4. Designing Component Kits and Architectures with *Catalysis*

Alan Cameron Wills

4.1 Introduction

Copying code from existing designs does not work well as a strategy for reuse. Gluing components that were never designed to work together produces clumsy and inflexible designs. Real flexibility is achieved with coherent kits of components from which families of products can be constructed.

What are the design methods we should use to choose components? What constitutes a "coherent kit", and how do we define one? How do we cope with the reality of existing assets? How do we work out how to couple components developed to different standards?

This chapter outlines how to define component kits using the Catalysis techniques [1]. Catalysis is an approach to component-based development that is gaining increasing use among industrial developers of component and high-integrity software.

4.2 What is a Component?

The idea of component-based development (CBD) is that we assemble software end-products from reusable components [6]. Each component is designed to work in a variety of contexts and work in conjunction with a variety of other components. Many end-products may be designed with the help of one component.

Components differ from modules

We have always carved our software up into digestible chunks: first so that the recompilation did not take all day, and then so that different people could take charge of different parts of the system. In a merely modular system, you know what your module is going to interface to; if there is any question about the detail of the interface to another module, you can peer over the partition and talk to its designer. But in a component-based system, you do not know who your component might be talking to: that is up to the people who use your component in their designs. Therefore we must be very careful about defining component interfaces — much more careful than we needed to be in more traditional methods of design.

Components have much in common with objects

- The state is encapsulated, the only access being through messages (procedure calls etc), with the benefit that many different components can implement the same interface, and each component's role in a collaborative organisation may be characterised with a separate interface;
- The key to good component-based design is separation of concerns, just as it is in object-oriented programming;
- There are instances, classes, subclasses and interfaces of both objects and components. A component class is its program code; the instance is its installation in a given context; a component class may be written in such a way that it can be extended with "plug-in" code, thereby forming subclasses;
- The most important feature of both object and component design is that interfaces can be described separately from the classes that implement them, so that a piece of code designed to work with a given interface can be used with any component/object that implements that interface. This feature, if properly used, is what gives both object and component designs their great flexibility (sometimes referred to as "polymorphism").

Components differ from objects in some ways

- Two components working together may be written in different languages and running on different machines;
- A component may have its own persistent state — a database, file system, etc;
- The interface to a component typically provides access to objects inside it, so that a call might be written *component.object.function( )* — rather than just the last two parts. This might be an illusion created at the interface: a component does not have to be object-oriented inside;
- The interface to a component includes the idea of outputs, rather than just the list of procedure-calls that is an object-oriented interface. For example, Java Beans can provide "events" and "properties" that other components' compatible interfaces can be wired into;
- In COM and Enterprise Java Beans, a component instance is installed in a *container*, which provides local context. Typically, the component provides business logic, and the container maps from logical to physical data structures. This is again a good separation of concerns;
- A component will generally be more robustly packaged than an object: more likely to check preconditions than rely on the caller, and able to give a sensible response outside its normal operational range.

4.3 Families of Products from Kits of Components

Let us look at this example in Figure 4.1 of a system built from components. These components could be hardware or software. They are rather small components, but the main principles apply to large components too. The notation of the labelled connectors used here is from the Real-Time extension to UML proposed by Selic et al [5].

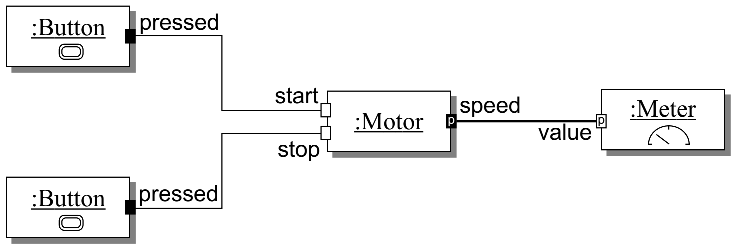


Figure 4.1. A simple system built from components

Component assemblers do not modify components When you buy or reuse a component, you want the benefit of it having been tried and tested. If you adapt an instance to suit your application better, you immediately introduce the possibility of new bugs, and rule out the option to accept any future fixes and enhancements published by its developers.

Therefore, we do not modify components, we employ "black box reuse". Instead, the larger strategy is to make components designed to be connected to others. The usual principles apply:

- each component should just perform one function well though it does not have to be a small function: a phone switch and a payroll system can be good components;
- each component should have clearly defined interfaces into which you plug other components and extend or parameterise its behaviour.

Preferably, the pattern of component non-modification should be enforced by making no source code available to assemblers. This works well for markets too, since component developers do not like to expose their designs.

Design connectors separately from components If components cannot be altered by their assemblers, then they must be designed in such a way that they can be wired together by some means. For example, after creating a Button-instance and a Motor-instance, we must be able to do something that connects the Button's "pressed" output to the Motor's "start" input.

There are many ways in which we could achieve this effect. As an example scheme, in Figure 4.2 we could decide that:

- the input and output ports of components are separate objects;
- each output holds a list of inputs that are registered with it; and,
- whenever a component wants to send an output, it sends a standard message to all the inputs registered with it.

To be an output port, in this scheme, means to accept a "please register me as an observer" message from input ports. This establishes the connection — clearly the instruction concerning which components to connect together must come from outside.

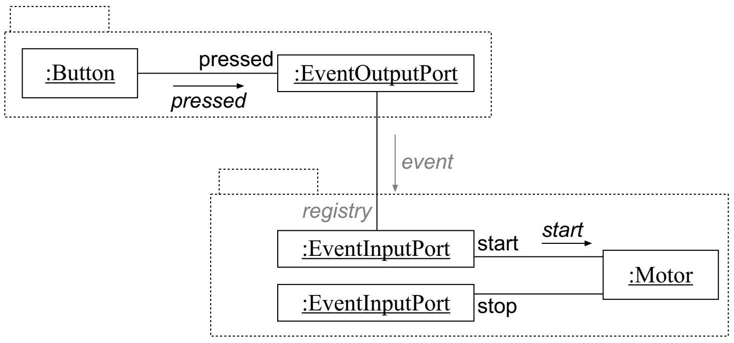


Figure 4.2. Representing the input and output ports by objects

So the component diagram is a higher-level view of this object-oriented scheme. Obviously, the component version is much more convenient for the designer who assembles the components, who should not need to worry about the details of the interaction. Such schemes also lend themselves to visual assembling tools: VisualAge, and Java Beans tools are examples.

Therefore, the connection schemes should be designed separately from the components themselves.

Of course, there are many possible schemes that could implement the properties desired of the connectors. The component specifications, and the systems built from the components, are independent of the connector schemes. Only the implementors of the components need to understand the details of the scheme.

Few connector types Figure 4.3 shows a larger system made from the same components, and a few others pulled out of the same bag. This kit is a bit like Lego: there are

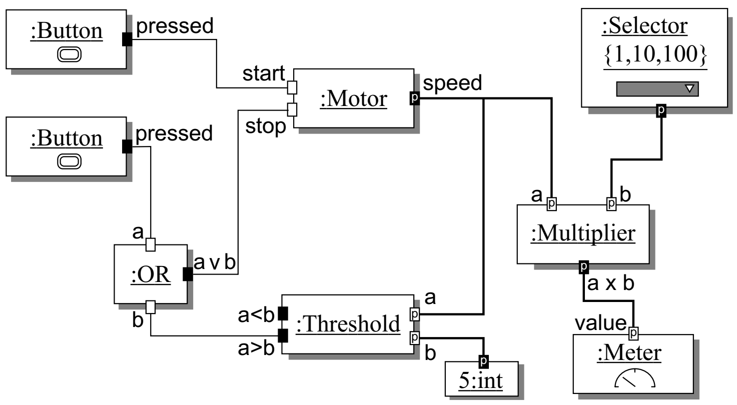


Figure 4.3. The example system of Figure 4.1 enlarged

lots of end-products that can be constructed from it. The essential feature of the kit is that many of the ports can be plugged into many of the other components' ports: the "pressed" output of a Button can be wired up to the "start" or "stop" inputs of a Motor, or the "a" or "b" inputs of an OR component. In fact, there are just two kinds of connector here: those that transmit single events, and those that transmit regular updates of continually-varying numeric properties, such as the speed of the Motor. Events and properties of this kind are standard in Java Beans.

In larger components (as we will see below), the connectors define more complex interfaces: the nature of the transactions that can be performed (such as purchase and work transfer) and the details of the protocol (such as sequences of messages).

To achieve the flexibility we look for in component-based development, we must provide components whose ports conform to a relatively small number of connector types.

Components come in kits A *kit* is a collection of components that have been designed to conform to a particular set of connector specifications. They do not necessarily come from one supplier, and have not necessarily been built all at the same time but they can be configured into working systems (or larger components) because their designers have all read the *Kit Architecture* — the document that describes the connector schemes. In fact, it is the kit architecture that defines the essential nature of the kit, more than its population of components.

Obviously, we cannot get the same flexibility by assembling components that were not designed to work together. Some companies may think that taking up CBD means setting up a procurement team to acquire useful components that could be assembled together. This sounds in danger of becoming like finding miscellaneous things from a junk yard: you can strap such pieces together, but a lot of "glue" is required. Unless you are careful, you will end up with many modules with individually crafted interfaces between them, rather than reconfigurable components.

To do CBD well, you must have a clear kit architecture.

4.3.1 Larger Components

The principles outlined above apply to larger components too. Examples include: the parts of an enterprise business system that support each business function; parts of a telecoms network; and, the work processing stations in a workflow system. The connectors between larger components will tend to be more complex transactions, in addition to single events and properties. Examples include: the transfer of a financial trade from one stage of the back office pipeline to the next; the purchase of stocks between trading systems or of power between electricity companies; the connection of a call through a telecoms system; or, the reservation of a machining resource by a process in a robot factory. All of these are existing protocols with well-defined standards in their respective businesses.

Kit architecture includes a business model In these more complex protocols, the objects transferred or referred to are not elementary values but, instead, things like trades, customers, orders, calls, and so on. It is important that all the components have the same ideas about what these things are.

Therefore, the kit architecture must include common definitions of these business objects. The definitions are not just data formats, they must include:

- definitions of the transactions that apply to the objects (making an order, paying it, delivering it); and
- the business rules to which all components must conform — whether, for example, an order may be delivered before it is paid for.

If these are not defined, misunderstandings and incompatibilities will arise between the components.

Notice that it is by no means necessary or desirable to make all the components use the same objects or formats internally. Most of them will have their own internal structures that suit them best, or are just there from history. Forcing designers of diverse components to use the business model internally will result in all sorts of strange perversions as local requirements are squeezed into the global format.

4.3.2 Component Strategies

A company can get into component-based development from a variety of directions:

families of products — an organisation may desire to be able to develop new variants of a basic system very rapidly, to keep up with changes in the market;

enterprise integration — a typical large organisation has a wide variety of systems developed over the years, many of which have their own point-to-point connections. The configuration is inflexible: it cannot be arranged easily to keep up with changes in the business organisation.

The requirement is to make these systems all talk the same language. The strategy is (see Figure 4.4):

- to define a component kit architecture; and,
- to wrap the different systems so that they have interfaces that conform to the kit architecture.

Notice that it is not sufficient just to say “we’ll make a CORBA backbone”: this provides a technical communications channel, but the systems cannot interact successfully until they have worked to a common business model, as described above. There can be so many systems in a large enterprise that it is not possible to define a model that suits everyone perfectly and not possible to take account of every system in the corporation. In this case, the model has to be open and extensible, and the medium (such as XML) has to be open too. It is also useful to think in terms of modelling “zones” — a zone is a region in which a particular model is adhered to — between the zones, gateways translate from one model to another. Zones are inevitable in a large organisation, since different groups will adopt their own local standards, and subsequently be reluctant to switch to another; the only solution is to allow local languages, but provide a common language where necessary.

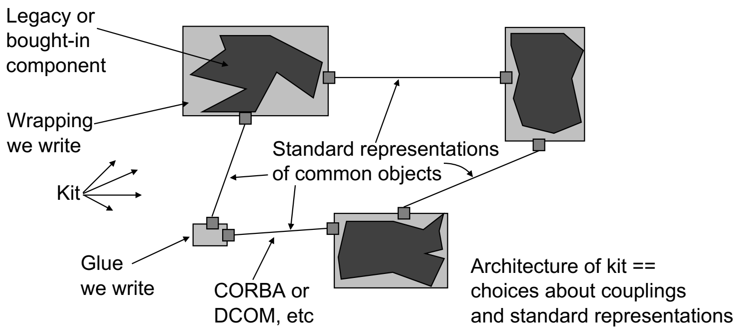

Legacy or bought-in component
Wrapping we write
Kit
Standard representations of common objects
Glue we write
CORBA or DCOM, etc
Architecture of kit == choices about couplings and standard representations

Figure 4.4. A system integrated using a kit architecture

Component roles There is a separation of design roles in component-based development, corresponding to the artefacts:

- component kit architecture designs the connectors;
- component design creates components that fit the kit;
- component assembly creates end-products (or larger components) from the kit;

Of course, the roles may be played by the same people but there are different emphases of skills involved:

- component architecture is a very skilled job, requiring strong insights into the future direction of the kit, and how to make it open and flexible;
- component design is a more careful activity, focusing on producing good general robust components that are likely to meet their specifications in a wide variety of configurations;
- component assembly is typically about working with users to meet their requirements satisfactorily and rapidly.

There is also a role of component strategist, deciding what components to populate the kit with, in order to be able to produce the products that will be needed in future. Note that this is not the same as a reuse librarian, since there is not necessarily any explicit component library with its implicit search mechanisms.

Parts in a component package A component will be assembled with others about which the designer has no knowledge. It will be up to the assembler to test it in this configuration. A component should therefore come with test and monitoring software. We also hope that it would come with some documentation describing what it can be expected to do.

4.4 Catalysis: Modelling Component Behaviour

Catalysis is a method for CBD. It uses the UML notation and adds to it a number of techniques for improving the precision of specifications, and tracing from specification through design to code. In summary, features of Catalysis include:

- abstraction — modelling is used to describe requirements, interfaces to components, high-level designs (as well as the detailed design). None of these can be translated directly to code, requiring other design decisions about, for example, the other interfaces that the component is required to provide;
- precision — models can be unambiguous (even though abstract): you can decide whether any particular component fits the model or not. Precision helps reduce misunderstandings between developers which, we have seen, is especially important in CBD. Writing precise models at a high level also tends to expose gaps and inconsistencies at an early stage in development;
- traceability — you can document the relationship between a model and its implementations, and work out how changes propagate;
- coherence — the various UML notations are used with specific meanings, and there are strong interconnections between them. This provides designers with more checks for consistency and completeness in the high-level models;
- reuse — Catalysis includes techniques for reusing modelling work, as well as executable components. Models can be constructed from powerful generic templates which are also good for defining component connectors;
- object and component design — Catalysis includes process patterns for developing object and component software from different starting points: greenfields, redevelopment, etc.

We will now describe some of the basic modelling techniques; we will then see how these are applied in specifying and designing components and their connectors.

4.4.1 Actions

The Catalysis term "action" corresponds to the UML use case, as shown in Figure 4.5. We use the same symbol, but attach a more specific meaning to it. It represents a task, message, interaction, transaction, job, process — anything that happens and causes changes over time. We use it not only in domain modelling, but also to represent interactions between components, between users and software, and between objects inside a design. The ellipse representing the action is linked to the types of object that participate in it or are affected in some way. Unlike the usual UML procedure, we do not immediately go on to describe a sequence of steps whereby a "buy" can occur. There are many possible such sequences: with credit card, mail order, cash in a shop, etc. Instead, we first focus on documenting the outcome that is common to all of these variants, called a postcondition or informally a "goal".

This approach allows us to document the most important things we know about the domain, without being pushed into more detail. However, we can be quite precise about what we have said. The goal uses a vocabulary about the relationships between the objects: the "ownership" of the Thing by the Vendor or Purchaser. We can draw this relationship as an association, and draw an instance diagram contrasting a typical situation before and after an occurrence of the action, as in Figure 4.6. The thicker lines show the "after" situation. Notice that there are no messages on this diagram; we are just showing the outcome at this stage.

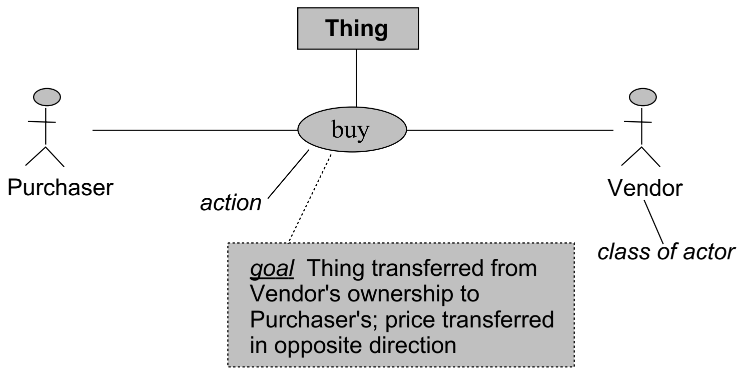

Thing
buy
Purchaser
Vendor
action
class of actor
goal Thing transferred from Vendor's ownership to Purchaser's; price transferred in opposite direction

Figure 4.5. A simple action as a domain model

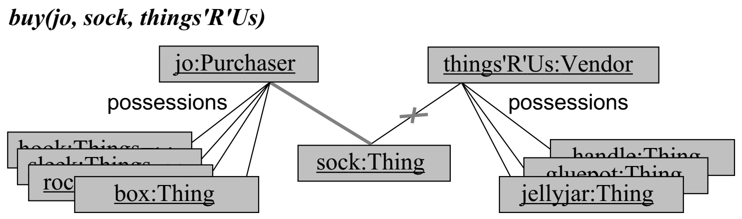


Figure 4.6. A before and after instance diagram

The "possessions" association in Figure 4.7 gives us a vocabulary that we can use more precisely in the description of the goal.

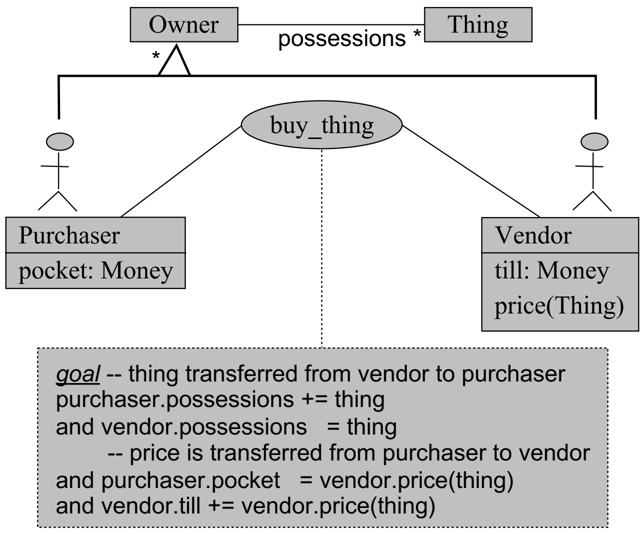

Owner
Thing
possessions *
*
buy_thing
Purchaser
pocket: Money
Vendor
till: Money
price(Thing)
goal -- thing transferred from vendor to purchaser
purchaser.possessions += thing
and vendor.possessions = thing
-- price is transferred from purchaser to vendor
and purchaser.pocket = vendor.price(thing)
and vendor.till += vendor.price(thing)

Figure 4.7. Elaborating the possessions association

Joint actions These actions are not particularly attached to any one of the participants. In object-oriented design, we attach each task to a particular object but in analysis and high-level design, we do not want to be forced into that decision too early: we want to be able to say what happens, without the exact details of who does what. It is an essential element of any design notation that it should provide ways of documenting the design decisions that have been made, without having to imply anything about the decisions that have not yet been made.

Coherence between logical and action models The rule in Catalysis is that the specifications of actions must use the vocabulary provided by the objects and associations in the static model. Though it is not conventional with less specific methods, we can draw the associations and actions on the same diagram. Tools such as Rose support this.

Coherence between statecharts, objects and actions In Catalysis, a state represents a Boolean attribute, and a transition represents an action. Many possible statecharts can be drawn about one type of object: a person can be awake, asleep, or dead; at the same time, and partly independently, they can be in various marital states, employed or not, and so on. Each of these charts could be drawn to help explain a different aspect of the domain, as in Figure 4.8. The states should be derivable as Boolean functions of

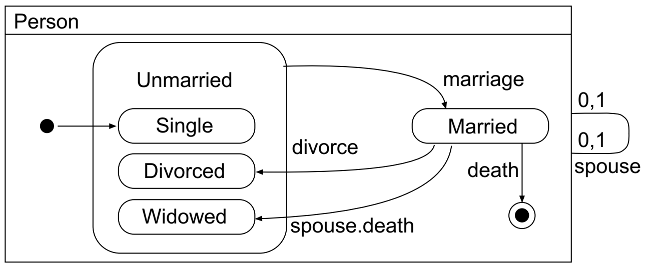


Figure 4.8. A statechart within a class

other attributes or associations, for example, *married = (spouse != null)*. The actions should appear as ellipses elsewhere in the model, and their pre- and postconditions should include the source and target states on the diagram, for example, postcondition of *marriage(this,x)* is *spouse=x*.

Drawing statecharts is a useful analytical tool for finding actions, as well as a useful presentation of the pre- and postconditions of some sorts of action.

There is no prescribed way of implementing a statechart in Catalysis, though there are some patterns, one of which involves an explicit state machine. But, in general, the statechart is a presentation tool, and each action is implemented independently.

4.4.2 Refinement and Traceability

We have a number of well-defined relationships between specifications of greater and lesser degrees of detail. One example is action specialisation shown in Figure 4.9:

actions can have all the same relationships as objects, including extension of specifications. An extension's postcondition is ANDed to that inherited from the "supertype" action.

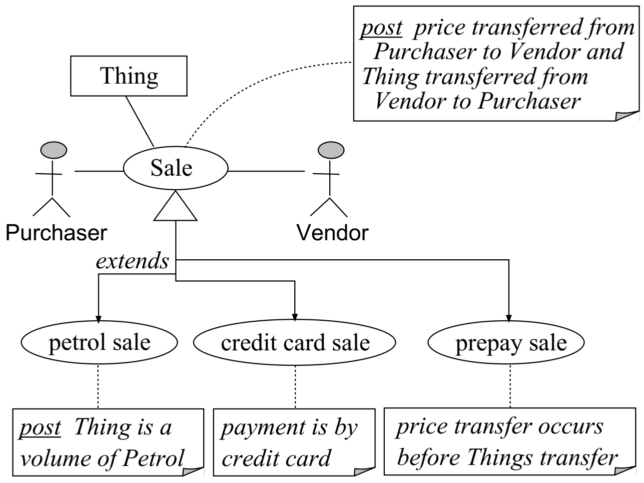

post price transferred from Purchaser to Vendor and Thing transferred from Vendor to Purchaser
Thing
Sale
Purchaser
Vendor
extends
petrol sale
credit card sale
prepay sale
post Thing is a volume of Petrol
payment is by credit card
price transfer occurs before Things transfer

Figure 4.9. Action specialisation

Decomposition All real changes or interactions happen over some period of time, and the purpose of an action is to represent these. All interactions can, on closer investigation, be found to be composed of smaller interactions. In these sequence diagrams, a vertical bar represents an instance of an object, and a horizontal bar an occurrence (an "instance") of an action. In Figure 4.10, the top left diagram shows two occurrences of a "buy" action. Looking closer (right hand diagram), we can see that, on this occasion, the buying was done by a sequence of three actions. Their postconditions in that sequence should together add up to the postcondition of "buy". We can "zoom in" on the detail of the objects too. The Vendor turns out on closer inspection to be a Sales, Distribution and Accounts department, and while you make an order with the Sales department, the delivery is by Distribution and you pay to Accounts. This view also enables us to see some of the interactions internal to Vendor that form part of "buy".

The notation here diverges somewhat from the conventional UML sequence diagram: because an action can encompass a transaction between any number of participants, the horizontal bars can touch multiple vertical ones. However, we actually use statecharts more commonly to describe decompositions, as shown below.

It is important to appreciate that these are not any kind of transformations: they are just views of the same underlying reality, with different levels of detail. The most abstract description is just as true a picture of what is going on, but contains less information. This corresponds well to our everyday descriptions of events. If I say "I bought a boot yesterday", you might leave it at that, or you might ask how I went about doing it.

The combination of object and action refinement usually goes together so this is not a straight functional decomposition. Furthermore, subtyping in the actions and

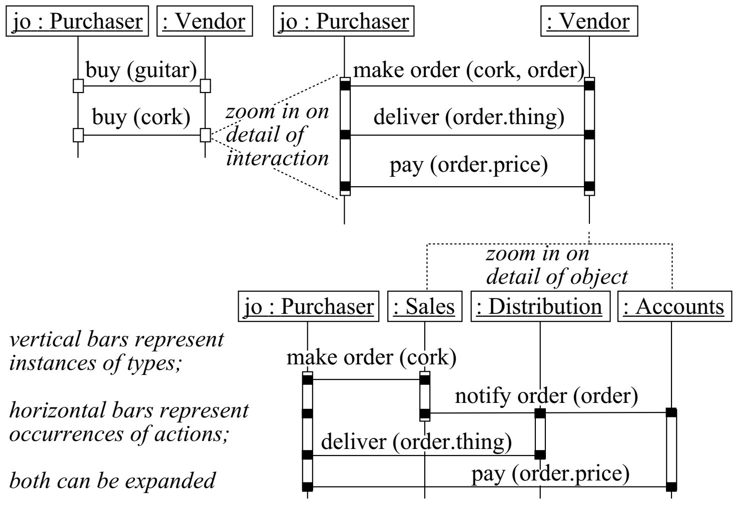


Figure 4.10. Decomposition and "zooming"

objects allows one description to apply to a variety of more detailed cases so we have not given up the polymorphic value of object design.

An action decomposition can be summarised on a type diagram using the aggregate notation, as shown in Figure 4.11. The postconditions of the more refined actions

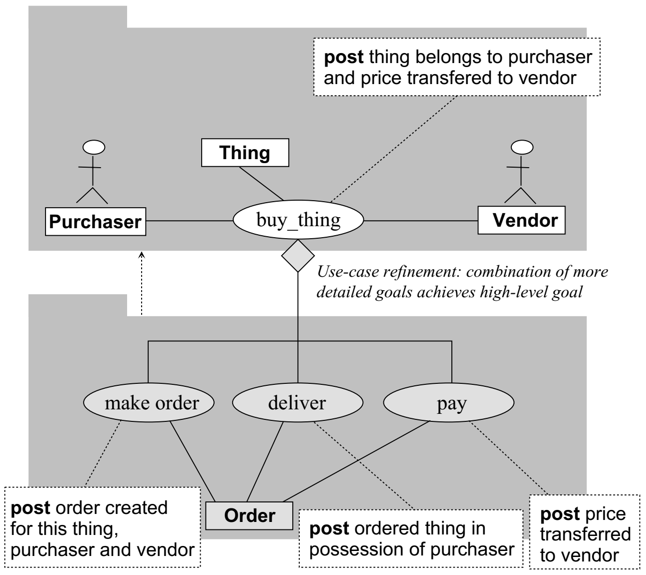

post thing belongs to purchaser and price transfered to vendor
Thing
Purchaser
buy_thing
Vendor
Use-case refinement: combination of more detailed goals achieves high-level goal
make order
deliver
pay
post order created for this thing, purchaser and vendor
Order
post ordered thing in possession of purchaser
post price transferred to vendor

Figure 4.11. An action decomposition

generally require more detailed models: more information is required to represent the intermediate states. In this example, there is an Order object that represents the state of the “buy” transaction as it progresses through the stages.

The “finer” actions may be composed in various ways: they may be a simple sequence, or some of them may be repeated or optional, or they may happen in parallel, in other words, all the things you can do with a program. The difference with a program is that we do not determine the order in advance: the participants decide that at the time.

To document what combination of the finer actions makes up a particular abstract action, we can use a statechart (or an activity diagram) as in Figure 4.12. Any sequence of occurrences that follows the chart constitutes a “buy”. Orders and deliveries may be made without matching payment, but when that happens we do not call it a “buy”. We should check that, given the postconditions of the constituent actions, any route

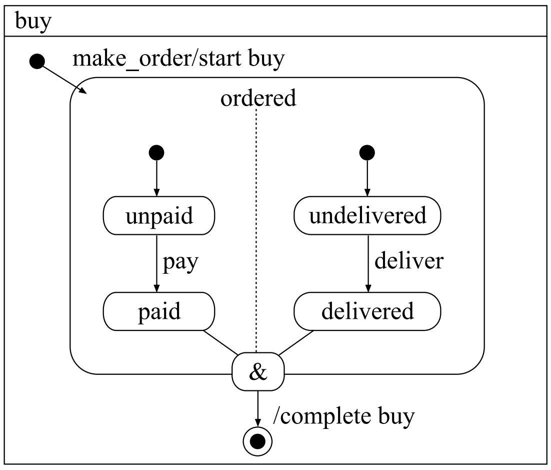


Figure 4.12. Decomposing an action using a statechart

through the statechart (between the start and complete markers) would accomplish the specification of “buy”.

Statecharts work well for this purpose, as they allow for repetition and alternatives. Sequence charts are best for illustrating the events on just one occasion.

4.4.3 Refinement from Domain to Components

Actions can represent interactions in the “real world” or within software. An object-oriented message or procedure call is one kind of action. We can trace from domain actions all the way down to operations within the software. In this example in Figure 4.13, the actions are broken down successively, in parallel with the participating objects. Some of the actions ultimately decompose to interactions between two constituents of the Accounts department: a clerk and a software component. Thus we can relate software requirements directly to the activities of the business as a whole. Of course, we can then continue the refinement to operations inside the software which we will discuss in the next section. In this section we have shown:

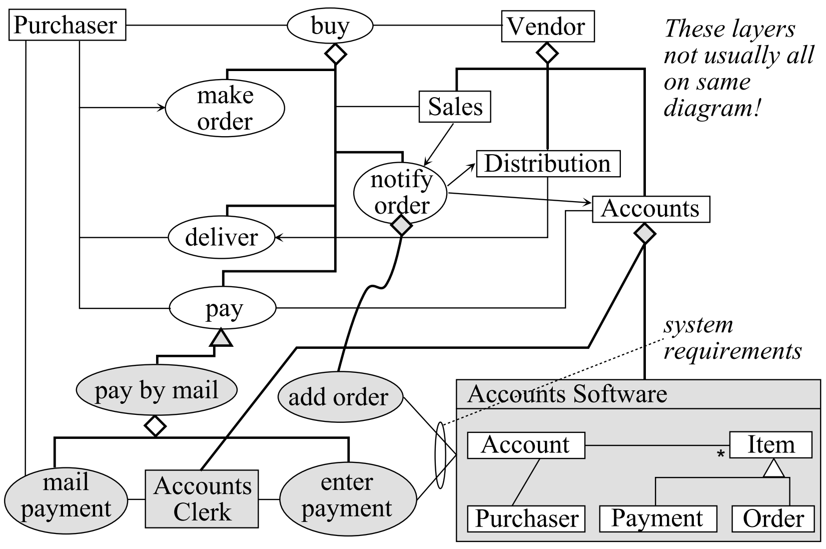


Figure 4.13. From domain model to components

- how the outcome of any kind of interaction or event can be specified by reference to a model;
- the strong relationships between the different modelling notations in Catalysis;
- how precise yet abstract descriptions of behaviour can be written; and,
- how the abstractions can be systematically traced to the more detailed models, ultimately from business models down to program code.

4.5 Modelling for Component-Based Development

4.5.1 Domain Modelling

Domain modelling is about analysing the concepts that occur in the domain of interest, without reference to any particular design. We are interested in domain modelling for three reasons:

- it improves understanding between people working in the domain;
- it is a good first stage in the development of component models within the business; and,
- it is essential to the definition of connectors: the components need to be talking the same language about the objects they are dealing with.

Domain modelling tells us about types of object that can be found in the domain, and types of action — tasks, jobs, events, things that happen.

4.5.2 Specifying Component Behaviour

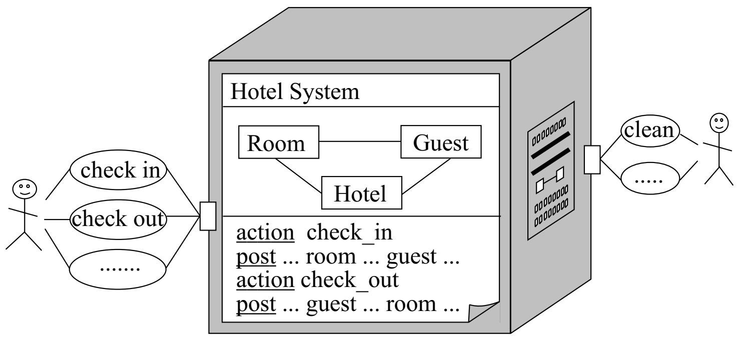


Figure 4.14. A hotel system as a black box

A specification is like a label on the side of a black box: it tells you what behaviour to expect, but does not really tell you what is inside. In Figure 4.14, we can see that the Hotel System deals with relationships between Rooms and Guests, but we cannot see how it represents them. From an implementor's point of view, it gives the criteria for acceptability. There may be several such views of a component.

The objects on the "label" represent the component's knowledge of the external object, which may be more limited than our domain model's view. The specifications of the actions now focus on their effects on the component (and do not include any other participants), and are expressed in terms of the model objects and their associations and attributes.

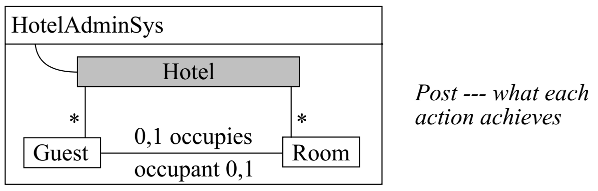
The Catalysis convention is to represent the component specification like an object type, but with the model objects drawn inside the box. You cannot do this with many tools, but in those cases you can use the aggregation symbol instead.

In the same way as for the domain model, the postcondition can be documented more or less formally (we prefer both), and before/after snapshots can be drawn to help illustrate the effect as shown in Figure 4.15.

Formal constraints The formal language is OCL, the Object Constraint Language, which is an add-on to the UML standard [7]. Invariants, preconditions and postconditions are boolean expressions about the relationships between objects; the model can be "navigated" with the syntax *object.link.link*... where each link is an attribute or association. Postconditions can additionally refer to two states of every attribute or association, and so establish how the outcome of an action and the prior state should be related: the tag "@pre" refers to the state before the occurrence.

Associations marked with multiple cardinality are treated by default as bags, so that many useful constraints are written with "member of" and "includes" relations.

The advantage of using OCL is that it allows very well defined statements at an early stage of development, clear of the clutter of implementation detail. Writing constraints formally tends to de-fuzz issues, exposing ambiguities and gaps: although



```
action HotelAdminSys::check_in (guest: Guest, inn: Hotel)
post guest.occupies.hotel==inn
     -- guest occupies a Room in the required hotel ...
and guest.occupies.occupant@pre==null
     -- which was unoccupied previously
```

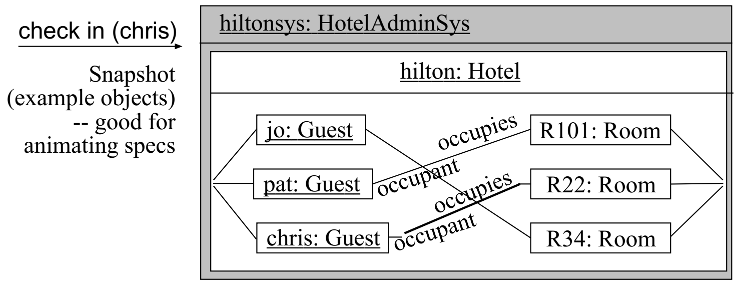


Figure 4.15. Adding to the domain model

more work is involved than writing a less formal requirements specification, experience suggests that there are considerable savings down the line.

Furthermore, the OCL expressions serve as the basis for test harnesses. From a quality assurance point of view, it is widely accepted as good policy to make test specification clear before much work is done on the implementation. In component-based development, it is essential to be able to define the requirements on an interface, because there may be many components that want to implement it.

Reaching inside It is often useful to document actions not as operating just at the surface of components, but to think of the components as containers for the objects inside. The CORBA and COM models do this. We can therefore draw actions like in Figure 4.16. However, there is still no implication about what is inside the component:

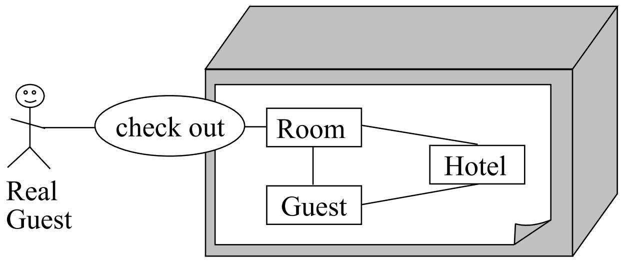


Figure 4.16. Moving an action inside a component

the objects in the “label” may be an illusion created by the component’s façade code.

This section has shown how transactions between a component and its context are specified as changes to a model of the component’s state.

4.5.3 Designing Components

To design the component, we decompose the interface actions further as shown in Figure 4.17. The interfaces may turn into substantial actions between sizeable sub-components or they may be elementary messages between objects. The objects or components may be in the same execution space, or in different machines. The par-

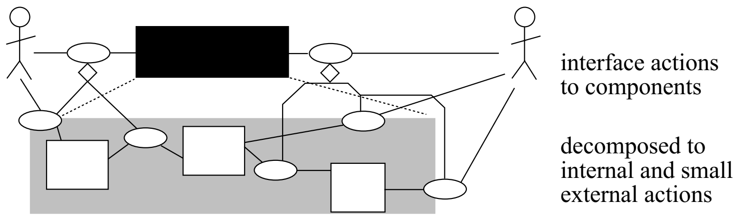


Figure 4.17. Decomposing interfaces

titioning of responsibilities between constituents is crucial to a flexible design: the CRC technique is used to minimise dependencies. “One object, one purpose” is the rule generally but if the bandwidth between components is limited (for example, if they are in different execution spaces), they may need to duplicate some information; that, in turn, implies a need for synchronisation.

Retrievals: mapping implementation to specification models Specification models, the “labels on the outside” of a component, are there to explain its behaviour to external clients. Provided the implementor produces the expected behaviour, it does not matter how the internal design actually works. Naturally, it is nice if the implementor uses object-oriented programming, and uses classes and links in the program that correspond directly to the types and their relationships that the analyst discovered in the real world but there are various reasons why this may not be the case. The specification may be a partial view, or the implementor may choose a different model for performance reasons, or the implementation may have been constructed independently of the requirement — for example, if a generic system is adapted to meet the requirement.

To ensure that a component meets a specification, we have to translate from its internal language to the specification language. The general way to do this is to program a set of classes representing the abstract model directly. Each association, attribute, or state in the abstract model should be implemented as a read-only function, which extracts the value of the abstract attribute from the implementation. Note that this has to be done separately for each implementation and each specification it claims to fulfill. The invariants and postconditions can then be coded as test software, and run at the start and end of each action during integration testing. The object-oriented programming language Eiffel [3] has the facilities for doing this built-in. For example, if

I write a specification of a geographical position using x-y coordinates, then my postconditions will be written with x's and y's. If you prefer to implement using direction and distance, we can check whether you have met my postconditions only after you have provided x( ) and y( ) retrieval functions.

For more complex examples, the same procedure can be followed, but a pictorial overview of the mappings can be useful. In the system of Figure 4.18, the business model has Customers that have Orders and Accounts. The system has been implemented using a bought-in Accounts system and legacy Dispatching support software. Unfortunately, the Accounts system does not have any notion of Customer: its spec-

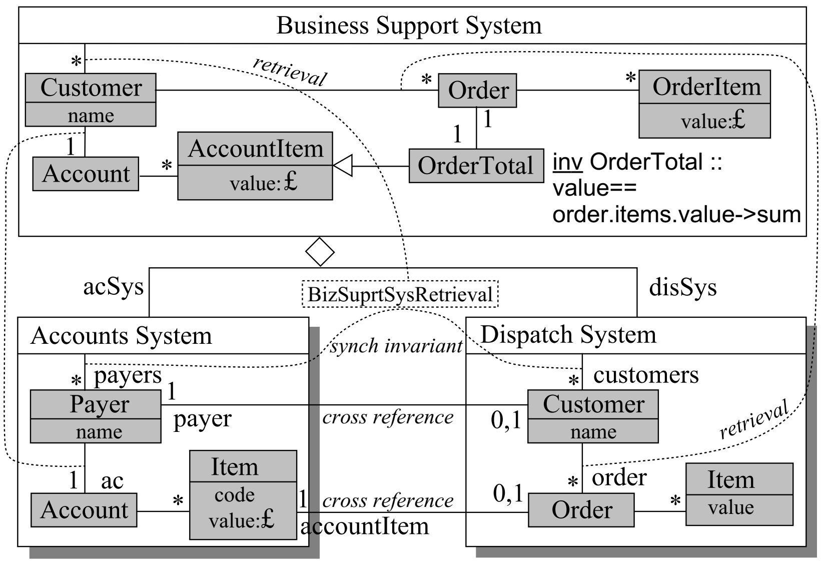


Figure 4.18. Implementation of a business support system

ification talks about Payers. But we can sketch the correspondence between them on a diagram. Normally there would be too much to put on one diagram, so we would focus on one area of correspondence at a time. The Payer-Account link corresponds to the business model's Customer-Account link, and we can see that the Customer's Orders are kept in the Dispatch system.

Something that becomes clear when we sketch retrievals is that some information is duplicated in both implementing components: this tells us that we will have to do some synchronisation — that is, programming that ensures the two lists keep up to date.

On a diagram like this, that shows the models of several components at a time, we can show cross-references: associations that cross the boundaries of the components. Here, an Item in an Account can be linked to an Order. For an association to cross boundaries means that each end has to understand some kind of identifier; spreading the model across component boundaries like this tells us where we will need such identifiers. There are also cross-references to the outside world: for example, a Customer's account number or name are cross-references that associate the software Customer with the real one.

In this section, we have outlined how component design continues the fractal decomposition of actions and objects. We have also seen how different models within the implementation can be related back to the business model and cross-referenced with each other.

4.5.4 Defining Component Connectors

Recall that the point of defining component connectors separately from components is to allow many components to use the same connector, and thereby provide reconfigurability.

Consider the example shown in Figure 4.19 (a) — this is about the link between Mobile Phones and a mobile Phone Network.

The special feature of the "mobile link" is that communication is maintained even while the phone is travelling around: the network has a number of base stations dotted around the country, and the phone works through the nearest.

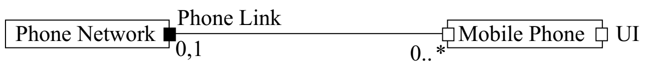
In (b) a Robot Vehicle trundling around a factory gets its commands from a Control Network, in a manner just like a mobile phone network, through antennae fixed in strategic positions around the factory. The protocol of the link is exactly the same as in a mobile phone, except that the domain model is different: a Robot Vehicle would not get sensible commands from a Phone Network, and a Mobile Phone would not be able to call through a factory's Control Network.

If we think of the connectors as objects, we obtain Figure 4.19 (c). The subclasses are concrete: there are instances of them shown in Figure 4.19 (a) and (b). The definition of the Phone Link includes enough information about the protocol and the domain model, that the designers of the Phone Network and Mobile Phone could each work independently and know that their efforts would couple properly when required.

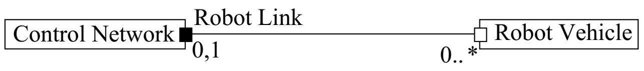
The Mobile Link superclass is abstract: it represents a scheme of interaction in which some things (like the domain model) are left undetermined.

Model templates Catalysis includes the concept of a model template: a generic piece of model in which some of the details can be filled in. For example, we can generalise the two kinds of Mobile Link, as shown in Figure 4.20 (a). We can define some properties of all Mobile Links, no matter what the exact type of the Network and the Mobile. To extend the scheme to make a subtype such as Phone Link, substitute the placeholders with the real types Phone Network and Mobile Phone (and provide additional information about the robot commands); this is shown in (b).

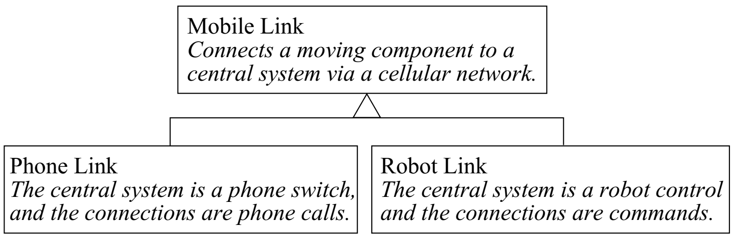
The "real" types' definitions include everything we attach to the placeholders in the template, but with the names substituted. Notice that this does not make Phone Network or Control Network subtypes of Network: if that were so, we could couple Phone Networks to Robot Vehicles. The idea is to extend the entire template (in the dashed box) together, rather than each type individually.



(a)

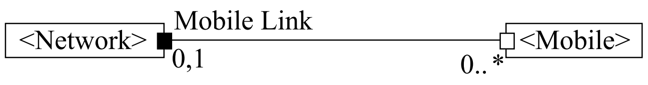


(b)

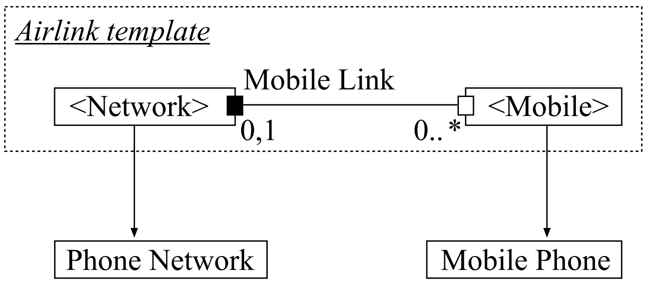

Mobile Link
Connects a moving component to a central system via a cellular network.
Phone Link
The central system is a phone switch, and the connections are phone calls.
Robot Link
The central system is a robot control and the connections are commands.

(c)

Figure 4.19. Connectors as objects



(a)

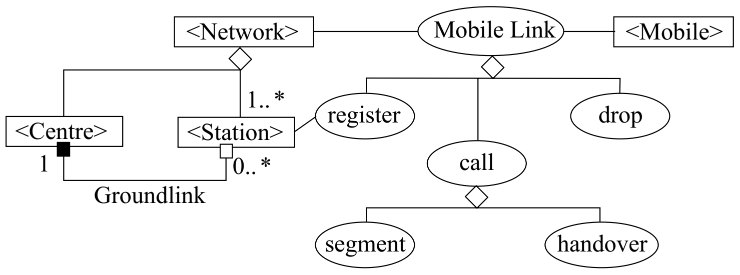


(b)

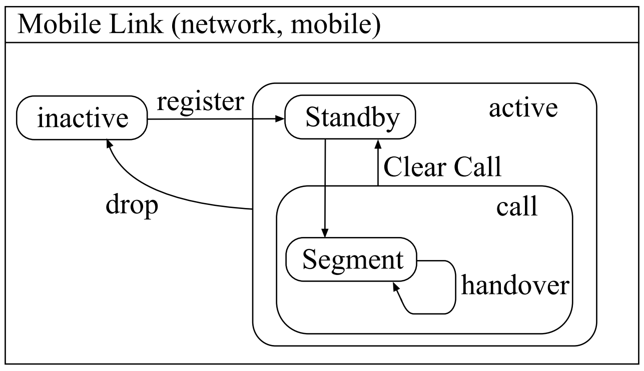
Figure 4.20. Model template example

Defining the connector protocol Of course, we can attach postconditions and/or refinements to the objects and actions in a template. A Mobile Link, as a connector, can be considered as an action, and can be decomposed into smaller actions. A Network maintains a link with its Mobiles via a number of Stations as in Figure 4.21 (a). When a Mobile is switched on, or comes within range of a Station, it registers with it; this enables the network's centre to establish communication with a Mobile wherever it is. As the Mobile moves, it often "drops" one Station and registers with an adjacent one.

A "call" is a two-way transfer of information. During a call, the Mobile may be handed over from one Station to another, to keep uninterrupted communication. Any part of a call conducted through one Station is called a "segment", so a call is one or more segments separated by handovers. A statechart in (b) illustrates the possible sequences. Let us go further into the handover. The effect is to transfer the Mobile's



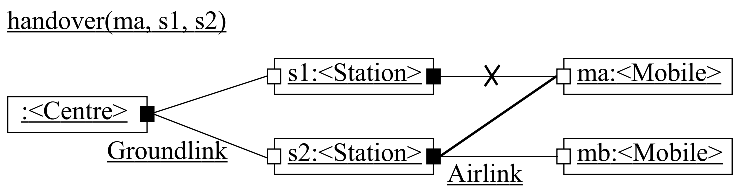
(a)



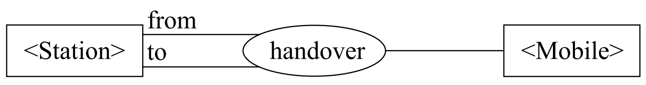
(b)

Figure 4.21. Connection protocols

communication to a different station. We can represent the Mobile-Station links with connectors, and draw a before/after snapshot as in Figure 4.22 (a), a type diagram as in (b) and an action specification as in (c). That tells us what the handover achieves,



(a)



(b)

```
action handover (from:<Station>, to:<Station>, mobile:<Mobile>)
post    mobile.airlink = to    -- the mobile is talking through ''to''
pre     mobile.airlink = from  -- the mobile was talking through ''from''
```

(c)

Figure 4.22. Detailing connection protocols

but how does it work? If we decide to look more closely at this stage, we might see a protocol defined for the handover. It could be a statechart or a sequence chart — let us choose the latter on this occasion; this version in Figure 4.23 includes some timing constraints too.

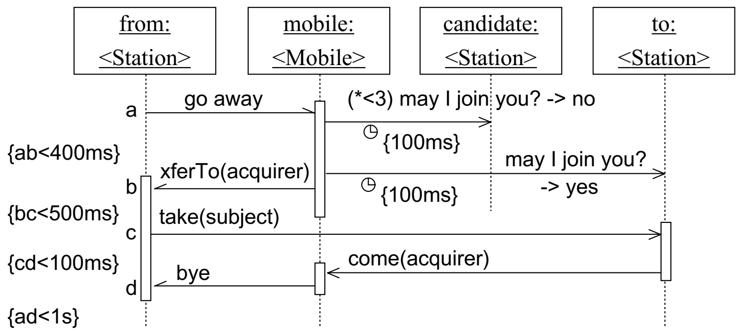


Figure 4.23. Detailing further connection protocols

The Mobile keeps track of up to three Stations' signal strengths, and when the signal from the current airlink becomes weak, it checks out the other best candidates. They may refuse to accept the connection because of their limited traffic capacity.

Applying model templates The UML “pattern” notation is used to indicate the application of a Model Template, shown in Figure 4.24. This is like a macro application:

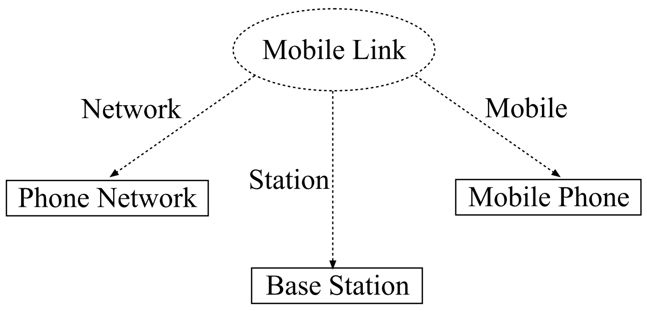


Figure 4.24. A model template as an UML pattern

it effectively creates a copy of all the diagrams we have documented for the placeholder types, replacing them as indicated. We can add extra information about these particular types (e.g. about the specifics of making phone calls) as required.

In this section, we have seen how connectors can be defined separately from components and, furthermore, how templates can be used to define generic connectors that can be specialised to particular applications.

4.5.5 Component Partitioning

The issue of how to partition functionality between one component and another is about separation of concerns, which gives flexibility, balanced with the need for reasonable performance and robustness, especially if the components are distributed.

In the “families of products from kits of components” philosophy, the objective is to make a variety of software products within the same general domain. To make one product rather than another, you unplug one component and plug in another. The components should, therefore, ideally correspond to the features that will vary from one product to another.

A bad partitioning would, therefore, be one in which the “components” were the major functional units that every product required and in which variation is achieved by tweaking parameters on the components’ sides. If the same set of pieces appears in every end-product, they are not components in the sense of this chapter.

One way of thinking about a kit of components is that it provides a language in which to write designs for the products in your domain. In workflow systems and visual program builders, this is not just an analogy: the work-stages in a workflow are the statements of the language. Nierstrasz [4] argues that there should always be such a language in a good component kit. A useful approach to component architecture is, therefore, to consider how you would design a language in which to write systems in your target domain.

Against pure partitioning are issues of performance and robustness. Functions and information may have to be duplicated to provide local service where the bandwidth is low, or where their principal source may sometimes be inaccessible.

4.5.6 Processes for Component-Based Development

The key features for a CBD process are as follows:

- separate component assembly, component design, and component kit architecture. Make separate development cycles for each, and a cycle for development of the component repository;
- create a kit architecture, and in particular a domain model, independent of any component's design, as part of the kit architecture. The domain model should not just be entities and relations: it should contain invariants and dynamic constraints too;
- use short-cycle incremental development (and accompanying principles) for the specifications and designs of the components;
- use the formal notations demonstrated here between colleagues: they are not generally for clients. Writing precise specifications helps clarify what the users are asking for, and raises questions you can go back to them with. They are also good for prototyping;
- write postconditions and invariants or actual test code before writing the software.

4.6 Summary

This chapter has provided an overview of the Catalysis approach to component-based development. The principal aim is to get flexible systems from reconfigurable components; this in turn requires well-defined connectors, and connectors that are defined separately from the components themselves. Connectors are interfaces but, unlike plain object interfaces, they can encompass the ideas of dialogues, protocols, and transactions.

A variety of techniques can be applied to defining connectors precisely. The central idea is the postcondition defined on abstract models of the components' states. When a candidate implementation is presented, it should be "retrieved" to the specification model so that test procedures based on the specifications can be applied.

Catalysis techniques are particularly valuable both for component-based development and high-integrity systems. They can be characterised as follows:

- scalable — because any system or subsystem can be abstracted as a single object and any transaction, no matter how complex, can be abstracted as a single action, the method is suitable for taming large designs;
- traceable — retrieval provides an unambiguous link between abstract models and code. Precise and abstract, meaningful statements can be made at a very high level, exposing gaps and inconsistencies early;
- reuse — of both models (as templates) and code (as components);

- coherence — there are strong relationships between the different models, providing different views that can be tied together.

References

1. D'Souza D, Wills A. Objects, Components and Frameworks in UML: the Catalysis Approach, Addison-Wesley, Reading, Massachusetts, 1998
2. Fowler M. UML Distilled, Addison-Wesley, 1997 (`http://www.omg.org`)
3. Meyer B. Object Oriented Program Construction, Englewood Cliffs, Prentice Hall, 1988
4. Nierstrasz O, Tichelaar S, Demeyer S. CDIF as the Interchange Format between Reengineering Tools, OOPSLA'98 Workshop on Model Engineering, Methods and Tools Integration with CDIF, September, 1998
5. Selic B, Gullekson G, Ward P. Real-Time Object-Oriented Modeling, John Wiley & Sons, New York, 1994 (`http://www.objectime.com`)
6. Szyperski C. Component Software — Beyond Object-Oriented Programming, Addison-Wesley, Harlow, 1998
7. Warmer J, Kleppe A. The Object Constraint Language, Addison-Wesley, Reading, Massachusetts, 1998