Chapter 10 Components and Connectors

Many software managers, harried by budgets and delays, envy hardware designers. To design a steam engine, the engineers did not start by designing screws from scratch. Electronic systems are built by plugging together chips, boards, or boxes that are widely interoperable. A well-chosen set of components can have many possible configurations: many end products that can be made quickly and reliably.

Over the past few years, the same thing has begun to happen in software. Word processors can talk to spreadsheets, and graphs to databases. Standards such as COM and CORBA allow you to plug together components in different languages and platforms. JavaBeans, or any similar protocol, allows separately designed objects to find out more about each other's capabilities before negotiating a collaboration. Visual building tools help you plug components together pictorially.

Large-grained components are becoming a practical part of an enterprise component strategy. These components interact with one another as much as their smaller cousins do, and they must be analyzed and designed so that they interoperate as expected.

This chapter explains how to meet requirements using component-based designs and how to design components that work well together. After introducing component concepts, discussing pluggable parts, and describing how components have evolved over the years, we look briefly at three component standards: JavaBeans, COM+, and CORBA.

Rather than limit ourselves to a specific component technology, we then introduce the port-connector model of component architectures. We discuss a typical example of such an architecture and show how to specify and design with components in this architecture. Then we show how even ad hoc and heterogenous component systems are amenable to systematic development in Catalysis.

10.1 *Overview of Component-Based Development*

By itself, the use of object-oriented programming is not enough to get stupendous improvements in software delivery times, development costs, and quality. Some people

wonder why, having bought a C++ compiler, they're not seeing all the glorious benefits they've heard of.[1] But it doesn't work that way. The greatest benefits depend on good management of the software development process. The bag of techniques, languages, methods, and tools lumped under the "object-oriented" heading is an *enabling* technology: It makes it easier to achieve fast, cheap, robust development, but only if you use it properly.

If you want to achieve significant improvements in software productivity, one of the most important shifts is to stop writing applications from scratch every time you embark on a new project. Instead, you should build by using software components that already exist. The building blocks that you use for software development should not be limited to those offered by the programming language but should also include larger-grained, encapsulated units.

Over the past five years or so, we have seen this change happening. Many applications are now built on purchased third-party frameworks or by gluing together existing applications. Many programmers have come across Microsoft's OLE/COM (and before it, DLLs), which provides a way of bolting together entire applications. The OMG's CORBA provides similar facilities.

For example, an application that reads stock figures from a newsfeed can be "wired up" to a spreadsheet; this component does some calculations and passes the results to a database, from which a Web server extracts information on demand. Each of these components may be a stand-alone application, perhaps even with its own user interface, but is provided with a way to interact with other software.

Many development teams think only of gluing together large third-party components that can also work as stand-alones. But the spreadsheet in our example doesn't need a user interface: It is used only as a calculating engine within a larger chain. The example could be built more efficiently by using a calculating mechanism designed to be used as a component in a larger design and so lacking all the GUI overhead (perhaps with a suitable GUI as an optional add-on). And the persistence mechanism need not be a part of the spreadsheet itself. It could use a separate data-access component for that; again, it could include a default one.

Most development teams could benefit from thinking more in terms of building their own components for their application area. This is the key to fast, reliable development: to do it the way hardware designers have been doing it for two centuries and build components that can be assembled together in many combinations. Most end products—and indeed most components—should be assemblies of smaller components, built either elsewhere or in-house.

The aim must be to invest in the development of a component library as a capital asset (see Figure 10.1). Like any investment, this one requires money to be spent for a while before any payback is seen. A conventional software development organization requires a considerable shift of attitudes and strategy to adopt a component-based approach. Like all

1. And some of them then go around saying, "It doesn't work"!

big shifts, it must be introduced in easy stages, and you must plan carefully the risks, fallbacks, and evaluation of each phase.

A lot of marketing hype surrounds the terms *component* and *component-based development*. It includes radical pronouncements suggesting that object technology is dead and components are the next salvation. Separation of concerns, encapsulation, and pluggable parts continue to distinguish good, flexible designs from bad.

10.1.1 General Components

What is new about components? If they are reusable software pieces, how are they different from modules? If they are like objects, in what ways are they different? At the most basic level, components are parts that can be composed with others to build something bigger. Let's start with some basic definitions.

© ***component-based development (CBD)*** An approach to software development in which all artifacts—from executable code to interface specifications, architectures, and business models; and scaling from complete applications and systems down to small parts—can be built by assembling, adapting, and "wiring" together existing components into a variety of configurations.

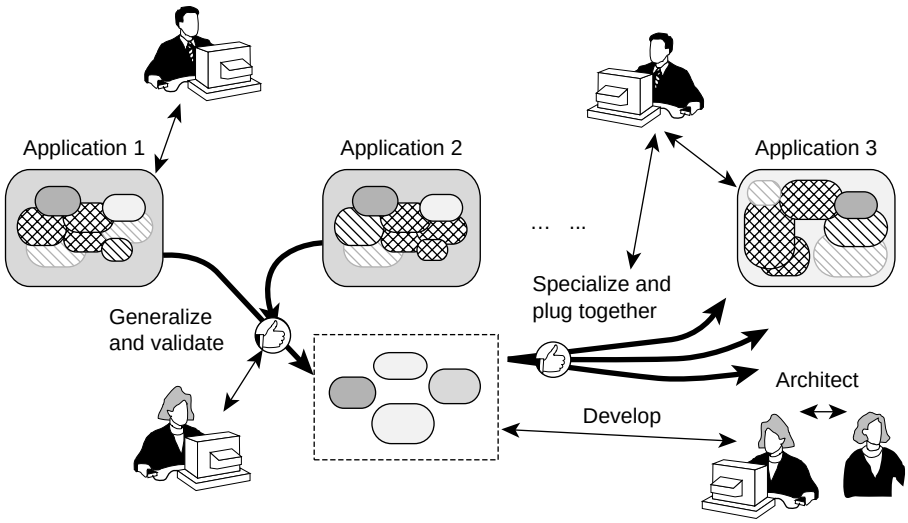


Figure 10.1 Component development and distribution.

© ***component (general)*** A coherent package of software artifacts that can be independently developed and delivered as a unit and that can be composed, unchanged, with other components to build something larger.

Based on this broad definition, all the following items conform to the general spirit of component-based development:

- Dropping a user-interface widget, such as a master-slave pair of list boxes, onto a canvas and connecting the lists to the appropriate data sources from the problem domain.
- Using the same C++ List template to implement any number of domain classes by specializing the template parameter:

```
class Order {
    private:  List<LineItem> items;
};
```

- Using an off-the-shelf calendar package, word processor, and spreadsheet in an assemblage of several heterogenous components and writing scripts so that these components together fulfill a particular business need.
- Using a class framework, such as Java's Swing components, to build the user interfaces for many applications and to connect those UIs to their domain objects.
- Using a model framework, such as resource allocation, to model problems ranging from seminar room allocation to scheduling machine time for production lots in a factory.
- Using predefined language constructs in an infinite variety of contexts:

```
for ( ...; ...; ... ) { ... }
```

A component can include anything that a package can include: executable code, source code, designs, specifications, tests, documentation, and so on. In Chapter 9, Model Frameworks and Template Packages, we show how the idea of composing software based on interfaces also applies to designs and models, leading to a more general idea of component-based modeling: All work is done by adapting and composing existing pieces. To make effective use of implementation components, we should start with componentization of business models and requirements. In this more general sense, a Catalysis package constitutes a component.

10.1.2 Implementation Components

In this chapter we focus on implementation components: executable code, source code, interface specs, code templates, and the like. In this context, a component is similar to the well-known software engineering idea of a *module*, although we have standards in place and technology infrastructure that makes building distributed component systems a reality.

For one component to replace another, the replacement component need not work the same internally. However, the replacement component must provide at least the services that the environment expects of the original and must expect no more than the services the environment provides the original. The replacement must exhibit the same external behavior, including quality requirements such as performance and resource consumption.

© ***component (in code)*** A coherent package of software implementation that (a) can be independently developed and delivered, (b) has explicit and well-specified interfaces for the services it provides, (c) has explicit and well-specified interfaces for services it expects from others, and (d) can be composed with other components, perhaps customizing some of their properties, without modifying the components themselves.

10.1.2.1 A Unit of Packaging

A component is a "package" of software that includes implementation, with a specification of interfaces provided and required. The mechanics for this packaging differ. In some component technologies, such as JavaBeans and COM+, the compiled code for a component includes an explicit runtime representation of interfaces; you can programatically inquire about these interfaces and use the information to establish suitable connections between components. In other component technologies, the compiled implementation may be stripped of this extra information and reduced to a minimal executable or DLL; in this case, the packaging unit must include separate and explicit descriptions of interfaces provided and required. COM (prior to COM+) required a separate and explicit type library containing component interface information to be registered in the shared system registry.

A component package typically includes the following.

- *A list of provided interfaces:* These are often imported from other packages, containing only the specs of these interfaces.
- *A list of required interfaces:* These are often imported from separate packages, just like the provided interfaces.
- *The external specification:* This is a specification of external behavior provided and required, relating all the interfaces in a shared model.
- *The executable code:* If built according to a suitable and consistent architecture, this can be coupled to the code of other components via their interfaces.
- *The validation code:* This is code, for example, that is used to help decide whether a proposed connection between components is OK.
- *The design:* This includes all the documents and source code associated with the work of satisfying the specification; it may be withheld from customers.

Components can also contain modifications to, and extensions of, existing classes. This is used often in Smalltalk—in which existing classes and methods can be dynamically modified and extended—and also is generally useful for any system that cannot be halted to install software upgrades. The interesting thing in terms of modeling is to provide mechanisms and rules for ensuring that the components do not interfere improperly with one another after they're installed in a system.

In some situations it can be useful to further distinguish the specification of a component, an executable that implements that spec, a particular installation of that executable, and a "running" incarnation of that executable that is available as a server. We will use the term *component* loosely to include all these. A tool or metamodel that dealt more fully with component deployment and management would separate them.

10.1.2.2 A Unit of Independent Delivery

Being independently deliverable means that a component is not delivered partially. Also, it must be specified and delivered in a form that is well separated from any other components it may interact with when deployed.

10.1.2.3 Explicit Provided and Required Interfaces

The only way one component can interact with another is via a provided interface. The specification of a component must explicitly describe all interfaces that a client can expect as well as the interfaces that must be provided by the environment into which the component is assembled.

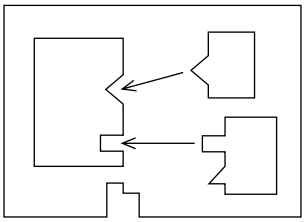
For components to be *plug-replaceable*, it is essential that the component specs be self-contained and symmetrical; it is the only way to be able to design reliably using parts without knowledge of their implementations. In contrast, a traditional *server*, in a client-server setting, is one-sided: It provides a set of services, but the services it expects from others are not documented explicitly. Object-oriented languages have also focused almost exclusively on services implemented by a class without any explicit description of the services it expects from other objects.

10.1.2.4 Complete Separation of Interfaces from Implementation

Component software demands a complete separation of interface specifications from implementations. The interface specs, rather than the source code, define what a component will provide and expect when used. In fact, if you are ever forced to use an existing component that does not have an explicitly documented interface, it is usually worth writing a specification of it "as perceived and used"; this document greatly simplifies testing and reduces the time to evaluate the suitability of new and alternative versions of the component.

10.1.2.5 Component Composition

The basic idea is that, to assemble a larger component or application, the composer of components (a) selects which components to compose; (b) connects required interfaces of one component to provided interfaces of others to plug them together; and (c) perhaps writes some *glue* using scripting or adapters between components. Composites often provide standard services to structure their child structure at runtime and to expose some of their children's services.



As with packaging, the exact form of component composition varies across different component technologies and tools. The key point is that the composition can be done by a different party, and at a different time, from the building of the components themselves. Section 10.7, Component Architecture, discusses a component-port-connector model for describing compositions in a simple way. Ports are the access points in a component where its services can be accessed or where it can access another's services; connectors couple ports. The kinds of compositions supported by a particular technology or style can be described by its connector kinds.

10.1.3 Components and Binding Times

Components are designed and built in one activity; they are composed with others in a separate step. When designing and building a component, you can relate it to the others it

will (eventually) be composed with only by their contractual interfaces. Actual implementations are selected at composition time or, sometimes, even at runtime. We usually want to delay the bindings made when components are composed so that the composition can be done as late as possible. One view of the range of binding times is shown in Table 10.1.

Exhibit 10.1 Binding Times for Composition

Binding Time	How It Works
Coding time	Straight-line code, without even procedural separation.
Compile time	The standard separation of procedures; all calls are bound to an implementation at compile time.
Link time	Separate compilation units, with interfaces declared separately from their implementations; calls are compiled against interfaces and are bound when the compiled units are linked into an executable.
Dynamic linking	Separate compilation units that do not need to be linked into a single executable up front; instead, a compiled unit (a DLL) can be dynamically linked into the running system. Calls are compiled against interfaces and are bound the first time the implementation is loaded.
Runtime	Calls are compiled using a level of indirection; this indirection is used to dynamically bind a call against an interface to a specific implementation at runtime based on the "receiving" object; this can be combined with dynamic linking (Java) or static linking (C++).
Reflective	Calls are not compiled against interfaces at compile time. Instead, the runtime keeps an explicit representation of interfaces offered; calls are issued dynamically against these at runtime based on "reflection" about the services and (of course) are resolved dynamically. Scripting services, in which components are coordinated by interpreted scripts, are best built on such a facility.

10.1.4 Objects Versus Components

There is confusion in popular writings about the similarity and differences between an object and a component. Some of this confusion stems from the fact that component technology is often best implemented using an object-oriented language; more fundamentally, it stems from loose usage of the terms *object*, *class*, and *component*.

10.1.4.1 Is an Object a Component?

Components are software artifacts and represent the work of software developers and their tools; objects are identifiable instances created in running systems by executing code that is a part of some component. So, in that strict sense, an object is not a component. It is the component code that is reused—the calendar package, perhaps with some customizable properties—rather than a specific calendar instance and its state.

That said, a running component is often manifested as a collection of objects and can be usefully treated as though it were one large-grained object, based on our approach to

refinement. So we sometimes use the term *component* a bit more loosely in this chapter to refer to the object or set of objects that manifest a particular usage of a component in an application.

© ***component instance*** The object, set of objects, or predetermined configuration for such a set of objects that is the runtime manifestation of a component when composed within a particular application.

10.1.4.2 Is a Class a Component?

Only if packaged to include explicit descriptions of the interfaces that it implements and the interfaces it expects from others. Consider the following Java class:

```
class C1
    implements I1, I2        -- interfaces this class implements
{
    public T0 foo (T1 x);
    private T2 y;
}
```

The minimal component that could contain C1 would also have to include the specifications of I1, I2, T0, T1, and T2. A package with a single class; the interfaces it implements (perhaps those interface specs are imported from another package); and the interfaces it requires of any other objects it deals with (input parameters, returned objects, factory objects it uses to instantiate other objects, and so on) would constitute a minimal OO component.

If class C1 inherited part of its implementation from another class, there would be a direct implementation dependency between the classes. Many people believe that implementation inheritance, although often useful, should not cross component boundaries. When the boundary must be crossed, it may be better to adopt a composition or delegation style approach (see Section 11.5.3, Polymorphism and Forwarding).

In general, a component could implement its interfaces by directly exposing them to clients or by implementing classes that provided the interfaces; to use the interfacer, clients would need to obtain a handle to an instance of such a class.

10.1.4.3 Component-Based Design versus OO Design

When you build a design from components, you don't need to know how they are represented as objects or as instances of a classes or know how the connectors between components work.[2] In federated systems, just as in OO programs, each component is a collection of software; it is chosen for the support it provides of the corresponding business function and uses local data representations best suited to the software. Just as in OO programs, objects must access the information held by other objects, so in a component architecture, components intercommunicate through well-defined interfaces so as to preserve mutual encapsulation.

The smaller-grained components in Section 10.6.1 and Section 10.6.2 are also very similar to objects. In fact, they would be easiest to implement as objects in an object-oriented programming language. This shows that the differences between component-based

design and object-oriented design are mainly of degree and scale and are not intrinsic to either type of design.

- Components often use persistent storage; although objects in an OO programming language always have local state, they typically work only within main memory, and persistence is dealt with separately.
- Components have a richer range of intercommunication mechanisms, such as events and workflows, rather than only the basic OO message. These mechanisms support easier composition of the parts.
- Components are often larger-grained than traditional objects and can be implemented as multiple objects of different classes. They often have complex actions at their interfaces, rather than single messages.
- A component package, by definition, includes definitions of the interfaces it provides as well as the interfaces it requires; a traditional single class definition focuses on the operations provided and not on the operations required.
- Objects tend to be dynamic; the number of customers, products, and orders you have, and their interconnections, changes dynamically. In contrast, larger-grained components may be more static; there will probably always be only one payment control and one finance component, and their configuration will be static.

Components, like objects, interact through polymorphic interfaces. All our modeling techniques apply equally well in both cases, including the more general connectors for components. Plus, we can usefully talk about a component instance, component type, and component class.

10.1.5 Components and Persistence

A component instance has state represented by its object(s) and is part of a larger component. Hence, its persistence needs to be in the context of its containing component. Java and COM provide differing versions of such protocols.

In simple cases, persistence can be achieved by a protocol by which a component instance serializes itself into a stream that is managed by its container. For larger, server-side components, each component can manage its own persistent storage and transactions; effective composition now requires that the container be able to coordinate nested transactions that cross its subcomponent boundaries. Enterprise JavaBeans, CORBA, Microsoft Transaction Server, and COM+ provide their own versions of this.

---

2. Components can also be built without explicitly using object-oriented design techniques at all; but OO makes it a lot easier. If you tried to build a component architecture without mentioning objects, much of the technology you'd use would amount to object orientation anyway. We've recently read a few reports proclaiming "Objects have failed to deliver! Components are the answer!" The authors either have a poor understanding of how componentware is built, or, being journalists and paid pundits, they enjoy a disconcerting headline. The reality is that OOP, like structured programming before it, has become part of the body of ideas that constitute good software engineering. Having learned how to do it, we can now move on to putting it to work.

10.2 *The Evolution of Components*

We should not assume that objects are intrinsic to component-based development; in fact, one of the advantages of components is that they can encapsulate legacy systems regardless of how the systems are implemented inside. The idea of components goes back almost as far as the idea of software, but a number of things have changed significantly over the years.

- The granularity of the components and the corresponding unit of pluggability: from monolithic systems, to client-server partitions, to the operating system and its services, to today's object-based component approaches.
- The effort and ease of dynamically connecting components to compose larger systems, from writing screen-scrapers for host-based systems (discussed in a moment) to creating complex applications by visually configuring and connecting server-side components to one another and to user interfaces.
- The overall effort and cost of creating applications: from monolithic custom-made systems to gigantic "package" solutions[3] to the assembly of smaller components built on standard infrastructure and interfaces.

The earliest mainframe-based systems were written as monolithic applications manipulating data shared across all the application procedures. Internal procedures could rarely be considered encapsulated; they typically operated on shared data, making composition at that level difficult and error-prone. The only visible interface was to an external dumb terminal, and the nature of the interface was primitive: paint to the terminal screen and read character commands from the keyboard.

Calling these host-based applications "components" is a stretch; composing them was painful. Because the only interface offered to the outside was to a dumb terminal, the only way to connect two such "components" was to write pieces of code called *screen-scrapers* and *terminal emulators*. These programs acted as dumb terminals to the host but interpreted the screen painting commands and generated character commands. The granularity of components was very large, connections could not be established dynamically, and the technology for connecting parts was primitive.[4]

At the time, it was possible to deliver complete applications only in the form of an executable. Moreover, the executable had to be prebuilt in a static manner and could be replaced or upgraded only as a single unit. Software libraries contained source code, which you used by including the text and compiling it with your own.

But software vendors aren't keen on letting people see their source code; they'd rather give you the executable and (if you insist) the spec. This meant providing ways in which

3. These are sometimes so inflexible that they either work wonderfully for your business or your business must adapt to fit what the package solution offers.

4. This is changing substantially, with mainframe applications reborn as *server-side components* using technologies such as Enterprise JavaBeans.

applications could communicate. In the early days, this meant that they passed files to one another, a slow process that required that every output and input from a program be converted to the form of external records. It lent itself to pipeline processing rather than dialog between programs.

This led to the development of the application program interface—which could be seen as a way in which a program could pretend to be an application's user using facilities standardized at the level of the operating system—and dynamic linking, which enabled code to be linked at runtime without further processing. Two distinct forms of large-grained components evolved from this.

The first form led to client-server-styled systems, with the client combining user-interface and application logic and communicating via SQL requests with a database server that dealt with persistence, transactions, security, and so on. All communication involved database processing requests in SQL, and clients did not communicate with one another (except indirectly through shared data on the server).

Finer-grained application components also started to interact, using operating system support. In the world of Windows, generic applications were built with APIs that enabled them to be interconnected and interact via the operating system, exchanging information through standardized data representation schemes. Thus, a spreadsheet application could communicate with a word processor and a stock feeder to produce a formatted financial report. In UNIX, we saw the emergence of the elegant, but limited, pipe-and-filter architecture.

The first APIs were sets of functions that an external component could invoke. If there was any notion of an object receiving the function calls, it was the entire running executable itself. But the most recent developments in this field have put the executable program into the background (see Figure 10.2). The objects are the spreadsheet cells, the paragraphs in the document, the points on the graph; the application software is only the context in which those objects execute. The component architecture determines what kinds of object interactions are allowed.

Clearly, the granularity of components became much finer with object technology. No longer was it only the spreadsheet interacting with the database; now it was the sheets and cells that were connected to the columns and rows in the database, generating paragraphs and tables in the word processor. These relatively dynamic objects connect to other objects, regardless of the applications within which they exist.

The virtual enterprise of the future is built with components and objects locating each other, connecting, and interacting on a standardized infrastructure. This happens for components of all granularity, from large server applications to fine-grained objects, across boundaries of language, processor, and even enterprise, and with binding times from completely static to highly dynamic.

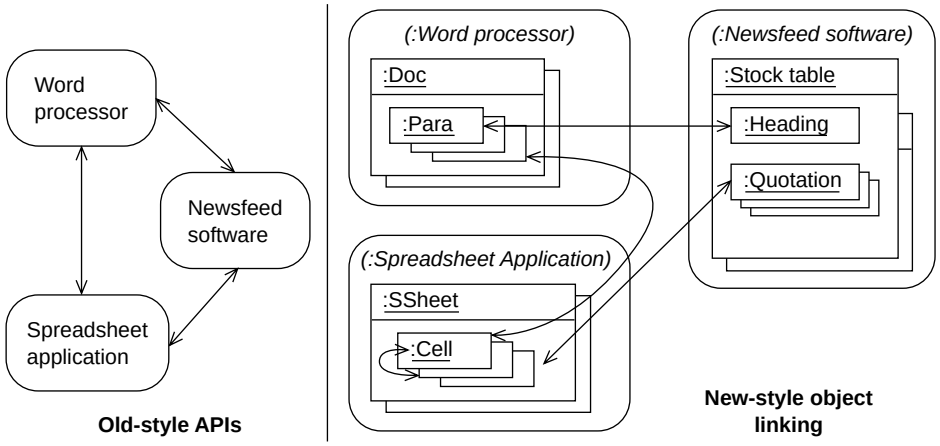


Figure 10.2 Old versus new styles of component interactions.

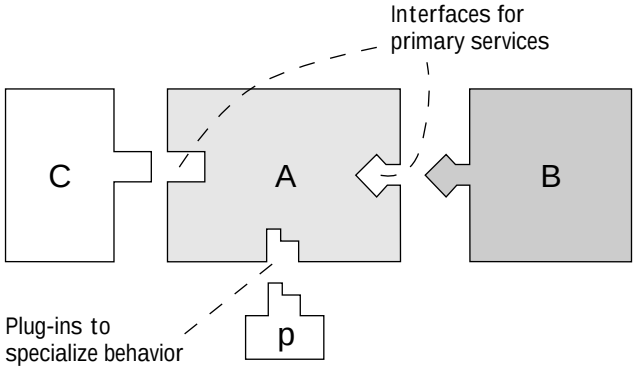


Figure 10.3 Primary and customization interfaces.

10.2.1 Components and Pluggable Reuse

Reuse comes in a wide variety of flavors ranging from cut-and-paste through complete application frameworks that can be customized. The component approach to reuse mandates that a component not be modified when it is connected to others; components should simply plug together, via defined interfaces for their services, to build larger components or systems. This makes it easier to replace or upgrade parts; if they support the same (or compatible) interface, one part can be replaced by another (see Figure 10.3).

The fact that a component should not be modified does not mean that it cannot be customized externally. A component can be designed to provide, in addition to the interfaces for its primary services, additional interfaces for plug-ins that customize the behaviors of its primary services; settable properties are a special case of this. This style of pluggable design is discussed in Chapter 11.

The precise meaning of a connection between components varies depending on the needs of the application and the underlying component technology. It could range from explicit invocation of functions via the connection to higher-level modes such as transfer of workflow objects, events being propagated implicitly, and so on. Section 10.7, Component Architecture, examines this in detail.

10.2.2 Components and Standardization

If we are to build systems by assembling components, we need a set of standards that are agreed to by component developers so that these components can interoperate and reduce the development burden of common tasks. Many of these issues would need to be addressed even without components, but the need for standardization would be less. There are three broad categories of standards: horizontal, vertical, and connectors (see Figure 10.4).

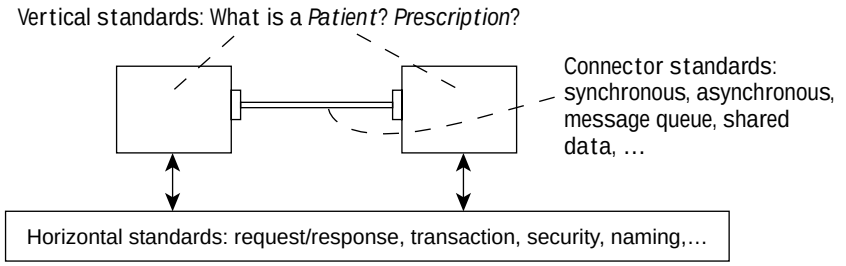


Figure 10.4 Horizontal, vertical, and connector standards across components.

10.2.2.1 Horizontal (Infrastructure) Standards

Components need a common mechanism for certain basic services, including the following.

• *Request broker:* This mechanism maintains information about the location of components and delivers requests and responses in a standard way.

• *Security:* This category includes mechanisms for authentication of users and authorization for performing various tasks.

• *Transactions:* Because each component potentially maintains its own persistent state information, business transactions must cross multiple components. A common mechanism is needed for coordinating such distributed transactions correctly. These transactions will be nested rather than flat.

• *Directory:* Many components need access to directory services—for example, to locate resources on a network or subcomponents within a component. They must be based on a common interface, with a uniform way to reference entities inside different components and across different naming schemes.

• *Interface repository:* Component interfaces and their specifications must be defined in a common way so that they can be understood both by people and by other components.

10.2.2.2 Vertical Standards

In addition to the underlying mechanisms being shared, components must agree on the definition of problem domain terms—usually manifested as problem domain objects—on which they will jointly operate. Components in a medical information system, for example, must share a common definition of what exactly a Patient is and what constitutes an Outpatient Treatment. They must share this definition at least in terms of the interfaces of those objects.

As of 1998, the OMG was actively working toward such *vertical* standards in domains that include telecommunications, insurance, finance, and medical care. Microsoft has a less coordinated effort under its DNA project. Every project must invariably also define its own domain-specific standards.

10.2.2.3 Connector Standards

We must also use standard kinds of *connectors* between components, defining a variety of interaction mechanisms for various kinds of components and compositions. The basic object-oriented message send is not the only, nor even the most suitable, way to describe all interactions.

• *Connectors that support explict call and return:* The call could be synchronous or asynchronous. Asynchronous messages could be queued until processed, and return values could be treated as *futures*[5] or callbacks.

• *Connectors with impicit event propagation:* Certain state changes in one component are implicitly propagated to all components that registered an interest in that event.

• *Connectors that directly support streams:* Producers insert values or objects into the stream, where they remain until consumed by other components.

• *Connectors that support workflow:* Objects are transferred between one component and the next, where this transfer is itself a significant event.

• *Connectors that support mobile code:* Rather than just receive and send data and references to objects, you can actually transmit an object, complete with the code that defines its behaviors.

10.2.3 Why the Move to Components?

Components and *component-based development* are rapidly becoming buzzwords; like those before them, they bring a mixture of hype and real technical promise. The main advantages of adopting a component-based approach to overall development are as follows.

5. An encapsulation of a ”promise” of a value; it may block if the value is accessed before being available.

- It permits reuse of implementation and related interfaces at medium granularity. A single domain object may not be a useful unit of reuse; a component—packaging together an implementation of services as it affects many domain objects—can be.
- Effective components also form the basic unit for maintenance and upgrading. There should no longer be a need to upgrade entire "systems"; instead, components get replaced or added as needed.
- Component partitioning enables parallel development. Identifying medium-grained chunks and focusing on early design of interfaces make it easier to develop and evolve parts in parallel.
- Interface-centric design gives scalable and extensible architectures. By letting each component have multiple interfaces, we reduce the dependency of any one component on irrelevant features of another component that it connects with. Also, adding new services incrementally can be accommodated more easily; you can introduce new components and add the relevant interfaces to existing ones. Scalability is somewhat more easily addressed, because replication, faster hardware, and so on can be targeted at a finer grain. Moreover, modern component technologies such as Enterprise JavaBeans move many of the burdens of resource pooling and scaling away from the business components to the containers within which they run.
- It lets you leverage standards. Because component technology implies a base set of standards for infrastructure services, a large application can leverage these standards and save considerable effort as a result.
- It can support capabilities that are impractical for "small" objects, such as (1) language-independent access of interfaces—so that you can use components written in other languages—and (2) transparent interaction between distributed components.

10.3 *Building Components with Java*

JavaBeans is the component technology for building components using Java. A JavaBean can be a single Java class whose external interfaces are described using the following.

- *Properties:* Object attributes that can be read and written by access methods
- *Methods:* Services with specified effects that a client can invoke
- *Events:* State changes that an object will notify its environment about, with no expectation of any resultant effect

Properties and methods represent services provided by the component. Events represent notifications from the component. There is no explicit way to represent services required by the Bean.

JavaBeans was designed to distinguish properties, methods, and events without any change to the basic Java language. It does this by using a facility called reflection.

10.3.1 Reflection

Java retains an explicit runtime representation of class, interface, and method definitions in its compiled class form; the *reflection* API is a facility for accessing this information. For every class that is loaded into a running system, Java instantiates a single instance of the predefined class Class. There is a static method for dynamically loading any class based on its name and several methods for examining the structure of a class definition.

```
class Class {
    static Class forName(String);        // load class by name, instantiate Class
    Class getSuperclass();               // return the superclass
    Class[] getInterfaces();             // return list of interfaces implemented
    Method[] getMethods();               // return list of methods
    Field[] getFields();                 // return list of stored fields
    Method[] getConstructors();          // return list of constructors
    Object newInstance();                // create a new instance
}
```

There are several other related classes, of which the most interesting one is Method.

```
class Method {
    Class getDeclaringClass();           // return home class
    String getName();                    // return method name
    Class getReturnType();               // return type
    Class[] getParameterTypes();         // list of parameter types
    Class[] getExceptionTypes();         // list of exception types
    void invoke(Object target);          // (very late bound) method invocation
}
```

The following example illustrates a runtime use of these facilities:

```
// locate and load the class dynamically
Class c = Class.forName ("UserDefinedClass");

// instantiate the class
Object o = c.newInstance ();

// get one of the methods on that class
Method m = c.getDeclaredMethod ("userMethod", { } );

// invoke that method on the new instance
m.invoke (o);
```

In addition to supporting the mechanisms needed by JavaBeans, as explained later, reflection enables very late binding of calls by dynamically looking up interfaces and methods and invoking them against objects of statically unknown types.

10.3.2 Basic JavaBeans

The simplest way to write a JavaBean is to program a single class, following certain naming patterns for the methods on your class. Using the reflection API on an instantiated

Bean (a process called *introspection*), a visual tool or other application can categorize the methods into properties, methods, and events.

- *Property:* Write a pair of methods named Y get<X>() and set<X>(Y) to define a property named X of type Y.
- *Event:* Write a pair of methods named add<X>Listener (XListener) and remove<X>Listener (XListener) and add the event signature to the operations that the XListener interface must support; this defines a single event your bean can publish to registered parties.
- *Method:* Write a method that follows neither a property nor an event pattern.

You can implement Beans without following the naming rules. You implement additional methods on the bean that (indirectly) explicitly identify the methods corresponding to properties, events, and methods on that Bean. We omit the details here.

10.3.3 Improved Components with JavaBeans

Attempting to program a Bean as a single class is not practical for nontrivial components; a component instance would typically consist of several connected objects of different classes, each implementing some of the external component interfaces. The more recent specifications for JavaBeans make it simpler to build complex Beans from several classes.

- Do not use language primitive "casts" to access other interfaces of a component, because they would understand only language-level objects. Instead, there is a prescribed explicit query protocol for getting to other interfaces.
- Bean instances will be nested. The containing "context" may (1) provide standard containment services and (2) interpose its own behavior before its parts execute their methods. Standard interfaces are defined for this purpose.

10.3.4 Persistence

Java provides a light-weight serialization mechanism; the implementor need do no additional work for objects to serialize and restore themselves correctly. The mechanism uses the underlying reflection services to implement generic save and restore functionality only once.

10.3.5 Packaging Using JAR Files

Compiled Java components are packaged into JAR files and include the class files that implement the component services, additional class files (if any) for explicitly defined properties, methods, and events, and some additional information.

10.3.6 Enterprise JavaBeans

Server components typically implement significant business functions and run on a server. In a multitier architecture, most business logic in an application runs on dedicated servers

rather than on the client machine. In general, a multitier design increases the application's scalability, performance, and reliability—because components can be replicated and distributed across many machines—but at the cost of some "middleware" complexity. Java's Enterprise JavaBeans (EJB) standard is a server component model that simplifies the process of moving business logic to the server by implementing a set of automatic services to manage the component.

The Enterprise JavaBeans model lets you implement business functions as JavaBeans and then plug them in to a standard *container* that provides automatic management of resources and contention from multiple threads, transaction programming based on two-phase commit across multiple independent components, and distributed programming. An EJB component, packaged into a JAR file, has four main parts.

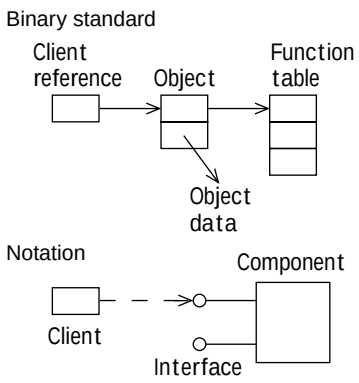
- *Home interface:* The client interface to a factory object that instantiates the main server Bean. A client locates this factory using a standard directory-based name lookup.
- *Remote interface:* The interface to the primary server Bean itself, providing the business operations to the client.
- *EJB class:* The class that implements the business operations.
- *Deployment descriptor:* A description of the preceding parts and additional attributes such as transactional and security behaviors that can be decided at deployment time rather than in the code for the main server component itself.

When the component, packed into its JAR file, is deployed, you must designate a

- *Container:* A component implemented by a middleware vendor; it acts as a container for your server component. It provides exposure of your services to clients (who actually do not directly access your implementation), remote access, distributed transaction management, security (including authentication and authorization), resource pooling, concurrent service for multiple clients, clustering, and high availability. In short, a container gives you all the things that, if they were absent, would make implementing a multitier system a nightmare.

Java presents a compelling technical base for component-based development, including enterprise-scale server components. Its main drawback is the single-language model, which is addressed by the integration of Java and CORBA.

10.4 *Components with COM+*



As it does almost everyplace else, Microsoft proceeds to define its own set of standards in the area of components. The foundation of its component software starts with COM: a binary standard for interfaces. A client reference to any object via an interface is represented by a pointer to a node that refers to a table of interface functions. If the reference is to an object with state, the intermediate node will also contain or refer to that object's state representation. A component offers some number of interfaces, each referred to by the client in this way. The implementation could involve many objects and classes or could involve none.

Every interface supports QueryInterface, a common function that queries for other interfaces based on unique identifiers assigned to interfaces. Each interface is immutable once published; a new version is a new interface. Because references to an object via different interfaces are physically different pointers, determining whether two references refer to the same object is not direct. Instead, COM prescribes that each component possess a single distinguished interface called IUnknown, which reliably serves as the identity.

COM is an interface standard and not a programming language; hence, it does not prescribe specific mechanisms for implementation reuse, such as class inheritance. However, it offers two mechanisms for composition.

- *Containment:* In this simple and straightforward approach, a container object receives every client request and explicitly forwards all requests as needed to its child objects.
- *Aggregation:* A container can directly expose references to (interfaces of) its inner objects; a client can then directly invoke operations on it. To behave like a single object, each inner object delegates all its QueryInterface requests to its container.

New objects are instantiated by a library call to CoCreateInstance, with a unique identifier for a particular implementation to be instantiated and the identifier of the interface of the new object that should be returned. The appropriate *server* is identified (from those registered in the system registry), started, and requested (via a factory) to create a new object; it returns an interface reference to the client.

COM interfaces are defined in an interface description language called IDL.[6] These interfaces can be compiled to produce *type libraries*—the runtime representation of the structure of interfaces and methods—and to produce appropriate proxy, stub, and marshaling code for the case of remote object references.

6. It is different from the OMG's IDL.

COM does not have the equivalent of Java's reflection, relying instead on the type library. Consequently, scripting and other applications that require very late binding—in which even the method called is not compiled against an interface but is looked up at runtime—require explicit support in the component itself. Each component can support what are called *dispatch interfaces*, in which a client requests an operation by a number; the component resolves the mapping from numbers to methods to invoke. COM uses *outgoing interfaces* to define events, just as JavaBeans uses its events.

COM includes a model for compound documents called OLE (a collection of standard COM interfaces) and includes ActiveX controls, another set of COM interface standards that include outgoing interfaces to permit events to be signaled by a control to its container. ActiveX controls also have properties that are similar to JavaBeans properties.

COM+, a relatively recent entry in the furiously renamed space of Microsoft's component technologies, has serious technology merit. It essentially defines a virtual machine model for components, similar in many respects to the Java virtual machine. COM+ provides garbage collection (eliminating the pesky reference counting approach of COM), extensive metadata (permitting reflection and eliminating dispatch interfaces), and infrastructure services (security and transactions) for server-side components. One of its more interesting features is called *interception*: the ability for the virtual machine to intercept requests sent to a component and interject special processing. This feature could be used to provide late-bound cross-component services, much as Enterprise JavaBeans uses the container as an intermediary to access component services.

10.5 *Components with CORBA*

CORBA (Common Object Request Broker Architecture) was designed by the Object Management Group (OMG) to support open distributed communication between objects across a wide variety of platforms and languages. Interestingly, despite the "Object" in its name, CORBA does not directly expose the notion of object identity; it could more properly be considered a distributed component framework.

To meet its goals of heterogenous computing, CORBA opted to become a source code standard rather than a binary one. Component interfaces are defined within modules using the CORBA IDL; different programming languages have standardized bindings to the IDL. Programmers either (a) manually write IDL and then compile it into the source code versions needed to write their implementations or (b) use a vendor's programming language compiler that offers direct generation of IDL.

The OMG's Object Management Architecture looks somewhat like the drawing in Figure 10.5. The architecture comprises four parts:

- *CORBA bus:* The base level of IDL-based interface definitions, the interface and server repositories, and the request broker
- *CORBA Services:* A variety of largely infrastructure services ranging from events to transactions, relationship, naming, life cycle, licensing, and externalization

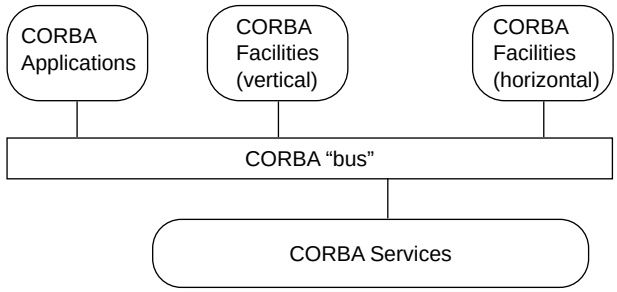


Figure 10.5 The CORBA architecture.

- *CORBA Facilities (horizontal):* Printing, e-mail, compound documents, structured storage, workflow, and so on
- *CORBA Facilities (vertical):* Standards for business objects in "vertical" domains, including health care, telecomm, financials, and so on

CORBA recently defined mappings for the Java language and aligned closely with JavaBeans and Enterprise JavaBeans for its component model. In fact, the Java Transaction Service is defined based on the CORBA model.

10.6 *Component Kit: Pluggable Components Library*

This section is about component *kits*: collections of components that are designed to work with one another. The contents of a kit need not be completely fixed; you can add to it and have various accessory kits and subkits of pieces that work particularly closely together. But a kit has a unifying set of principles—the kit's component architecture type—that makes it easier to plug together the members of a kit successfully compared with components built separately or chosen from different kits. Plugging arbitrary components together usually requires that you build some sort of glue. We deal with that in Section 10.11, Heterogenous Components.

We'll begin our discussion of component kits with an example to illustrate the basic principles and later show how they apply to larger-scale (and more business-oriented) components. These examples use various kinds of connectors between component instantiations, coupling service requirement points (ports) in one to service provision points (ports) in another. We use arrows and beads to represent connectors: $\longrightarrow\!\!\circ\!\!\longrightarrow$. Section 10.8.2, Defining the Architecture Type, shows how varying kinds of connectors can be defined.

10.6.1 Graphical User Interface Kit of Components

GUIs form the most widely used kits of components. Windows, scrollbars, buttons, text fields, and so on can be put together in many combinations and coupled to your database,

your Web server, or some other application. You rarely need to program the software that sets up and builds the forms: Instead, you use a GUI wizard to design them directly.

Using the connector notation discussed earlier, a typical design might look like the configuration of component instances shown in Figure 10.6. The connectors represent couplings between properties of two components (⟶; a pair of values is kept continually in sync) or events of two components (⟹; a published occurrence from one component triggers a method of another component).

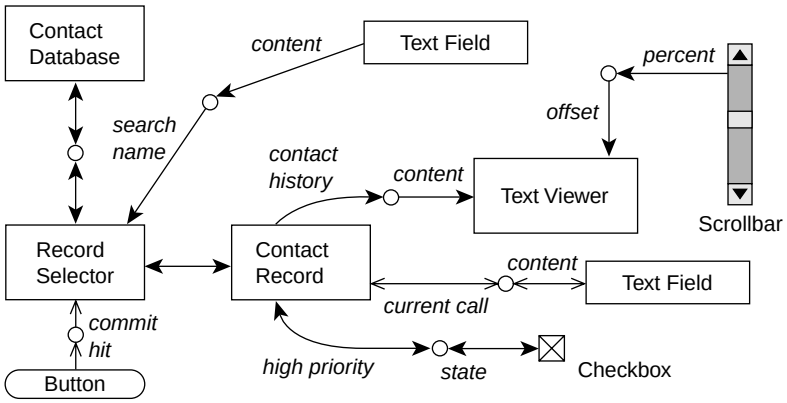


Figure 10.6 GUI component kit.

There is some need for adapters between, for example, the content of the Text Viewer and the contact history of the Contact Record. The design calls for continuously updated properties, such as the scrollbar's connector to the Text Viewer, as well as events such as the Button's hits. Some of the connectors are bidirectional: The checkbox both sets the priority and immediately shows a change in the property.

10.6.2 Kit of Small Components

Suppose you are given a collection of pieces of hardware: a motor of some sort, a couple of push-button switches, and a meter. Suppose the motor has a few wires attached: one labeled "start," another "stop," and a third tagged "speed «output»." The buttons and meter also have labeled connections. Now imagine connecting the components as shown in Figure 10.7.

Of course, a push-button can be used for many other purposes; if you had a lamp, you might use the push-button to switch it on and off, and your Meter might be used to display a temperature. There might be other ways of controlling the motor and other ways of using its speed to control other things. Let's root around in the box of parts and do some creative wiring (see Figure 10.8).

When the motor is running slowly, the meter doesn't show the low speed very clearly, so we've decided to multiply the speed by 10 or 100 before it gets to the meter. From the

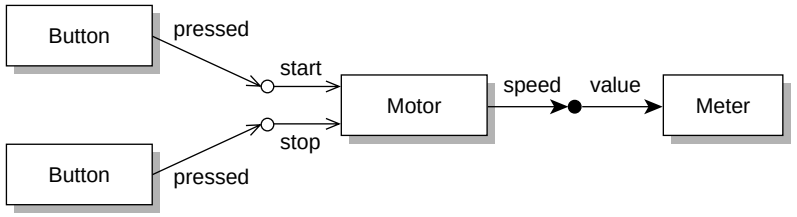


Figure 10.7 Electronic hobbyist component kit.

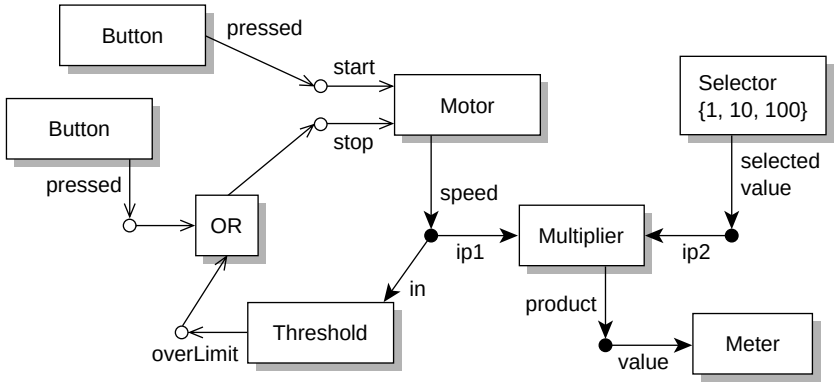


Figure 10.8 Electronics kit: a nontrivial configuration.

box of components we've pulled a Multiplier and a Selector. A Selector is a user-interface widget that provides a fixed choice of values, which in this case we've set to the factors we want to allow.

We're also worried that the Motor might run too fast sometimes (perhaps if its load is removed), so we've pulled out a Threshold.[7] It converts a continuously varying value, such as the speed, into a Boolean off/on, switching on when its input rises above a certain limit. Then we've connected it and the stop button (through an OR gate) to the Motor's stop input. So the Motor will be stopped either by the button being pushed or the Motor running too fast.

Any model railroad enthusiast will recognize this as a neat kit of parts with which you could build a lot of different projects—many more potential products than the number of components in the box. Note the ease of modifying the first version to realize the second. It's not difficult to imagine such a kit in hardware nor to visualize it in software. The Motor could be a software component controlling a hardware motor; the Buttons, Meter, and Selector could be user-interface widgets; and the other components could be objects not directly visible to the user.

7. Hardware people call it a Schmitt trigger.

10.6.3 Large Components

Components need not be little things; they can be entire applications or legacy systems. The nature of the connectors between these components differs. The components shown in Figure 10.9 are the support systems for some of the departments in a large manufacturing company.

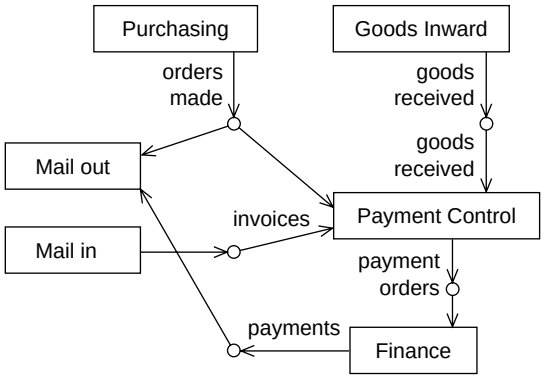


Figure 10.9 Large business components.

The components and their lines of communication mirror those of the business. The diagram could represent a business structure of departments and flows of work, or a software structure of components and connectors. Just as departments are composed of teams and people, so a closer look at any one of these components would reveal that it is built from smaller ones.

The company places orders, which are mailed and forwarded to payment control for subsequent payment. When the shipping department receives goods, it forwards a notification to payment control. Invoices received are also forwarded. When an invoice is received for goods that were delivered, a payment is generated.

Whereas the smaller example sent primitive values across the connectors, these components send larger objects, such as orders, invoices, and customer information, to each other. The connectors between these components serve to transfer objects, in one case with a duplication or split to two or more destination components.

The configuration is a modern one. Rather than a single central database with clients all over the enterprise, there are separate components, each holding appropriate data and performing appropriate operations tailored to support the business operations. There are several benefits of this *federated* scheme.

• *Flexibility:* It is more extensible and flexible than centralized systems clustered around one database. Business reorganization is rapidly reflected in the support systems.

• *Scalability:* It is more scalable to serve more people and buy more machinery. You are not constrained by having a single server that will scale only so far.

• *Graceful degradation:* Each business function is supported by its own machinery, so one malfunction doesn't stop the whole enterprise.

• *Upgradability:* You do not have to set up and subsequently update the system as a single entity. As the business grows and is reorganized, new software can be added. You can plug in commercial off-the-shelf components rather than build every enhancement into a single program.

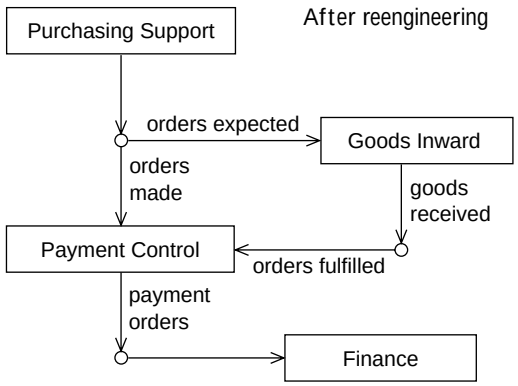


Figure 10.10 Reconfigured business components.

• *Appropriateness:* Because it requires less central control, it is less prone to local political and bureaucratic difficulties. There is no one authority that must agree to every change or must be persuaded to address the evolving requirements of every department. Instead, the business users hold much more responsibility for providing support appropriate for themselves.

An inefficiency was diagnosed in the process; goods were being delivered and paid for that had never been ordered, and reconciling purchase orders with goods received was becoming expensive. The company reorganized its departmental roles and the software to match (see Figure 10.10). Goods Inward now has records of all orders that have been made and will turn away spoof deliveries.[8] Because the component structure reflected the business structure at the level of granularity of the change required, the change was accommodated fairly well. Finer-grained business changes, such as the introduction of a new pricing plan for existing products, require a corresponding finer-grained component (or object) structure to accommodate the change.

10.6.4 Component Building Tools

Component technology is often associated with visual building tools. Once a systematic method of connecting components has been established, tools can be devised that let you

8. Thanks to Clive Mabey, Michael Mills, and Richard Veryard for this example.

to plug the components together graphically. Digitalk's Parts and IBM's Visual Age were early examples; Symantec's Visual Café works in Java, as does Sun's Java Workshop. Similar-sounding *visual programming* tools are restricted to building user interfaces rather than composing general components. There are tools (for example, Forté) that specialize in distributed architectures, in which the components may be executing on different machines. Others are good at defining workflow systems in which components of one kind—work objects—are passed for multiple stages of processing between components of another kind, the work performers.

10.7 *Component Architecture*

An *architecture* is an abstraction of a system that describes the design structures and relationships, governing rules, or principles that are (or could be) used across many designs. Here are examples:

> *"Four-tier with Web servelets."*
>
> *"We shall all write in Java."*
>
> *"Here's how we make one property observe another."*
>
> *"Use these interfaces and protocols to implement a spell-checking feature."*
>
> *"Use Fred's class whenever you want to wongle foobits."*
>
> *"Never use return codes to signal exceptions."*

The architecture is what lends the coherence and consistency to the design.

An architecture broadly comprises two parts (we defer a more detailed discussion to Chapter 12, Architecture).

- The generic design elements or patterns that are used in the architecture, such as subject-observer, Fred's class, the event-property connectors, or the generic design for spell checking.
- The rules or guidelines that determine where and how these architectural elements are applied, such as "For any composite user-interface panel that may be reused, make all internal events available via the composite." In the extreme case, these rules can be formalized to fully define a translation scheme.

Design is about relating several independent pieces together and claiming, "This particular way of combining these pieces will make something that does so-and-so."

The pieces might be the result of applying architectural rules, or they might be a sequence of statements making up a subroutine. They might be a group of linked objects, an assembly of hardware components, or large software subsystems that intercommunicate in some way. Or they might be a composition of several collaboration patterns on problem domain objects. The essence of design is that the composition of various pieces, each one designed separately, somehow meets a requirement.

That said, we sometimes refer to a high-level partitioning structure as architecture, either technical architecture (having to do with underlying component technology, inde-

pendent of business logic) or application architecture (having to do with partitioning of application logic).

10.7.1 The Component-Port-Connector Model

The rest of this chapter deals with how to model and specify components: executable units that can be plugged together with different interaction schemes connecting them. Our modeling approach is based on the ideas put forth in three definitions.

© ***ports*** The exposed interfaces that define the *plugs* and *sockets* of the components; those places at which the component offers access to its services and from which it accesses services of others. A plug can be coupled with any socket of a compatible type using a suitable connector.

© ***connectors*** The connections between ports that build a collection of components into a software product (or larger component). A connector imposes role-specific constraints on the ports that it connects and can be refined to particular interaction protocols that implement the joint action.

To us, a component architecture defines the schemes of how components can be plugged together and interact. This definition may vary from one project or component library to another and includes schemes such as CORBA, DCOM, JavaBeans, database interface protocols such as ODBC, and lower-level protocols such as TCP/IP as well as simpler sets of conventions and rules created for specific projects.

Component models are specifications of what a component does—based on a particular component architecture, including the characteristics of its connectors—and descriptions of connections between components to realize a larger design.

© ***component-based design*** The mind-set, science, and art of building with and for components and ensuring that the result of plugging components together has the expected effect.

It's impossible to allow a designer to stick components together just any old way: An output that yields a stream of invoice objects can't be coupled to an input that accepts hotel reservation cancellation events. But we would like to be able, at least within one kit of components, to couple any input with any type-compatible output if their behaviors are compatible. To be able to do this confidently across independently developed components means that certain things must be decided across the whole kit.

• *Specification:* What must you do to specify a particular component, with its input and output ports?

• *Instantiation:* What must you do to create an instance of a component and to couple components together?

• *Connectors:* How do connectors (connections between components) work—as function calls, messages on wires, data shared between threads, or some other way? If there are different kinds of connectors, what are they, and how do their implementations differ?

• *Common model:* What types are understood by all the components (only integers or Customers too?)? How are objects represented as they are passed from one component to another?

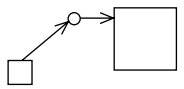
• *Common services:* How does a component refer to an object stored within another? How are distributed transactions coordinated?

The answers to these questions are common across any set of components that can work together (see Section 10.2.2, Components and Standardization). Together, they form a set of definitions and rules called a "component architecture." Microsoft's DCOM, the Object Management Group's CORBA, and Sun's JavaBeans are examples. Project teams often devise their own component architecture either independently or (more sensibly) as specializations of these types. Highly generalized architectures cannot provide, for example, a common model of a Bank Customer, but this would be a sensible extension within a bank.

In this chapter, we concern ourselves with architecture at the generic level of the kinds of ports and connectors supported, although, more broadly, an architecture definition could include much more.

10.7.1.1 Component Connector

We use arrows and beads to represent connectors, ⟶○⟶ based on the UML notation for defining an interface and a dependency on it. However, our use of connectors between ports is more refined.



A type of connector is defined as a generic collaboration framework (see Chapter 9). The interactions between components can be complex, and part of the complexity comes from the techniques that permit them to be coupled in many configurations. The same patterns are repeated over and over—each time, for example, we want work to flow from one component to another or we want a component to be kept up-to-date about the attributes of another.

Connectors hide complex collaborations. The stream of payment orders from Payment Control probably requires a buffer along with a signaling mechanism to tell the receiving component to pick up the orders. The stream of invoices from Purchasing follows the same pattern. The continous update of the Meter's value from the Motor's speed requires a change-notification message; other values transmitted in that example need the same message.

Rather than describe these complex collaborations from scratch for each interface, we invent a small catalog of connectors, which are patterns of collaboration that can be invoked wherever components are to be plugged together. Then we can concentrate on only the aspects that are specific for each connector, mainly the type of information transmitted.

A design described using connectors doesn't depend on a particular way of implementing each category of connector. What's important is that the designer know what each one achieves. We can distinguish the component architecture model (which connectors can be used) from the component architecture implementation (how they work).

10.7.1.2 Example Connectors

Each project or component library can define its own connectors to suit itself. In the Motor example in Section 10.6.2, Kit of Small Components, we can identify two principal kinds of component connector:

- *Events:* Exchanges of information that happen when initiated by the sender to signal a state change (shown with an open arrow: ⟶)
- *Properties:* Connectors in which an observer is continually updated about any change in a named part of the state of the sender (shown with a solid arrow: ➔)

The approach we discuss here also applies to other kinds of connectors such as the following:

- *Workflow transfers:* In which information moves away from a sender and into a receiver
- *Transactions:* In which an object is read and translated into an editable or processable form, is processed, and then updates the original

These are examples; you can define your own kinds of connectors to meet the needs of your enterprise. Being able to separately and explicitly define connector types lets us further decouple intercomponent dependencies from the component implementations and better abstract the structure of components when they are composed later.

10.7.2 Taxonomy of Component Architecture Types

There can be many different implementations of a given architecture model, and the same architecture model can be applied to many different applications. A given component architecture describes its type, or style, and its implementation.

© ***component architecture type, or style*** Which categories of connectors are permitted between components, what each of them does, and the rules and constraints on their usage. Some (unary) connector types can even be used to define standard infrastructure services that are always provided by the environment.

© ***architecture implementation(s)*** How each category of connector works internally, including the protocol of interactions between ports.

Each component architecture does this, whether it uses widespread standards or is defined by the architect of a particular project. Although the use of the big standards is important for intercoupling of large, widely marketed components, we believe that purpose-designed architectures will continue to be important within particular corporations and projects and also within application areas (such as computer aided drafting, geographical systems, and telecommunications) that have particular needs (such as image manipulation, timing, and so on). Therefore, it is necessary to understand what an architecture defines, how to define your own architecture (see Section 10.8, Defining Cat One—A Component Architecture), and how various architectures are related.

Like everything else, architectures have types (requirements specifications) and implementations. A type defines what is expected of the implementation, and there may be

many architecture implementations of a single architecture type. One architecture implementation may be clever enough to implement more than one architecture type if they do not make conflicting demands. For example, most television broadcasts now carry both pictures and pages of text, thereby accommodating separate architectural requirements within a single design of signal. Similarly, some architectural implementations allow two architectures, such as CORBA and COM, to interoperate.

Like object types, architecture types can be extended: Extra requirements can be added (just as the transmission of color pictures was added to the original monochrome). A simple version of the architecture discussed earlier defines event and property connectors; an extension might add transfers and transactions.

It's important to remember that an architecture does not necessarily define any code. The type lays down rules for what the connectors achieve, and the architecture implementation defines the collaborations to achieve that. The collaborations tell the designers of the components which messages they must send and in what sequence.

An architecture gives a component writer a set of ground rules and facilities. It does not necessarily limit the kinds of interactions a component can have with another component; rather, it provides patterns for the most common kinds of interaction.

Suppose you are writing a component that accepts print jobs, queues them, and distributes them among printers. An empty architecture is one in which nothing is predefined and every component and interface must be defined from scratch. Without a laid-down architecture, the documentation of your component must define

- Which operating system clients must use
- Which programming language (or set of calling conventions) must be used
- Which calls the client must make to inquire whether you can accept a job, to pass a job to you, and confirm that you've received it

If you have a simple component architecture, it could minimally define operating system and language or clarify how to couple components working in different contexts. If it defines no notion such as our transfer connector, you must define all the messages you expect to send and receive.

But if it is a more sophisticated architecture that defines transfers, your job is much simpler: You only need to say exactly what type of object you're transferring. The architecture defines what "transfer"means and what messages achieve that effect; wherever you need to, you simply use the transfer connector (as in using a framework application, discussed in Chapter 15) and omit all the details.

So an architecture doesn't limit the kinds of work that components can do together, but it makes it easier to document certain categories of interaction and thereby encourages the use of components.

10.8 *Defining Cat One—A Component Architecture*

There are an infinite number of interesting architectures, and the principles of component-based design we discuss apply regardless of specifics. Components can be connected in a great variety of ways—using CORBA, COM, or even a daily manual FTP transfer—but all of them can be seen as examples of the basic notion of self-contained components coupled together using a set of connectors. The kinds of components and connectors and the way they are implemented vary from one architecture to another; for example, JavaBeans (see Section 10.3) offers a specific set.

In Catalysis, we view all component architectures as follows.

• *Component:* Any active element that performs a useful task we call a component. Components can be individual objects or large subsystems; they can even be the departments in an organization. In general, components differ from plain objects in being packaged more robustly.

• *Connector:* Any means of communication between components we call a connector. A connector can be something as simple as a function call or a group of calls that provide for a collaboration, or it can be something more complex such as a dialog across an API. Or it might be a message sent via CORBA or COM or a file transfer, a pipe, or even the delivery of a deck of punched cards by courier. And of course, a GUI is a connector, just as a user is a component.

• *Port:* In general, each component has several identifiable means of being connected to others. We call these *ports*; connectors connect ports together. Each port can be given a name (and, if there are a variable number of them, an index). For example, the operations on a plain object have their message names; the ports to an Internet host have numbers; the pins on a logic chip and the sockets on the back of your PC are tagged with specific functions; and many large systems have interfaces directed at different user roles.

Some architectures allow for restrictions on *fan-out:* the number of inputs supplied by each output. In general, connections can be made and unmade dynamically, although this can be restricted in a particular architecture.

• *Port category:* In any architecture, there are different *categories* of ports characterized by the style of information transfer: whether isolated events or streams of values, whether buffered, whether interactive, and so on. They are also differentiated by implementation: C++ function call, database transaction, FTP transfer, dictation over the phone, and so on.

• *Component architecture:* A component architecture is a choice of categories of connectors. They are specified first by what they achieve (signaling an event, updating values, transferring objects, and so on) and secondarily by how they are implemented (COM operation, e-mail, carrier pigeon, and so on).

Most categories of port have a *gender* (such as «input» and «output») and a type of value that is transmitted, from simple numbers to complex objects such as reservations or stock dealings. These elements also are defined by the specific component architecture, which also must define the rules whereby the ports can be connected (for example, an input always to an output).

10.8.1 Cat One: An Example Component Architecture

For the sake of concreteness, we will introduce a specific hypothetical architecture called Cat One. It provides a fairly typical basis, and its connectors are similar to those in COM and JavaBeans. However, the component model is itself extensible.

Cat One has several port categories: «Event», «Property», «Transfer», and «Transaction Server». All the port implementations are described in terms of function calls, but you could easily extend Cat One to support more-complex implementations crossing application and host boundaries.

The information carried by each port has a type. Each port can be coupled only to another port that has a compatible type. Compatibility is defined separately for each port category; for Event and Property couples, the sender's output type must be a subtype of the receiver's input.

A connector is implemented as the registration of the receiver's input with the sender's output. When a connection is to be made, the sender must be informed that the required output is to be sent to the required input of the required receiver. This explains what an output port is: It represents an object's ability to accept registration requests and to maintain a list (a separate one for each of its output ports) of the interested parties. An input port represents an object's ability to accept the messages sent by the corresponding output ports.

There are various ways of implementing the messages that occur in a connector. One way is to use only one universal "event" message, with parameters that identify the sender, the name of the output port (as a string), the name of the input port, and the information to be conveyed. This approach is convenient in languages such as C++. In Smalltalk it is easier to use a different message name for each input port; the name corresponds to the port's label. (Only in reflexive languages can the sender be told at registration what message to send.)

A connection can be implemented by an adapter object, whose job is to receive an output message and translate it into the appropriate input, translating the parameters if necessary at the same time.

Events are the most general category of port: An event is a message conveying information about an occurrence. The only difference between an event and an ordinary object-oriented message is that in an event the receiver is registered to receive it. More generally, an event can be implemented as a dialog of messages initiated by the sender; in Catalysis we know how to characterize that with a single action.

Properties convey the value of some attribute of the source component. A property output sends a message (or initiates a dialog) each time the attribute changes. (The attribute may be the identity of a simple object such as a number or of a complex object such as an airplane.) A variation on this theme calls for updates at regular intervals rather than immediate notification of every change. A further variant provides for the source to ask the permission of the receivers each time a change is about to happen. Obviously, this approach calls for more messages at the implementation level; but in a component design, we consider all that to be part of one port.

A Transfer port passes objects from the source to the sink. Once accepted by the sink, a sent object is no longer in the sender. Strings of components with Transfer ports can be used to make pipelines and workflows. In an implementation, the source asks the sink to accept an object; if and when the object is accepted, the source removes it from its own space.

Each Transaction Server port provides access to a map from keys to values. Key-value pairs can be created and deleted; the value associated with any key can be read and updated. Many Transaction Client ports can be coupled to one Server; each one can seize a particular key so that others cannot update it. On release, the client can choose to confirm or abort all the updates since seizing.

10.8.2 Defining the Architecture Type

An architecture is a set of definitions that you can take for granted once you know you're working within that context. Defining an architecture is therefore about writing down the things that are common to every component and its connectors. In other words, it's about defining a design package that can then be imported wherever that architecture is used and gives a meaning to the shorthand port and connector symbols.

An architecture package generally specifies a number of connectors. In addition, it may define collaborations that implement the connectors. Third, it may define generic program code that a designer can use to encode one end of each connector.

10.8.3 Connector Specification

Each component can have several named ports; each port can take part in a connection with several others (see Figure 10.11). The connect action between two ports links them together with a connector. One of the ports must be unlinked; if the other one is already linked, the existing connector is used. Other connectability criteria, not yet defined, must be satisfied as a precondition.

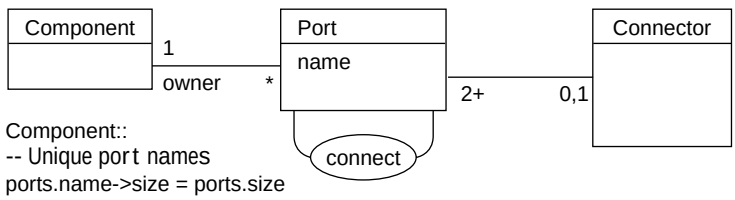


Figure 10.11 Basic port and connector model.

```
action   connect (a : Port, b : Port)
pre:     (a.connector = null or b.connector=null)
         and connectable(a,b)          -- to be defined for each category
post:    a.connector = b.connector and
```

```
        a.connector.ports =
            a.connector@pre.ports + b.connector@pre.ports + a + b
```

```
action  disconnect (a : Port)
pre:    a.connector <> null
post:   a.connector = null
```

A port may have a gender. No more than one source port may be coupled to the same connector.

10.8.4 Connector Design

Figure 10.12 shows a design (one among many possible) of gendered ports, in which a connection is established by registration. To distinguish the types of this implementation from the types they represent in the model in Figure 10.11, we alter the names slightly.

The connect action is realized as a collaboration between the owners of the ports.[9] The owner of the sink port is sent a registration message:

```
action Component1::couple(sink: SinkPort, source:SourcePort)
pre:    sink.source = null and connectable(sink, source)
post:   sink.source = source and
        source.owner–>register(source, sink)
```

Registration does two things: It records the source and port to which this sink is connected and results in a message to the source owner:

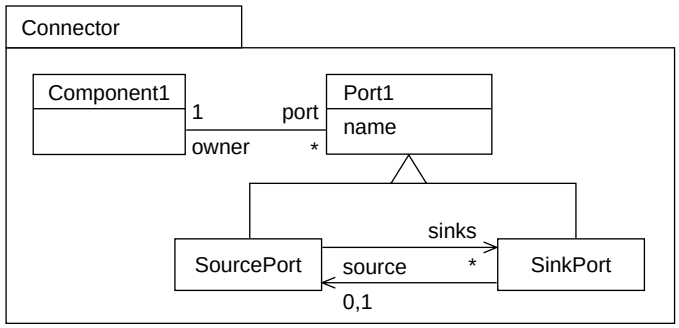


Figure 10.12 A design for establishing connections.

```
action Component1::register (source: SourcePort, sink: SinkPort)
pre:    sink.owner = self
post:   source.sinks += sink
```

9. If the connect takes place in a component assembly tool, the registration may be hidden within initialization code generated by the tool.

This code adds the sender to the registry of sinks for this source.

The abstract model's Connector is realized as the pair of links sinks and source. A Connector exists for each non-empty set of sinks; its ports are the linked SourcePort and SinkPort.

10.8.5 Interpretation of Connector Diagrams

Now we must define how the notation we've been using for components should be interpreted in terms of our component model. We define each box on a component diagram as a Component1 instance; each emerging arrow is a SourcePort, and each ingoing arrow is a SinkPort. A connection between components is a Connector in the sense of our abstract model, which we've realized as a complementary pair of links between the ports.

We can translate a typical fragment of componentry[10] (see Figure 10.13). This illustration shows that an architecture gives a meaning to a component diagram by making it an abbreviation for an object diagram.

10.8.6 Property Connector

Having given a meaning to the basic idea of a connector, we can go on to define the various categories we are interested in for Cat One.

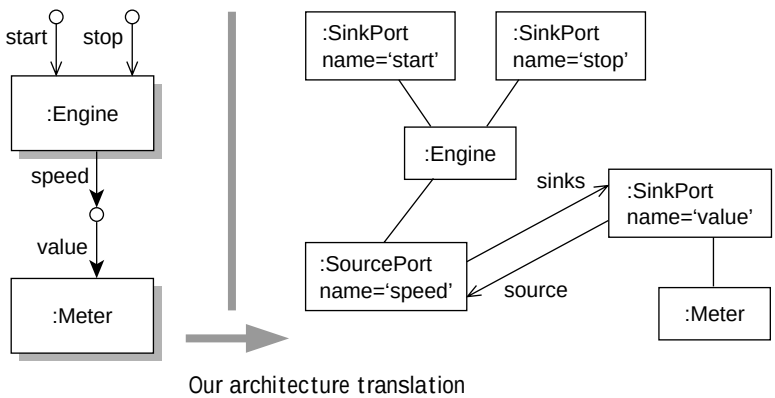


Figure 10.13 Unfolding of a component diagram based on framework.

A property connector has a value in the source port that is maintained by its owner. Whenever the value changes, all currently connected sink ports are updated (see Figure

10. We could use explicit stereotypes to indicate the component frameworks being applied; instead, we assume that a distinguished connector and arrow notation has the same semantics within a particular component architecture. Also, the semantics work equally for types and instances.

10.14). The PropertyConnector framework is applied for each connector marked «property» (for which an abbreviation is the filled arrow). The ports are labeled with the appropriate substitutions for SourceType and SinkType; the type names should also be used to generate separate types for the ports.

Suppose that we want to see how our architecture interprets the example shown in Figure 10.15. The intent of this figure is that the aircraft shown in the aircraft display always tracks the current plane from the runway control; the ready status of the approach control tracks the clear status of the runway control.

We can translate first to a framework application: In the context of our composed component type, the framework relationships shown in Figure 10.16 on the next page hold between

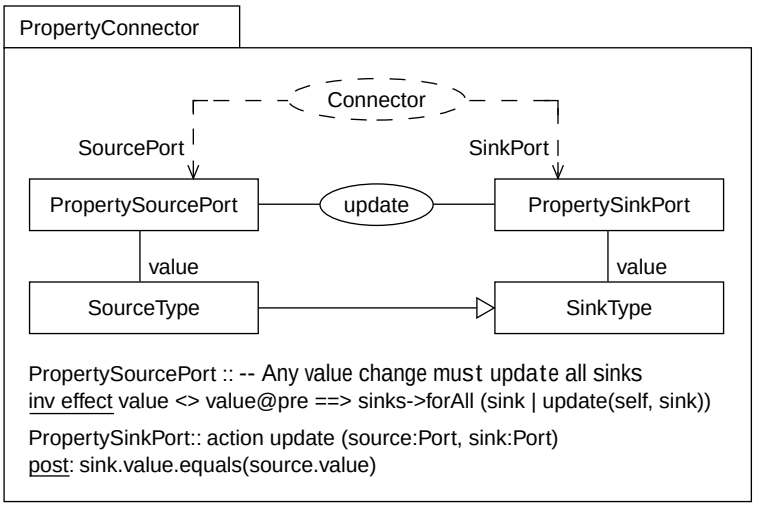


Figure 10.14 Template for a property connector.

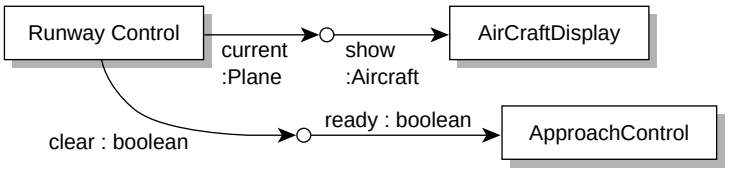


Figure 10.15 Component diagram to interpret.

the parts. The unfolded framework definitions are shown in Figure 10.17.

The complexity of this result is persuasive of the utility of the component notation. The same technique can be used to give meaning to any additional layer of notation and not only to components. All this could be defined in a syntax section of the architecture pack-

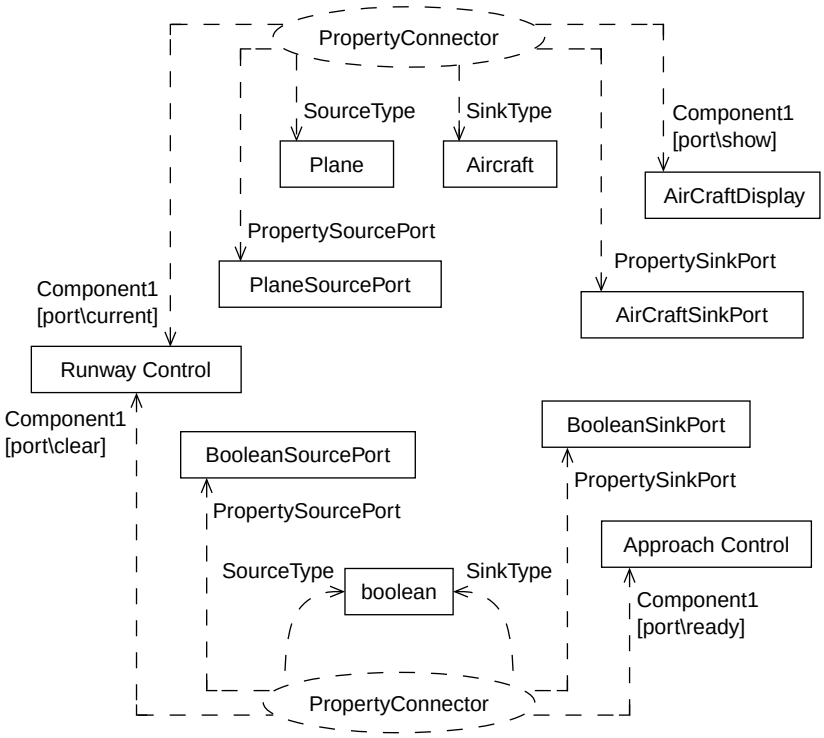


Figure 10.16 Equivalent template applications.

age itself along with a suitable mechanism for defining a visual grammar (see Chapter 9, Model Frameworks and Template Packages).

A straightforward generalization uses a recursive definition of port and connector, allowing us to make connections either at the level of individual events and properties or at higher-level bundles of them.

10.9 *Specifying Cat One Components*

When we specify a component, we take for granted the underlying architecture—mechanisms for registration to receive an output and the like—and focus on a higher-level specification. Including outputs in a specification (marked «output property», «output event», and so on or the equivalent solid arrow notation) implies that all this is assumed. A component specification takes the form of a single type description; but as always in Catalysis, it can be refined and implemented as a collection of objects.

Let's now look at the main categories of ports in our Cat One architecture one by one and see how to use specification techniques to define them.

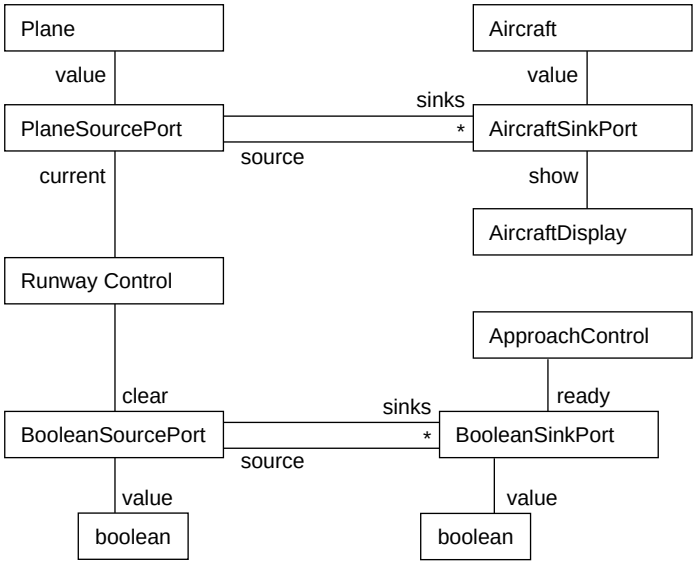
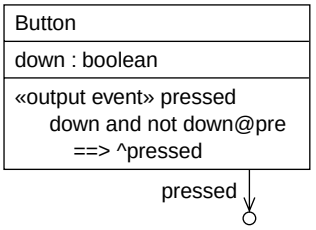


Figure 10.17 Unfolded version of component diagram.

10.9.1 Specifying Input and Output Events

We already know how to specify input events: this category corresponds to the actions or operations that in previous chapters have been used to specify objects. The only difference comes in the implementation and runtime effect. An object designed as a component accepts events according to the protocol defined in a chosen component architecture so that it can be coupled to other components' outputs, including any registration required for those output events. Mechanics for mapping from the output event to the corresponding input events—including string-based mapping, reflective techniques, or an adapter object—are defined by the implementation of the architecture.

An output event occurs when a given change of state occurs. We can specify the change that stimulates the output, using the "old and new state" notation of postconditions within an effect invariant (see Section 3.5.4, Effect Invariants). Notice the caret mark denoting that this message is scheduled to be sent. Here, we've declared the output event superfluously both in text and pictorially. («output event» pressed could have been omitted.) Output events can have arguments, which deliver information to the receiving input events. The usual type matching rules apply.

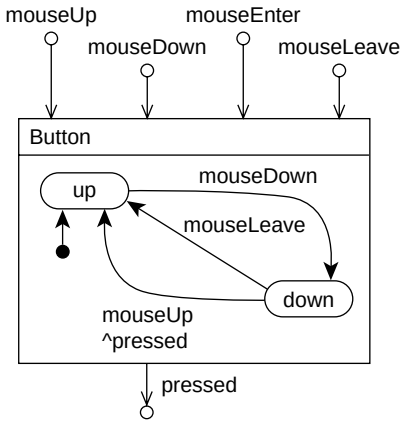


Certain details of an output event are not specified in this style. For example, how soon after a qualifying transition takes place must the output be made?[11] Nor have we said who

11. We can easily add performance specifications either informally or formally.

will receive the output, because this will differ for each design in which the component is used; the architecture guarantees that all connected ports are notified.

Output events aren't always coupled purely to a state change: an output may happen only when the change is caused by a given input action. For example, our Button is a graphical user interface widget that responds to various mouse messages from the windowing system: let's assume mouseUp, mouseDown, mouseEnter, and mouseLeave. The latter two happen if the user drags the mouse in or out of the Button's screen area; the Up and Down messages are sent only when the mouse is within that area.



Let's suppose that we want the operation stimulated by the button to take place when the user has pressed and released the mouse. As the user presses, the Button changes color, and it changes back to normal as the release occurs. But after pressing, the user can make a last-moment decision not to do the action, signaling the change of mind by dragging the mouse away from the Button before releasing the mouse key. In this case, the Button returns to the normal state but does not send the output event.

The required behavior can be readily illustrated with a state chart. Alternatively, we can show the ^pressed requirement as part of the postcondition of each action that causes it:

```
action mouseUp
post: up and (down@pre ==> ^pressed)
```

Here are some useful abbreviations for events:

«input event» action (params)	action(params)
«output event» operation(params)	^ operation(params)

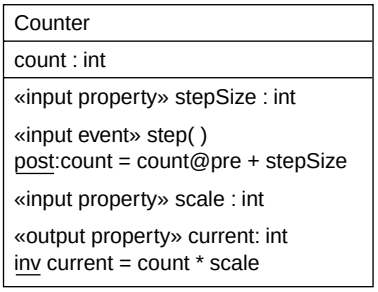
At a business level, a Warehouse component might publish an event outOfStock(Product) specified as

```
«output event» outOfStock (p: Product)
inv effect p.stock@pre >= p.minumum & p.stock < minimum ==> ^outOfStock(p)
```

10.9.2 Specifying Properties

Input and output properties provide a simple way to connect state variables across components.

10.9.2.1 Specifying Output Properties



An output property is an attribute that the component architecture specifically allows to be visible to other components. A chosen component architecture provides a pattern for implementing attributes tagged with the «output property» stereotype.[12]

Like any other attribute, a property can be used in and affected by postconditions of events.

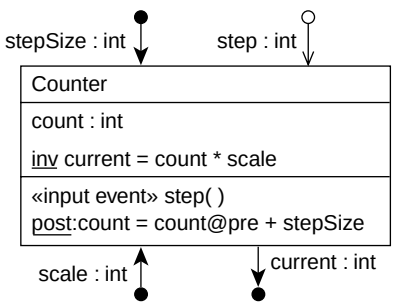
In the component's type definition, properties are shown textually below the line that separates model from behavior; the implementor is obliged to make the property externally visible. But, as usual, the attribute need not be implemented directly as a stored variable but can instead be computed when required.

10.9.2.2 Specifying Input Properties

An input property is an attribute exposed according to a component architecture so that it can be controlled by the output of another component. It can be linked with invariants to other attributes. There is an implied invariant: that an input property will be equal to whatever output (of some other component) it may be coupled to. Therefore, although an input property can be used in a postcondition, it doesn't make sense to imply that it is changed by the action:

```
wrongOp (x : int) post:stepSize = stepSize@pre + x
```

An input property must always be coupled to exactly one output property, although it can be coupled to different outputs at different times. (The next section deals with creating and connecting components.)



Pictorially, input and output properties can be shown with solid arrows, as distinguished from the open arrows of events. (The shadow emphasizes that this is a component—that is, something intended to be implemented according to a component architecture. But it's only for dramatic effect and can be omitted.)

A property value may be an object (and may itself be a component). Updates should be notified whenever a change in a property would significantly change the sending component. A "significant" change is one that would alter the result of an equals comparison between this component and another component.

If the property changes to point to another object, that would normally be a significant change. If there is a change of state of the object pointed at, then the significance of the change depends on whether the property object's state is considered part of the state of the

12. An example is the get/set method pattern used in JavaBeans for component properties.

component. This is the same issue as the definition of equals in Section 9.7, Templates for Equality and Copying.

10.9.2.3 Require Condition

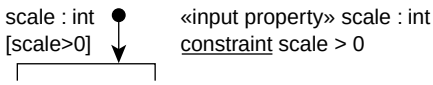
A require condition is an invariant that it is the responsibility of the component's user to maintain. By contrast, a regular invariant is one that, given that the require condition is true, is maintained by the component. A require condition governs the relationship between properties. A typical one might be

```
require scale * stepSize < 1000
```

As with a precondition, it is a matter of design policy whether the implementor assumes that the require condition will always be observed by a careful client or whether the implementation performs checks to see that it is true.

10.9.2.4 Constrained Connectors

A constraint can be imposed on an input and written either against the port in a diagram or in the type description:



Different component architectures treat constraints differently. The simplest approach is to treat the constraint as a form of require condition: the user must ensure that it is not violated.

Alternatively, the architecture can support constraints with a protocol whereby an output requests permission before each change. One such architecture gives the implementation of every port a method with a signature such as change_request_port_x (new_value). By default, this returns true. Before altering an output, a component should send a change request to all the inputs currently registered with that output. If any of them returns false, this change at this output must not happen, and the component must think of something else to do.

Another alternative is for the architecture to support an exception/transaction abort scheme. This approach could be described using the techniques for describing exceptions in Chapter 8.

Using such a mechanism, constraints can also be applied to outputs and more generally to combinations of inputs and so on.

10.9.2.5 Bidirectional Properties

An «in out property» can be altered from either end: changes propagate both ways.

10.9.2.6 Port Attributes

In our approach, every port has several attributes; as before, attributes and specification types need not be directly implemented.

- port.component: The component to which this port belongs.
- inputPort.source: The output port to which an input port is currently coupled.
- outputPort.sinks: The input ports to which an output port is currently coupled.
- propertyPort.value: The object or primitive that is output or input by this property port. Generally, when there's no ambiguity it's convenient to use the name of the property port by itself, omitting the .value.
- propertyPort.constraint(value): A Boolean function returning, for an input property port, whether the associated constraint is true for the given value—that is, whether it is permissible to send this value to this port. For an output port, it is true if true for all the currently coupled inputs.

Port attributes can be used in specifications. For example, suppose we want to insist that the Counter always be wired so that the component that sets its scale is the same one that sets its stepSize:

```
requires
stepSize.source.component = scale.source.component
```

10.9.3 Specifying Transfers

A transfer connector sends an object from the source component to the sink. In contrast with events and properties, each «output transfer» port can be connected to only one «input transfer»; but the basic library of components includes a Duplicator component that accepts one input and provides several outputs.

Like properties, transfers are characterized by the type of object transferred.

10.9.4 Specifying Transactions

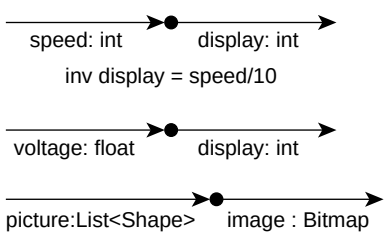
A transaction connector provides for a property of the component to be locked against alteration or reading by others; altered; and then either released in its new state or rolled back to its original state.

10.10 *Connecting Cat One Components*

Big components are made by connecting smaller ones. In this section we explain how the connector types in our example architecture support composition.

10.10.1 Connector Properties

An input property can be driven by an output property, and each output can generally drive any number of inputs. A connection must match types: The output type must be the same as, or a subtype of, the input type. The input and output can have different labels.



It is sometimes useful to make simple transformations between output and input—for example, multiplying a value by a fixed constant or translating an object from one type to another. In the implementation, such things can be done either by an appropriate small component or by some flexibility in the architecture that permit inputs to accommodate straightforward translations on-the-fly. For example, in C++, it is easy to define a translation from one type to another (with a constructor or user-defined cast), which is automatically applied by the compiler where necessary.

In our notation, a transformation can either be shown as an explicit annotation to the connector or, where the output and input types differ, can be left implicit, as the default translation between those types.

10.10.2 Connecting Events

Unless a particular restriction is specified, an output event can be connected to any number of inputs, and an input can be connected to any number of outputs.

An output event has a name and a set of arguments; an input event has a name and a set of parameters. In the simplest connector, only the names differ; then the occurrence of the output event causes the input to be invoked on all currently registered targets. The arguments must match the parameters in the usual way.

If the parameter lists of the output event differ from those of the input to which it is coupled, a mapping must be defined at the connector. If the transformation is too complex, an intermediate component should be defined for the purpose.

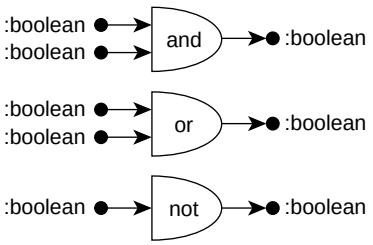
An output event can transfer information in two directions: via parameters and via the return value normally associated with function calls. For that reason, an output event can have a postcondition at the sending end.[13]

10.10.3 A Basic Kit of Components for Cat One

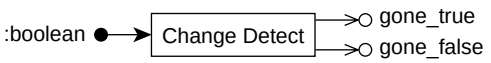
A basic kit of components can be defined to which you can add components that are more domain-oriented. The following is a selection of basic pieces that can be used in many ways. It's intended to provide a general flavor of what can be achieved.

13. In contrast, in JavaBeans an output event cannot expect any postconditions; this means that it cannot be used to describe actual services expected from another component.

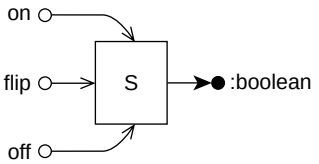
Standard Boolean processing of Boolean properties: outputs are true when their inputs have the relationship marked. (The symbols come from the tradition of electronic logic.)



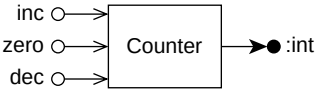
In general, properties and events cannot be coupled. Change Detect generates events on transition of a Boolean property.



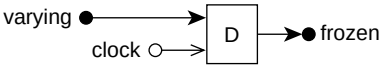
The output of the S gate can be turned on, turned off, or inverted from its current value.



The Counter tracks inc and dec events.

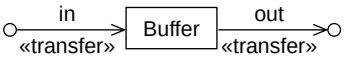


The D gate freezes a copy of the variable input at the moment the clock event occurs.

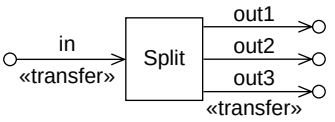


Transfer components:

- Buffer: a FIFO list; it accepts, or emits, any object (property) on request.



- Split: duplicates each input to all its outputs, where each output meets the equal criteria on the input.



10.10.4 Dynamically Creating and Connecting Components

Components can be instantiated and connected dynamically.

10.10.4.1 Connecting Components

The port attributes allow us to specify a connection in a postcondition, as in the following example:

```
Desk :: login(userName:String)
post:directory.userWithName(userName).gui.source = self
```

10.10.4.2 Instantiating Components

Components are instantiated in the same way as types are instantiated in a postcondition: by using ComponentType.new. Consider an operation that causes a component to permanently invert the value of a Boolean output property.

```
action Comp::invert (out: Port)
post:    -- a new Not is created
         let (n : Not.new) in (
              -- whose output connects to the original sink ports
              n.sinks = out.sinks@pre
              -- and it is attached to the port
              out.sinks = n.input
         )
```

A particular component architecture (such as COM+) might intercept the instantiation and connection operations. (Remember, many of them need to be a part of the standard infrastructure services; see Section 10.2.2, Components and Standardization.) That component infrastructure can monitor the known components and their connections to provide richer extensions of behavior.

10.10.4.3 Visual Notations

All the standard notations for dynamic creation of objects, links between objects, and cardinality constraints on the connections extend also to components. In addition, visual builder tools may provide alternative visualizations for instantiating components and connecting their ports.

10.11 *Heterogenous Components*

A kit of components is designed to a common architecture and can readily be plugged together in many ways. But more often, we must use components that were not designed to work together and may not have been designed specifically to work with any other software.

An assembly of disparate components is prone to inconsistencies and gaps in its facilities. And as components are rewritten or substituted, it is easy for its specification to drift.

When you're building with a heterogenous collection of components, you think less about making a beautiful architecture into which all the pieces fit. You don't have the opportunity of designing them. Instead, you worry about how you can nail together the pieces you are given to achieve your goals.

You can considerably alleviate these problems by building a requirements model and then using retrievals to relate the assembly of components back to the requirements. Used

systematically (see Section 13.2.1, Multiple Routes through the Method), this approach helps keep a consistent vision of the system's objectives. The work required to construct a requirements spec and models of the components is repaid by the savings from greater coherence of the result and the rapidity of assembly that is inherent in component-based design.

10.11.1 A Requirements Spec

Let's suppose we are starting a company that sells office equipment. There will be no showroom—only a catalog mailed to customers—and we'll have a warehouse and a telephone sales organization. We want to put together a system to assist its operations.

10.11.1.1 Requirements Model

Figure 10.18 shows a quick and rough model of what we'll deal with.

Most of the types and attributes shown here have an obvious meaning. The primary use cases in this business include makeOrder, makeCustomer, recordContact, and dispatchShipment; others, such as findOrderDispatches, are more like queries and look up information. Let's elaborate a bit on one sample:

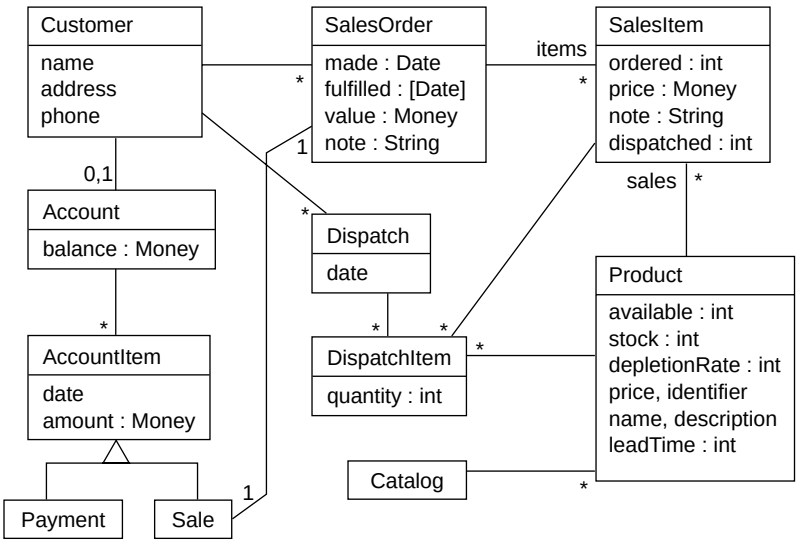


Figure 10.18 Business model for sales company.

use case	dispatchShipment
participants	dispatch clerk, shipping vendor
parameters	list of <sales items, quantities>
pre	sales items not yet fulfilled, all items for same customer

```
post        a new Dispatch created, with dispatch items for each
            sales orders that have no more pending items marked fulfilled
```

10.11.1.2 Business Rules

When a sale is made, the sales staff creates an appropriate Customer object (unless one already exists) and a SalesOrder. The SalesOrder may have several SalesItems, each of which defines how many of a cataloged Product are required. Products are chosen from a Catalog. To avoid confusion, there can't be two SalesItems in a SalesOrder for the same Product:

```
inv SalesOrder::
item1 : SalesItem, item2 : SalesItem :: item1 <> item2
    implies   item1.product <> item2.product
```

An availability level is recorded for each Product: this is the number of items available in stock that have not been earmarked for a SalesOrder. Any operation that adds a new SalesItem also reduces the availability of the relevant Product:

```
inv effect Product:: newItem: SalesItem ::
            sales = sales@pre + newItem
            implies availability = availability@pre – newItem.ordered
```

By contrast, the stock level is the number of items actually in the warehouse; they may have been ordered but not yet dispatched. Availability can fall below zero (if we've taken orders for products we haven't got yet), but stock can't be negative. It is reduced by any operation that adds a new dispatch:

```
inv Product :: stock >= 0
inv effect Product :: newDispatch : DispatchItem ::
            dispatches = dispatches@pre + newDispatch
            implies stock = stock@pre – newDispatch.quantity
```

(When availability gets low, we start purchasing more stock; but let's not go into all that in this example.)

When items are sent to a Customer, a Dispatch object is created. One Dispatch may satisfy several SalesOrders (to the same Customer); and one SalesOrder may be dealt with over several Dispatches as stocks become available. The total number of items dispatched must be no more than the number ordered:

```
inv SalesItem ::        dispatched = dispatchItems.quantity->sum
        and             dispatched <= ordered
```

We must also send the right things to the right Customer:

```
inv DispatchItem ::
            dispatch.customer = salesItem.salesOrder.customer
    and product = salesItem.product
```

A fulfilled SalesOrder is one all of whose SalesItems have the same dispatched and ordered counts. The fulfilled attribute is an optional Date:

```
inv         SalesOrder :: (fulfilled <> null) =
            ( item : salesItems ::item.dispatched = item.ordered )
```

Any operation that changes the order or dispatches must observe this invariant and set fulfilled to something other than null after the order is fully satisfied. But we should also say that the "something" should be the date on which this happened:

```
inv effect SalesOrder :: (fulfilled@pre = null and fulfilled <> null) ==>
        fulfilled = Date.today
```

Every SalesOrder is recorded as an item in the Customer's account:

```
inv      SalesOrder:: sale.amount = value
  and value = items.price->sum
```

A further feature is that the sales staff keeps a note of contacts with potential Customers to remember when and why to pester them next and how much chance there is of getting some business (see Figure 10.19). Some people do this with sticky notes; others use electronic organizers. A Contact here is an occasion on which a Customer was spoken to or sent mail; Customer includes prospects who have not yet made an order.

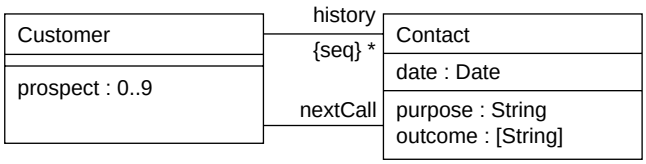


Figure 10.19 Contact-tracking model in sales business.

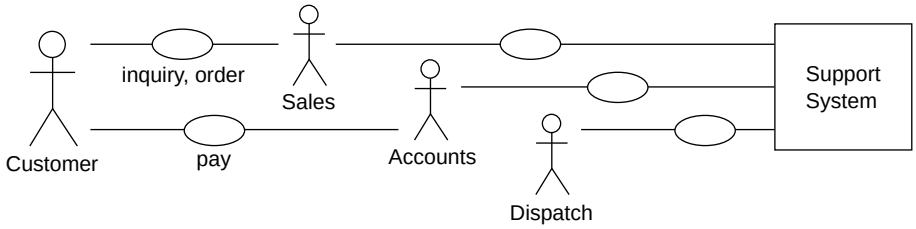


Figure 10.20 Target context diagram.

10.11.1.3 Target Operations and System Context

The new system should support sales, dispatching, and accounts staff (see Figure 10.20 on the previous page). Here's a rough list of the actions we'd like the system to perform for the Sales staff:

(Sales, Support System)::

nextProspect	Display a Customer due to be contacted today.
addContact	Add details of call and date for next try.

makeCustomer	Create details of a new Customer.
findCustomer	Find a Customer from a name or order reference.
makeOrder	Make a SalesOrder for the currently displayed Customer.
findProduct	Display a product from the catalog.
addOrderItem	Add the currently displayed product to the currently displayed SalesOrder.
confirmOrder	Enter payment details such as card payment or payment on account; complete order creation.

The Accounts staff should be able to check on the state of a Customer's account and enter payments. The Dispatch staff should be able to see the orders and create Dispatches.

10.11.2 A Component-Based Solution

In ancient history (earlier than, say, five or six years ago), we might have set to and launched a two-year project to write from scratch a mainframe-based system that integrates all these facilities. But that seems very unlikely these days.

Our chief designer immediately recognizes that the contact-tracking requirement corresponds closely to a single-user PC-based application she has seen in use elsewhere. This will suffice; customers are assigned to sales staff by region, so each salesperson can keep his or her own contact database. There are several mainframe-based general accounts systems; and the designer knows of an ordering system that can be brought in and adapted quickly.

In the interest of meeting rapidly approaching deadlines, therefore, separate systems are set up to accommodate the preceding requirements (see Figure 10.21). Each salesperson works at a PC running four applications: his or her own contacts database that tracks customers in the assigned region; a products catalog browser (hastily constructed as a local Web site); and two virtual terminals, each to the Orders and the Accounts systems. The staff in the warehouse and the Accounts department have their own user interfaces on these terminals.

Does this design fit the bill? That depends on what each of the chosen components actually does.

10.11.2.1 Model of the Whole

Let us first build a model of the components in the complete solution as envisioned (see Figure 10.22). We have annotated it with the types from the requirements model, with a first guess of where these types will "primarily" be maintained. Life will not be so simple, of course.

10.11.2.2 Models of Constituent Components

Before we can plan any meaningful interaction between the existing components or start to develop glue code, we need a model of what they do. Of course, none of them comes with such a model handy!

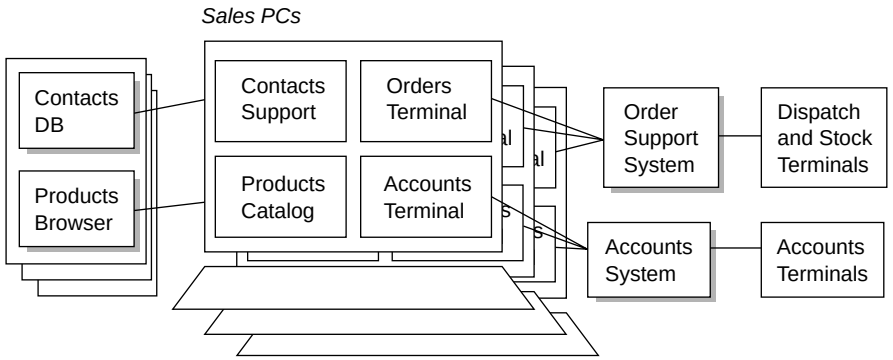


Figure 10.21 Heterogenous component architecture.

By a mixture of experiment and reading the manuals, we build a behavioral model for each component (their designers have omitted to provide one for us). This procedure is highly recommended when you're adopting a component made elsewhere; the same applies when you're reviewing an aging component built locally. The exercise clarifies

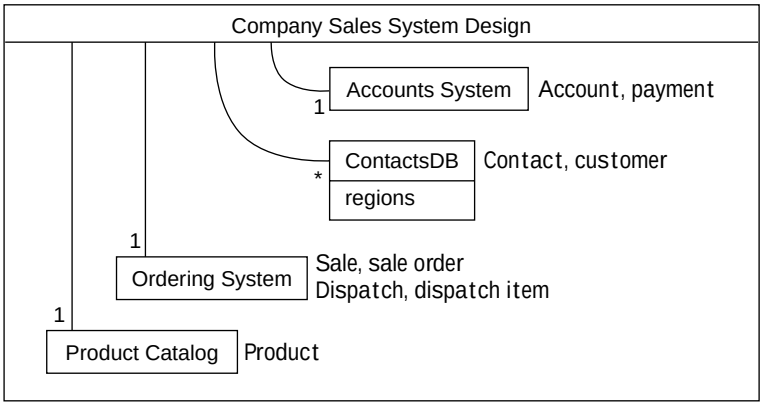


Figure 10.22 Large-grained components modeled.

your understanding of the component, reveals useful questions about its behavior that you can research further, and also tends to make it clear where its shortcomings lie.

The Contacts system has the type model shown in Figure 10.23. Each known person has a history of past contacts and a scheduled next contact. There is a set of due contacts: those that should be worked on. In the reference fields you can put a unique reference number that can be used externally to identify objects.

Like many simple data storage applications, its operations don't seem very interesting: they are different ways of entering, searching, and updating the attributes. There is a current

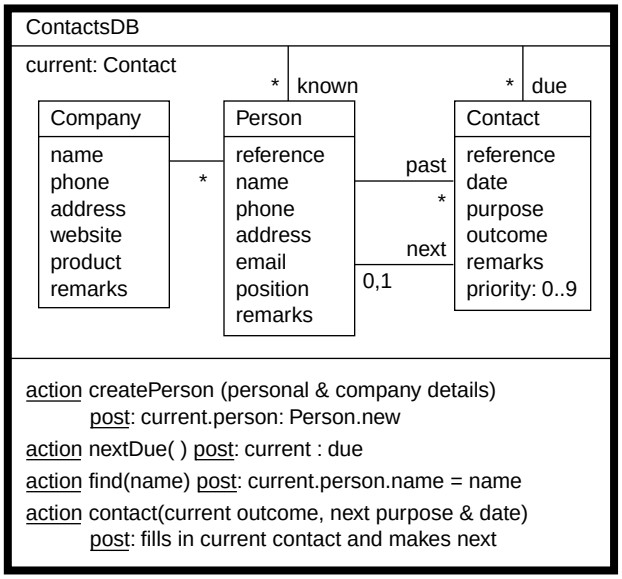


Figure 10.23 The Contacts system.

Contact and, by implication, a current Person and Company: these are displayed on the screen. createPerson makes a new one; nextDue selects a Person who is due for pestering again; and there are various find operations that can select a person by name, reference, company, postal code, and so on.

When a contact is made (a call, e-mail exchange, and so on) the outcome of the current contact is filled in, and a new Contact attached to the same Person, with suggested date and purpose of call. There is no way of deciding never to call this Person again or of leaving it to the Person to call when interested. Sales staffers insist that such a course of action is unthinkable.

The Ordering system attaches Orders to Customers and provides information about the demand for Products, although not the actual stock (see Figure 10.24). There are operations for creating new Customers, Orders, and Items. On a longer-term basis, new Products can be created. When an Order is fulfilled, it is removed from the outstanding list and linked to a new OrderFulfillment on the completed list.

The Accounts system keeps a record of payments against accounts (see Figure 10.25). The reference attribute of an account enables it to be cross-referenced to external records.

10.11.2.3 Cross-Component Links

The system is now made up of a disparate set of components; each one keeps a part of the information needed by the whole system. The Accounts in the Accounts system refer to the People in the Contacts database. How do we represent these cross-references?

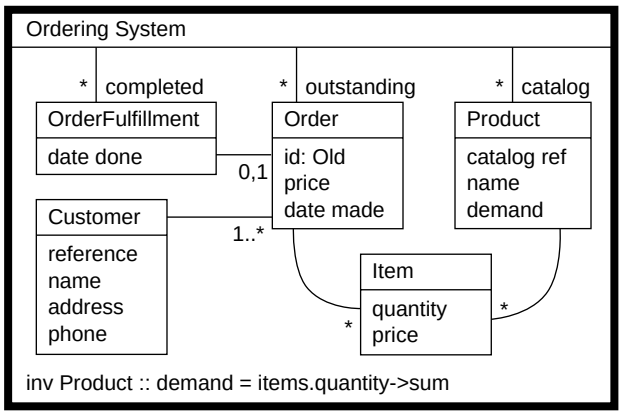


Figure 10.24 The Ordering system.

Links between types represent any association of the members of one type with the members of the other type no matter how the association is implemented. Within a single component, associations typically may be implemented with memory address pointers; within a single database, associations may be implemented as pairs of keys in a table. Between components, an association represents a form of identification that each one recognizes as referring to a single object within itself. For example, we could draw an association between Person (real ones) and the Personal_Records in a national Social Security database: the Social Security number identifies members of one type with those of the other.

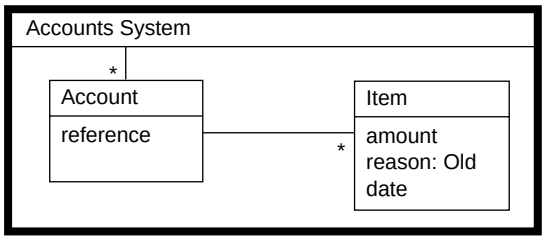


Figure 10.25 The Accounts system.

Figure 10.26 Mapping component models and new business process to requirements.

In the company Sales system (see Figure 10.26), the link from Customer to Account is not held in one component. The Accounts system doesn't have a notion of Customer. But we can decide to use reference fields to cross-link them as *foreign keys*. Within each component, the CID (customer identifier) attributes uniquely identify one Person, Customer, and Account. The CID type itself is understood by all the components (and may be just a

number or a string). So the Person::Account link, which tells us where to find accounting information for a Person, is implemented as the Account in the Accounting System that has the same reference as the Person's ref. We have chosen to show both the link and the CID attributes; the link is a derived one, so we should write an invariant that relates them.

Notice that this is an invariant of the company Sales system design. There is nothing that constrains a Contacts database in general to have cross-references to Accounts databases; after all, it's a third-party component. Therefore:

```
Company Sales System Design :: -- within any instance of this design,
inv
    Person :: ref = account.reference
    -- each Person's ref is the same as his or her account reference
    -- (We already know from the type diagram that the account is within the
    -- Accounting System that belongs to the same Company Sales System Design)
```

We can do similar things with the other cross-component links. The oscustomer link is optional, and that makes it more convenient to state more directly how it is represented:

```
Company Sales System Design :: -- within any instance of this design,
inv
    Person ::          -- for any Person,
        oscustomer = orderSys.customerDB [c | c.ref = ref ]
        -- my oscustomer is the only member of the (Company Sales System's)
        -- order system's customer database whose ref attribute is the same as
        -- my ref attribute. (So if there is no such customer, oscustomer = null.)
```

In effect, reference numbers, identifiers, and keys of all kinds are implementations of links that cross system and subsystem boundaries (see Figure 10.27).

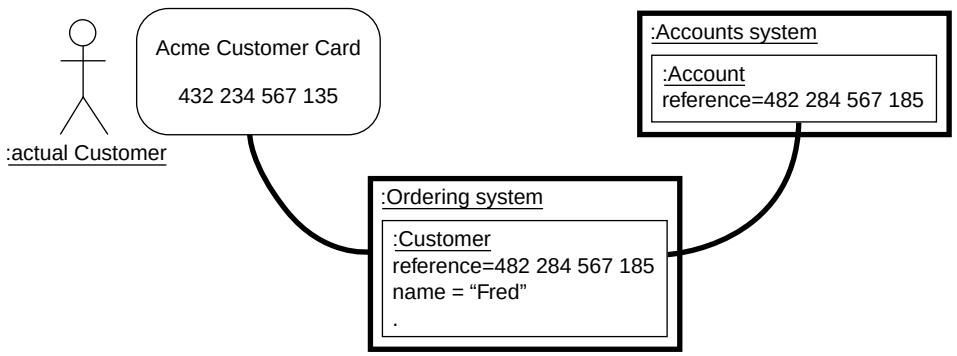


Figure 10.27 Cross-component links.

10.11.2.4 Retrievals

Now we must show how all the information mentioned in the business model is actually represented somewhere in the system design. (See Section 6.4, Spreadsheet: Model

For example, our business model mentions a customerBase, a set of Customers. Where and how are they represented in the system? There is something called Customer in the design, but that is a Company Sales System Design::Ordering System::Customer, which is not the same as a Company Sales Business Model :: Customer. Where do we look to find the complete set of Customers?

For every Customer we know about businesswise, there is a Person in the System Design. We've shown the relationship as the association abs. There are similar direct correspondences for several other types. In fact, the whole business model is implemented by the whole system design, so we've shown that association, too. (The outer type boxes are shown in gray to reduce clutter in Figure 10.26. In real documentation, you'd show them separately.) So we can say that

```
Company Sales System Design ::    -- for any member of this particular design,
abs.customerBase                  -- the customer base of the business model
= contactsDB.known.abs            -- is represented by all the Persons in all
                                  -- the contacts databases
```

(In another implementation, there might not be any one place where all the customers are stored. For example, there might be Customers in the Ordering system that are not in the Contacts database. In this case, we'd say

```
abs.customerBase = contactsDB.known.abs + orderSys.customerDB.abs.)
```

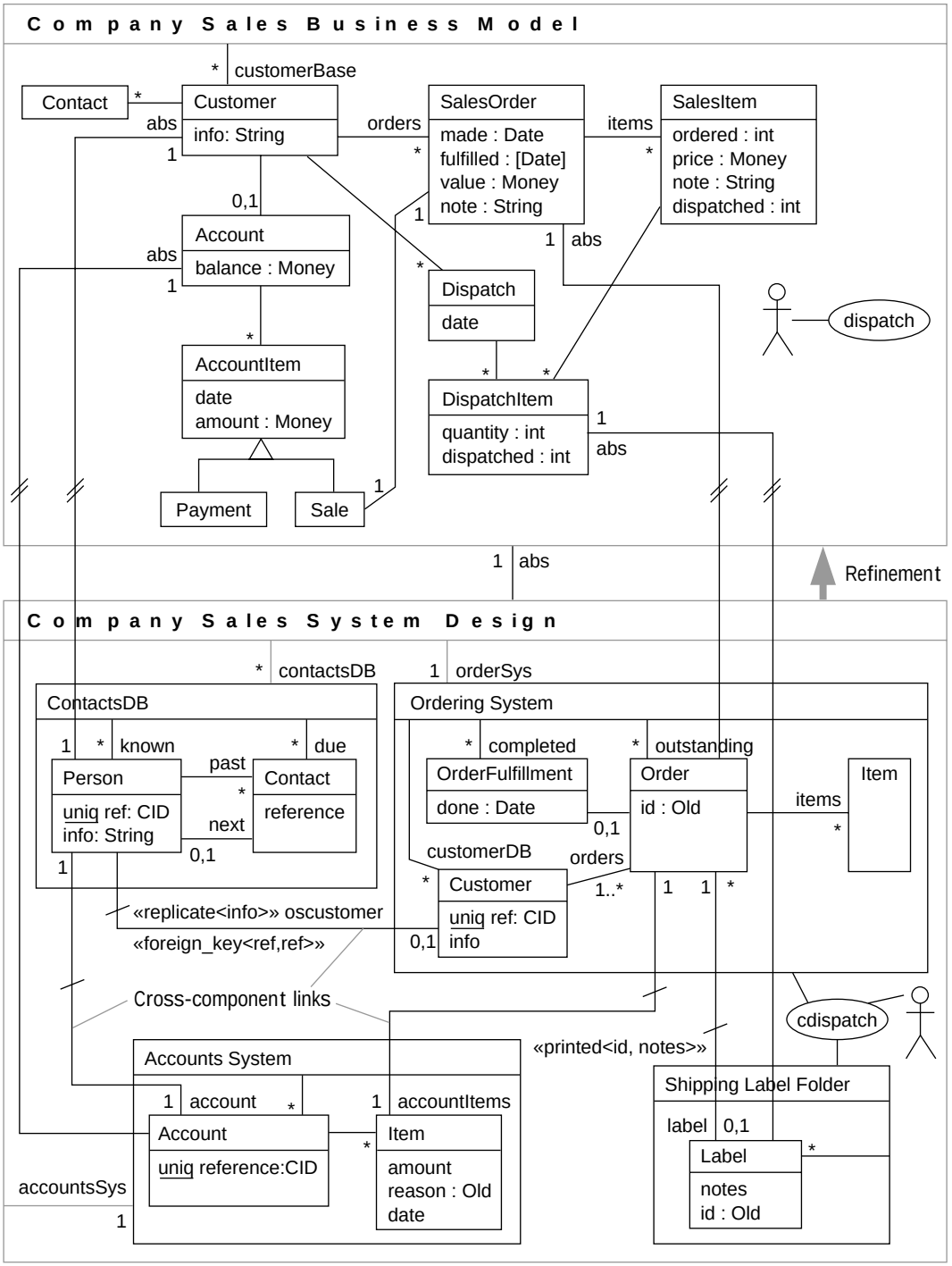
The abs at the end says that it's not the Person that is the Customer but rather the abstraction represented by that part of the implementation. So we should now define what abs means for a Person:

```
Company Sales System Design ::
Person ::                 -- and for each Person, the corresponding Customer's
    abs.info = info       -- attributes are got by following various links and then
    and abs.account = account.abs    -- finding what they represent
    and abs.orders = oscustomer.orders.abs
    and abs.dispatch = oscustomer.orders.label.abs
```

Notice the style here: starting with a type in the implementation, we say that it represents something in the more abstract model—a Customer in this case—which we call abs. Then we go around all the attributes of that abstraction, saying how each attribute is represented within the implementation: abs.info = ..., abs.orders = ..., and so on.

Each of these pieces of information can be retrieved from the implementation by following some links. Sometimes, that is enough: the info is a String, which is what we modeled. But often, we navigate to an implementation type, such as Order; so we end up by saying that it's not actually the implementation's Orders, but rather the abstractions they represent. So we say oscustomers.orders.*abs*. Then in a separate retrieval, we can define how Orders are represented, in the same way.

```
Company Sales System Design ::
Order ::
```



Refinement.) This is essential for review and testing, and generating these relationships tends to expose mistakes.

```
            abs.fulfilled = orderFulfillment.date
and         abs.value = accountItem.amount
and         abs.items = items.abs ... etc
```

In this way, we can gather together, or *retrieve*, the attributes of the business model's types that are scattered about in the implementation.[14, 15]

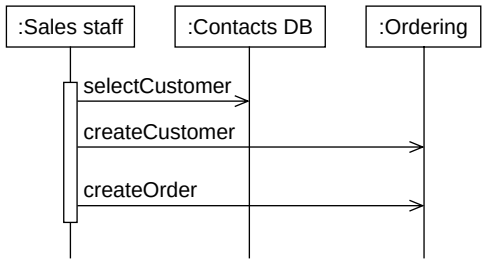
10.11.2.5 Implementing Actions and Business Rules

Now that we know how the requirement's model is represented in the components stuck together for the design, we can work out whether and how the required actions are properly catered to. We need to coordinate the business transactions across our ad hoc components. Let's look at makeOrder, addOrderItem, and confirmOrder.

The requirement for makeOrder is to "create a new Order for the currently displayed Customer." In our hastily contrived system, there can be two Customers displayed on a sales PC screen: one in the Contacts database and another one in the Ordering system. Because the components are entirely separate, the system provides no guarantee that they are consistent.

But makeOrder is an action: a specification of something that must be achievable *with* the system, although not necessarily something it must take the entire responsibility for. Remember that we have modeled it as a joint action (see Section 4.2.3, Joint Actions), and the responsibility partition has not been decided. Our implementation of makeOrder is shown on the next page.

- The salesperson gets the same Customer displayed in the Ordering system window as in the Contacts database. This may involve creating a new Customer in the Ordering system, using the PC's cut-and-paste facilities to transfer name, reference number, and address from one window to the other.
- The salesperson uses the Ordering system's Order creation operation.



14. You might like to try completing the retrieval. In doing so, you may find that some business attributes are missing from the implementation.

15. The technique illustrated here is very general. For example, in the highly regulated business of financial services (banking, insurance), there are many rules about the permitted and prohibited flow of information in different business areas. At the technology level, the CORBA security service provides some machinery to implement this. Explicit as-is and essential business models were constructed, as was a separate technology model. The formal retrievals highlighted many holes that needed addressing and established confidence that the solution met the complex requirements.

The Ordering system provides this action directly, although you must first look up the Product in its list, which carries less information than the separate Web-browsable product catalog.

The spec says, "Enters payment details, such as a credit card or payment on account." According to one of the business rules (see Section 10.11.1.2), the order must be entered as a sale in an Account associated with this Customer.

The Accounts system is entirely separate from the Ordering system, so it is up to the sales staff to copy the right numbers into the right accounts.

Again, we're using the idea of action as specifying the outcome of a dialog between actors (people and components in this case). When people are involved, this generally means relying on them to do the right thing.

How would we describe the use case findOrderDispatches in our new business process? Let's first look at the original requirements model; this use case is a query, and it is best to make the specifications of queries trivial by adding convenience attributes to the model. So we add an attribute on SalesOrder:

```
SalesOrder::dispatches
    -- it is all dispatches for any of my SalesItems
    items.dispatchItems.dispatch

action Agent::findOrderDispatches (order, out dispatches)
post:    result = order.dispatches
```

This requirements specification applies whether the underlying process is manual or automated. The new version refines the required one:

```
action Agent::findOrderDispatcher (order#)
            -- the dispatches with labels whose orders include the target order#
post:    result = shippingLabelFolder.labels[order.id->includes(order#)].dispatch
```

Similarly, the abstract dispatch use case is refined to cdispatch, which now involves the agent, the ordering system, and the shipping labels folder.

10.11.3 Glue Components

Once the dust of setting up shop has settled a bit, we can make some improvements to the first support environment. We will move some work from the staff into the machine.

Sales staffers will now work through a Sales Client component (see Figure 10.28). This is an evolutionary change: the other components will stay exactly as they are. The Sales Client provides a single user interface to all the components used by the sales staff. It implements some of the cross-component business rules such as the Account-Order tie-up, eliminating that area of human error.

In this design, the Sales Client (which runs in each sales PC) integrates the third-party components; but notice that they still do not talk directly to each other. Often, there is no facility for this in older components. In a similar way, a Dispatch Client (used by the ware-

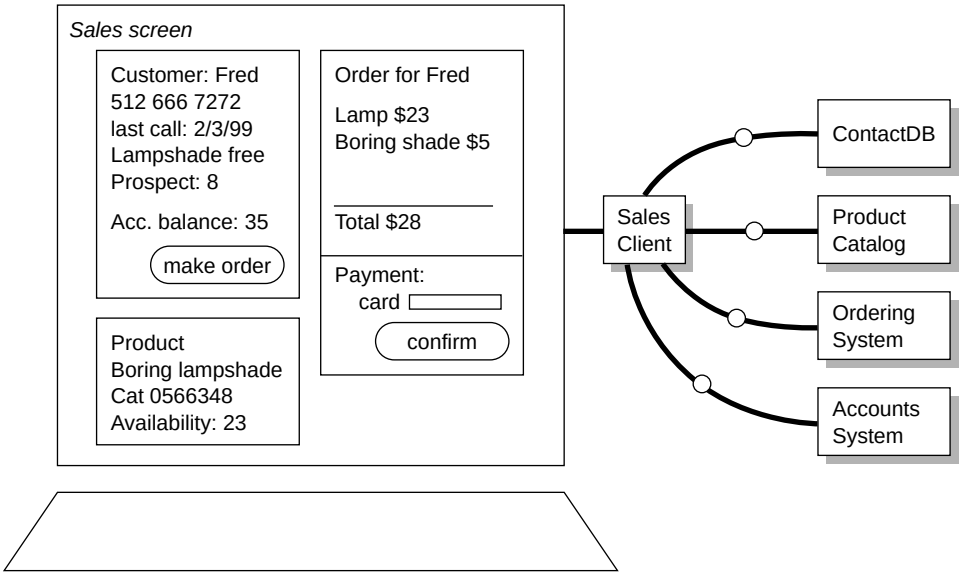


Figure 10.28 First revision: adding glue components.

house staff) can integrate the Ordering system with a dispatch-tracking system and stock control.

Now that the Sales Client has become the sales operator's sole interface with the system, it should take complete responsibility for implementing the actions and business rules associated with sales. Its spec is the sales part of the requirements. It should therefore provide complete operations for making an order, adding the currently displayed product to the order, and creating and adding the proper amounts to the Customer's account.

It is characteristic of third-party components that there is little that is coherent about their interfaces: they all talk in integers, floats, and characters, and that's as far as it goes. There are few standard protocols between components, and locally built glue such as Sales Client tends to be written to couple to specific components. Nevertheless, if the glue can be kept minimal, adaptations are not difficult.

Glue components are not always user-interface or client components. Components can be "wrapped" in locally built code to work to a standard set of connectors, making them look more like members of a coherent kit.

10.11.4 Federated Architecture

A federated system is one in which the division between components more nearly matches the division between business roles.

Improving our system still further, we write our own components. We divide the functions of the Ordering system (see Figure 10.29). One part runs on the salespeople's PCs, and the other part runs on the dispatching department's machine.

Each sales support component has a list of Products and Customers and can generate Orders. Orders are sent (either immediately or in batches) to Dispatch Support, and corresponding debits are posted to the appropriate accounts. The lists of products and customers are shared between all sales staff by intermittent replication to a Sales Master component.

(The pattern of replication is the same for all objects, involving comparison of modification dates followed by transmission one way or the other. This suggests «replication» as

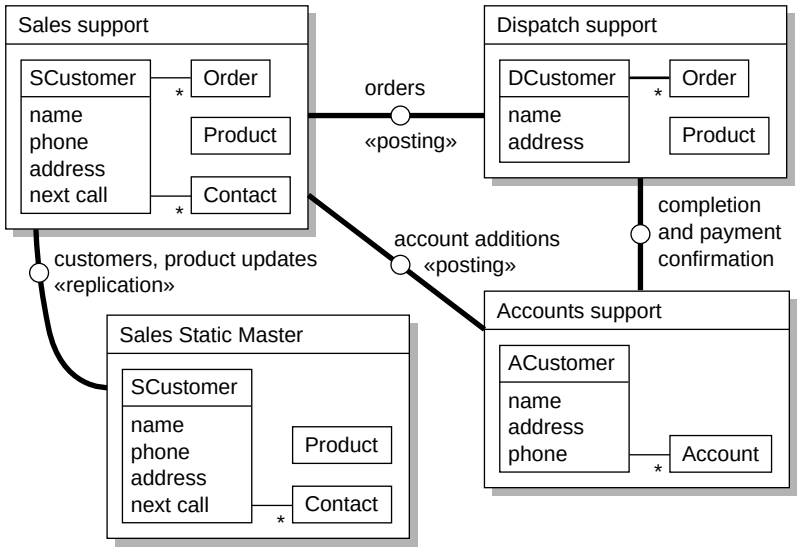


Figure 10.29 Eventual version: federated architecture.

a connector category. The same principle applies to «posting», which is about appending an item to a list that cannot otherwise be altered.)

Each component has a version of the types that support its function. Taken together, these types retrieve to the requirements types.

Federation brings a number of benefits, including decoupling of outage: Each staff member can keep working even if other machines are down. It also supports scalability—the number of sales and dispatching staff is not limited by the power of one machine—and geographical decoupling: a high-bandwidth connection is not necessary between a salesperson and other parts of the system, so salespeople could work from home.

On the downside, some replication is necessary of the product catalog and the customer database. (*Replication* means ensuring consistency between datasets using intermittent

updates, such as when a user logs in occasionally.) However, disk space isn't very expensive, and the technology of replication has been much developed in recent years, particularly since its popularization by Lotus.

10.11.5 Summary: Heterogenous Components

A designer using a set of components that were *not* designed as a kit is faced with two problems:

- Matching their different and redundant views of similar concepts (such as Customer)
- Making their different connectors work together

Glue components can be built to translate the concepts and adapt the connectors; in the simplest case, the users can perform those functions.

We have seen how a clear, high-level requirements specification, and the retrieval relationship, helps clarify the relationship between the disparate designs of the components. We have also seen what needs to be done to unify them.

Pattern 10.1 Extracting Generic Code Components

Reuse code by generalizing from existing work to make pluggable components.

Intent

Make an existing component reusable in a broader context. Resources have been assigned for work outside the immediate project need (see Pattern 10.2, Componentware Management).

Considerations

It is often better to make a generic framework model, which requires less investment in the plug technology needed to make code components plug together. A framework model provides only the specs for each class that fits into a framework and is typically specialized at design time. This arrangement is better for performance and requires less runtime pluggability.

It takes hard work to find the most useful generalization that fits many cases—and to get it right. This effort is worthwhile only for components that will be reused at least four or five times; the investment doesn't pay off for some time.

Strategy

Components Cross Projects. It is unusual for someone without much experience to design a good generic component in advance; such components become apparent only after you find yourself repeating similar design decisions. It is also not very helpful to find components in isolation: They work best as part of a coherent kit. Within a small project, there is not much payoff in developing components: It's easy to spend too much time honing beautifully engineered components that aren't used anywhere else. It's therefore a farsighted architectural job to decide which components are worth working on and integrating into the local kit.

Identify Common Frameworks. Separate objects and collaborations that have common features into framework packages and reimport them to apply.

Don't Overgeneralize. If you simply dream up generalizations, they will not work.

Identify Variable Functionality and Delegate to Separate Plug Objects. Any time you can encapsulate such variability into a separate object, do so.

Specify Plug Interfaces. Define what you need from anything that plugs into your component as well as what you provide to it. Provide the simplest model that makes sense. "Lower" interfaces generally need to know less than "upper" ones.

Package Your Component. It should be delivered with the following.

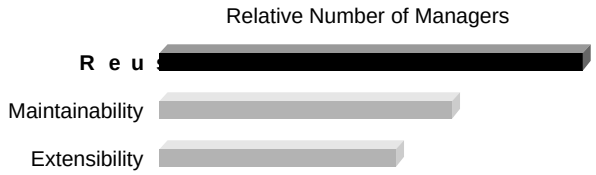
- Its plug specifications.
- Its test harness for clients' plugs—based on the plug spec. This takes one of two forms:
  - A stand-alone harness that drives the plug-in and pronounces judgement
  - A switchable monitor that checks pre- and postconditions during operation
- A selection of demo or default plug-ins.

The component may be part of a suite of components that can plug together in different ways.

Pattern 10.2 Componentware Management

In this pattern, you devote resources to build, maintain, and promote use of a component library. There is no free lunch.

Intent

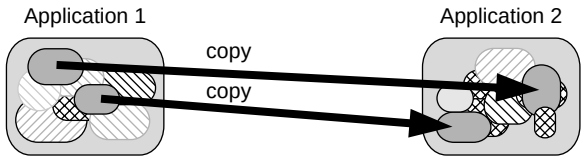


Reuse Motivates the Adoption of OT. Surveys show that of the main three motivations managers quote for taking up object technology, reuse is the leader. But objects do not automatically promote reuse; rather, they are an enabling technology that will reduce costs if well applied. If OT is badly applied, it increases costs. These costs always increase in the short term: Investment is required to move to a reuse culture. The good news is that there are a growing number of success stories when objects are done right.

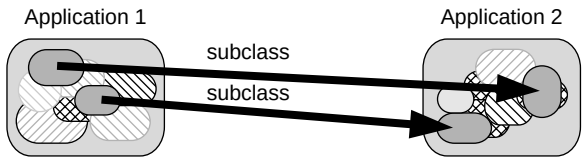
Considerations

Maturity. You need a well-defined process already followed by your developers.

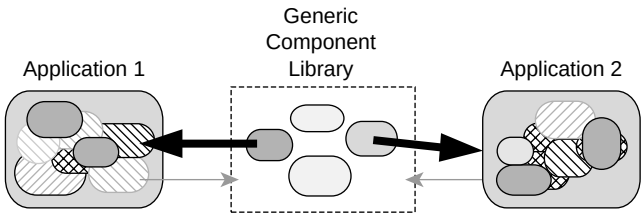
What Is Reuse? Cut and paste, also known as "Adopt, adapt, and improve," is cheap and easy but provides limited benefits; enhancements to the original do not benefit the reusers at all.



Programming by adaptation means that if something looks similar, you inherit and override methods as required. It takes more effort and gives limited benefits. Unless you adhere to strict rules, superclass enhancements need review of overrides in subclasses.



The third approach is to build generic components and import them to reuse. You must devote substantial resources to a library, but this approach yields the best benefits. It requires the most investment.



Caveats. Expect limited success initially. For example, payback should not be expected until after a year or two; pilots will show some success in the short term. It also takes time for management and technical staff to consistently support the idea long-term, and the skills required—for generic component design and interface specification—are more demanding.

Strategy

- Apply the spiral model, with deliberate activities for abstraction and re-refinement.
- Develop a reuse team whose members know and develop the library. Use people with a perfectionist turn of mind and the right skill set.
- To encourage reuse, offer the services of some reuse team people to development projects.
- Do not underresource; it is your design capital.
- Apply models, patterns, frameworks, and designs to all stages of the process.

Pattern 10.3 Build Models from Frameworks

Do not build models from scratch but instead build them by composing frameworks.

Intent

The idea is to generalize modeling work so that you focus on reuse and componentware as early as possible (see Pattern 10.2, Componentware Management).

Considerations

Large models can be repetitive within similar business environments just as program designs are. And models also have variability that can be captured by different forms of factoring and parameterization.

A framework can be only a specification or can also include code implementation (see Chapter 11, Reuse and Pluggable Designs: Frameworks in Code). The latter requires more investment in its design and may run slower. Pure specification frameworks help you build a spec, and then you must implement the result; it does not have the same runtime overhead of pluggability.

Strategy

Model-only Strategy. In this strategy, you build and use model frameworks.

- Look out for similar patterns within business models, type specifications, and high-level designs. Also, make frameworks for common design and analysis patterns.
- Extract common models and use placeholders, effects, invariant effects, and abstract actions to allow you to separate the parts of the models.
- Compose your framework with others. When one type has definitions from more than one framework, use join composition (see Section 8.3.4, Joining Action Specifications).
- Implement the composed framework. Each type in the composition will have a spec, which can be implemented as for the basic design.
- Use a tool that will help you compose model frameworks.

Pattern 10.4 Plug Conformance

Two components fit together if their plug-points conform. Document them with refinements.

Intent

Ensure that two components you've acquired (or built) will work with each other.

There are two specifications at a plug-point: the "services offered" advertisement of one component and the "required" of the other. We must ensure that one matches the other.

How hard this is depends on whether the two components use similar terms. If one happens to be designed specifically for the other, it's easy. If that's not the case but they are based on the same business model, it's not very difficult. If the models are entirely different, there's more work to do (for example, see Section 10.11, Heterogenous Components).

Every model is based on imported others; with luck, components concerned with the same business will import the same packages. Indeed, for this reason it is important to base your specifications on imported models as much as possible.

Strategy

Document a refinement that shows how one component meets the other. See Chapter 6, Abstraction, Refinement, and Testing.

Pattern 10.5 Using Legacy or Third-Party Components

Make a model of an existing component before using it. Create proxies to act as local representatives of the objects accessed through the component.

Intent

This pattern produces a uniform strategy across component boundaries, including legacy components.

Some of your software may be in the form of a third-party or legacy component. It may be an infrastructure that you use to serve your middleware or may be part of the core of the system that implements part of the main business model.

For example, a library management system deals with loans, reservations, and stock control. A third-party component is bought to deal with membership. This is a conventionally written component (probably built atop a standard database) with an API that allows members to be added, looked up, updated, and deleted.

Considerations

Model Translation. The component may (if you're lucky!) come with a clearly defined model. If it doesn't, it may be useful to build your own type model. The model will show the component's view of the information it deals with, together with the operations at the API. The model will not correspond precisely to your system model.

- It will not contain all the information. For example, the library manager knows each member's address, which books the member is currently holding, and what fines are owed. Only part of this data will be stored in the membership manager.
- The component may be capable of storing other things we aren't interested in for this application, such as credit history.
- If the component's model was written by someone else, its attributes and associations may be radically different from those of your system model. For example, it may be designed to associate "reference numbers" with several "short strings," which you intend to use for the member name and address.

Associations across Component Boundaries. Any kind of association crossing a component boundary—for example, between members and loaned books—must be represented in some way, typically by a handle that both components map internally to their own ends of the links. So there may be a reference number used to identify members at the API of the membership manager; this reference number would be stored wherever we need to associate our other objects with members.

Strategy

Use proxies outside the component to represent objects stored more completely within it. Create proxies only when they're needed. For example, the library looks up a member by name and gets back a reference handle from the membership manager, which you wrap in a Member object created for the purpose. Further operations on the Member are dealt with by that object, which sends changes of address through the API; it can be garbage-collected when we process another member.

To check that the component as described does what you require, make a partial retrieval between its model and the system's. Check for conformance of the API specs to your requirements.

10.12 *Summary*

The Catalysis approach to design is about standing back from the detail so that you can discuss the most important parts without the clutter of fine detail. You prolong the life of a design by making your overall vision clear to maintainers, enhancers, and extenders. Earlier in this book we saw how to use models to abstract away implementation details. Pre/post specifications abstracted what was required of an object rather than how it achieved it. Joint actions represent, as one thing, an interaction that may be implemented by a series of messages.

This chapter takes the abstraction one level higher (see Figure 10.30). We have defined a notation in which components—separately deliverable chunks of software—can be specified and designed and then plugged together to make bigger components and complete systems. We have also defined a variety of ways in which components can be interconnected, abstracting away details of the connectors, and outlined a framework for the definition of more categories of connectors.

We have applied the Catalysis ideas of modeling and behavioral abstraction to enable us to specify components aside from their implementations. We have also shown how to check that plugging a set of heterogenous components together meets a given set of requirements.

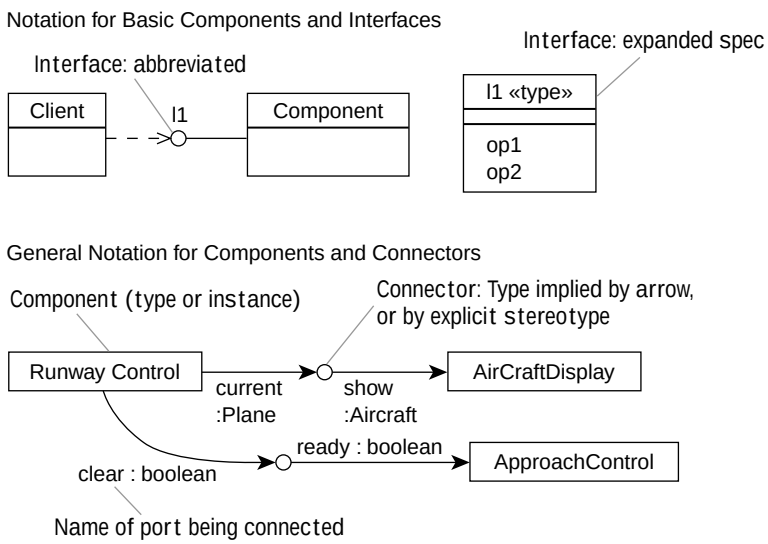


Figure 10.30 Notation for the products of component modeling.