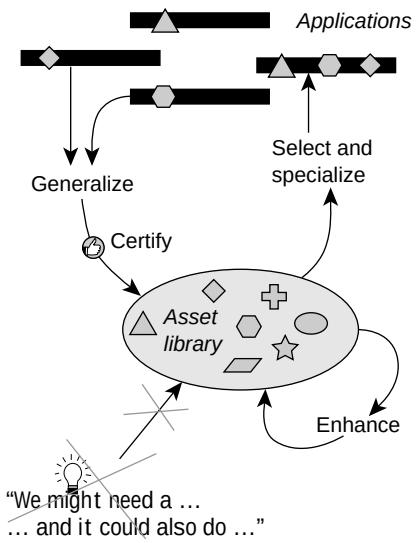
Naturally, the organization that measures developer productivity in terms of lines of code written has some rethinking to do before reuse can succeed. Suitable incentives schemes should be based more on the ratio of code reused to new code written.

11.1.5 Distinct Development Cycles

In a reuse culture, development tends to split into two distinct activities.

• *Product development:* The design and creation of applications to solve a problem. This phase is centered on understanding the problem and rapidly locating and assembling reuse capital assets to provide an implementation.

• *Asset development:* The design and creation of the reusable components that will be used in different contexts. This is carried out with more rigorous documentation and thought. Because software capital assets will be used in many designs, the impact of a change can be, for better or worse, quite large.



It is therefore worth putting much serious effort and skill into assets. Of course, the products must work properly; but whereas much may be gained by, for example, tuning a reusable asset's performance as far as possible, a product that is used only in one context often must be good enough only for that purpose. Strong documentation also pays off more with assets than with products.

For developing reusable assets, you would generally want to apply many of the techniques in this book. Reuse means investing in the quality of software; the old argument that "we don't have time to document" can have only a negative effect in a reuse culture. The development of products or applications will also use many of the same techniques, but the process can be quite different (for example, see Section 10.11, Heterogenous Components).

11.2 *Generic Components and Plug-Points*

For a component to be reused in different contexts, it must be sufficiently generic to capture the commonality across those contexts; yet it must also offer mechanisms so it can be specialized as needed.

This means that you must understand what parts of the component are common across those contexts. Design it so that those parts that vary across contexts are separated from

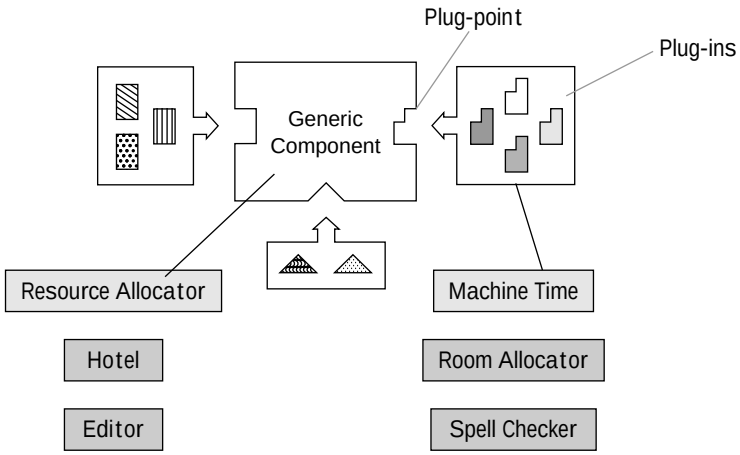


Figure 11.2 Plug-points for plug-ins.

the component itself at carefully selected *plug-points* where that variation can be encapsulated and localized—places where specialized components can be plugged in to adapt the overall behavior. As Figure 11.2 illustrates, a resource allocator component might be specialized to allocate machine time on a factory floor; a hotel component might accommodate various room allocation policies; and an editor might accommodate any spell checker.

11.2.1 Plugs: The Interfaces

Let's look at a couple of ways in which components can be made to plug together. In particular, it helps to distinguish between (1) composing components to build something bigger and (2) plugging parts into a generic component in order to specialize its behavior to current needs. Although both approaches place similar demands on modeling and design, the intent of each one is different. These distinctions may get blurred in the presence of callback-styled programming and reentrant code; when an architecture is layered, with calls restricted to a single direction from higher layers to lower, the distinction remains sharp.

11.2.2 Upper Interfaces: For "Normal" Use

The components we use are made from other components. Figure 11.3 illustrates how a larger component (a video rental system) might be assembled by assembling components for membership, reservations, and stock management. We think of this as the "upper" interface; it is the direct and visible connecting of parts that provide well-defined services. Examples of such upper interfaces are the public operations of a class, APIs of databases, windows systems, and, of course, the primary services offered by the membership, reservations, and stock management components.

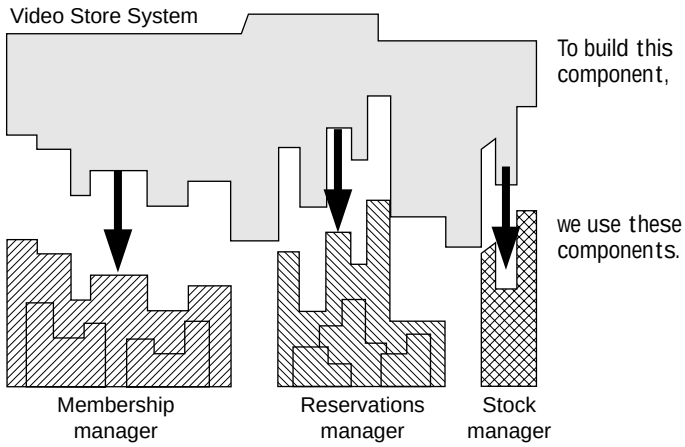


Figure 11.3 Using upper interfaces to build a larger component.

11.2.3 Lower Interfaces: For Customization

Each of the components whose "normal" interface we used is itself an incomplete implementation; to provide its services, it needs additional parts to be plugged in to it. Figure 11.4 illustrates the use of "lower" interfaces via which a generic component is specialized with several plug-ins that customize its behavior to the problem at hand. In this case, a generic membership manager is being adapted by plugging in specifics for video store members and their accounts.

Generic components have plug-points—parameterized aspects that can be filled in appropriately in a given context—both for implementation components and for generic model components that are built to be adapted and reused. When designing a complex component, we might reach into a component repository and build our specific models from generic model components in this library. Of course, the generic versions must provide mechanisms for extension and customization to the specific domain at hand.

Modern desktop software bristles with plug-points. Web browsers, as well as word processors and spreadsheets, accept plug-ins for displaying specialized images; desktop publishing software accepts plug-ins for doing specialized image processing. In those cases, the plug-ins usually must be designed for a particular parent application. Using dynamic linking technologies, the plug-ins are coupled with the parent application when it begins to run.

Every OO language provides some form of plug-ins. The most common form is the use of framework classes: the superclasses implement the skeleton of an application—implementing methods that call operations that must be defined in the subclasses—and a set of subclasses serves to specialize the application. The plug-points are the subclasses and their overriding methods. In C++, a template `List` class can be instantiated to provide lists of numbers, lists of elephants, or lists of whatever class the client needs; the plug-point is a simple template parameter.

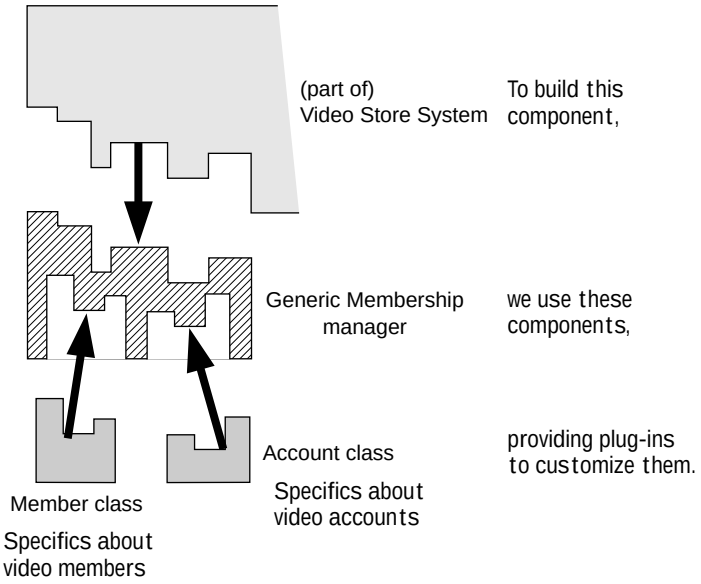


Figure 11.4 Using lower interfaces to customize a component.

The principle is not limited to single classes but rather can span multiple abstract classes that collaborate and must be jointly extended before use. Many class libraries provide user-interface frameworks (such as Smalltalk's Model-View-Controller). You make a user interface (either manually or with a visual builder tool) by inheriting from the framework classes and plugging specializing methods in to your subclasses.

11.2.3.1 Infrastructure Services: A Special Kind of Lower Interface

To operate properly, many components need an underlying set of infrastructure services (see Section 10.2.2, Components and Standardization). These services do not customize the behavior of the component in any interesting way; they simply provide an implementation of a common *virtual machine* for use by all components.

Obvious examples include the POSIX interface, which provides a common view of many different operating systems, and the Java virtual machine, which provides all the services needed to run Java components. However, this underlying virtual machine may be more specialized to the problem at hand—for example, a state-machine interpreter or a graph transformation engine.

11.3 *The Framework Approach to Code Reuse*

In object-oriented design and programming, the concept of a framework has proven to be a useful way to reuse large-grained units of design and code while permitting customiza-

tion for different contexts. The style of reuse with frameworks, and the mind-set for factoring out commonality and differences, is quite distinctive.

11.3.1 OOP Frameworks

An object-oriented framework is often characterized as a set of abstract and concrete classes that collaborate to provide the skeleton of an implementation for an application. A common aspect of such frameworks is that they are adaptable; the framework provides mechanisms by which it can be extended, such as by composing selected subclasses together in custom ways or defining new subclasses and implementing methods that either plug in to or override methods on the superclasses.

There are fundamental differences between the framework style and traditional styles of reuse, as illustrated by the following example.

> *Design and implement a program for manipulating shapes. Different shapes are displayed differently. When a shape is displayed, it shows a rendering of its outline and a textual printout of its current location in the largest font that will fit within the shape.*

In the next two sections we will contrast the traditional approach to reuse with the framework style of factoring for reuse.

11.3.1.1 Class Library with Traditional Reuse

A traditional approach to reuse might factor the design as follows. Because the display of different shapes varies with the kind of shapes, we design a shape hierarchy. The display method is abstracted on the superclass—because shapes display themselves differently—and each subclass provides its own implementation.

There are common pieces to the display method, such as computing the font size appropriate for a particular shape given its inner bounding box and printing the location in the computed font. Hence we implement a computeFont and printLocation method on the superclass (marked protected in Java so that it is subclass-visible).

```
class Shape {
    // called from subclass: given a Bounding Box and String, compute the font
    protected Font computeFont (BoundingBox b, String s) { .... }
    // called from subclass: print location on surface with Font
    protected void printLocation (GraphicsContext g, Font f, Point location) { ... }
    public abstract void display (GraphicsContext g);
}
```

A typical subclass would now look like this:

```
class Oval extends Shape {
    // shape-specific private data
    private LocationInfo ovalData;
    // how to compute my innerBox from shape-specific data
    private BoundingBox innerBox() { .... }
    // rendering an oval
    private void render (GraphicsContext g) { ... trace an oval ... }
    // display myself
```

```
        public void display (GraphicsContext surface) {
            render ();
            BoundingBox box = innerBox();
            // let the superclass compute the font
            Font font = super.computeFont (box, ovalData.location.asString());
            // let the superclass print the location
            super.printLocation (surface, font, ovalData.location);
        }
    }
```

A class model is shown in Figure 11.5. The dynamics of the inheritance design can be shown on an enhanced interaction diagram, separating the inherited and locally defined parts of an object to show calls that go up or down the inheritance chain.

11.3.1.2 Framework-Style Reuse and the Template Method

With framework development, the skeleton of the common behavior is one of two things: either (1) an internal method specified on an interface that must be implemented by a specialized class, or (2) a *template method* in the superclass with the variant bits and pieces deferred to the subclasses. We will illustrate the latter design here.

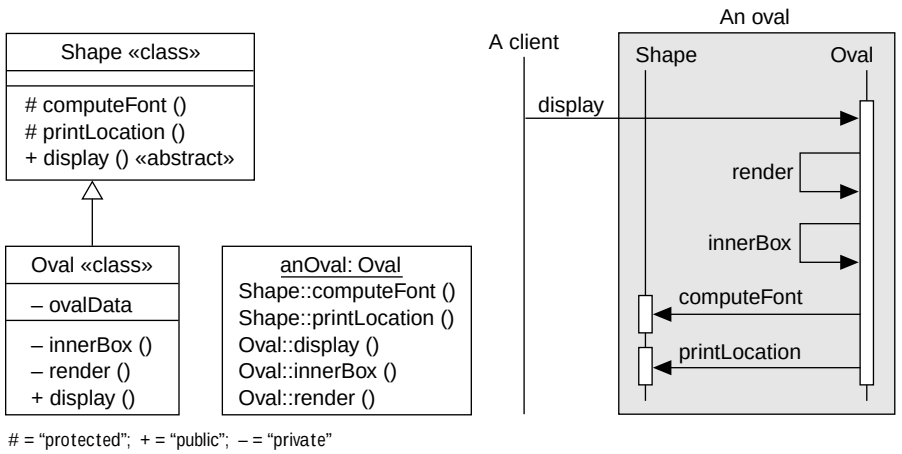


Figure 11.5 Inheritance design and enhanced interaction diagram.

With a framework approach to reuse, our factoring looks quite different from that of the traditional approach. We start with the assumption that all shapes fundamentally do the same thing when they are displayed: render, compute a font for their inner bounding box, and print their location in that font. Thus, we implement at the level of the superclass:

```
class Shape {
    public void display (GraphicsContext surface) {
        // delegate to subclass to fill in the pieces
        render (surface);                    // plug-point: deferred to subclass
```

```
            BoundingBox box = innerBox(); // plug-point: deferred to subclass
            Point location = location();        // plug-point: deferred to subclass

            // then do the rest of "display" based on those bits
            Font font = computeFont (box, location);
            surface.printLocation (location, font);
        }
    }
```

The actual rendering and computation of the inner box and location must be deferred to the subclasses: *plug-points*. However, if a subclass provides the appropriate bits as *plug-ins*, it can inherit and use the same implementation of display. Thus, we are imposing a consistent skeletal behavior on all subclasses but permitting each one to flesh out that skeleton in its own ways.

```
class Oval extends Shape {
    // implement 3 shape-specific "plug-ins" for the plug-points in Shape
    protected void render (GraphicsContext g) { ...trace an oval ...}
    protected BoundingBox innerBox () { ... }
    protected Point location () { return center; }

    // private shape-specific data
    private Point center;
    private int majorAxis, minorAxis, angle;
}
```

Figure 11.6 illustrates this approach. Although this example focuses on a single class hierarchy, it extends to the set of collaborating abstract classes that are characteristic of frameworks.

To display itself, the Shape hierarchy, for example, requires certain services from the GraphicsContext object. There could also be different implementations of GraphicsContext—for screens and printers—using a similar framework-styled design. It is this partitioning of responsibility—among different shape classes and among shapes and the GraphicsContext—that gives the design its flexibility. Thus, any packaging of a class as a reusable unit must also include a description of the behaviors expected of other objects—that is, their *types*.

Table 11.1 Contrast between Traditional and Framework Styles

Traditional	Framework
Begin with the mind-set that the display methods would be *different* and then seek the *common pieces* that could be *shared* between them.	Assert that the display methods are really the *same* and then identify the *essential differences* between them to *defer* to the subclasses.
Focus on sharing the *lower-level* operations such as computeFont and printLocation. The higher-level application logic is duplicated, and	Share the *entire skeleton* of the application logic itself. Each application plugs in to the skeleton the pieces (such as render, innerBox, particular GraphicsContexts) required to complete the

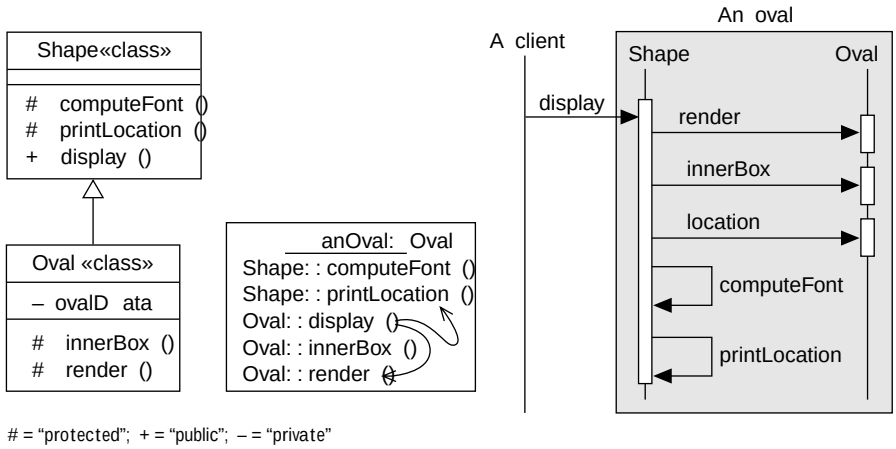


Figure 11.6 Framework-style inheritance design.

each one calls the shared lower-level bits.	skeleton. Most (many) calls go from the framework skeleton to the individual applications; in fact, one of the hallmarks of a framework is, "Don't call me—I'll call you." Define an interface representing demands that the reusable skeleton framework makes on the applications—the plug-points for extension. Application contains newer code; however, the existing (base) code calls newer code to delegate specialized bits. The framework implements the architecture and imposes its rules and policies on the applications.

11.3.1.3 Contrast of Styles

Table 11.1 summarizes the contrast between the approaches in terms of factoring of code, degree of reuse, and consistency of resulting designs. These differences are summarized in Figure 11.7, which show the contrast between base and application levels in the two approaches.

A significant part of framework design is factoring the plug-points that are provided for adaptation or customization. This example requires that a subclass provide the missing behaviors. A more frequently recommended design style for frameworks is based on delegation and composition; by composing an instance of a framework class with an instance of a custom class that implements a standard framework interface, we adapt the behavior of the framework.

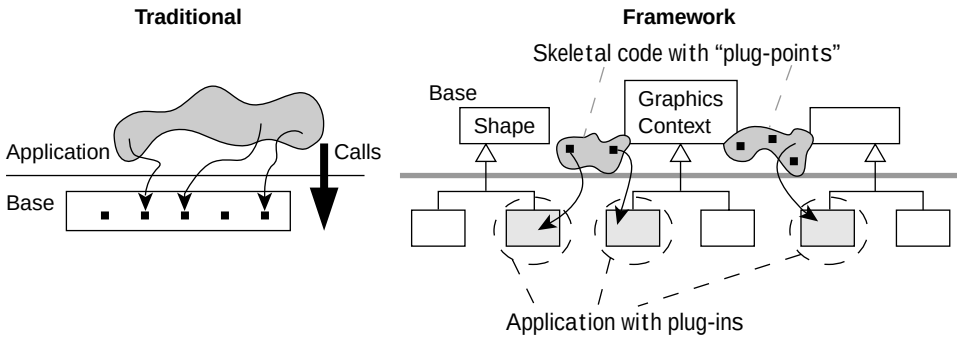


Figure 11.7 Traditional versus framework-style designs.

11.3.2 Non-OOP Frameworks

Object-oriented programming provides novel ways to implement with plug-points and plug-ins by using class inheritance. However, you can use other techniques to achieve the underlying style of implementing a skeletal application and leave places in it that can be customized. The central such technique is delegation (see Section 11.5.3, Polymorphism and Forwarding). The framework approach to building systems extends to component-based development as well (see Section 1.9.2, Component Frameworks).

11.4 *Frameworks: Specs to Code*

In the preceding section we saw how framework techniques in object-oriented programming can help build a skeletal implementation, with plug-points for customization to specific needs. We saw in Chapter 9, Model Frameworks and Template Packages, that pieces of code are not the only useful reusable artifacts; recurrent patterns occur in models, specifications, and collaborations. Moreover, the basic OOP unit of encapsulation—a class—is not the most interesting unit for describing designs; it is the collaborations and relationships between elements that constitute the essence of any design.

11.4.1 Generalizing and Specializing: Models and Code

To systematically apply framework-based techniques to development, we start with template packages to construct domain models, requirements specifications, and designs from frameworks. We could construct the specifications for a particular problem by applying the generic framework and plugging in details for the problem at hand. On the implementation side, an implementation for the generic specification should be correspondingly customizable for the specialized problem specification (see Figure 11.8).

A framework implementation thus provides a customizable solution to an abstract problem. If it is done right, the points of variability in the problem specifications—plug-

points on the specification side—will also have corresponding plug-points on the implementation side.[1]

11.4.1.1 Combining Model Frameworks

Consider the operations of a service company that markets and delivers seminars. Different aspects of this business, and hence its software requirements, can be described separately: allocation of instructors and facilities to a seminar, on-time production of seminar materials for delivery, trend analysis for targeted marketing of seminars, invoicing and accounts receivable, and so on.

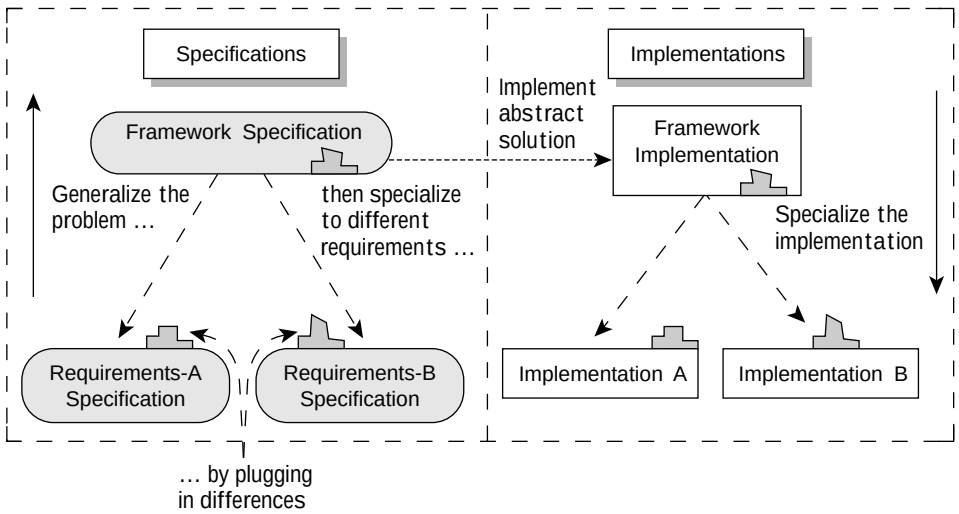


Figure 11.8 Frameworks for specification versus implementation.

Each such aspect can be generalized to be independent of seminar specifics, creating a library of reusable abstract specification frameworks, shown in Figure 11.9 with details of invariants and action specs omitted.

This model framework uses abstract types such as Job, Requirement, and Resource and abstract relationships such as meets, provides, and so on. These types will map in very different ways to a car rental application (Resource=Vehicle, Job=Rental, meets=model category matches) than to assigning instructors to seminars (Resource=Instructor, Job=Session, meets=instructor qualified for session topic).

These frameworks must now be mapped to our problem domain and must be related to one another by shared objects, attributes, and so on. Figure 11.10 shows the overall problem model as an application of these frameworks. Of course, these frameworks must inter-

1. The relation between these plug-points is analogous to refinement.

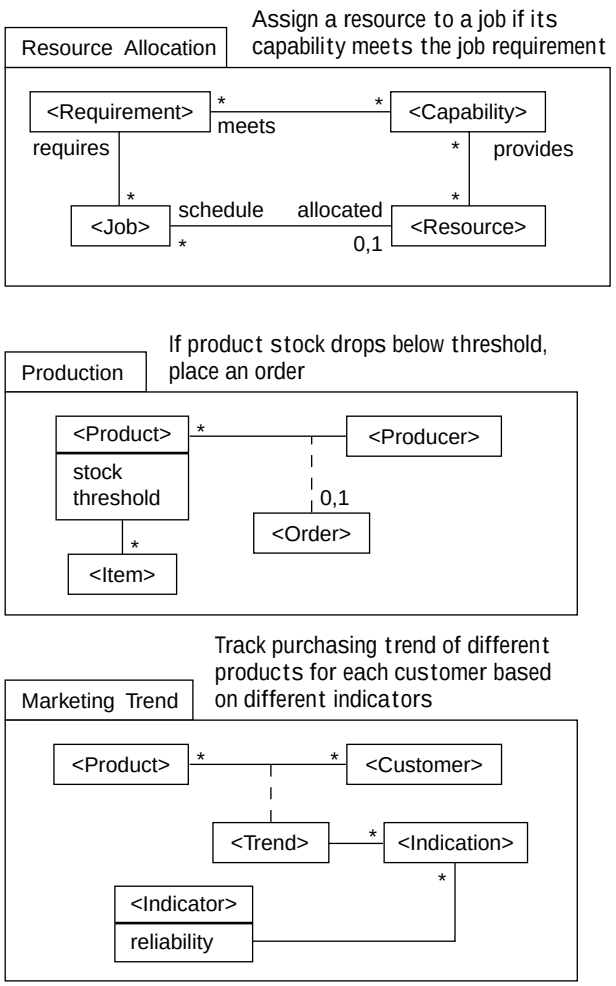


Figure 11.9 Specification frameworks for the seminar business.

act. A session must have both an instructor and a room assigned; failure of either means that the session cannot hold.

When a session holds, copies of course materials must be produced, and the customer trends are updated. Note that each problem domain object can play multiple roles in different frameworks. For example, a Course Title serves as a Requirement in the two applications of the Resource Allocation framework and serves as a Product in the applications of the Marketing and Production framework.

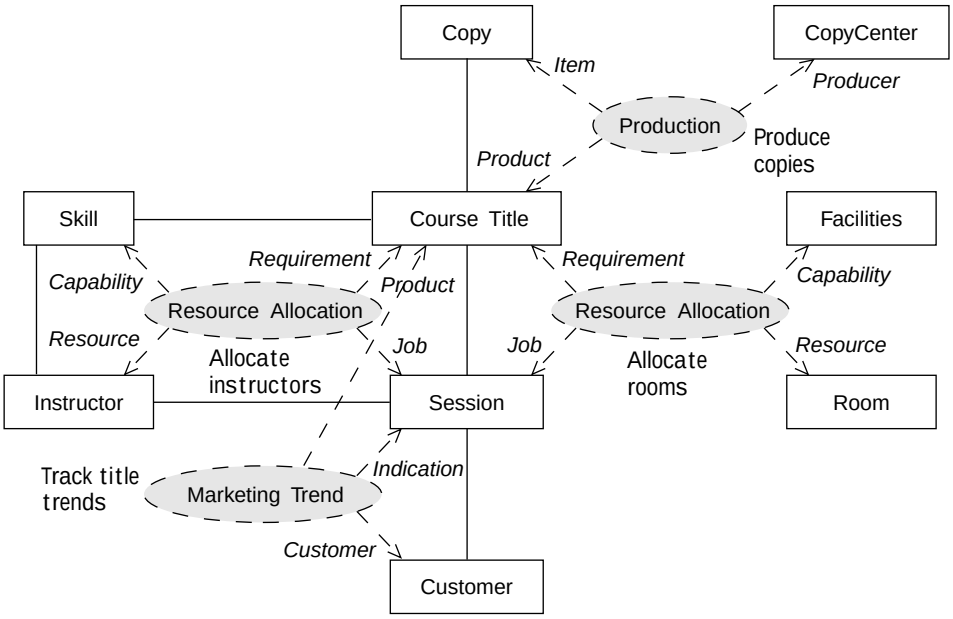


Figure 11.10 Specification by composing frameworks.

11.4.1.2 Combining Code Frameworks

Each of the model frameworks in Figure 11.10 could come with a default implementation framework. Our design, at the level of framework-sized components, would look like Figure 11.11. Each of the framework components has its plug-points suitably filled by implementation units from this problem domain. Thus, the instructor allocator has Instructor and Session as plug-ins for Resource and Job; and the trend monitor has Session and Topic plugged in for Indication and Product.

Here are two (of many) schemes to implement the plug-ins while making these frameworks interact correctly with each other.

1. Separate objects (SessionJob, Occupancy, Indication) within each framework capturing the framework's view of a session so that we have a federated component system. We also need some form of cross-component links between them and a mechanism to keep them in sync. This is often best implemented with a central session object, SessionGlue, that acts as the glue between each of the role objects; it registers for events from the roles and uses the events to keep the other roles in sync. Each of these role objects can inherit a default implementation that is part of the generic framework itself:

```
// the role of a session within the instructor allocation framework
class SessionJob implements IJob extends DefaultJob{
```

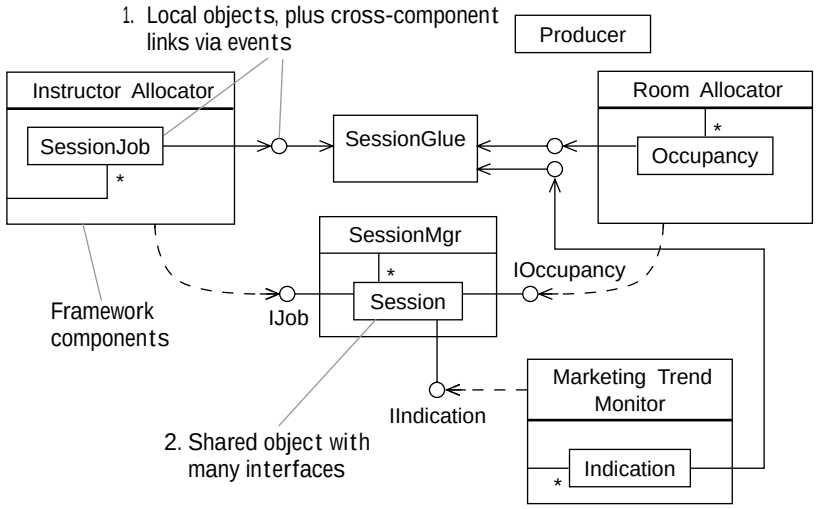


Figure 11.11 Two alternative ways to compose code frameworks.

```
    // link to the "glue" object, SessionGlue, via an event notification interface
    ISessionJobListener listener;

    // just became confirmed from the allocation framework
    void confirm () {
        super.confirm () // do normal inherited confirmation stuff
        // notify the listener object about confirmation
        listener.confirmed (self);
        // could also update the marketing indicators directly; simpler, more coupled
        // MarketingTrendMonitor::indicationConfirmed (trendIndication);
    }
}

class SessionGlue implements ISessionJobListener, ... {
    // maintain the cross-component links
    // code to receive events from one of the role objects
    // update any shared state, invoke methods on other role objects
}
```

2. Shared objects for the sessions, perhaps implemented in a SessionMgr component (which might be the shared database). Each session offers different interfaces for each role a session plays in each of the frameworks: IJob, IOccupancy, IIndication. Its class will implement methods on each interface to make explicit invocations into the other frameworks if necessary.

```
// one interface for each role this shared object plays
class Session implements IIndication, IJob, IOccupancy {
    // just became confirmed from the allocation framework
```

```
        void confirm () {
            ...do normal confirmation stuff
            // but confirmation must also update the marketing indicators
            MarketingTrendMonitor.indicationConfirmed (self);
        }
    }
```

A third way is to uniformly compose roles (see Pattern 11.1, Role Delegation).

11.4.2 Issues of Composing Multiple Code Frameworks

Combining independently designed frameworks is not easy. As discussed in Section 11.3.1.3, Contrast of Styles, a framework attempts to codify and prescribe the policies and rules for interactions among its application's objects; in particular, the framework usually takes control, and the applications simply plug in the parts they need. We need explicit attention to interframework coordination at the level of the interframework architecture.

One way to open up the control is to adopt a more component-oriented approach across frameworks. Each framework instance might publish a variety of "internal" events to other interested framework instances; these events expose selected state changes from one framework and offer the others a chance to react to the change.

A somewhat more sophisticated approach would refine the event notification to a negotiative-style protocol: Rather than announce "I've done X," a framework announces, "I'm about to do X; any objections?" Other frameworks then have a chance to veto the proposed event if it would compromise some of their rules. This approach is particularly useful when each framework imposes its own restrictions on what can be done in the other rather than only extending the behaviors (as described in the discussion on joining of specifications in Section 8.3.5, Joining Type Specifications Is Not Subtyping). JavaBeans offers a facility of vetoable events that uses exceptions to signal the vetoing of a proposed event. At the level of federated business components, an alternative is to use cross-component transactions to the same effect: either all, or none, complete.

11.5 *Basic Plug Technology*

There are several implementation mechanisms for achieving the effect of plug-points and plug-ins. This section discusses the main ones, emphasizing the value of black-box composition over white-box inheritance for large-grained reuse.

11.5.1 Templates

C++ provides a compile-time template facility that can be used to build generic classes or families of generic classes. One way to implement a framework for resource allocation is to use a family of C++ template classes that are mutually parameterized:

```
template <class Job>
```

```
class Resource {          // a resource is parameterized by its job
    Set<Job*> schedule;
    makeUnavailable (Date d) {
        ...
        for ( each job in schedule overlapping d, if any )
            job.unconfirm ();
    }
}

template <class Resource>
class Job {               // a job is parameterized by its resource
    Resource* assignedTo;
    Range<Date> when;
    unconfirm () { .... }
}

template <class Resource, class Job>
class ResourceAllocator {// a resource allocator manages resources and jobs
    Set<Resource*> resources;
    Calendar<Resource*, Job*> bookings;
    ...
}
```

We can use inheritance to have a domain class, Instructor, act as a resource for a seminar session:

```
class Instructor : public Resource<Session*>, ...
```

We might use multiple inheritance[2] to have our Session play the role of Job for two resources:

```
class Session : public Job<Instructor*>, public Job<Room*> {
    ....
}
```

11.5.2 Inheritance and the Template Method

For an inheritance-based design, the template method (see Section 11.3.1.2, Framework-Style Reuse and the Template Method) forms the basis of plug-ins. This design style, common initially, has now fallen somewhat out of favor.

11.5.2.1 Inheritance Is One Narrow Form of Reuse

Inheritance was initially touted as the preferred object-oriented way to achieve reuse and flexibility. In the early days of Smalltalk (one of the earliest popular OO programming languages), several papers were written promoting "programming by adaptation." The principle was that you take someone else's code, make a subclass of it, and override whichever methods you require to work differently. Given, for example, a class that

2. Some circularities in type dependencies will not work with C++ templates.

implements Invoices, you could define a subclass to implement BankAccounts: Both are lists of figures with a total at the end.

Although the code runs OK, this wouldn't be considered good design. The crunch comes when your users want to update their notion of what an Invoice is. Because a BankAccount is a different thing, it's unlikely that they'll want to change that at the same time or in the same way or that the overrides retain the behavior expected of an Invoice. It then takes more effort to separate the two pieces of code after the change, losing any savings you gained in the first place.

The programmer who uses inheritance in this way has forgotten the cut-and-paste keys: They provide the proper way to start a design that borrows ideas from another one. If the concepts are unrelated, then the code should also be unrelated.

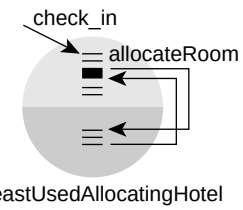
> *Do not inherit code unless you also intend to inherit its specification, because the internal implementation itself is always subject to change without notice.*

11.5.2.2 Inheritance Does Not Scale for Multiple Variants

What else might inheritance be good for? Perhaps multiple variants of a basic class. Consider a hotel booking system. When a guest checks in, the system does various operations, including allocating a room. Different hotels allocate their rooms using different strategies: Some of them always choose the free room nearest the front desk; others allocate in a circular way to ensure that no room is used more than another; and so on.

So we have several subclasses of Hotel, one for each room-allocation strategy. Each subclass overrides allocateRoom( ) in its own way. The main checking-in function delegates to the subclass.

```
class Hotel
{    public void check_in (Guest g)
     { ... this.allocateRoom (g); ...}
     protected abstract Room allocateRoom (Guest g);
}
class LeastUsedAllocatingHotel extends Hotel
{
     public Room allocateRoom (Guest g) {....}
```



But the problem is that it is difficult to apply this pattern more than once: If Hotels can have different staff-paying policies, does that mean we must have a different subclass for each combination of room allocation and staff payment? That would not scale very well even if you used multiple inheritance.

11.5.3 Polymorphism and Forwarding

The solution is to forward these tasks to separate specialist *strategy* objects that implement different policies behind a common interface [Gamma95]; this is the essence of good polymorphic design.

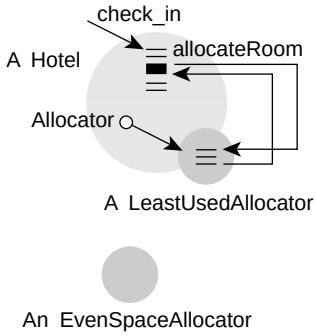
```
class Hotel {
```

```
        Allocator allocator;
        public void checkInGuest (Guest g)
        {... allocator.doAllocation(g); ...}
}

interface Allocator {
        Room doAllocation (...); // returns a free room
}

class LeastUsedAllocator implements Allocator {
        Room doAllocation (...) {...code ...}
}

class EvenSpaceAllocator implements Allocator {
        Room doAllocation (...) {...code ...}
}
```



Each Hotel object is coupled to a room allocator object, to which it delegates decisions about allocating rooms. Separately, it is coupled to a staff payer, and the same is true for whatever other variant policies there may be. Different policies are implemented by different classes, which may be completely different in their internal structure. The only requirement is that all room allocator classes must implement the doAllocation( ) message—that is, they must conform to a single interface specification.

This polymorphic coupling between objects is far more important as a design principle than inheritance is. It is what enables us to link one component to many others and thereby to build a great variety of systems from a well-chosen set of components. Both component-based and "pure" object-oriented approaches can take good advantage of this delegation-based approach via interfaces.

Let's return to the BankAccount and Invoice example: If there is any common aspect to the two things, the proper approach is to separate it into a class of its own. A list of figures that can be added might be the answer; so whereas BankAccount and Invoice are separate classes, both of them can use ListOfFigures.

11.5.4 Good Uses for Inheritance

Is there any good use for inheritance? Extremists would say we can do without it—and write good object-oriented software—provided that we have the means (a) to document and check interface implementation and (b) to delegate efficiently to another object without writing very much explicit forwarding code. All object-oriented programming languages support these techniques, although some do so better than others. Java, for example, has good support for interfaces, whereas C++ mixes implementation and interface. Smalltalk has support for inheritance but no type-checking. Few languages properly support delegation; it can be done in Smalltalk, and Java gets halfway there with its inner classes. Perhaps the next fashionable successor to Java will have explicit support for delegation.

More pragmatically, class inheritance has its place and value but should not be used when delegation via a polymorphic interface would work. It's reasonable to inherit from an abstract class, which provides an incomplete or skeletal implementation, and then extend it to plug in bits specific to your need. Inheritance with arbitrary overriding of methods is not advisable.

11.5.5 A Good Combination

One good way to combine these techniques is as follows.

- For every role, define an interface:

```
interface IResource { .... }
```

- For every interface, define a default implementation with inheritance plug-points:

```
abstract class CResource implements IResource {
    protected abstract plugIn ();
    public m () { ..... plugIn(); ...}
}
```

- Each default implementation should itself delegate to other *interfaces*:

```
abstract class CResource implements IResource {
    private IJob myJob;
}
```

- Concrete classes typically inherit from the default implementation, but they could also independently implement the required interface.
- Use a factory to localize the creation of new objects of the appropriate subclasses:

```
class ResourceFactory {
    IResource newResource () {
        return new CResource;
    }
}
```

In that way you can make a local change to the factory and have entirely new kinds of resources be created and used polymorphically.

11.5.6 Replacing Inheritance with Type-Based Composition

Any inheritance structure can be replaced by a more flexible and late-bound composition structure provided that you don't use language primitives for checking object identity.[3] The key idea is to treat the different portions of an instance of a subclass as though they were separate objects that are composed; explicitly forward calls for inherited *up calls* as

3. We hope that the next generation of component-aware languages can make an assembly of objects appear as one, with intelligent query of object identity and interfaces.

well as template-method *down calls*; and define explicit types to describe the call pattern between super- and subportions.

Figure 11.12 shows an example. For external clients, class A implements a type TA with its two public methods. The class A implements this type but expects to have the method Z() implemented by B; hence, the type of B as required by this implementation of A would be TBA. Similarly, the implementation of B provides the operation Q in addition to X and Y and expects the type TAB from its "super" portion. This results in four distinct types, which can be mapped directly to the two implementation classes.

When an instance of B is created, it must be passed an object that implements TAB. The code for B includes a reference to this object. Similarly, when an instance of A is created, it must be passed an instance that implements TBA. If the initialized references are thereafter frozen, the effect is very similar to inheritance except through a more robust and documented black-box type interface (see next page).

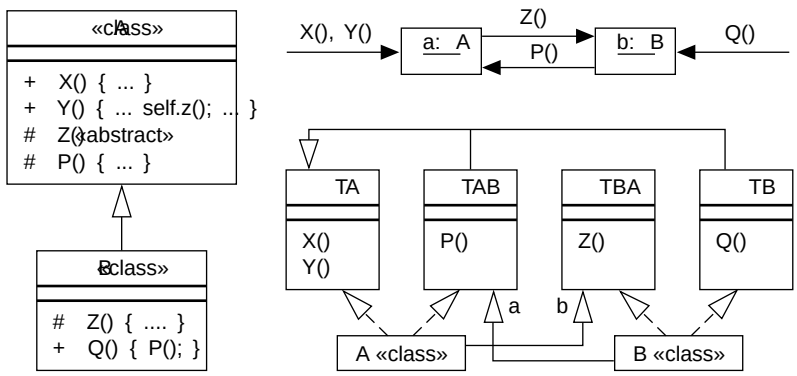


Figure 11.12 From inheritance to type-based composition.

```
class B implements    TB, TBA {
    private TAB a;    // the "super" part

    // interface for general clients: TB
    public X()    { a.X(); }
    public Y()    { a.Y(); }
    public Q()    { a.P(); }

    // interface for the "template-method" calls from A: TBA
    public Z()    { .... }

    // constructor: accepts an instance that implements TAB; or can create one itself
    B (TAB super)    { a = super; }
}

class A implements    TA, TAB {
    private TBA b;    // the "sub" part

    // interface for general clients: TA
    public X()    { .... }
    public Y()    { b.Z(); }

    // interface for the "up-calls" calls from B: TAB
    public P()    { .... }

    // constructor: accepts an instance that implements TBA;
    // circularity may need an additional "setter" method
    B (TBA sub)       { b = sub; }
    public setB (TBA sub) { b = sub; }
}
```

Although this structure may seem heavy for everyday use, keep it in mind as a possible transformation. You may use some variant of it to replace implementation inheritance with black-box reuse across component boundaries.

11.5.7 Specifying the Super/Sub Interface

One of the problems with inheritance frameworks is that they are white-box in nature: The extender of a framework must study and understand the source code in order to make extensions and to understand which methods to override, which other (template) methods that override will affect, and how the set of overridden methods must behave for the framework to function properly. The real reason for this legacy is that the "vertical" interfaces of class frameworks have rarely been documented explicitly, abstracting away from the code.

We have explained at length how type-based specification of interfaces lets us specify accurately, and yet precisely, the behavior expected of an object; and you have just seen how any inheritance structure can be analyzed, and even coded, using a type-based com-

position structure. Even if you do not change the implementation to composition, you can now document the super/sub interfaces using all the tools of type specification.

11.5.8 Component- and Connector-Based Pluggability

Let's not forget that components and connectors (see Chapter 10) provide yet another level of abstraction in building and plugging together components. Different, possibly customized, forms of connectors can make it much simpler to describe and implement the component configurations you need.

11.6 *Summary*

This chapter deals in some detail with the business of building reusable code components that are pluggable.

Reuse of software is not simply a matter of cut-and-paste; it should involve the reuse of interface specifications before implementation code is reused. Successful reuse poses many organizational challenges (culture, development processes, and so on) as well as technical ones (designing components that are adaptable to many different contexts and devising techniques for plugging in the adaptations).

The framework approach to code reuse provides a concrete, yet incomplete, implementation of the architecture: The rules and policies about how application objects interact are codified and enforced by the framework itself. Frameworks can be both white-box—a template method in the superclass must be overriden by a subclass after understanding the calls made in the superclass implementation—and black-box, in which interfaces for the plug-in calls are explicitly specified and implemented according to the spec.

The frameworks approach also works at the level of problem domain models. The ideal approach is to formulate requirements themselves in a modular fashion by using model frameworks and plugging in the specifics for your problem; implementing a code framework solution to the generic problem specification; and specializing that implementation framework to construct your system. A typical system consists of numerous such code frameworks and demands careful use of component-based techniques—such as event protocols across frameworks—so that the parts work together correctly.

The basic idea of plugging in to a code framework shows up in different ways in different languages, including C++ templates, component/connector technologies, and class hierarchies. The latter tend to be overused; it is often better to replace the inheritance with composition and explicit forwarding and to use types to document the subtle call patterns between superclasses and subclasses.

Pattern 11.1 Role Delegation

Adopt a uniform implementation architecture based on composition of separate role objects to allow plugging together of code components.

Intent

The idea is to compose separately implemented objects for different roles.

Objects play several roles, each of which may have several variants. We don't want a separate class to implement every combination of all the variants—for example, a Person can be a Full-time or a Part-time Employee; a Natural, Foster, or Step-Parent; and so on. The set of roles (and the choice of variant) may change at run time. We need to change the type without losing the object's identity.

Combining two specifications is easy: You just and them together (that is, you tell the designer to observe both sets of requirements). You can't do that with code, so we look for a standard mechanism for cooking up an object by systematically combining roles from several collaborations. This approach enables us to stick with the big idea that design units are often collaborations (and not objects) but retain the convenience of plugging implemented pieces together like dominoes.

Strategy

The technique is to delegate each of the role-specific pieces of behavior to a separate object (see Figure 11.13). One conceptual object is then implemented by several: one for each role, and (usually) a *principal* to hold them all together. The principal object keeps those parts of the state to which access is shared among the roles. Each role conducts all dialog with the other participants in the collaboration from which it arises. Generally, the roles are designed as observers of various pieces of the principal's state.

Make the group behave to the outside world as a single object, which was the original intent, by always keeping them in sync and being careful with identity checks. You must design an interface for all plug-ins to the same principal so that new plug-ins can be added for new roles. Never use a language-defined identity check (== in Java). Instead, have a sameAs(x) query; plug-ins pretend they're all the same object if they share the same principal. Calls to self within plug-ins usually go to the principal.

For example, the basic trading principal has a stock of products and cash assets. Into this can be plugged a role for retailing that knows about a Distributor and monitors the stock level, generating orders when necessary. Or we could make it a Distributor, plugging in the appropriate role; perhaps a Dealer would be something with both the Retailer and the Distributor roles.

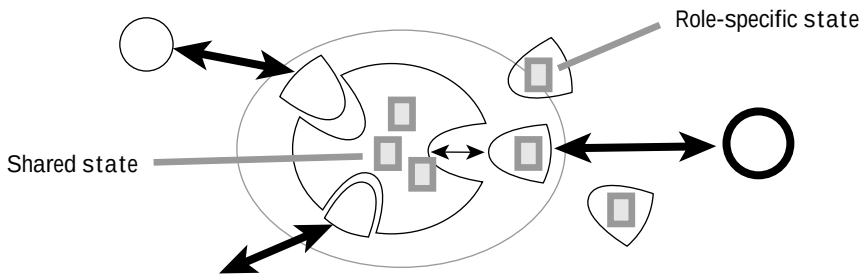


Figure 11.13 Building objects by connecting role objects.

Pattern 11.2 Pluggable Roles

Make role objects share state via observation of a shared object.

Intent

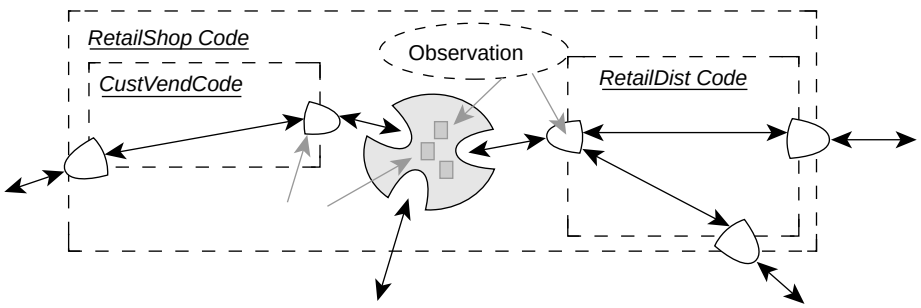
We need to supply complete implementations of frameworks, but frameworks are often about collaborations among roles rather than complete object behaviors.

Strategy

- Implement components as collaborations between role plug-ins. Each role implements the responsibilities of its framework spec, and each role is an observer of the shared state.
- Ensure a common interface for plug-ins. To build new collaborations, designers couple principals to collaborations.

Roles Observe the Shared State. So that a fully coded component can mimic the structure of the corresponding specification frameworks, each role should incorporate the code necessary for implementing placeholder actions. Most placeholder triggers boil down to monitoring changes of state. Each role can therefore be built as an observer of the parts of the common state that it is interested in.

The principal provides a standard pluggable interface allowing each role to register its interests and makes each shareable attribute a potential subject.



Collaborating Components Mirror Framework Specs. After building a specification by composing framework models, you can implement it by plugging together the corresponding fully implemented collaborations (if they are available).

This scheme could exact a performance penalty compared with purpose-built systems. There is overhead in the wiring of the observers wherever components are plugged together, although more-efficient versions of Observation, such as the JavaBeans event model, may adequately address this. In exchange for performance, you get rapid development; and you always have the option of designing an optimized version by working from the composed framework specifications.