Chapter 11 Components and Connectors

Outline

Many software managers, harried by budgets and delays, envy hardware designers. To design a steam engine they did not start 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 has begun to happen in software. Wordprocessors can talk to spreadsheets, and graphs to databases. Standards like COM and CORBA allow you to plug together components in different languages and platforms. Java's Beans, or any similar protocol, allows separately designed objects to find out more about each others' capabilities before negotiating a collaboration. Visual building tools help you plug components together pictorially.

Large-grained server components are becoming a practical part of an enterprise component strategy, thanks to technologies such as JavaBeans. These interact with each other much like their smaller cousins, and need to be analysed and designed so they interoperate as expected.

This chapter is about meeting requirements with component-based designs, and designing components that work well together. Section 11.1 introduces component concepts, pluggable parts, and how components they have evolved over the years. Section 11.2 shows how kits of components, designed to be used together, can be configured in very rich and varied ways.

Section 11.3 introduces the 'connector' model of component architectures, followed by a typical example of such an architecture in Section 11.4. Section 11.5 and Section 11.5 show how to specify and design with components in this architecture. Lastly, Section 11.7 shows how even ad-hoc and heterogenous component systems are amenable to systematic development.

11.1 *Components — an Overview*

OOP is not a silver bullet

Object-oriented programming alone 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]

Proper usage is essential

But it doesn't work like that. Those benefits depend on good management of the software development process. The bag of techniques, languages, methods and tools lumped under the "object oriented" heading are an *enabling* technology: they make it easier to achieve fast cheap robust development; but only if you use them properly.

One shift is towards reusing components

To achieve significant improvements in software productivity, some fundamental shifts in the management of software development have to take place. One of the most important is to stop writing applications from scratch every time you embark on a new project. Instead, you should build using software components that already exist. The building blocks that you use for software development should not be just those offered to you by the programming language, but larger-grained encapsulated units.

This is becoming widespread today

Over the past five years or so, we have seen this happening already. Many applications are now built upon bought-in 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 applications. The OMG's CORBA provides similar facilities (and is more carefully defined, though slower off the mark).

Wire together large grained applications

For example, an application that reads stock figures from a newsfeed can be 'wired up' to a spreadsheet; this 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 standalone applications with their own user interface, but are provided with a way to interact with other software.

The key is to identify separable parts

It is still the case that many development teams only think of gluing together large bought-in components that can also work as standalones. But the spreadsheet in that example doesn't need a user interface: it is just used as a calculating engine within a larger chain. The example could be built more efficiently with a calculating mechanism, designed to be used as a component in a larger design, that comes without all the GUI overhead (and perhaps with a suitable GUI as an optional add-on). And the persistence mechanism need not be a part of the spreadsheet itself; it can utilize another separate data-access component for that; again, it could include a default one.

...and build domain-specific building blocks

Most development teams could benefit from thinking more in terms of building their own components, appropriate for their own application area. For this is the key to fast, reliable development: to do it the way hardware designers have been doing it for two centuries: to 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.

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

The aim has to be to invest in the development of a component library; and, like any investment, this requires money to be spent for a while, before any payback is seen. A conventional software development organisation requires a considerable shift of attitudes and strategy to adopt a component-based approach. Like all big shifts, it has to be introduced in easy stages, planning carefully the risks, fallbacks, and evaluation of each phase.

Investing in assets needs shift in attitudes

This chapter looks at the nature of components: how they are built and assembled, how they differ from objects, and what comprises a component architectures.

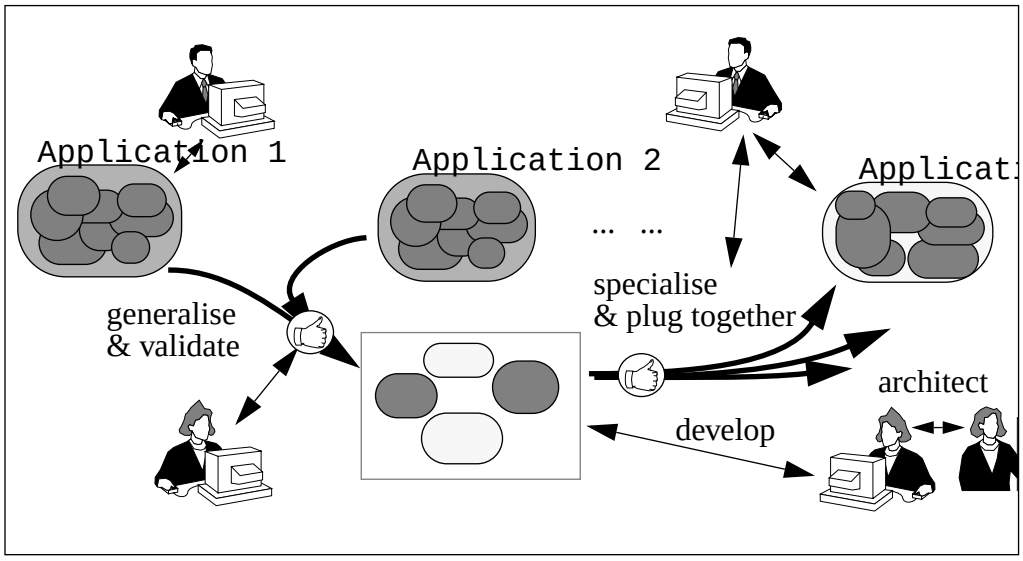


Figure 230: Component Development and Distribution

11.1.1 What is a Component?

There are many definitions; we will start with a broad one, then narrow our focus.

Component — A coherent package of software that can be independently developed and delivered as a unit, and that offers interfaces by which it can be connected, unchanged, with other components to compose a larger system.

This definition is a bit more general than most. A component can include anything that a package can, including executable code, source code, designs, specifications, tests, and documentation. In Chapter 15, *Frameworks*, we showed how the idea composing software based on interfaces also applies to designs and specifications, leading to a more general idea of component-based modeling and specification — all work is done by adapting and composing existing pieces.

A component can package several things

For a component to be 'independently developed' it must be self-contained i.e. the packaging must specify the interfaces that it implements, as well as any interfaces that it requires from other components in order to work properly.

A component implements, and uses, interfaces

Implement against other interfaces

Components are connected to other components only via their interfaces. Each component implements one or more interfaces, which are the published ways another component may be connected to it. Each implementations is done entirely in terms of other interfaces.

A component package will typically include:

- A list of imports — other components this one depends on.
- The external Specification — a description of what it provides for users, and what they need to provide to make it work. (In some cases, part of the specification may be available from the executing objects themselves. See Section 5.10.1, "Reflection," on page 556.)
- The Executable code — which, if built according to a suitable and consistent architecture, can be coupled to the code of other components.
- Validation code — to help decide whether a proposed connection is OK.
- The Design — all the documents and source code associated with the work of satisfying the specification. These may be witheld from customers.

Components can also contain modifications and extensions of existing classes. This occurs often in Smalltalk, where existing classes and methods can be dynamically modified and extended, and is generally also 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 each other, once installed in a system.

We will focus on executable components

Many popular uses of the term 'component' in today's literature refer to executable pieces of code, conforming to one or another component technology. Hence, in this chapter we shall focus largely on executable units. We will use the term component to loosely include the self-contained package, the executable itself, its specification, or even an instantiation of the executable e.g. as a 'component-server'.

11.1.2 The Evolution of Components

Components thinking has evolved a lot

The idea of components goes back almost as far as the idea of software. What has changed significantly over the years is the granularity of the components, the corresponding unit of 'pluggability', the ease of connecting components to each other to compose larger systems, the ability to do more such connecting dynamically, and more standardized infrastructure for component interoperation.

Mainframe systems were components...

The earliest mainframe-based systems were written as monolithic applications manipulating data shared across all procedures in that application. 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 simple: paint to the terminal screen, and read characters commands from the keyboard.

...but not very good ones

These early host-based applications were components — although composing them with others was quite painful. Since the only interface offered to the outside was to a dumb-terminal, the only way to connect two such 'components' to each other was to write pieces of code called 'screen-scrapers' and 'terminal emulators', that acted like

dumb terminals to the host, but interpreted the screen painting commands and generated character commands. The granularity of components was very large, the connections were quite static, and the technology to connect parts was primitive[1].

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

Appications were single executables, re-use was of source code

But software vendors aren't keen on letting people see their source code; they'd rather just give you the executable and (if you insist) the spec. This meant providing ways in which applications could communicate. In the early days, this meant that they passed files to one another: which was slow, and required every output and input from a program to be converted to the form of external records. It lent itself to pipeline processing rather than dialogue between programs.

This led to inter-application communication

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 executables to be linked at run time, without further processing. Two distinct forms of large-grained components evolved from this.

...and to API's, and dynamic linking

The first 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, etc. All communication involved database processing requests in SQL, and clients did not communicate with each other (except indirectly through shared data on the server).

Client-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 O.S., 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.

...and interacting applications followed

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. The objects are the spreadsheet cells, the paragraphs in the document, the points on the graph; the application software is just the context in which those objects execute. The component architecture determines what kinds of object interactions are allowed.

Internal objects were next exposed

Clearly the granularity of components became much finer with object technology. No longer was it just 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.

These were finer-grained and more dynamic configurations

1. This is changing substantially, with mainframe applications re-born as 'server-side components' using technologies such as Enterprise JavaBeans

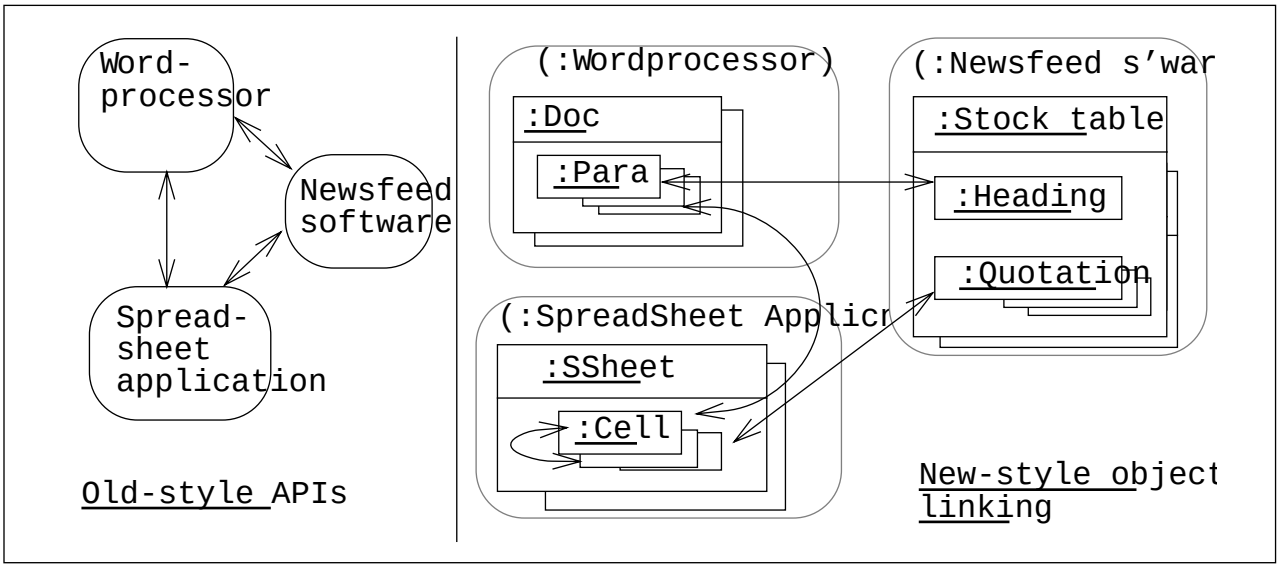


Figure 231: Old vs. New Styles of Component Interactions

The virtual enterprise is built on this idea

And, of course, the virtual enterprise of tomorrow 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 very dynamic.

11.1.3 Components and Pluggable Reuse

Components are meant to plug together

Reuse comes in a wide variety of flavors, ranging from cut-and-paste, through complete application frameworks that can be customized for use. The component approach to re-use mandates that a component should not be modified when it is connected to others i.e. 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 can be replaced by another.

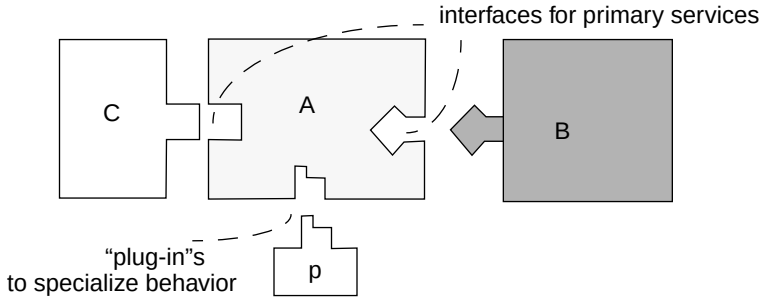


Figure 232: Primary and customization interfaces

The precise meaning of a connection between components will vary, depending upon 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, etc. Section 11.3, “Component Architecture,” on page 452 will examine this in detail.

Different kinds of connections

The fact that a component should not be modified when reused does not mean 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 servics. This style of pluggable design is discussed in Chapter 12.

Components can be customizable

11.1.4 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. These are required both so that these components can interoperate, and to reduce the development burden of common tasks; most of these issues would still need to be addressed even without components, but the need for standardization would be less.

Components need standards

There are three leading contenders in component-ware standards: Corba, COM, and JavaBeans, and they address the need for standardization to different extents. Specific domains and applications often should specialize these standards, or define their own as appropriate.

There are three broad categories of standardization: horizontal, vertical, and connectors.

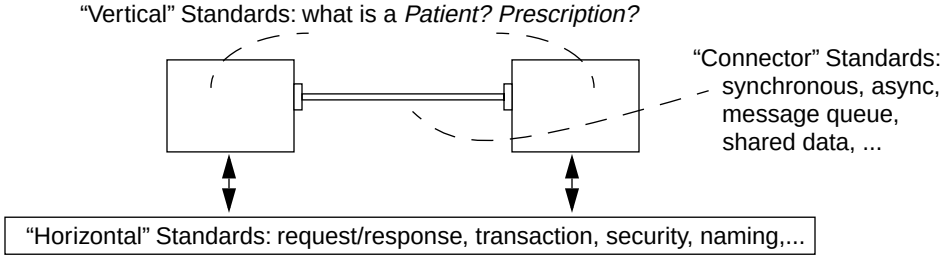


Figure 233: Horizontal, vertical, and connector standards across componen

‘Horizontal’ (infrastructure) standards

Components need a common mechanism for certain basic services. These include:

Common base services

- Request broker — maintains information about location of components, delivers requests and responses in a standard way.
- Security — mechanisms for authentication of users, and authorization for performing different tasks.
- Transactions — because each component potentially maintains its own persistent state information, business transactions will now need to cross multiple components. This needs a common mechanism for co-ordinating such distributed transactions correctly.

- Directory — many components will need access to directory services e.g. to locate resources on a network; others will contribute to these services e.g. a printing component. They need to be based on a common interface.
- Interface repository — component interfaces and their specifications must be defined in a common way, so that they can be understood both by man, and by other components.
- Naming — a uniform way to reference and access entities inside different components, across different internal naming schemes.

'Vertical' standards

Common domain models

Besides the underlying mechanisms being shared, components also need to agree upon the definition of problem domain terms — usually manifested as problem domain objects — which they will jointly operate upon. Components in a medical information system 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 1997, the OMG is actively working towards such 'vertical' standards, in domain s that include telecommunications, insurance, finance, and medical care. Every project must invariably define these domain-specific standards as well.

'Connector' standards

Common interaction schemes

Lastly, we need to use standard kinds of 'connectors' between components, supporting standard interaction mechanisms. These interactions can span a wide range of different semantics, suitable for different kinds of components and compositions. The basic object-oriented 'message-send' is certainly not the only, or even the most suitable, way to describe all interactions.

- Connectors which support explict call / return: the call could be synchronous or asynchronous, asynchronous messages could be queued until processed, etc.
- 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 it behaviors.

11.1.5 Why the move to Components?

How much is hype?

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:

Useful unit of reuse

- Re-use of implementation and related interfaces at medium-granularity: components span a range of granularity (as do objects). A single domain object may not be a useful unit of reuse; a component, packaging together an implementation of some services as it affects many domain objects, can be.

Parallel development

- Component partitioning enables parallel development: identifying medium-grain chunks, and focusing on early design of interfaces makes it easier to develop and evolve parts in parallel.

Scalable

- Interface-centric design gives scalable and extensible architectures: by letting each component have multiple interfaces we reduce the dependency any one component has on irrelevant features of another that it connects with. Also, adding new services incrementally can be accomodated more easily: introduce new components, and add the relevant interfaces to existing ones. Scalability is more easily addressed; replication, faster hardware, etc. can be targeted at a finer grain.

Infrastructure standardized

- Leverage standards: because component technology implies some base set of standards for infrastructure services, a large application can leverage off these standards and save considerable effort as a result.

Manageable units

- Technically practical unit to configure, version, and package: being self-contained, a packaged component must include definitions of the interfaces it implements, as well as those it expects from others. This is a sensible unit to manage for configuration and versioning purposes, particularly for somowhat larger-grained components, or 'kits' of components.

Higher-level capabilities

- Can support capabilities that are impractical for "small" objects: within a component technology it becomes practical to support capabilities that may not practical at the level of programming-language specific objects, such as (i) language-independend access of interfaces, so you can use components written in other languages; (ii) transparent interaction between distributed components.

A kit is a set of interoperable components

This section is about component kits — collections of components that are designed to work with each other. The kit doesn't have to be completely fixed — you can add to it, and have different accessory kits, and sub-kits of pieces that work particularly closely together. But there is a unifying set of principles, the kit's *component architecture*, which make the members of a kit more easy or more likely to be plugged successfully together, than components built separately or chosen from different kits. Plugging arbitrary components together usually requires some sort of "glue" to be built, and we'll deal with that in Section 11.7, "Heterogenous components," on page 472.

We'll begin with a small example to discuss the basic principles, and see how they apply to larger-scale (and more business-oriented) components later on. These examples will use different kinds of 'connectors' between the components. We use arrows and beads to represent connectors. ⟶○⟶ . Section 11.4.1, "Defining the Architecture Type," on page 458 will show how different kinds of connectors can be defined.

11.2.1 Graphical user interface Kit of Components

GUI components are quite familiar

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 or your web server or some other application. And these days, you rarely need to program the software that sets up and builds the forms: you just use a GUI wizard to design them directly.

A typical design

Using the connector notation above, a typical design might look like this configuration of component instances:

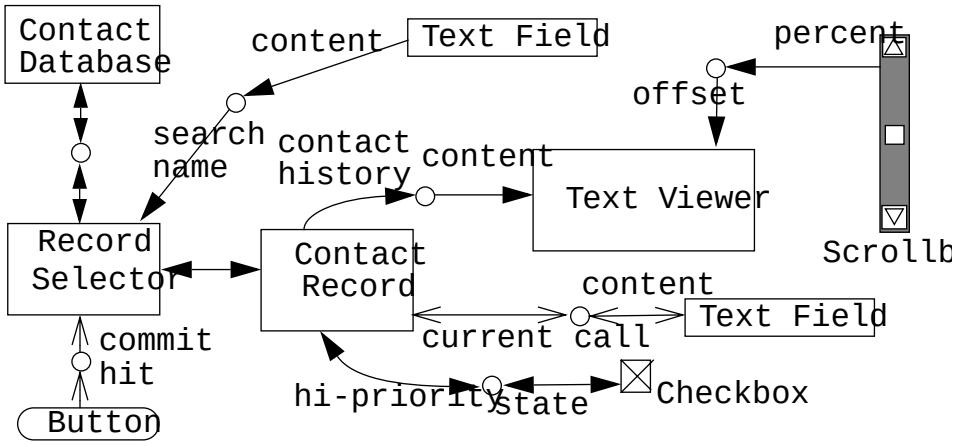


Figure 234: GUI component kit

Properties and events are coupled together

The connectors represent couplings between properties (⟶ ; some pair of values is kept continually in sync) or events (⇒ ; a published occurence from one component triggers some method of another) of two components. There is some need for adapters: between for example, the 'content' of the Text Viewer and the 'contact history' of

the Contact Record. There are both continuously updated properties, such as the scrollbar's connector to the Text Viewer; and events such as the Button's hits. Some of the connectors are bidirectional: the checkbox both sets the priority and will immediately show if anything alters that property.

11.2.2 Kit of Small Components

A domain-specific component-wiring example

Suppose you are given a collection of pieces of hardware: let's say, a motor of some sort, a couple of push-button switches, a meter. Suppose the motor has a few wires sticking out of it: one labeled 'start' another 'stop', and a third tagged 'speed «output»)'.The buttons and meter also have labelled connections. Now you can imagine connecting the components up together, something like this:

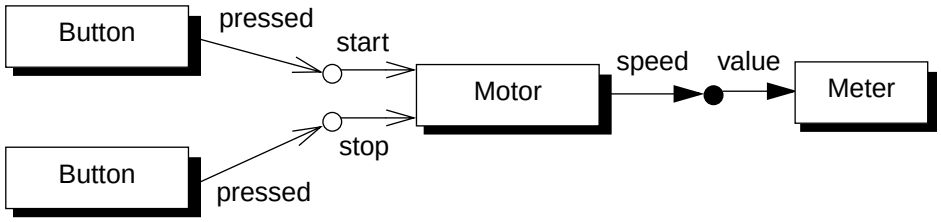


Figure 235: Electronic hobbyist component kit

And a more interesting re-configuration

Of course, a push-button can be used for many other purposes: if you had a Lamp, you might use it to switch that 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 bits and do some creative wiring:

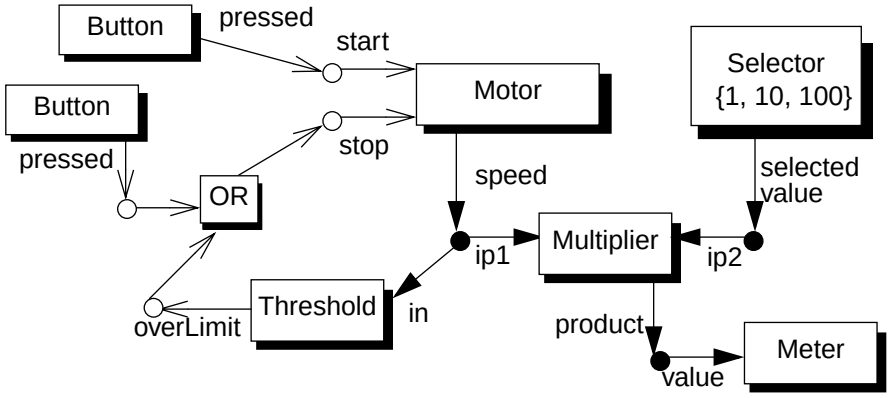


Figure 236: Electronics kit: a non-trivial configuration

Most parts are generic

When the motor is running slowly, the meter doesn't show the small speed very clearly, so we've decided to multiply the speed by 10 or 100 before it gets to the meter. From the box of components we've pulled a Multiplier and a Selector. A Selector is a user-interface thing 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:[1] this converts a continuously varying value like the speed into a boolean off/on, switching on when its input goes over 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.

Parts can be combined very flexibly

Now, any model railroad enthusiast will recognise 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 too difficult to imagine such a kit in hardware; nor 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, just objects not directly visible to the user.

11.2.3 Large Components

Components can also be large things

Components do not have to bit little things; they can be entire applications, or legacy systems. The nature of the connectors between these will differ.

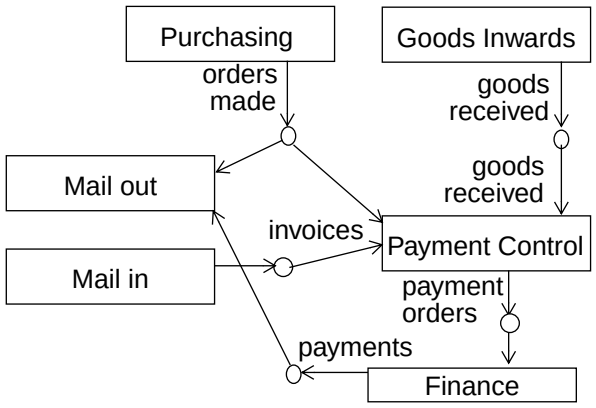


Figure 237: Large business components

These components are the support systems for some of the departments in a large manufacturing company. 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. And 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 out and forwarded to payment control for subsequent payment. When the shipping department receives goods coming in, they forward a notification to payment control. Invoices received are also forwarded. When an invoice is received for goods that were delivered, a payment is generated.

They communicate entire objects via transfers, splits

Where the smaller example sent primitive values across the connectors, these components send larger objects such as orders and 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.

1. What hardware people call a Schmitt trigger.

The configuration is a modern one: rather than having 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.

Such a ‘federated’ system has many benefits

- Flexible. It is more extensible and flexible than centralised systems clustered around one database. Business reorganisation is rapidly reflected in the support systems.
- Scalable. It is also more scalable: to serve more people, buy more machinery. You are not constrained by needing to have a single server that will only scale so far.
- Graceful degradation. Each business function is supported by its own machinery, and so one malfunction doesn’t stop the whole enterprise.
- Upgradable. The system does not have to be all set up and subsequently updated as a single entity. As the business grows and is reorganised, new software can be added. Commercial off-the-shelf components can be plugged in, rather than having to build every enhancement into a single program.
- Appropriate. Because it requires less central control, it is less prone to local political and bureaucratic difficulties: there is no one authority that has to agree to every change, or has to be persuaded to address the evolving requirements of every department. Instead, the business users hold much more responsibility for providing support appropriate for themselves.

Re-configuration now corresponds to visible business process change

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 very expensive. They reorganised their departmental roles, and the software to match: Goods Inwards now has records of all orders that have been made, and will turn away spoof deliveries.[1]

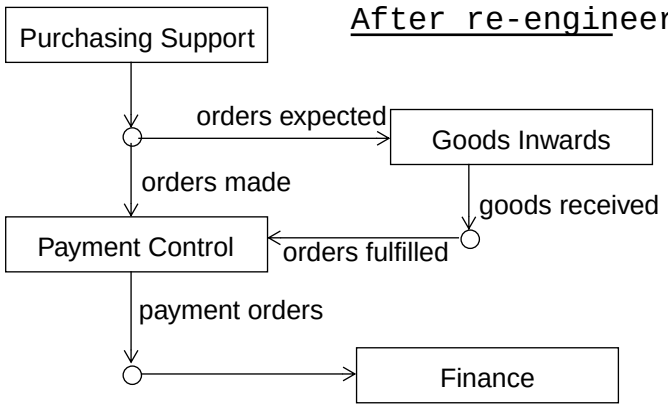


Figure 238: Reconfigured business components

Because the component structure reflected the business structure at the level of granularity of the change required, the change could be accomodated fairly well. Of course, finer-grained business changes — e.g. the introduction of a new pricing plan for existing products — requires corresponding finer grained component (or object) structure to accomodate the change.

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

11.2.4 Component building tools

Assembly tools work on such component kits

Unfortunately, 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 to plug them together graphically. Digitalk's Parts and IBM's Visual Age were early examples; Symantec's Visual Café works in Java, like Sun's Java Workshop; there are also similar-sounding "visual programming" tools that are actually restricted just to building user-interfaces, rather than composing general components. There are tools (for example Forté) that specialise 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.

11.2.5 Are components just objects?

Components may be built from objects

Many components may be single objects, or built from several objects. But when you build a design from components, you don't need to know how components are represented as objects, or instances of a classes, or how the connectors work.[1]

Federated components have much in common with objects

The shift from central database to federated systems is similar to the move from languages like COBOL to Java and its kin. In both COBOL and a central database architecture, all data definitions are lumped together in one place, and all the procedures in another. There is no restriction on the accessibility of each piece of data from each piece of program, so the interdependencies grow like bindweed. In federated systems, just as in OO programs, each component is a collection of software, chosen for the support of the corresponding business function, and using local data representations best suited to that software. And just as in OO programs, objects need to access the information held by other objects, so in a component architecture, components intercommunicate through well-defined interfaces, so as to preserve mutual encapsulation.

Similarly for small components

The much smaller-grain components in Section 11.2.1 and Section 11.2.2 are also very similar to objects. In fact, they would be easiest to implement as objects in an object-oriented programming language.

There are differences of degree

This shows that the differences between component and object oriented design are mainly to do with degree and scale, and are not intrinsic to either:

- Components often have persistent storage; although objects in an OO programming language always have local state, they typically only work within main memory.

1. Components can also be built without explicitly using object-oriented design techniques at all; but OO makes it a lot easier; and 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 just become part of the body of ideas that constitute good software engineering. Having learnt how to do it, we can now move on to putting it to work.

- Components have a richer range of intercommunication mechanisms, such as events and work-flows — rather than just the basic OO message.
- Components will often be larger-grained than the traditional object, and may be implemented as multiple objects of different classes.
- 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, but not the operations required. In fact, the smallest object-oriented package that could qualify as a component package would consist of a class, the interfaces it implements, and the interfaces it expects of other objects.
- Objects tend to be very dynamic in nature — the number of customers, products, and orders you have, and their interconnections, changes dynamically. In contrast, larger-grained components may be more static in nature — there will probably always be just one payment control and one finance component, and their configuration will be quite static. Of course, there is no intrinsic reason why these components should not be modeled as objects, or why they should be static.

Catalysis applies equally to both

Components, like objects, do interact through polymorphic interfaces. All of our modeling techniques will apply equally well in both cases, including the more general connectors for components. In a good OO development method, there is indeed no premature presumption about whether an object is in main memory, disk, tape, or graven on tablets of stone; and we have seen that collaborations and actions abstract over any kind of interaction, not just messages.

Components and objects both distribute

In our larger example, the components might be executing on separate host machines; or they might not. They might communicate through function calls, or through a network. That is determined by the Component Architecture; but the principles are the same as for large-grained distributed objects, irrespective of such issues.

11.3 *Component Architecture*

11.3.1 "Architecture"

Discussion sometimes arises about the difference between "architecture" and "design". We find the following distinction useful.[1]

Architecture is about consistent rules and patterns

Architecture is about recurring design structures, rules, or principles that are used (or could be) across many designs; such as, "We shall all write in Java", or "here's how we make one property observe another", or "use these interfaces and protocols to implement a spell-checking feature", or "use Fred's class whenever you want to wongle foobits", or even "never use return codes to signal exceptions". The architecture is what lends the coherence and consistency to the design.

An architecture is broadly comprised of two parts (we defer a more detailed discussion to Chapter 13, *Architecture)*:

- the generic design elements or patterns that are used in that architecture, such as subject-observer, or Fred's class, or the event-property connectors, or the generic design for spell-checking; and,
- the rules or guidelines that determine where and how those 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 even fully define a translation scheme.

Design is applying those rules in specific ways to meet a spec

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 some architectural rules; or a sequence of statements making up a subroutine; or a group of linked objects; or an assembly of hardware components; or large software subsystems, intercommunicating in some way; or a composition of several collaboration patterns on problem domain objects. But the essence of design is that the composition of different pieces that were or can be separately designed somehow meets a requirement.

That said, we will still sometimes refer to a high-level partitioning structure as architecture, either technical (having to do with underlying component technology) or application (having to do with partitioning of application logic).

11.3.2 The Component - Port - Connector Model

We define component architectures based on ports and connectors

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 will be based around these ideas:

- Components — pieces of software that can be 'plugged into' a wide variety of others. They range in scale from small user-interface widgets to large transaction-processing applications.

1. Thanks to John Daniels for this.

- Ports — the exposed interfaces that define the ‘plugs’ and ‘sockets’ of the components. 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).

This spans many levels of standards and rules

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

Component models specify a component

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 components and ensuring that the result of plugging components together has the expected effect.

Plugging component relies on underlying architecture

Now of course, it’s impossible to allow a designer to stick components together just any old how: 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 i.e. 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:

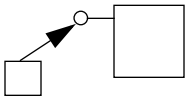
Specification	What you have to do to specify a particular component, with its input and output ports.
Instantiation	What you have to do to create an instance of a component, and to couple components together.
Connectors:	How connectors (connections between components) work — as function calls, messages on wires, data shared between threads, ... And if there are different kinds of connector, what are they, and how do their implementations differ?
Common model:	What types are understood by all the components? (Just 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 all of any set of components that can work together (see Section 11.1.4, “Components and Standardization,” on page 443). 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 Java Beans are examples. Projects often devise their own component architecture, either independently or (more sensibly) as specialisations of these, because very general architectures cannot provide a common model of what a Bank Customer is (for example); but this would be a sensible extension within a bank.

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

11.3.2.1 Component connector

Notation



We use arrows and beads to represent connectors,  , based upon the UML notation for defining an interface and a dependency on that interface.

Connectors are defined by collaborations frameworks

A type of connector is defined as a generic collaboration framework (Chapter 15). The interactions between components can be quite complex: part of the complexity comes from the techniques that permit them to be coupled to each other in many configurations. The same patterns are repeated over and over, each time we want, for example, work to flow from one component to another, or each time we want a component to be kept up to date about the attributes of another.

Each connector type encapsulates complex interaction

Connectors hide complex collaborations. The stream of payment orders from Payment Control probably requires some buffer, and a signalling mechanism to tell the receiving component to pick them up; 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.

Rather than describing these complex collaborations from scratch for each separate 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 just the aspects that are specific for each connector — mainly the type of information transmitted.

The design only depends on the connector type

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 knows what each one achieves. We can distinguish the component architecture model (what connectors there are) from the component architecture implementation (how they work).

11.3.2.2 Example connectors

Each project or component library can define its own connectors to suit itself. In the Motor example above, we can identify two principal kinds of component connector:

- Events — exchanges of information that happen when initiated by the sender to signal some 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 applies to other kinds of connectors as well, sich as:

- 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.

11.3.3 Taxonomy of Component Architecture Types

Architectures have 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 will describe:

An architecture type — what categories of connector there are between components, their rules, and what each of them achieves;

Architecture implementation(s) — how each category of connector works.

Other component architectures do the same — whether they are widespread standards, or just defined by the architect of a particular project. Whilst the big standards are 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 particular application areas (such as CAD, geographical systems, telecoms) where there are particular needs (e.g. for image manipulation, timing, etc). Therefore it is necessary to understand what an architecture defines, how to define your own architecture (Section 11.4, "Arch One — A Catalysis Component Architecture," on page 457), and how different architectures are related.

Architectural types form a taxonomy

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 — separate architectural requirements accommodated within a single design of signal; similarly, some architectural implementations will allow two different architectures, such as CORBA and COM, to interoperate.

...and can be extended

Like object types, architecture types can be extended: extra requirements can be added (just as the transmission of colour pictures was added to the original monochrome). A simple version of the architecture above just defines event- and property-connectors; an extension may 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 that achieve that. The collaborations tell the designers of the components, what messages they must send and in what sequence.

It provides common interaction patterns

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

Without an architecture you have to spell out many things

Suppose you are writing a component that accepts print jobs, queues them up, 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:

- what operating system clients must use;
- what programming language (or set of calling conventions) must be used;
- what calls the client must make to enquire whether you are able to accept a job, to pass a job to you, and conifrm that you've got it.

A simple architecture saves some basics

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 like our transfer-connector, you will have to define all the messages you expect to send and receive.

More sophisticated archictecture gives higher-level description

But if it is a more sophisticated architecture that defines transfers, then your job is much simpler: you only have to say exactly what type of object you're transferring. The architecture defines what "transfer"means and what messages achieve that effect; any place you need to, you simply use the transfer connector (e.g. using a framework application, 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 their use.

11.4 *ARCH ONE — A Catalysis Component Architecture*

There are an infinite number of potentially interesting architectures, and the principles of component-based design we discuss apply regardless of specifics. With the goal of being concrete, we outline here one specific basic scheme that will help us discuss the issues, including what it means to define your own architecture. We will call it ARCH ONE — a basic Catalysis component architecture. It has some similarities with JavaBeans and COM.

It is one concrete scheme to discuss issues with

ARCH ONE is a component architecture in which:

- Each connector between components links a labelled output port to one or more labelled input ports. (Some architectures allow for restrictions on 'fan-out' — the number of inputs supplied by each output.) Connections may be made and unmade dynamically.

  Dynamic connections

- There are different categories of port, such as «Event», «Property», «Transfer», «Transaction», and others. Most categories have two 'genders', «input» and «output». Each port can be coupled only with a port of a compatible category and gender — which generally means the same category and opposite gender.

  Many port categories, two genders

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

  Simple compatibility is based on subtyping

- A connection 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.

  'Output' port registers other's input ports

- There are various ways of implementing the messages that occur in a connector. One way is to use just 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 is convenient in languages like C++; in Smalltalk it is easier to use a different message name for each input port, corresponding to the port's label. (Only in reflexive languages can the sender be told at registration what message to send.)

  Messages may be encoded as strings

- 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.

  ...or translated by adapter objects

- Events are the most general category of port: an event is a message conveying information about some occurrence. The only difference between an Event an ordinary object-oriented message is that the receiver is registered to receive it.

  Events announce happenings to registrands

More generally, an Event may be implemented as a dialogue of messages initiated by the sender; in Catalysis we know how to characterise that with a single action.

Properties track value changes

- Properties convey the value of some attribute of the source component. A property output sends a message (or initiates a dialogue) each time the attribute changes. (The attribute may be the identity of a simple object like a number, or a complex object like 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. This obviously 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 'moves' an object

- 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 will ask the sink to accept an object; if and when accepted, the source removes it from its own space.

Transaction ports co-ordinate comit / aborts

- 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; and each may seize a particular key, so that others may not update it. On release, the client may choose to confirm or abort all the updates since seizing.

11.4.1 Defining the Architecture Type

Architecture is defined in a package, then imported

An architecture is a set of definitions that can be taken 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, to give a meaning to the shorthand port and connector symbols.

It provides connector specs, designs, and code

An architecture package will generally specify a number of connectors. In addition, it may define collaborations that implement the connectors; and thirdly, it may define generic program code that a designer may use to encode one end of each connector.

11.4.2 Connectors: general

11.4.2.1 Connector Specification

Each component can have several named Ports. Each Port may take part in a connection with several others.

Basic port-connector model

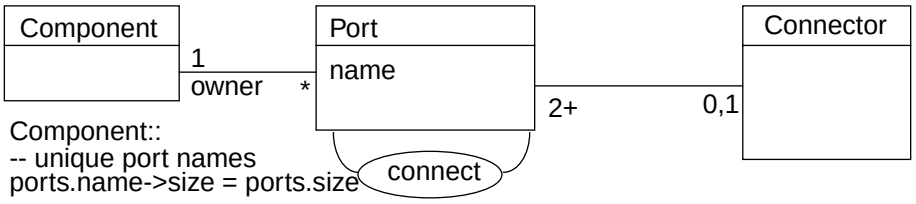


Figure 239: Basic port and connector model

The connect action between two Ports links them together with a Connector. One of the Ports must be unlinked; if the other is already linked, the existing connector is used. Other connectability criteria, not yet defined, must be satisfied as a precondition.

```
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. At most one Source Port may be coupled to the same Connector.

11.4.2.2 Connector Design

One connector design using registration protocol

This is a design (one amongst many possibles) of gendered Ports, in which a connection is established by registrationTo distinguish the types of this implementation from the types they represent in the model, we have alter the names slightly.

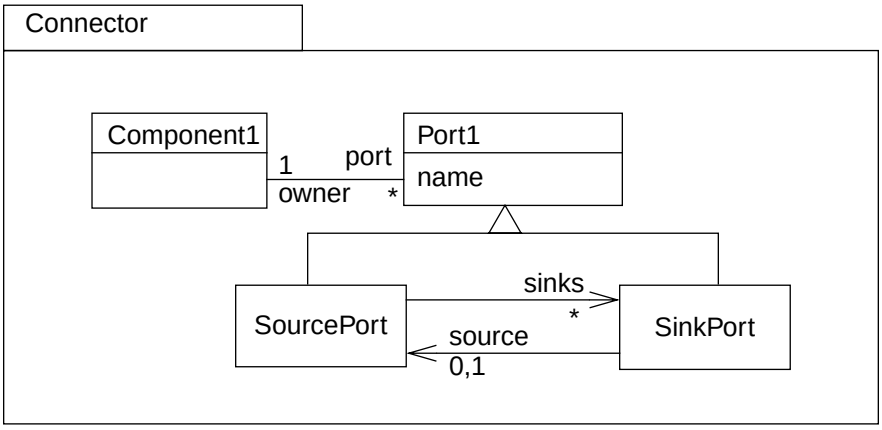


Figure 240: A design for connecting connectors

The connect action is realised as a collaboration between the owners of the Ports[1]. 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, surce)
post:    sink.source = source and
         source.owner–>register(source, sink)
```

Registration both records the source and port to which this sink is connected, and results in a message to the source-owner:

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

which adds the sender to the registry of sinks for this source.

A simple retrieval

Retrieval. The abstract model's Connector is realised 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 SinkPorts.

11.4.2.3 Interpretation of connector diagrams

A component diagram is "unfolded" based on this architecture

Lastly, we must define how the notation we've been using for components should be interpreted in terms of our component model. We will define each box on a component diagram as a Component1-instance; each emerging arrow is a SourcePort; and

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

each ingoing arrow is a SinkPort. A connection between components is a Connector in the sense of our abstract model, which we've realised as a complementary pair of links between the ports.

So we can translate a typical fragment of componentry:

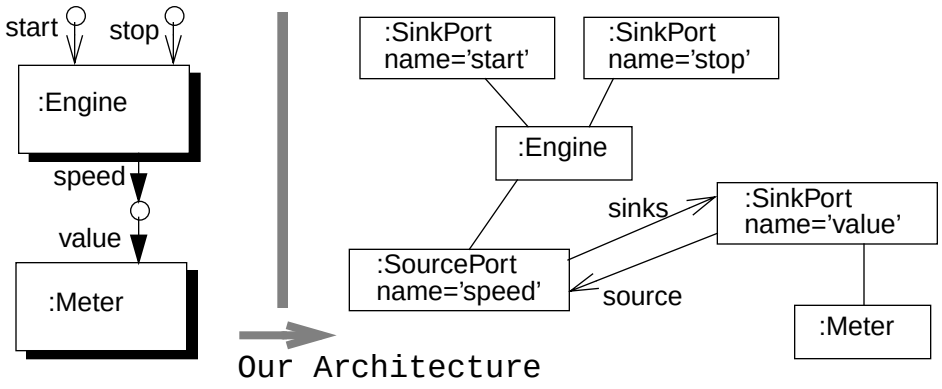


Figure 241: 'Unfolding' of a component diagram based on framework

This shows that an Architecture gives a meaning to a component diagram by making it an abbreviation for an object diagram.

11.4.3 Property connector

Having given a meaning to the basic idea of a connector, we can go on to define the different categories we are interested in for ARCH ONE.

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. Value changes propagate

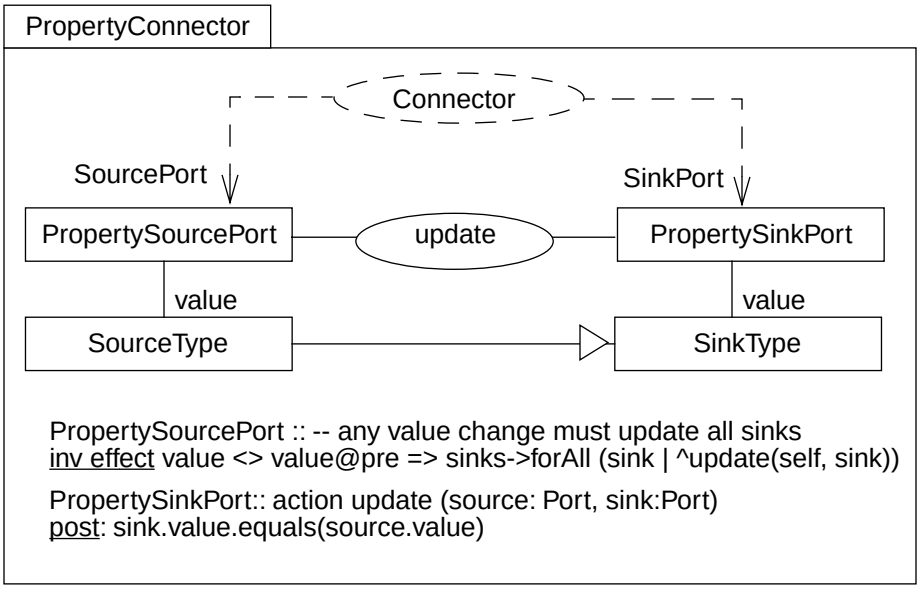


Figure 242: Template for a Property Connector

Property connections are ‘unfolded’ accordingly

Interpretation of Component diagrams. 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.

So if we want to see how our architecture interprets this example:

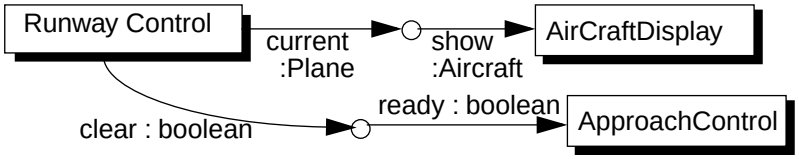


Figure 243: Component diagram to interpret

we can translate first to a framework-application:

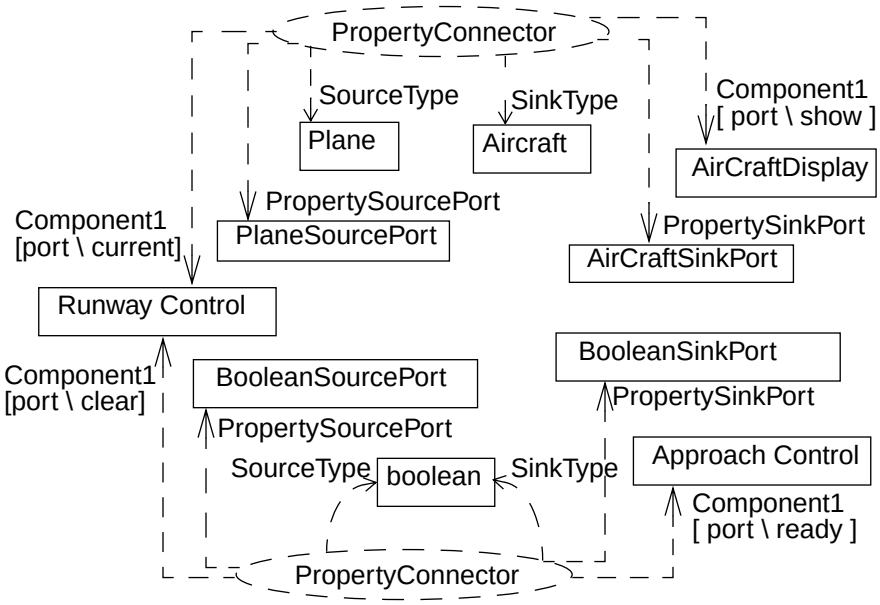


Figure 244: Equivalent template applications

and then unfold the framework definitions:

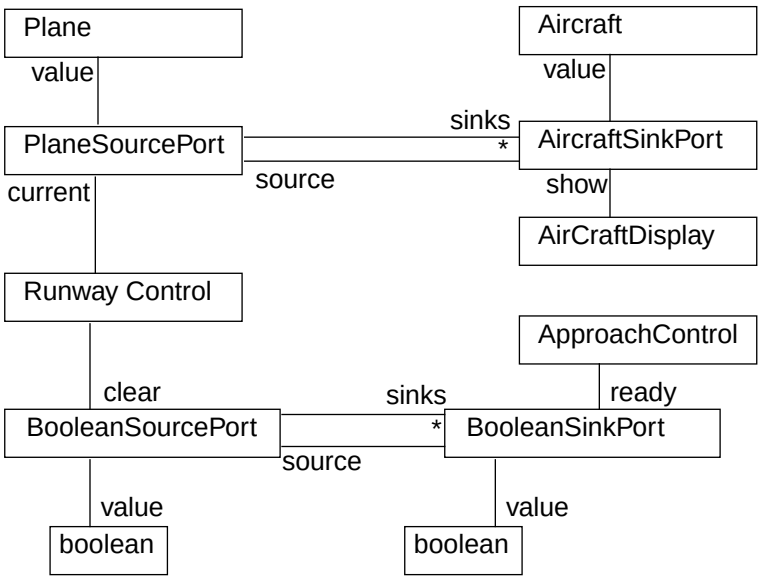


Figure 245: ‘Unfolded’ version of component diagram

The relative complexity of this is persuasive of the utility of the component notation. The same technique can be used to give a meaning to any additional layer of notation, not just to components. All this could be defined in a syntax section of the architecture package itself, with a suitable mechanism for defining a visual grammar (See Chapter 15, *Frameworks*).

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

11.5 *Specifying ARCH ONE Components*

Specs are higher-level, thanks to the architecture definitions

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 (marked «output property», «output event», etc., or the equivalent solid arrow notation) in a specification implies that all this is assumed. A component specification takes the form of a single type-description; but as always in Catalysis, it may be refined and implemented as a collection of objects.

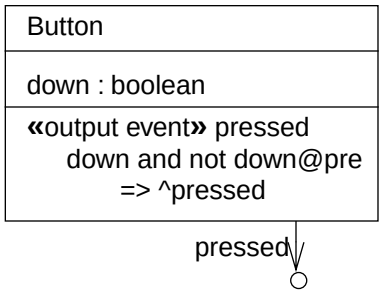
Let's now look at the main categories of port in our ARCH ONE architecture one by one, and see how to use specification techniques to define them.

11.5.1 Specifying input and output events

Input events are specified just like operations

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 run-time effect: an object designed as a component will accept events according to the protocol defined in a chosen component archiecture (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 adaptor object, are defined by the implementation of the architecture).

Output events can be specified by state changes



An output event occurs when some 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 (Section 4.8.5, "Effect Invariants," on page 171). Notice the caret mark, denoting "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 may have arguments, which deliver information to the receiving input events. The usual type matching rules apply.

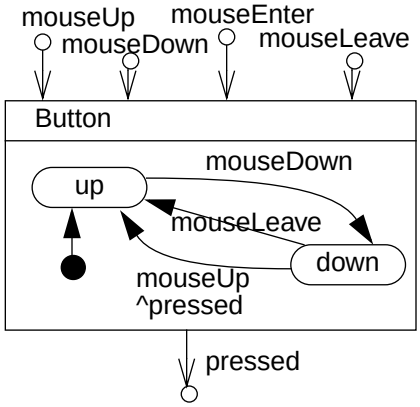
Certain details of an output event are not specified in this style. For example, how soon after a qualifying transition takes place, must the output be made?[1] Nor have we said who will receive the output, since this will be different for each design in which the component is used; the architecture guarantees all connected ports are notified.

1. We can easily add performance specifications, either informally or formally

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, mouseLeave. The latter two happen if the user drags the mouse in or out of the Button's screen area; and the Up and Down messages are only sent when the mouse is within that area.

Output events may be further conditionalized

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



The required behaviour can be readily illustrated with a state-chart.

A state-chart can make a convenient spec

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)

Some useful abbreviations for events:

<<input event>> action (params)action(params)
<<output event>> operation(params)^
operation(params)

11.5.2 Specifying properties

11.5.2.1 Specifying output properties

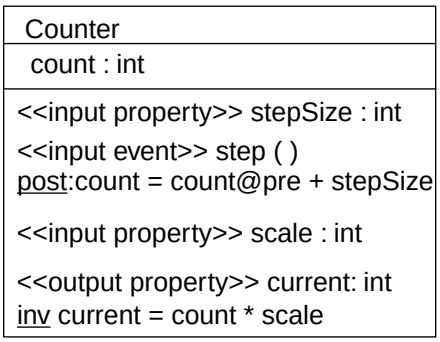
An output property is an attribute that the component architecture specifically allows to be visible to other components. A chosen component architecture will provide a pattern for implementing attributes tagged with the «output property» stereotype[1].

Output properties are visible

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

1. e.g. the get/set method pattern used in JavaBeans for component properties.

Attributes are not necessarily visible, while properties must be



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 does not need to be implemented directly as a stored variable, but may instead be computed when required.

11.5.2.2Specifying input properties

Input properties can be automatically controlled

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.

They should not be explicitly changed

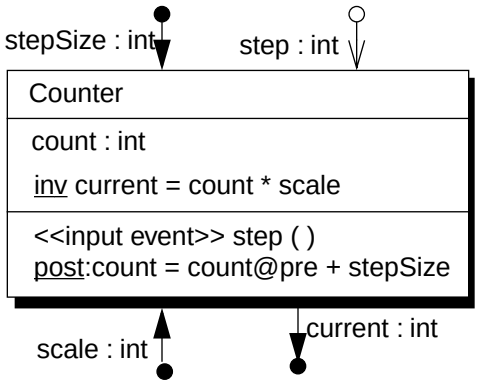
There is an implied invariant, that an input property will be equal to whatever output (of some other component) it may be coupled to. Therefore, although an input property can be used in a postcondition, it doesn't make sense to imply that it is changed by the action:

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

It's value tracks the single output property it is coupled to

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

And a visual notation



Pictorially, input and output properties can be shown with the solid arrows, distinguished from the open arrows of events. (The shadowing emphasises that this is a component — that is, something intended to be implemented according to some component architecture. But it's just 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 this 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.

If the property changes to point to another object, that would normally be a 'significant' change. If the object pointed at changes its state, then it depends on whether the property object's state is considered part of the state of the component. This is the same issue as the definition of 'equals' in Section 10.7, "Templates for Equality and Copying," on page 419.

11.5.2.3 Require-condition

A require-condition is an invariant which 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, will be maintained by the component. A require-condition governs the relationship between properties. A typical one might be:

'require' is an invariant obligation on the client

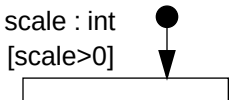
require scale * stepSize < 1000

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

11.5.2.4 Constrained connectors

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

Inputs can be constrained



«input property» scale : int
constraint scale > 0

Different component architectures can 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.

One approach: treat as 'require'

Alternately, the architecture may 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 like change_request_port_x (new_value). By default, this will return 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: device a 'veto' protocol for changes

Alternately again, the architecture may support an exception / transaction abort scheme. This could be described using the techniques for describing exceptions in Chapter 14.

A third: transaction / abort upon violation

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

11.5.2.5 Bidirectional properties

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

11.5.2.6 Port attributes

The Catalysis metamodel gives every port several attributes:

ports have accessible attributes

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 where 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, true iff true for all the currently-coupled inputs.

So a spec can constrain port configurations

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

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

11.5.3 Specifying transfers

Transfers are also defined in the architecture

A transfer connector sends an object from the source component to the sink. Unlike Events and Properties, each «output transfer» port may only be connected to 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 characterised by the type of object transferred.

11.5.4 Specifying transactions

As are transactions

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

11.6 *Connecting ARCH ONE Components*

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

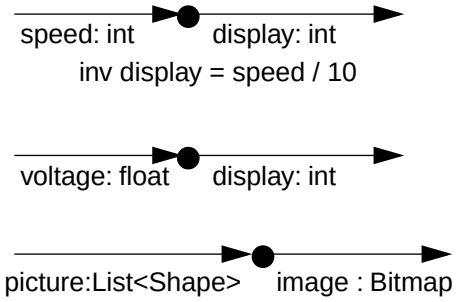
11.6.1 Connector properties

An input property can be driven by an output property, and each output may generally drive any number of inputs.

Output properties drive 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 may have different labels.

It is sometimes useful to make simple transformations between the 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 may be done either by an appropriate small component, or by some flexibility in the architecture permitting 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.



Simple transformations of property values can be made implicit

In our notation, transformations can be shown either 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.

11.6.2 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.

Many-to-many event connections

An output event has a name and a set of arguments; an input event has a name and a set of parameters. The simplest connector is when 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.

Parameters may map directly

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

...or may need a transformation

An output event may transfer information in two directions, via parameters and the return-value normally associated with function calls. For that reason, an output event may have a postcondition at the sending end[1].

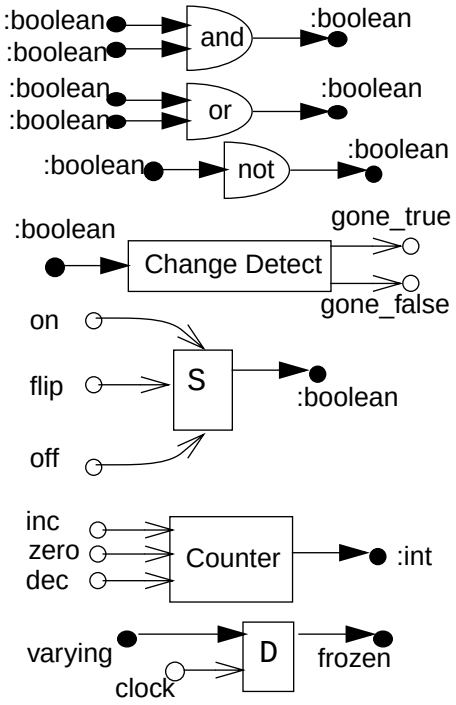
Output events can require a post-condition

11.6.3 A Basic kit of components for ARCH ONE

Here is a small, flexible component kit

A basic kit of components can be defined, to which more domain-oriented components can be added. Here's a selection that can be used in lots of ways, just intended to give a general flavor of what can be achieved.

Some components inspired by hardware



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.)

Property to event conversion: in general, properties and events cannot be coupled.

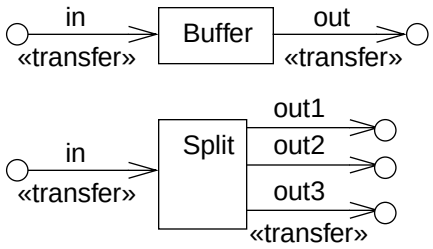
Change Detect generates events upon transition of a boolean property.

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

The Counter tracks inc, dec events.

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

Some 'transfer' components.



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.

11.6.4 Dynamically creating and connecting components

11.6.4.1 Connecting components

You can refer to port connections in a spec

The port attributes allow us to specify a connection in a postcondition. For example,

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

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

11.6.4.2 Creating components

...and create new components and connections

Components are instantiated the same way as types are in a postcondition, using ComponentType.new. Consider an operation that causes a component to 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 component architecture may dynamically track and intercept

A particular component architecture (e.g. COM+) might intercept the instantiation and connection operations (remember, many of these need to be a part of the standard infrastructure services; Section 11.1.4, “Components and Standardization,” on page 443). That component infrastructure can monitor the known components and their connections to provide richer extensions of behavior.

11.6.4.3 Visual Notations

All the standard notations for dynamic creation of objects, links between objects, and cardinality constraints on the connections, extend to components as well.

11.7 *Heterogenous components*

We more often need to compose ad-hoc components

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

This may be a bit more difficult

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.

The work is more bottom-up

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 are going to nail together the pieces you are given to achieve your goals.

Catalysis refinement and retrieval can make this systematic

These problems are considerably alleviated by building a requirements model, and then relating the assembly of components back to the requirements using 'retrievals'. Used as a systematic approach (Section 14.2.1, "Multiple Routes through the Method," on page 526), this 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 which is inherent in component-based design.

11.7.1 A Requirements Spec

Lets try an example

Let's suppose we are starting a company that sells, say, office equipment. There will be no showroom: just a catalog mailed to customers, and a warehouse, and telephone sales organisation. We want to put together a system to assist its operations.

11.7.1.1 Requirements model

Here's a quick & rough model of what we'll need to 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, dispatchShipment; there are others that are more like queries, looking up some information: findOrderDispatches. Elaborating a bit on one sample:

A straightforward domain type model

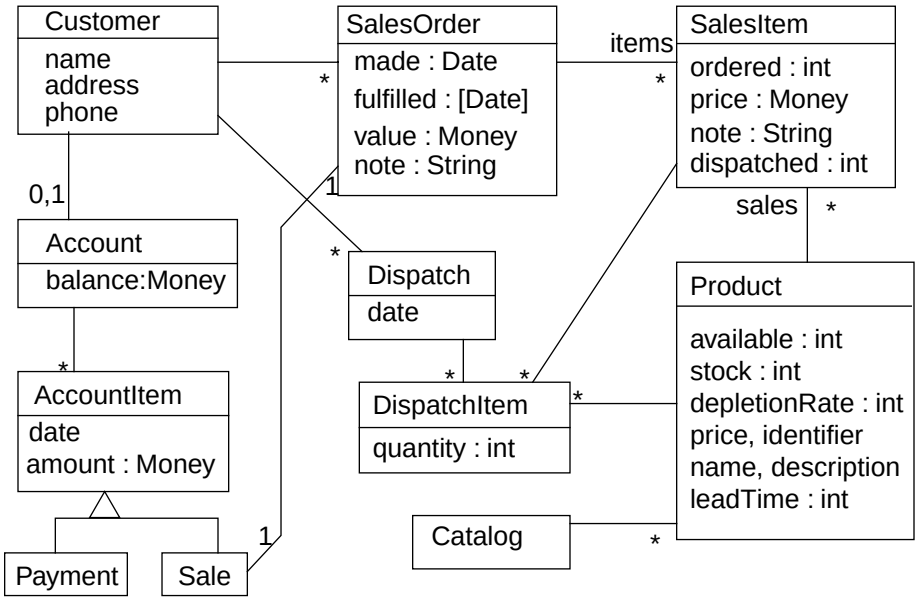


Figure 246: Business model for sales company

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

11.7.1.2 Business Rules

When a sale is made, sales staff create an appropriate Customer object (unless there is one already in existence), 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:

Some invariants and effect invariants capture business rules

```
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 which 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 — which may have been ordered, but not dispatched yet. Availability can go 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
```

and we must 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 of course observe this invariant and set fulfilled to something other than null, once the order is fully satisfied. But we should also say that the 'something' ought to be the date on which this happened:

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

Account – Order tie-up

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 sales staff keep 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. Some do this with sticky notes; others use electronic organizers. A Contact here is an occasion on which a Customer was spoken to or mailed; Customer includes potential customers who have made no order yet:

Also track contact info

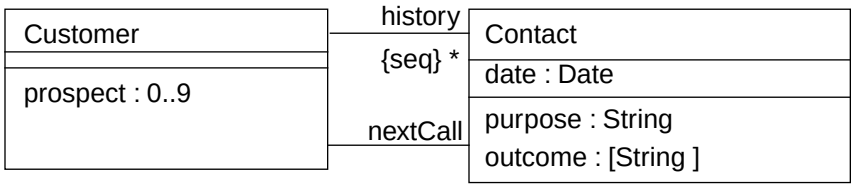


Figure 247: Contact-tracking model in sales business

11.7.1.3 Target Operations and System Context

The new system should support sales, dispatching, and accounts staff.

The system context

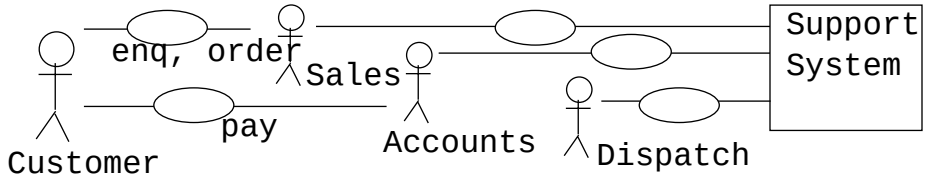


Figure 248: Target context diagram

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, 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 what orders there are, and create Dispatches.

11.7.2 A component-based solution

We have no time to build a custom 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.

We identify some existing applications

Our chief designer immediately recognises the contact-tracking requirement as corresponding 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 sales person can keep her own contact database. There are several mainframe-based general accounts systems; and she knows of an ordering system that can be brought in and adapted quite quickly.

And plan the solution architecture

In the interests of rapidly approaching deadlines, therefore, separate systems are set up to accommodate the above requirements.

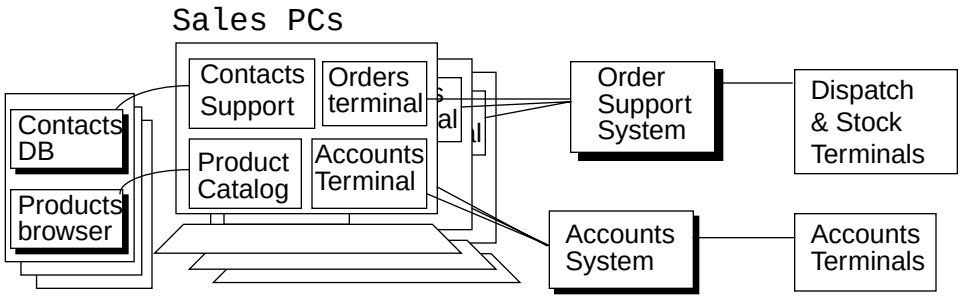


Figure 249: Heterogenous component architecture

Each Sales person works at a PC running four applications: their own contacts database, tracking customers in their assigned regions; a products catalog browser (hastily constructed as a local Web site); and a virtual terminal each to the Orders and Accounts systems — to which the staff in the warehouse and accounts department have their own user interfaces.

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

11.7.2.1 Model of the whole

Let us first build a model of the components in the complete solution, as envisioned. 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.

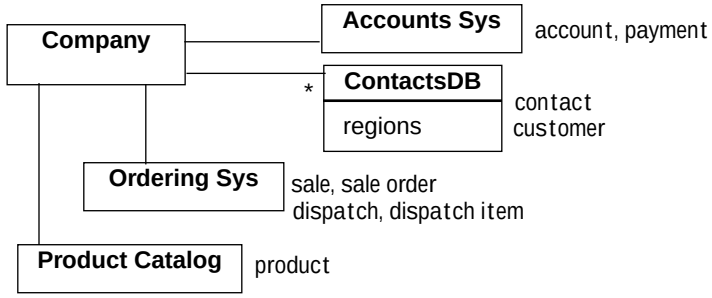


Figure 250: Large grained components modeled

11.7.2.2 Models of constituent components

How to extract type models?

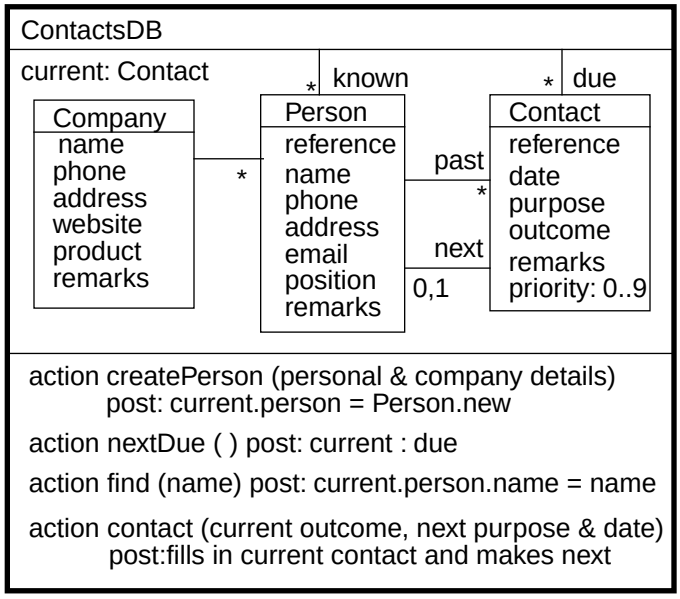
Before we can plan any meaningful interaction between the existing components, or start to develop 'glue' code, we need some model of what they do. Of course, none of them comes with such a model handy!

So we do some reverse-engineering

By a mixture of experiment and reading the manuals, we build a behavioural model for each component (since their designers have all omitted to provide one for us). This procedure is highly recommended when adopting a component made elsewhere; the same applies when reviewing an aging component built locally. The exercise clarifies your understanding of the component, reveals useful questions about its behaviour that you can do further research on, and also tends to make it clear where its shortcomings lie.

Contacts component

The Contacts system has this type model. 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.



The reference fields are where you can put a unique reference number that can be used externally to identify objects.

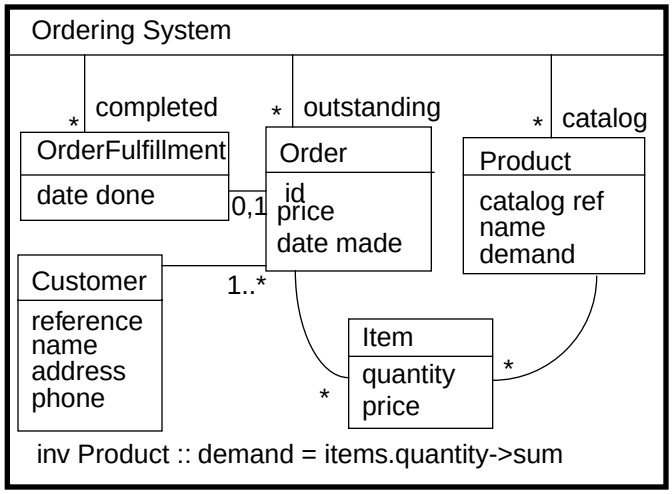
Like many simple data storage applications, its operations don't seem

very interesting: they are mostly just different ways of entering, searching and updating the attributes. There is a current 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, etc.

When a contact is made (call, email exchange, etc) 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 leaving it to them to call when they're interested: sales staff insist that such a course of action is unthinkable.

Ordering component

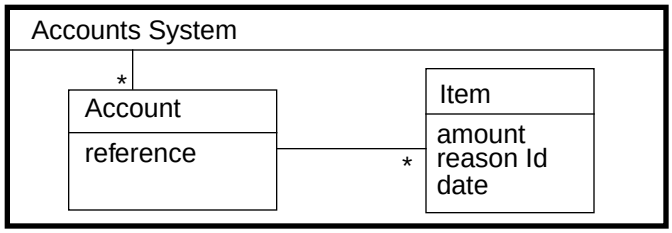
The Ordering system attaches Orders to Customers, and provides information about the demand for Products, though not the actual stock.



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.

Accounts component

The Accounts system keeps a record of payments against accounts. The reference attribute of an account enables it to be cross referenced to external records.



11.7.2.3 Retrievals

These models *must* map to the domain model!

How can we reconcile this disparate collection of models with the requirements spec we started with? First, we must do "retrieval": the task of checking that all the information required by the spec is there somewhere in the implementation. Looking at each of the queries in the requirements type model, how is that information represented in our design?

In the implementation, Customers' names, addresses and phone are kept in two places: the Contacts database and the Ordering system. Accounts are elsewhere; and Sales. We will need some way of keeping them in step. We combine the requirement and component models, and document retrievals and cross-component links (only parts shown here). Stereotypes based upon template packages have been used freely.

A 'customer' is implemented in two places, which must be kept in sync

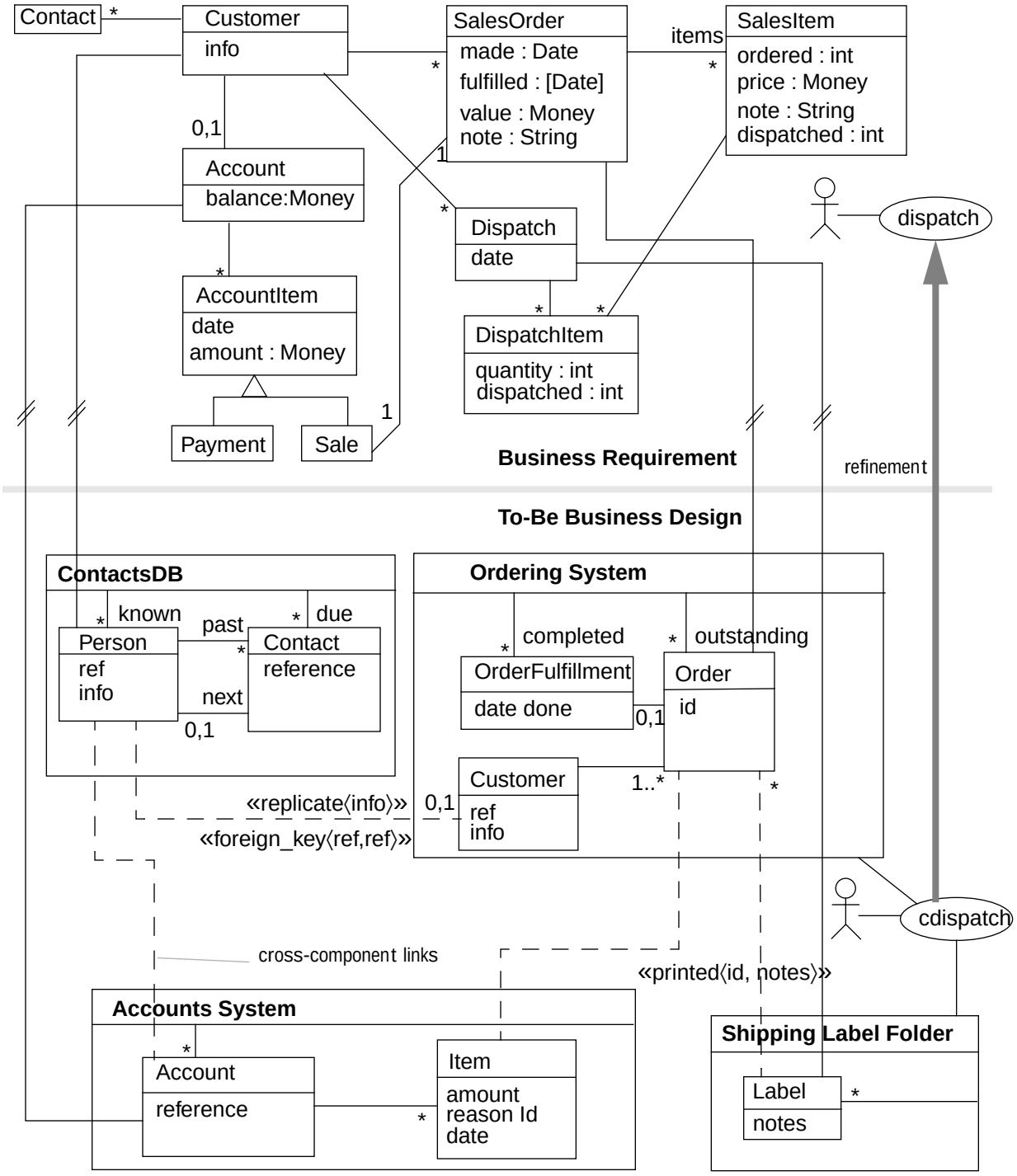


Figure 251: Map component models and new business proces to requirements

Here are some of the main constraints and retrievals:

Customer:: -- for every customer i.e. in the requirements model
-- must have a matching entry in the contact DB for that region

```
person.contactDB.regions->includes(address.zip)

-- if there are sales orders, there must be a customer (and orders) in the Ordering system
salesOrders <> 0 implies person.customer <> null
    and salesOrders.orders = person.customer.orders

SalesOrder:: -- for every (requirements) sales order
    -- if fulfilled, there must be an OrdefFulfilment
    fulfilled <> null implies order.orderFulfillment <> null
```

Some other spec types are represented differently

The prospect scores are in the Contacts base: no problem, since they are only used by the actions involving sales people. The link to sales orders is in the OrderingSystem: the requirement's SalesOrder is represented by the OrderingSystem's Order.

And we can negotiate system vs. actor responsibilities

Dispatches, keeping track of stock and what has been sent, don't seem to be there at all. We could either decide to factor something into the implementation, or make the dispatchers work with a paper system. We deal with this loose end manually: any time a shipment is made, the agent will write in the 'notes' field on the shipping label the reference id numbers of all orders (from the Ordering System) that had an item included in that dispatch. We keep a folder of all shipping labels.

This is shown in the bottom right of Figure 251. Notice how correctly the 'subjective' interpretation of 'containment' works: not all labels have the link to orders; but all those in the Shipping Label Folder are supposed to. Also note the line from Dispatch (requirements) to Label (today's business); it mandates that whenever a new Dispatch takes place, the corresponding label must be updated.

11.7.2.4 Implementing cross-component links

Associations must now cross component boundaries

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 the reference fields to cross-link them, as 'foreign keys'.

In effect, reference numbers, identifiers, keys of all kinds are implementations of links that cross system and subsystem boundaries:

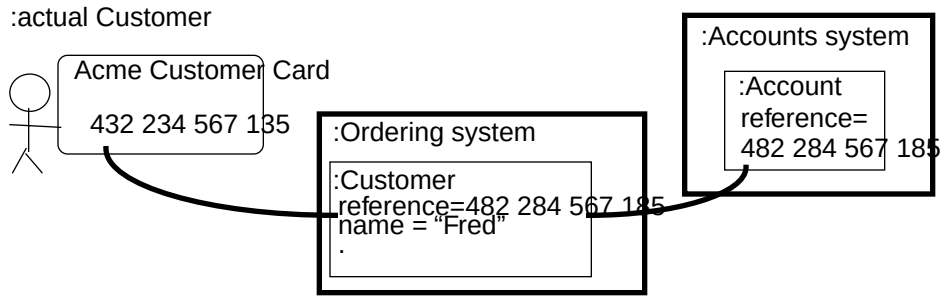


Figure 252: Cross-component links

Now it's easier to see the spec's Customer-Account link, as implemented here. But (another note for the detailed design) it does depend on using the reference fields consistently.

11.7.2.5 Implementing actions and business rules

Only after the retrieval can we meaningfully plan operations

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 for. We will need to co-ordinate the business transactions across our ad-hoc components. Let's look at makeOrder, addOrderItem, and confirmOrder.

makeOrder

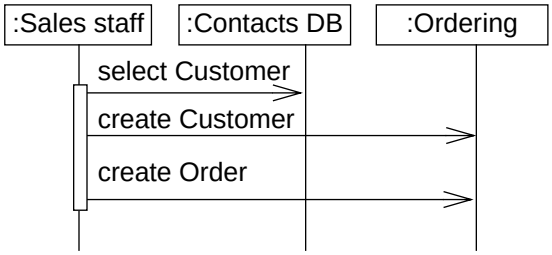
The requirement for makeOrder is to "create a new Order for the currently displayed Customer". Now, in our hastily-contrived system, there can be two Customers displayed on a sales PC screen: one in the Contacts Database, and another in the Ordering system. Because the components are entirely separate,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

joint-actions in the spec give us design flexibility

But makeOrder is an action: a specification of something that must be achievable *with* the system, though not necessarily something it must take the entire responsibility for. Remember that we have modeled it as a *joint action* (Section 5.4, "Abstract Joint Action = Use Case," on page 211), and the responsibility partition has not been decided. So here's our implementation of makeOrder:

Part of the action must be done by the user in this design

- The sales person 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 sales person uses the Ordering system's Order-creation operation.

addOrderItem

The Ordering system provides this action directly; although you have to first look the Product up in its list, which carries less information than the separate web-browsable Product Catalog.

confirmOrder

The spec says "enters payment details, such as a credit card or payment on account". And according to one of the business rules (page 312), 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); and where people are involved, this generally means relying on them to do the right thing.

findOrderDispatches

How would we describe the use case findOrderDispatches in our new business process? Lets 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 regardless of 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.

11.7.3 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 into the machine from the staff.

The next release will automate some more

Sales staff will now work through a Sales Client component. 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 the sales staff use. It implements some of the cross-component business rules such as the Account-Order tie-up, eliminating that area of human error.

Exactly one component is upgraded

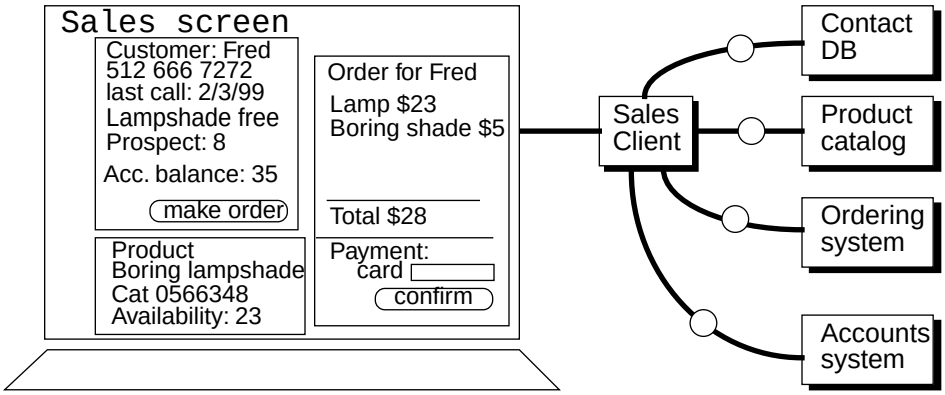


Figure 253: First revision: adding glue components

In this design, the Sales Client (which runs in each sales PC) integrates the bought-in 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 warehouse staff) can integrate the Ordering system with a Dispatch-tracking system and Stock control.

All interaction is through a 'glue' client component

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

The 'glue' bit takes over more responsibility from the user

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.

11.7.3.1 Heterogenous component architecture

Glue and wrappers should be kept simple

It is characteristic of bought-in 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" like Sales Client tends to be written to couple to specific components. Nevertheless, if the glue can be kept minimal, adaptations are not too 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.

11.7.4 Federated architecture

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

Federation example.

Improving our system still further, we write our own components. We divide up the functions of the Ordering system. One part runs on the sales people's PCs, and the other runs on the dispatching department's machine.

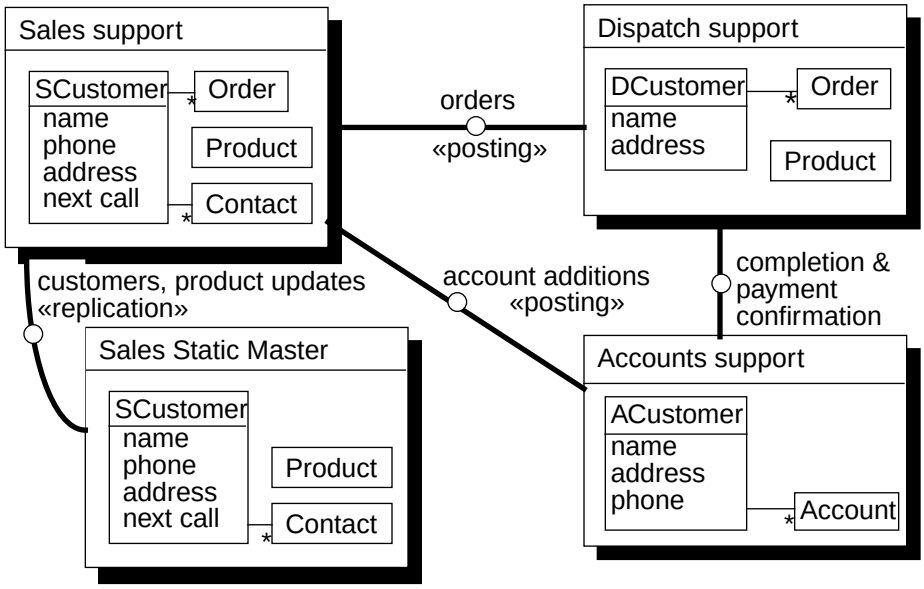


Figure 254: Eventual version: federated architecture

Connectors

Each Sales support component has a list of Products and Customers, and can generate Orders. Orders are (either immediately or in batches) sent 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 a connector category. Similarly «posting», which is about appending an item to a list which may not otherwise be altered.)

Connector categories

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

Type Views

The benefits of federation include decoupling of outage: each staff member can carry on working even if other machines are down; 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 sales person and other parts of the system, so they could work from home.

Federation decouples.

On the downside, some replication is necessary, of Product catalog, and Customer database. However, disk space isn't so expensive these days, and the technology of replication has been much developed in recent years, particularly since popularisation by Lotus. (Replication means ensuring consistency between datasets using intermittent updates, such as when a user logs in occasionally.)

Federation requires replication.

11.7.5 Summary: Heterogenous Components

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

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

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

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

Pattern-3 Extracting generic code components

Summary Re-use code by generalising from existing work to make pluggable components.

Intent Make an existing component re-usable in a broader context. Resources have been assigned for work outside immediate project need (Pattern-4, *Componentware management* (p.487)).

Considerations It is often better to make a generic framework model; this needs less investment in 'plug technology', needed to make code-components plug together. A framework model just provides the specs for each class that fits into a framework, and is typically specialized at design time. Better for performance; less run-time pluggability.

It takes hard work to find the most useful generalisation that fits many cases, and to get it right. This is only worth it for components that will definitely be re-used at least four or five times; and the investment doesn't pay off for some while.

Strategy **Components cross projects.** It is unusual without much experience to design a good generic component in advance; they become apparent only after you find yourself repeating similar design decisions. It is also not much use to find components in isolation: they work best as part of a coherent kit. Within a small project, there is not much payoff: it's easy to spend too much time honing beautifully engineered components that aren't used anywhere else. It's therefore clearly a long-sighted architectural job to decide what components are worth working on and integrating into the local kit.

Identify common frameworks. Pull out objects and collaborations that have common features into separate framework packages, and re-import them to apply.

Don't overgeneralise! 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, so do.

Specify plug interfaces. Define what you need of anything that plugs into you — and what you provide to them. Provide the simplest model that makes sense. 'Lower' interfaces generally need to know less than 'upper' ones.

Package. Your component should be delivered with its:

- plug specifications;
- test harness for clients' plugs — based on the plug spec. Either:
  - stand-alone harness that drives the plug-in & pronounces judgement; or
  - switchable monitor that checks pre and postconditions during operation.
- some selection of demo or default plug-ins.

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

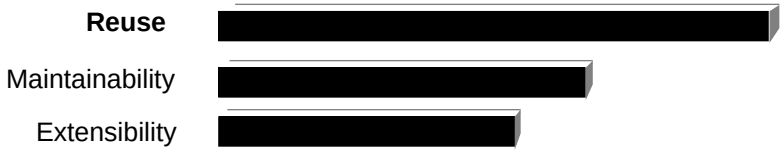
Pattern-4 Componentware management

Summary

Devote resources to build, maintain, and promote usage of a component library; there is no free lunch.

Intent

Re-use motivates uptake of OT. Surveys show that of the main three motivations managers quote for taking up object technology, re-use is the leader.

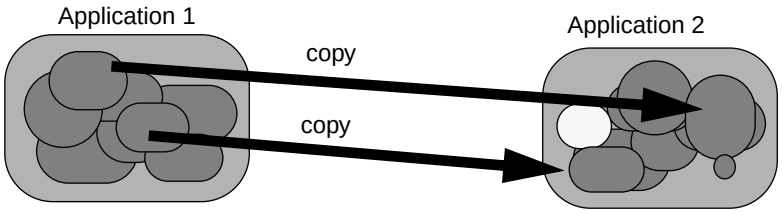


But objects do not automatically promote re-use — it is an enabling technology that will reduce costs if well applied. If badly applied, it increases costs. These costs always increase in the short term: investment is required to move to a re-use culture. The good news is, there are a growing number of success stories when done right.

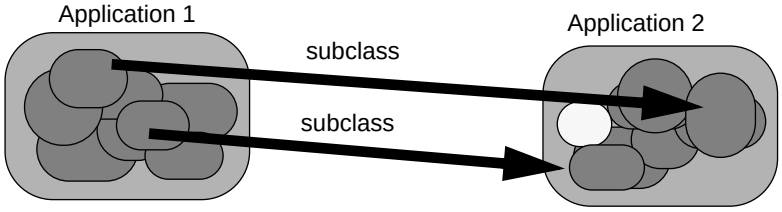
Considerations

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

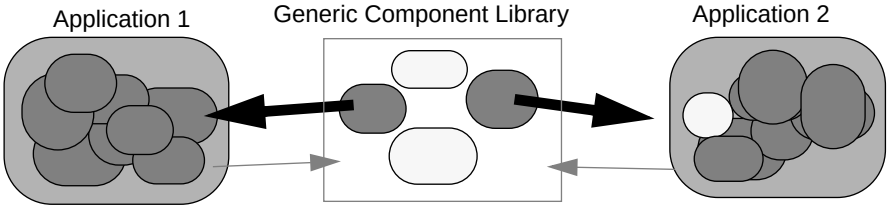
What is re-use? Cut & paste? — "Adopt, adapt, and improve"? Cheap and easy, but limited benefits; enhancements to the original do not benefit the re-users at all.



- Programming by adaptation? — if it looks similar, inherit and override methods as required. Takes more effort, gives limited benefits; Unless you adhere to strict rules, superclass enhancements need review of overrides in subclasses.



- Building **generic components** and import them to re-use— devote substantial resources to a library. Best benefits, and most investment required.



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

Strategy

- Apply the spiral model, with conscious activities for abstraction and re-refinement.
- Develop a re-use team who know and develop the library. Use people with a perfectionist turn of mind, and the right skill set.
- Offer some re-use team people to development projects, to encourage re-use.
- Do not under-resource; it is your design capital.
- Apply to all stages of the process: models, patterns, frameworks, designs.

Pattern-5 Build models from frameworks

Summary

Do not build models from scratch, but by composing frameworks.

Intent

Generalisation of modelling work, to start focus on re-use and componentware as early as possible (Pattern-4, *Componentware management* (p.487)).

Considerations

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

A framework can be just a specification, or can come with a code implementation (Chapter 18, *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 have to implement the result; it does not have the same run-time overhead of pluggability.

Strategy

Model-only strategy. Building and using model frameworks.

- Look out for similar patterns within business models, type specifications, high-level designs. Also make frameworks for common design and analysis patterns.
- Extract common models and use placeholders, effects, invariant effects, abstract actions to allow you to separate parts.
- Compose framework with others. Where one type has definitions from more than one framework, use join composition (Section 9.4.4, “Joining action specifications,” on page 373).
- Implement the composed framework. Each type in the composition will have a spec, which can be implemented as for basic design.
- Use a tool that will help you compose model frameworks.

Pattern-6 Plug conformance

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

Intent Ensure that two components you’ve aquired (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 need to ensure one matches the other.

How hard this is depends on whether they use similar terms. If one is actually designed specifically for the other, it’s easy. If not, but they are based on the same business model, it’s not so bad. If the models are entirely different, there’s more work to do (e.g. see Section 11.7, “Heterogenous components,” on page 472).

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

Strategy Document a refinement that shows how one meets the other. See Chapter 12, *Abstraction and Refinement*.

Pattern-7 Using legacy or bought-in components

Summary

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

A uniform strategy across component boundaries, including legacy components.

Some of your software may be in the form of a bought-in or legacy component. It may be an infrastructure that you use to serve your 'middleware'; or 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, stock control. A component is bought in 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 not, 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 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 some kind of handle that both map internally to their own ends of the links. So there may be some reference number used to identify members at the API of the membership manager; this 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. These need be created only when 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 go on to processing 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.

11.8 *Summary*

The Catalysis approach to design is about standing back from the detail. This enables you to think about and discuss the most important parts of your design without the clutter of fine detail; and to prolong the life of your design by making your overall vision clear to maintainers, enhancers, and extenders.

In earlier parts of this book, we saw how to use models to abstract away from the details of data structures. 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 has taken the abstraction one level higher. We have defined a notation in which components — separately deliverable chunks of software — can be specified and designed, and plugged together to make bigger components and complete systems. We have also defined a variety of ways in which components can be intercoupled, abstracting away from the details of the connectors; and set a framework for the definition of more categories of connector.

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