

Designing Component Kits and Architectures with *Catalysis*

Alan Cameron Wills

TriReme International Ltd
<http://www.trireme.com>

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 [Catalysis]. Catalysis is an approach to component-based development that is gaining increasing use among industrial developers of component and high-integrity software.

What's a component?

The idea of component-based development (CBD) is that we assemble software end-products from reusable components [Szyperski]. Each component is designed to work in a variety of contexts and work in conjunction with a variety of others. 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 didn't 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's 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 don't know who your component might be talking to: that's 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.

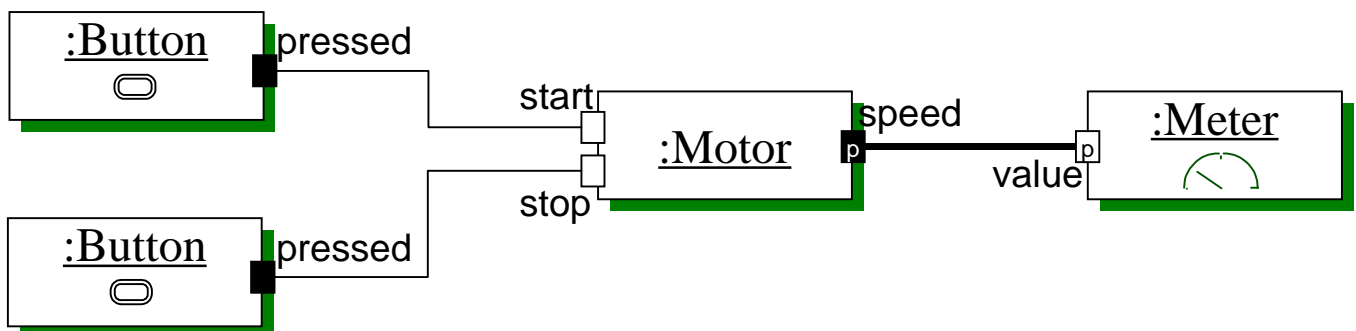
- The most important feature of both object and component design is that interfaces can be described separately from the classes that implement them, so that a piece of code designed to work with a given interface can be used with any component/object that implements that interface. This feature, if 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, filesystem, 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 doesn't have to be OO inside.)
- The interface to a component includes the idea of outputs, rather than just the list of procedure-calls that is an OO 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 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.

Families of Products from Kits of Components

Let's look at this example 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 [<http://www.objecttime.com>].)



Component assemblers don't modify components.

When you buy or reuse a component, you want the benefit of its 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 don't modify components. Instead, the larger strategy is to make components designed to be connected to others. The usual principles apply:

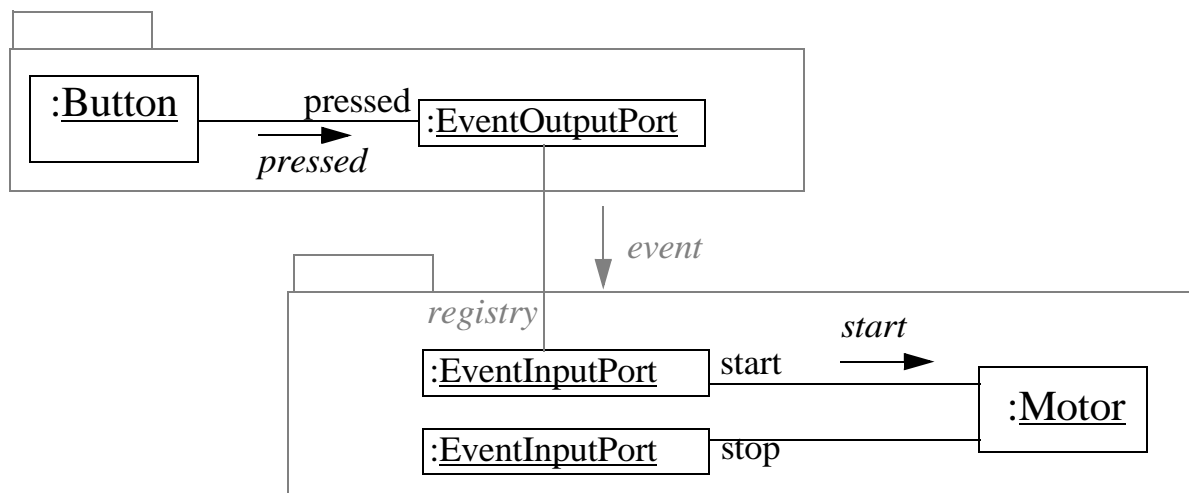
- Each component should just perform one function well (though it doesn't have to be a small function: a phone switch and a payroll system can be good components).
- Each component should have clearly-defined interfaces into which you plug other components and extend or parameterise its behaviour.

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 don't like to expose their designs.

Design connectors separately from components.

If components can't be altered by their assemblers, then they must be designed in such a way that they can be wired together by sending messages to their instances. 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, we could decide that the input and output ports of components are separate objects; that each output holds a list of inputs that are registered with it; and whenever it 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.



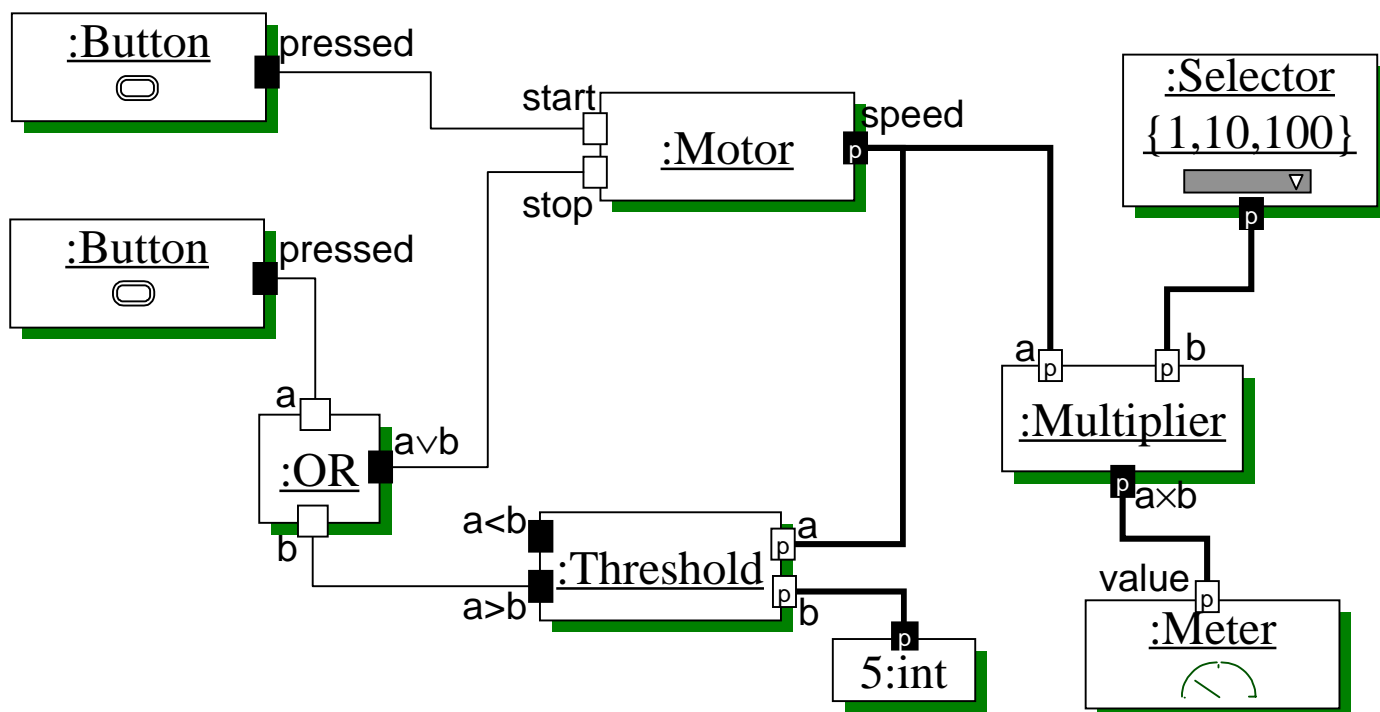
So the component diagram is a higher-level view of this object oriented scheme. Obviously the component version is much more convenient for the designer who assembles the components, who shouldn't need to worry about the details of the interaction. Such schemes also lend themselves to visual assembling 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.

Here is 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 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 others: 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 (which we’ll look at shortly), the connectors define more complex interfaces: the nature of the transactions that can be performed (purchase, work transfer, etc) and the details of the protocol (sequences of messages etc).

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 don’t necessarily come from one supplier, and haven’t 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’s 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. I heard one IS department saying they were taking up CBD, and were going to set 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

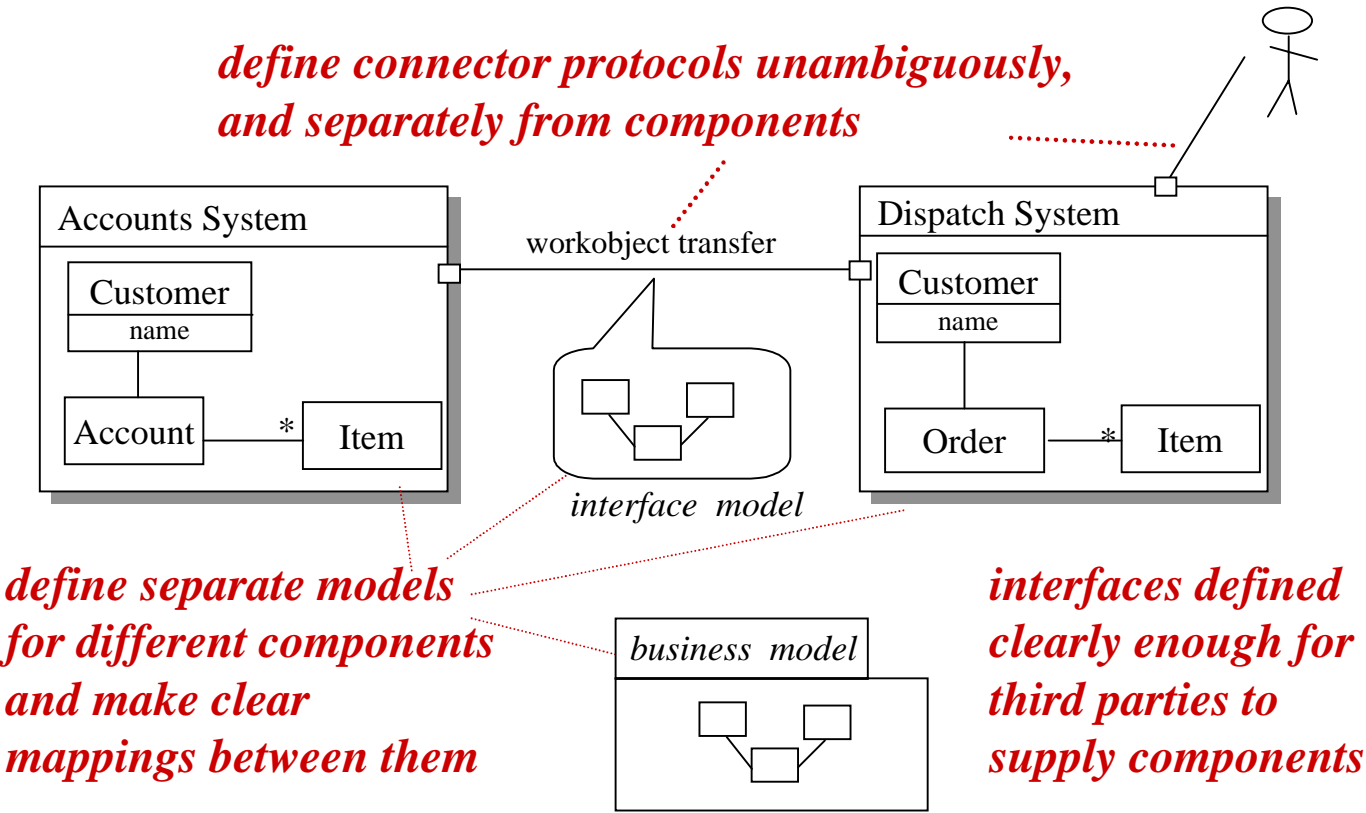
strap such pieces together, but a lot of ‘glue’ is required. And unless you are careful, you will end up with many modules with individually-crafted interfaces between them, rather than reconfigurable components.

To do CBD well, you must have a clear Kit Architecture.

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; 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: the transfer of a financial trade from one stage of the backoffice 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; 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 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.

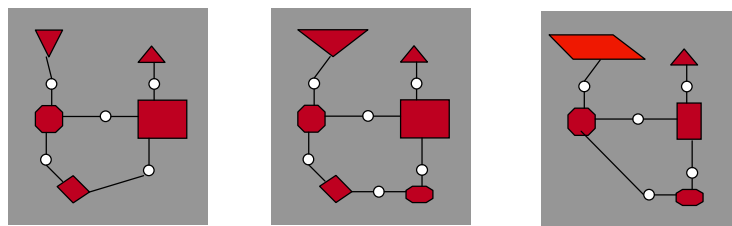
Notice that it is by no means necessary or desirable to make all the components use the same objects or formats internally. Most of them will have their own internal structures that suit them best, or are just there from history. Forcing designers of diverse components to use the business model internally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

Component Strategies

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

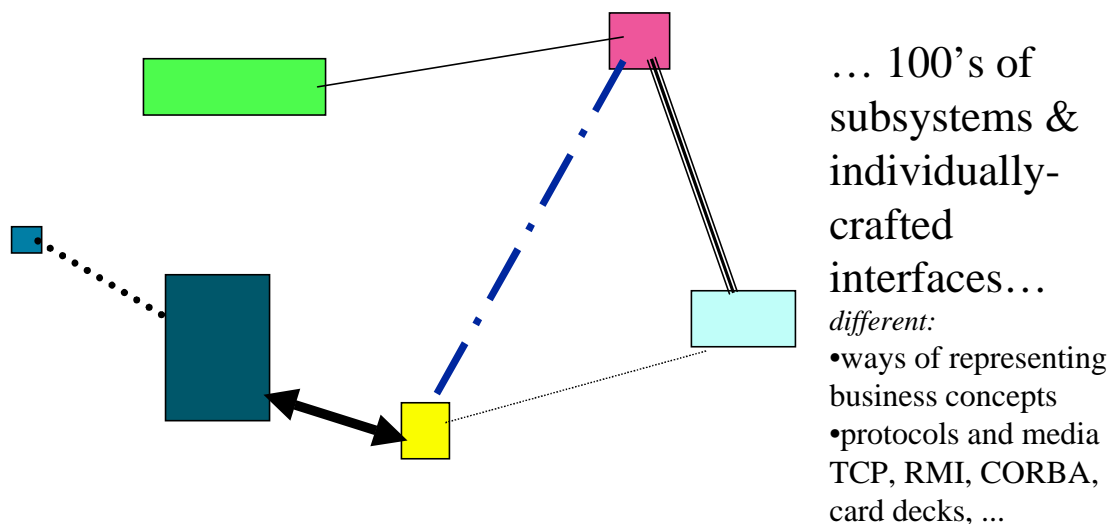
Families of products

The desire to be able to develop new variants of a basic system very rapidly, to keep up with changes in the market.



Enterprise integration

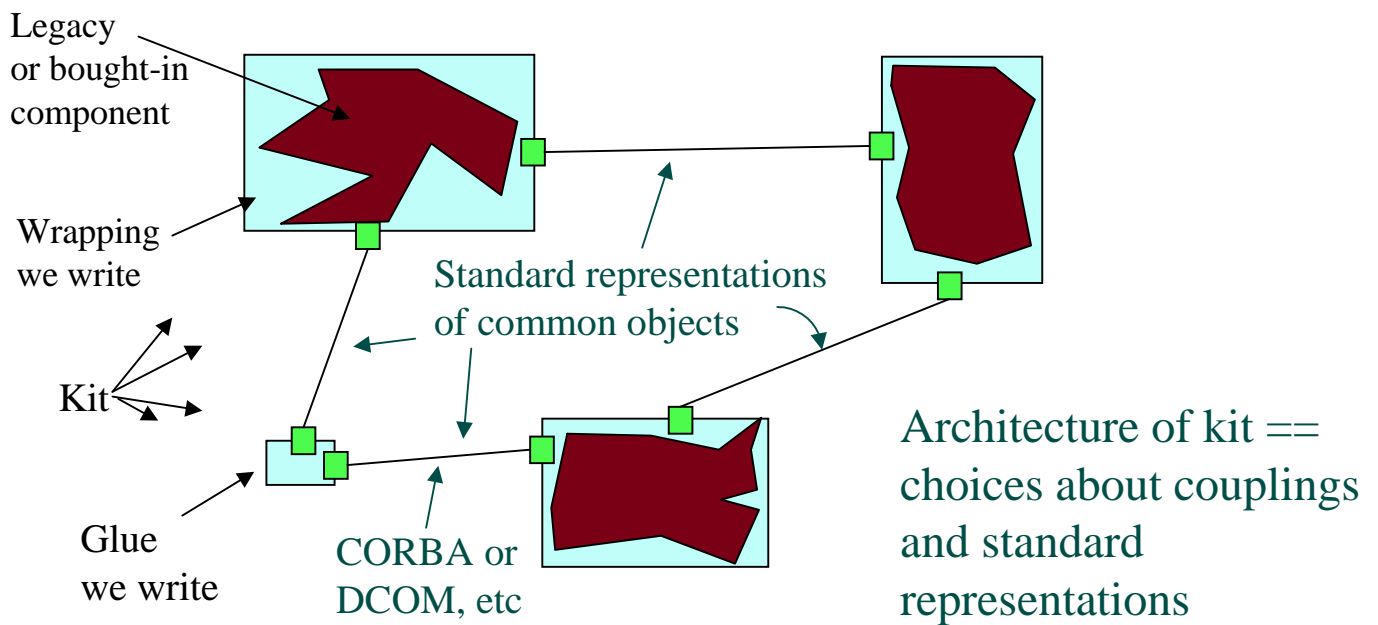
A typical large organisation has a wide variety of systems developed over the years, many of which have their own point-to-point connections. The configuration is inflexible: it cannot be arranged easily to keep up with changes in the business organisation.



The requirement is to make these systems all talk the same language. The strategy is:

- define a component kit architecture
- 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 they cannot interact successfully until they have work to a common business model, as described above.



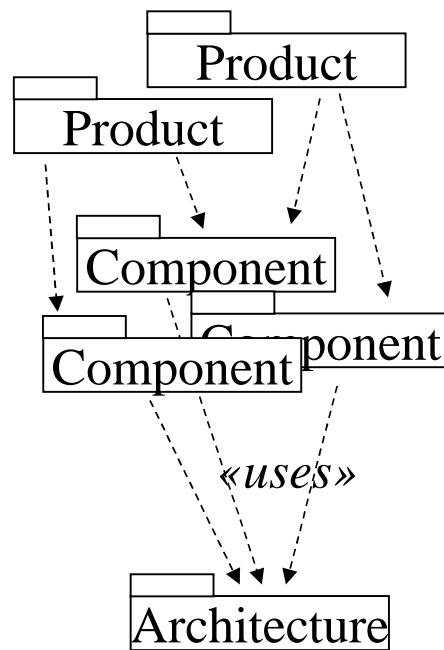
There can be so many systems in a large enterprise that it is not possible to define a model that suits everyone perfectly; and not possible to take account of every system in the corporation. In this case, the model has to be open and extensible, and the medium (such as XML) has to be open too. It is 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.

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 Assembly is typically about working with users to meet their requirements satisfactorily and rapidly; 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 Architecture is a very skilled job, requiring strong insights into the future direction of the kit, and how to make it open and flexible.

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.

Parts in a component package

A component will be assembled with others that its designer has no knowledge of. 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.

Spec
— what you expect of it
+ what it needs from you

Interface
— where & how you plug into it

Robust packaging

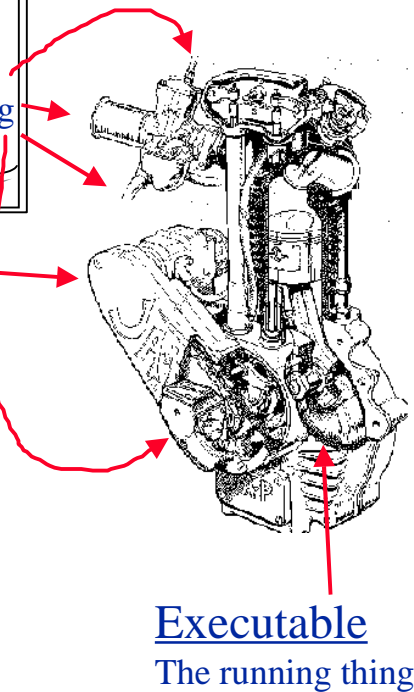
Proof against less careful product assembly:
complain, not collapse

Sample Plug-ins

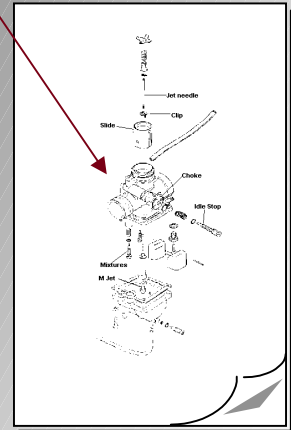
Defaults, standard options, etc

Validation suite

To test for conformance things you propose to plug into it; and to check its performance when operating in your context.



Notice smaller components inside



Design

How it works,
what it's made of

Private to designer

Catalysis: modeling 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've 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 modeling 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.

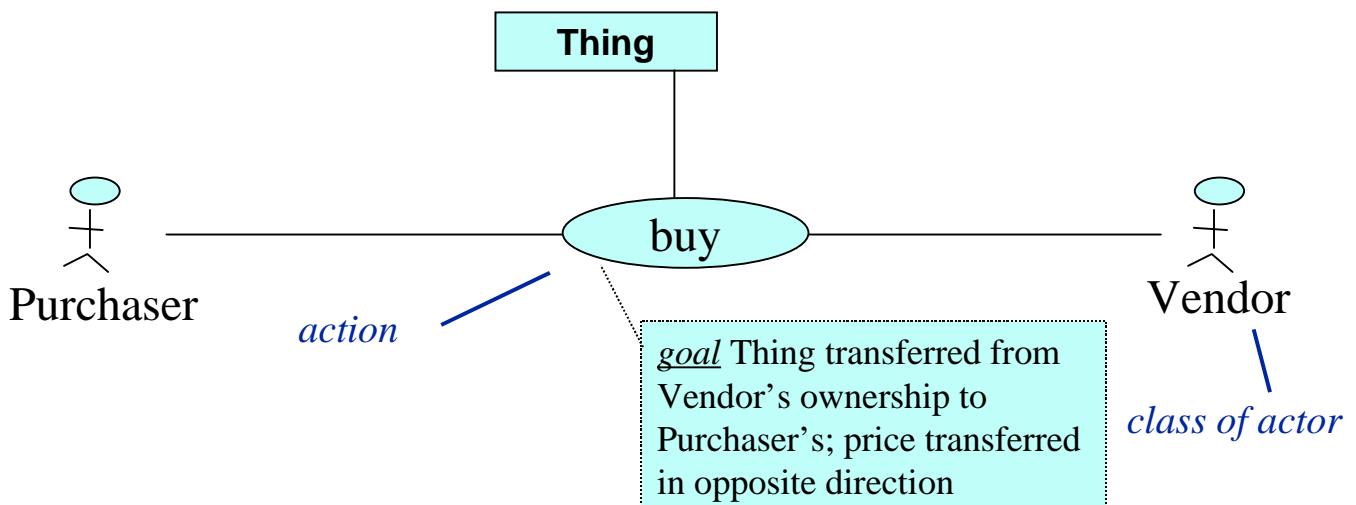
The next section will describe some of the basic modelling techniques; we will then see how these are applied in specifying and designing components and their connectors.

Modelling techniques in Catalysis

Actions

The Catalysis term ‘action’ corresponds to the UML use-case. 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.

An example, in domain modelling:

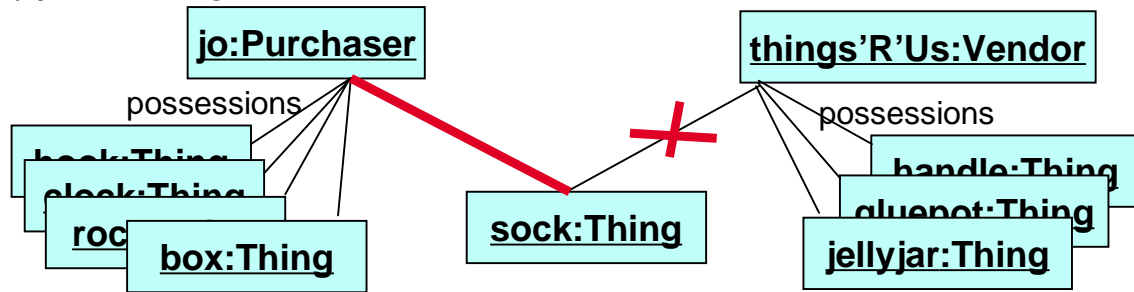


The ellipse representing the action is linked to the types of object that participate in it or are affected in some way. Unlike the usual UML procedure, we don't immediately go on to describe a sequence of steps whereby a ‘buy’ can occur. There are many possible such sequences: with credit card, mail 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've said. The goal uses a vocabulary about the relationships between the objects: the ‘ownership’ of the

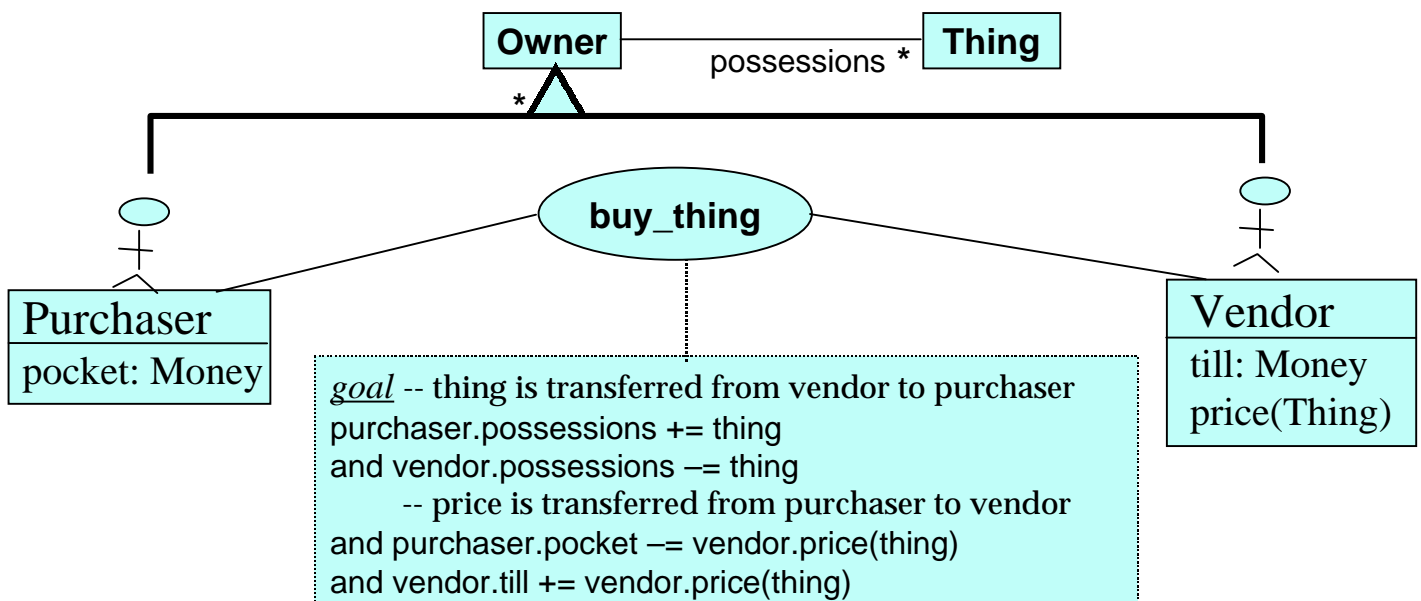
Thing by the Vendor or Purchaser. We can draw this relationship as an association, and draw an instance diagram contrasting a typical situation before and after an occurrence of the action:

buy(jo, sock, things'R'Us)



(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.)

The 'possessions' association gives us a vocabulary that we can use more precisely in the description of the goal:



Joint actions

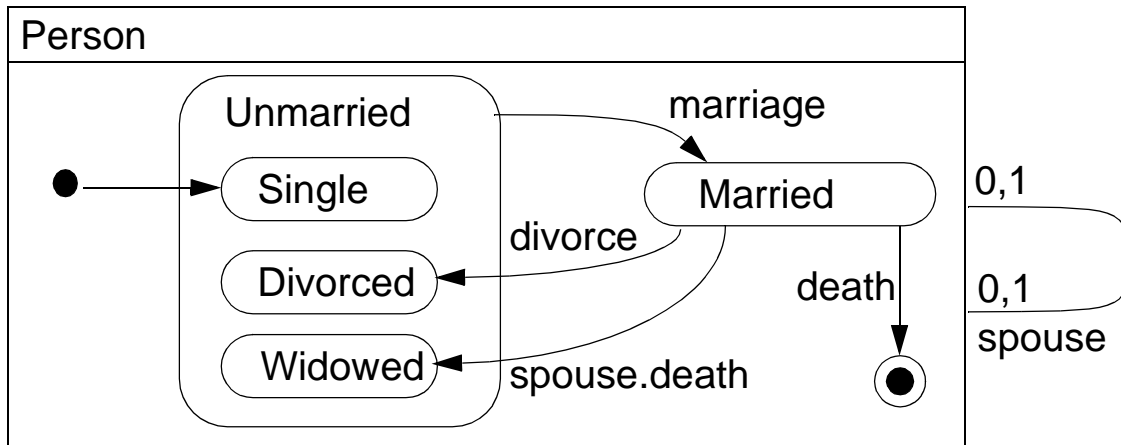
These actions are not attached particularly to any one of the participants. In object oriented design, we attach each task to a particular object; but in analysis and high-level design, we don't want to be forced into that decision too early: we want to be able to say what happens, without the exact details of who does what. It's an essential 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.



The states should be derivable as boolean functions of other attributes or associations; for example, $married = (spouse \neq null)$. 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 $spouse=x$.

The actions on statecharts should appear as ellipses elsewhere in the model; the states should be derivable as boolean functions of other attributes and associations; and the preconditions and postconditions of the actions should include the source and target states of the actions.

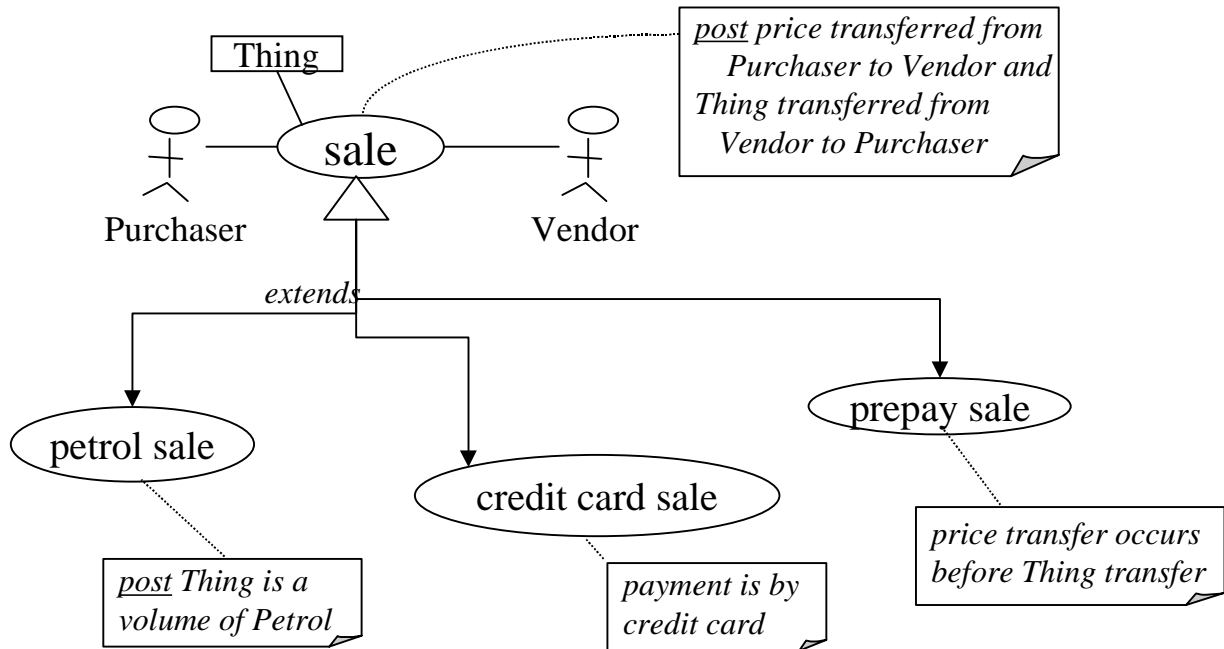
Drawing statecharts is a useful analytical tool for finding actions, as well as a useful presentation of the pre & postconditions of some sorts of action.

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.

Refinement and traceability

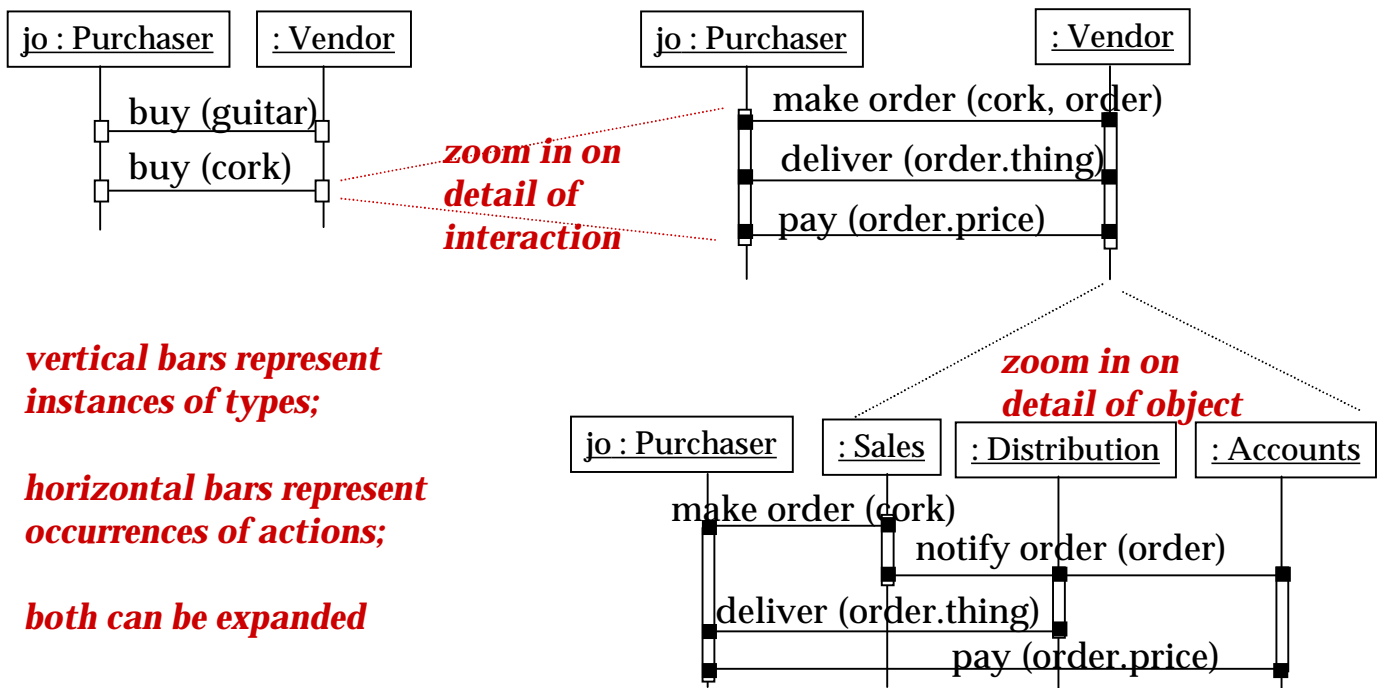
We have a number of well-defined relationships between specifications of greater and lesser degrees of detail. One example is action specialisation: actions can have all the same relationships

as objects, including extension of specifications. An extension's postcondition is ANDed to that inherited from the 'supertype' action.



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. 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'.



vertical bars represent instances of types;

horizontal bars represent occurrences of actions;

both can be expanded

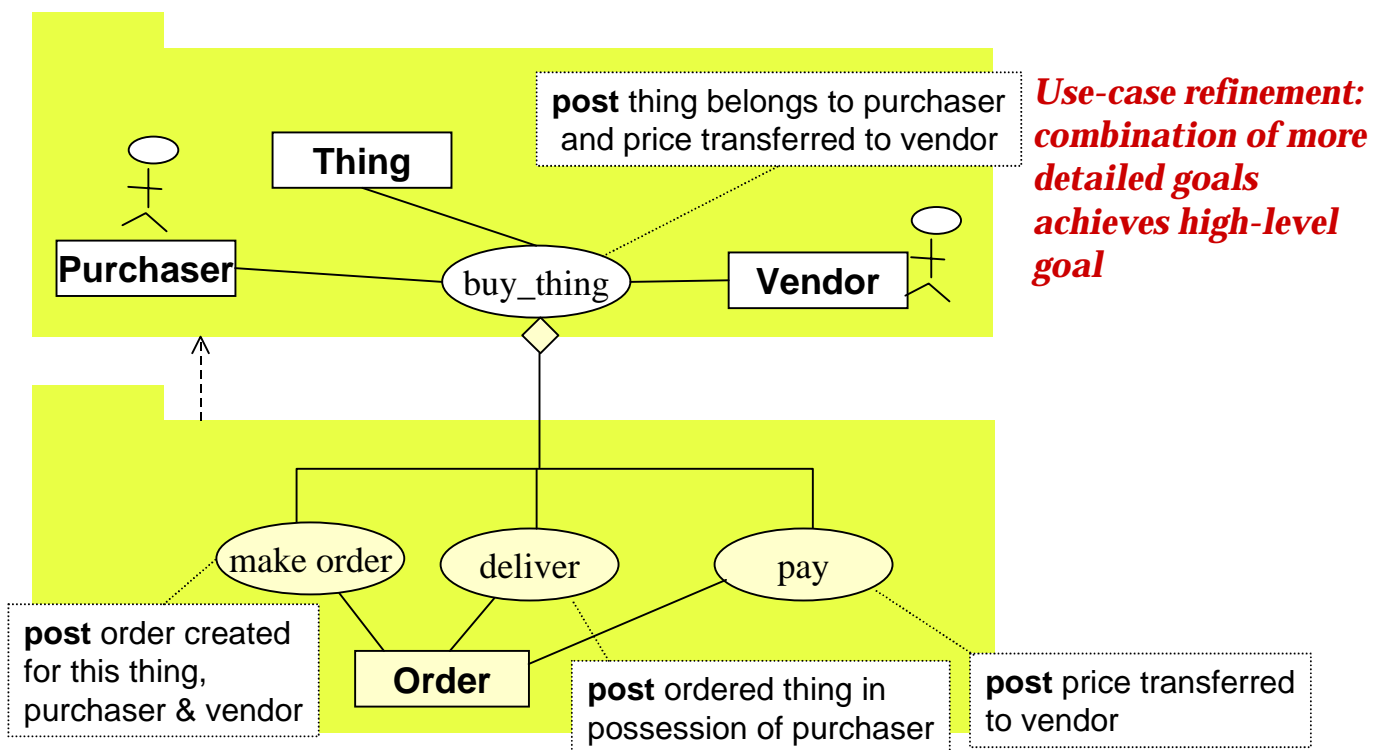
We can ‘zoom in’ on the detail of the objects too. The Vendor turns out on closer inspection to be a Sales, Distribution and Accounts department, and while you make an order with the Sales department, it is delivered by Distribution and you pay to Accounts. This view also enables us to see some of the interactions internal to Vendor that form part of ‘buy’.

(The notation here diverges somewhat from the conventional UML sequence diagrams: 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’s 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’s 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 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.

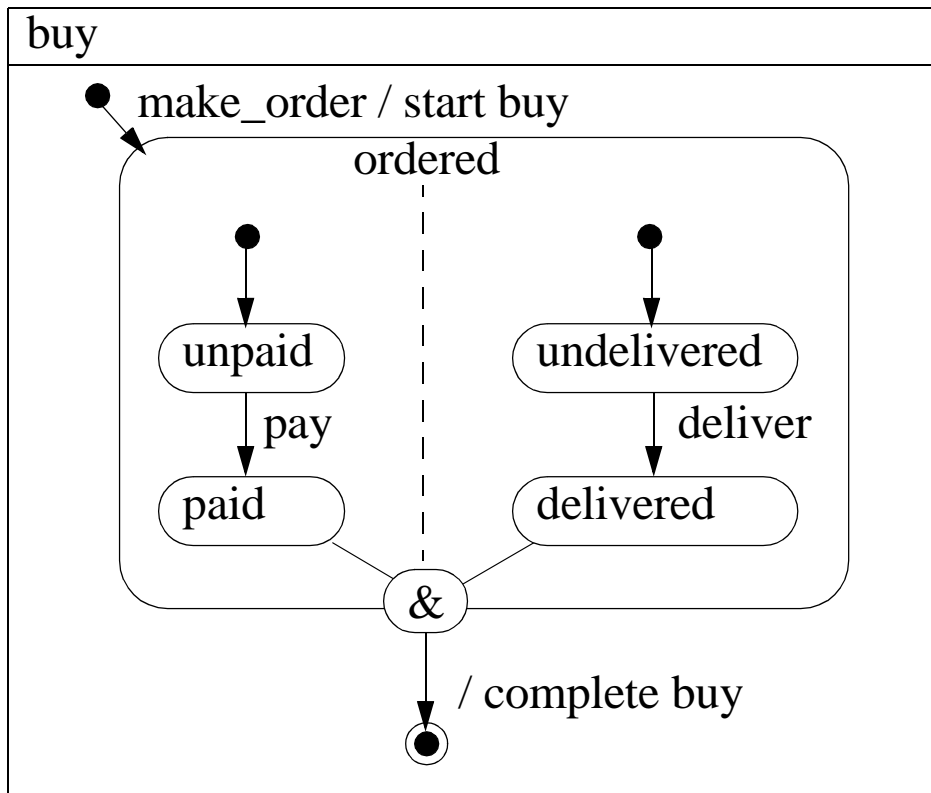
An action decomposition can be summarised on a type diagram using the aggregate notation.



The postconditions of the more refined actions 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 don’t 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). 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 don't call it a 'buy'.)



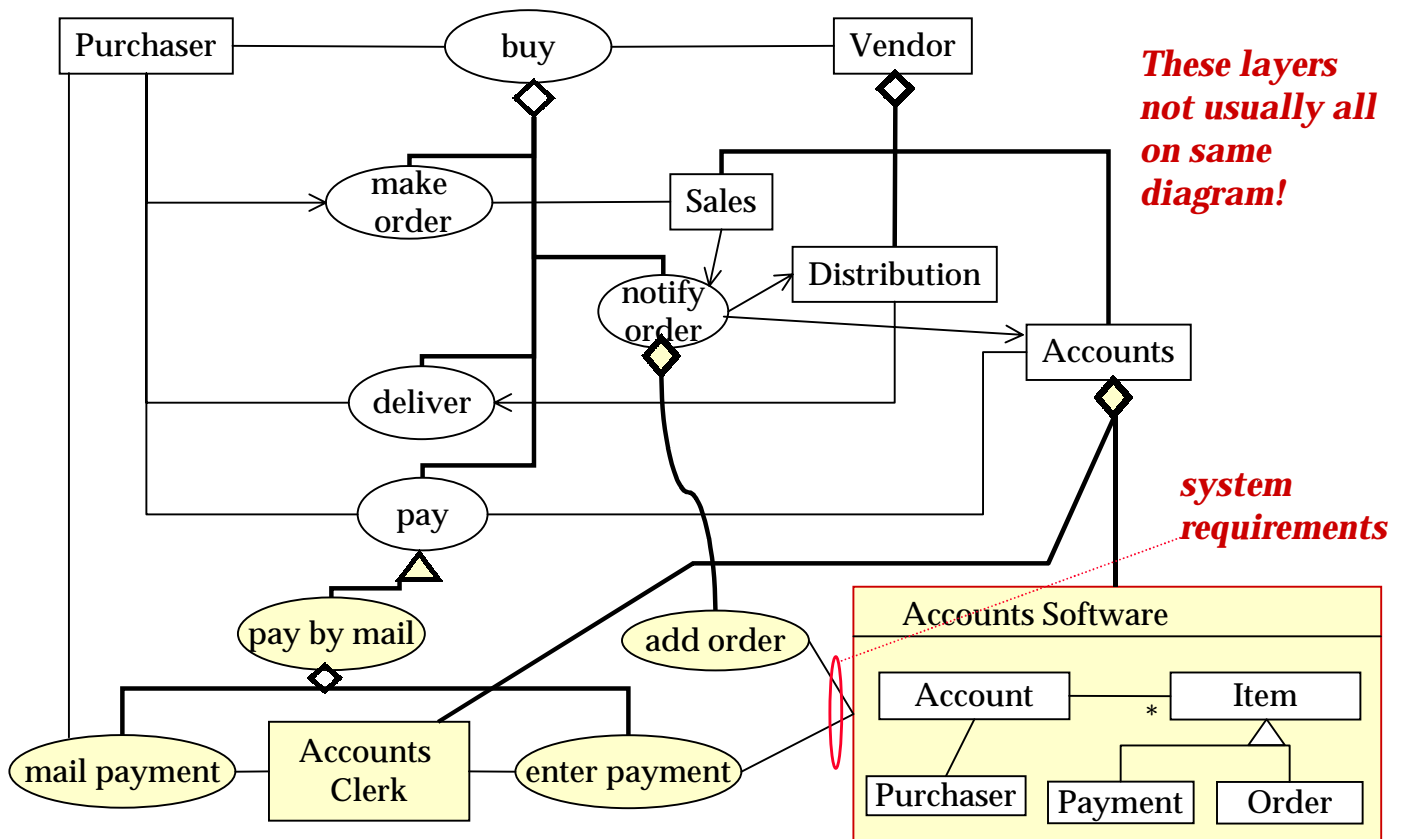
We should check that, given the postconditions of the constituent actions, any route through the statechart (between the start and complete markers) would accomplish the spec of 'buy'.

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.

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, 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'll discuss in the next section).



Summary. This section has demonstrated:

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

Modelling for Component Based Development

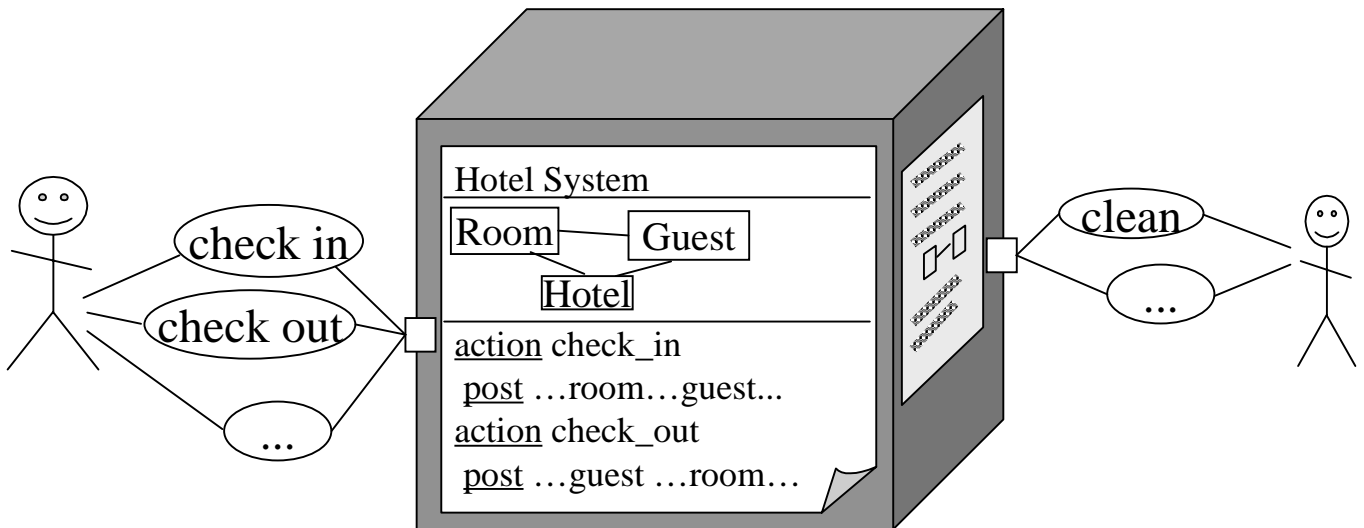
Domain Modelling

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.
- It is essential to the definition of connectors: the components need to be talking the same language about the objects that 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.

Specifying component behaviour

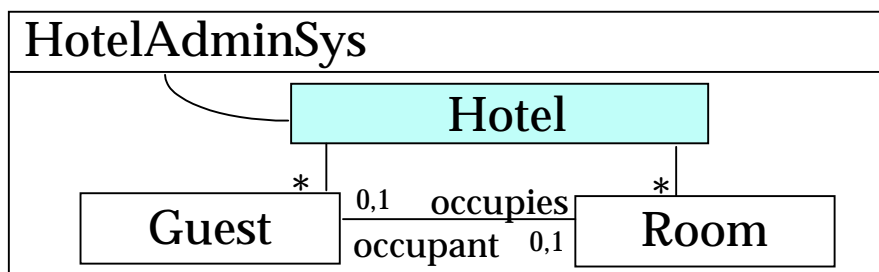


A specification is like a label on the side of a black box: it tells you what behaviour to expect, but doesn't really tell you what's inside. We can see that the Hotel System deals with relationships between Rooms and Guests, but we can't see how it represents them. From an implementor's point of view, it tells 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 don't 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 spec like an object type, but with the model objects drawn inside the box. You can't 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.



Model --- defines terms for describing...

Post --- what each action achieves

action HotelAdminSys::check_in (guest : Guest, inn : Hotel)

post guest.occupies.hotel == inn

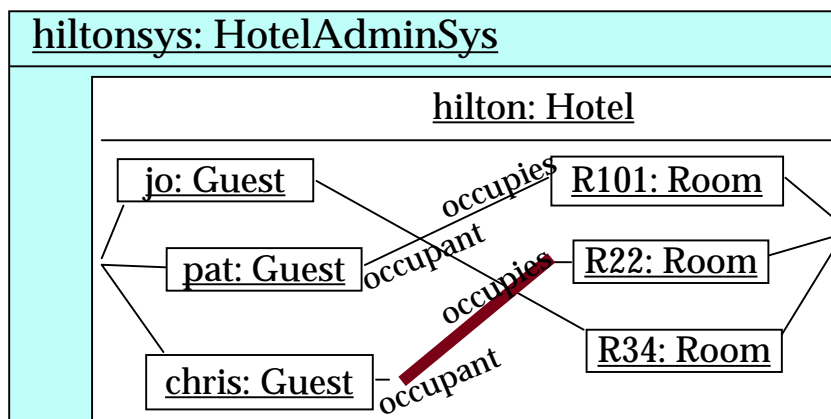
- - guest occupies a Room in the required hotel ...

and guest.occupies.occupant@pre == null

- - which was unoccupied previously

check_in (chris) →

**Snapshot
(example objects)
-- good for
animating specs**



Formal constraints

The formal language is OCL, the Object Constraint Language, which is an add-on to the UML standard [Warmer]. 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.

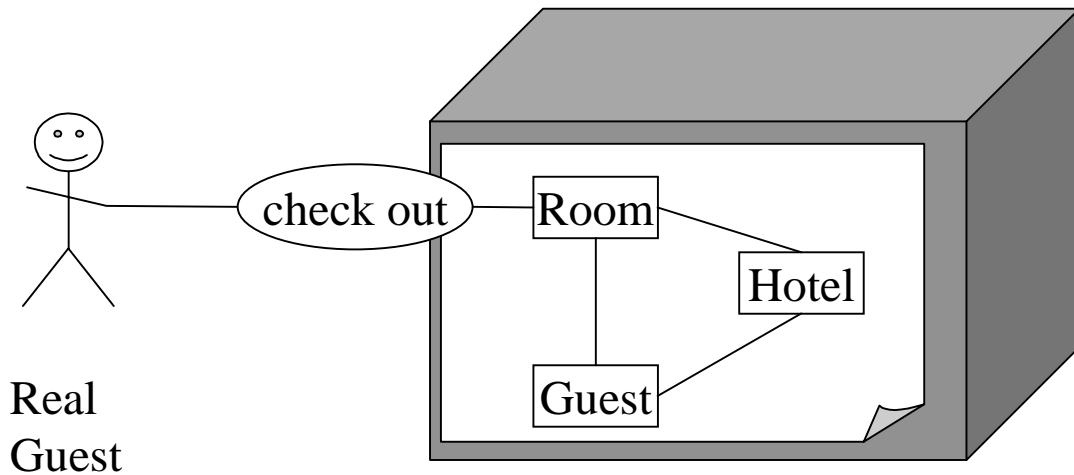
Associations marked with multiple cardinality are treated by default as bags, so that many useful constraints are written withy 'member of' and 'includes' relations.

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 more work is involved than writing a less formal requirements spec, 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. And 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's 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 this:

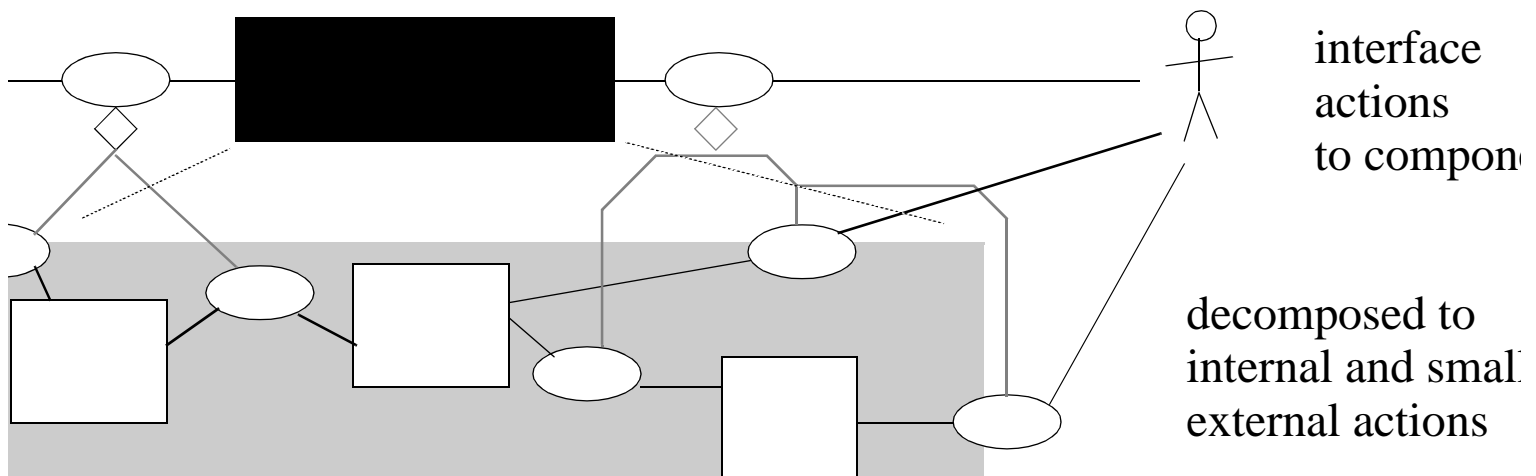


However, there is still no implication about what's inside the component: the objects in the 'label' may be an illusion created by the component's facade code.

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

Designing Components

To design the component, we decompose the interface actions further. They may turn into substantial actions between sizeable subcomponents; or they may be elementary messages between objects. The objects or components may be in the same execution space, or in different machines.



The partitioning 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

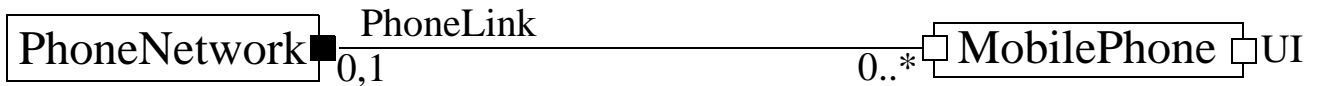
Specification models, the 'labels on the outside' of a component, are there to explain its behaviour to external clients. Provided the implementor produces the expected behaviour, it doesn't matter how the internal design actually works. Of course, it's nice if the implementor uses object oriented programming, and uses classes and links in the program that correspond directly to the types and their relationships 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 spec, 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. (This has to be done separately for each implementation and each spec 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 OO programming language Eiffel [Meyer] includes the facilities for doing this, built into the language.)

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've met my postconditions only after you've provided x () and y () retrieval functions.

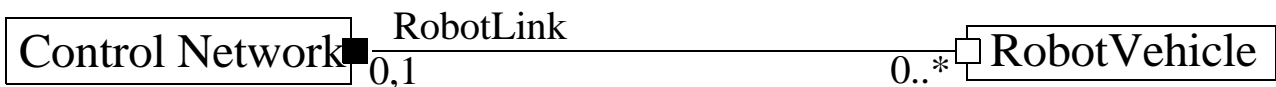
For more complex examples, the same procedure can be followed, but a pictorial overview of the mappings can be useful. In this system for example, the business model has Customers that

This example is about the link between Mobile Phones and a mobile Phone Network:

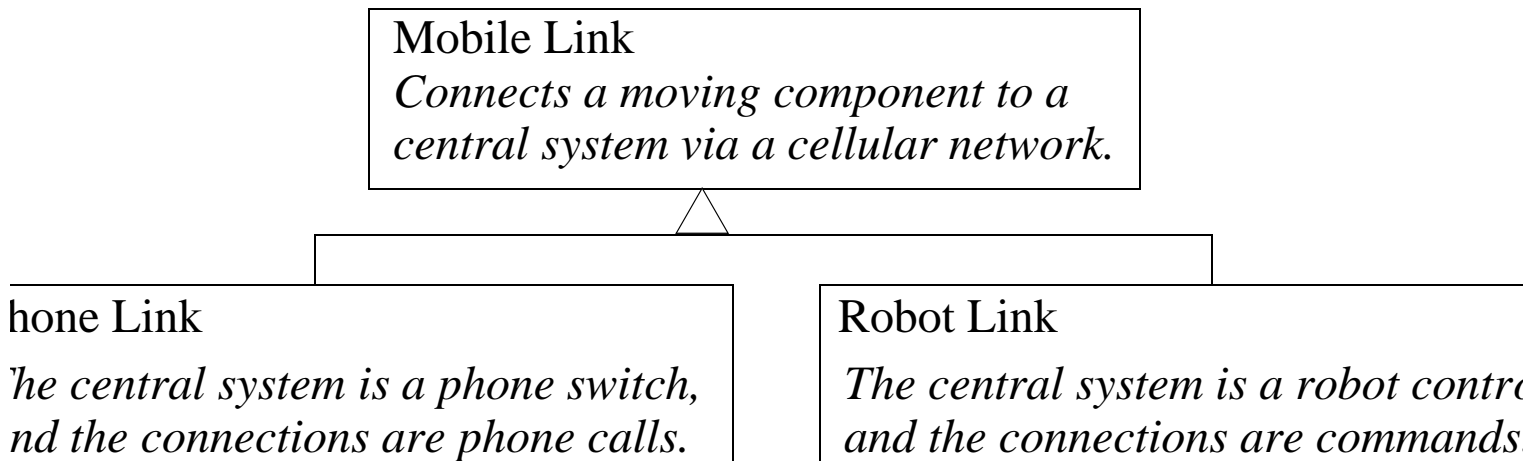


The special feature of the ‘mobile link’ is that communication is maintained even while the phone is travelling around: the network has a number of Base Stations dotted around the country, and the phone works through the nearest.

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 RobotVehicle would not get sensible commands from a PhoneNetwork, 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 could draw:



The subclasses are concrete: there are instances of them shown above. 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.

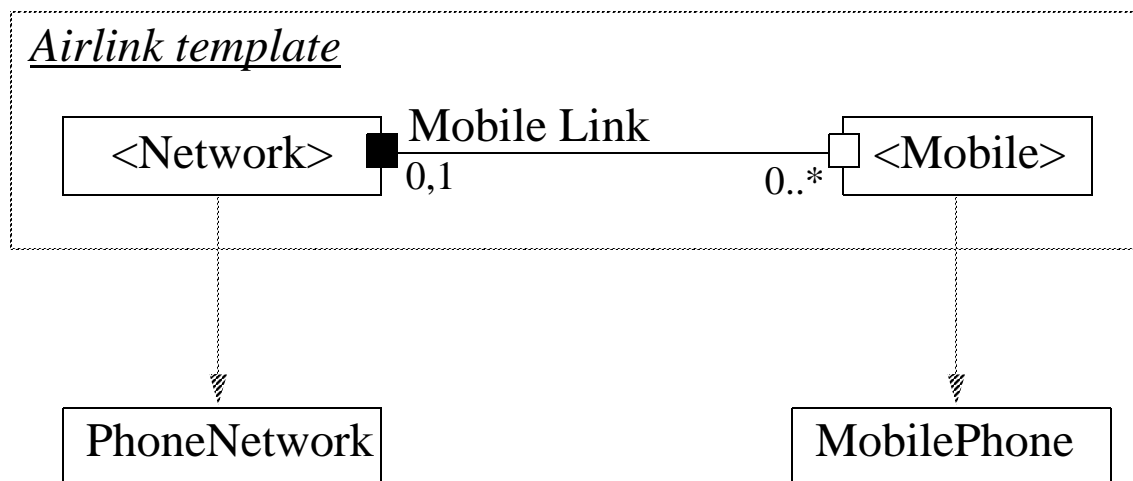
Model Templates

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:



We will shortly 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 RobotLink, substitute the placeholders with the real types Control Network and RobotVehicle (and provide additional information about the robot commands).

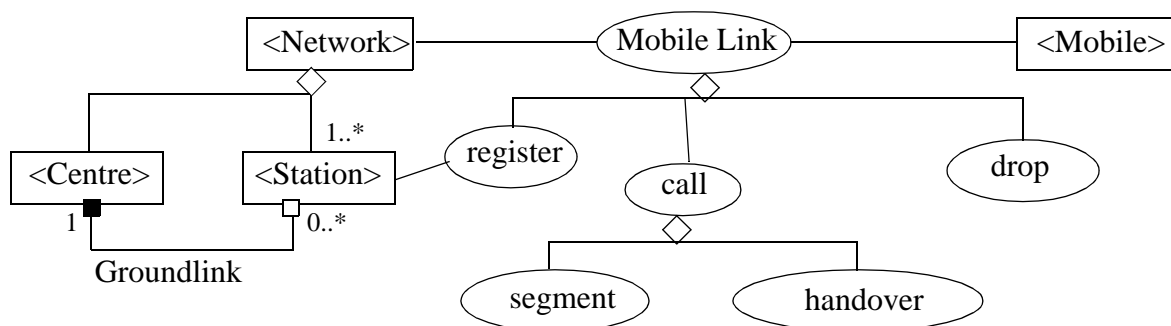
The ‘real’ types’ definitions include everything we attach to the placeholders in the template, but with the names substituted.



Notice that this does not make PhoneNetwork or ControlNetwork subtypes of Network: if that were so, we could couple PhoneNetworks to Robot Vehicles. The idea is to extend the entire template (in the dashed box) together, rather than each type individually.

Defining the connector protocol

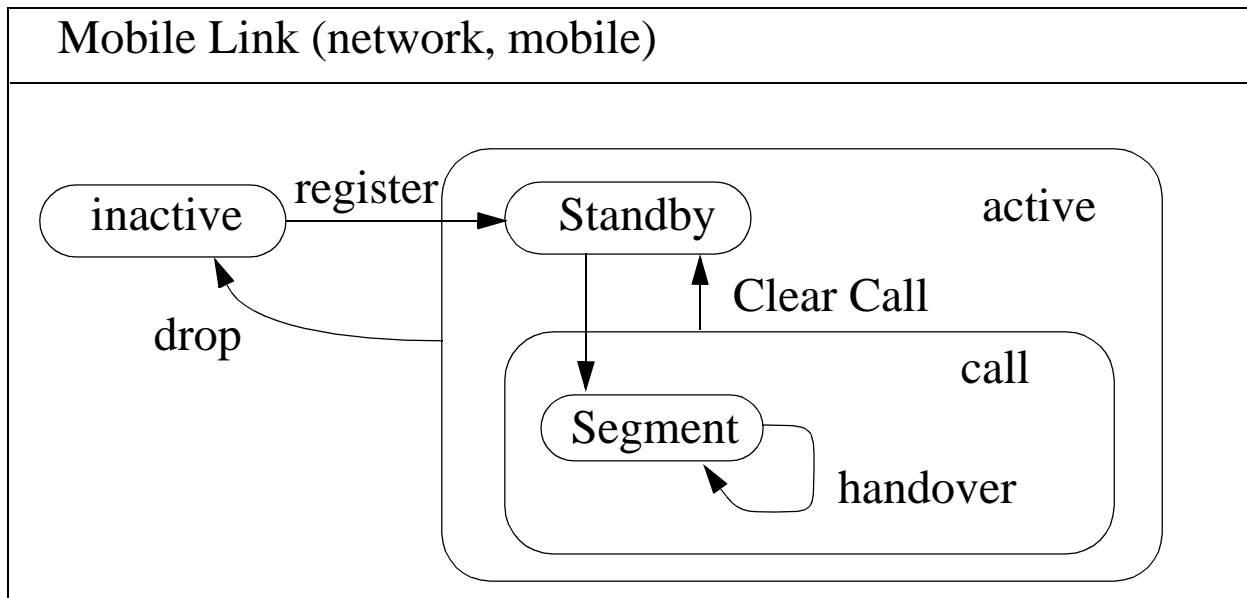
Of course, we can attach postconditions and/or refinements to the objects and actions in a template. A Mobile Link, as a connector, can be considered as an action, and can be decomposed into smaller actions:



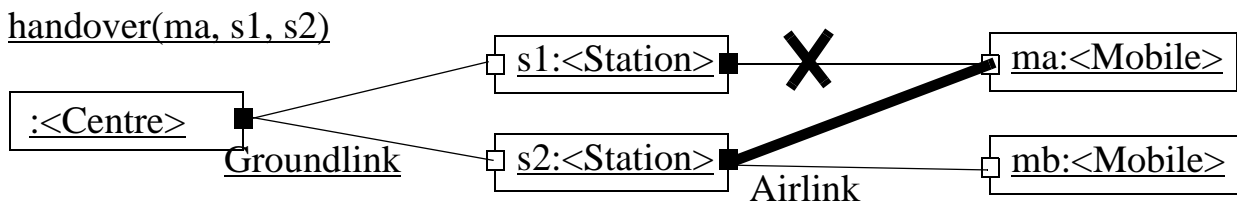
A Network maintains a link with its Mobiles via a number of Stations. When a Mobile is switched on, or comes within range of a Station, it registers with it; this enables the network’s centre to establish communication with a Mobile wherever it is. As the Mobile moves, it often ‘drops’ one Station and 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 can illustrate the possible sequences:



Let's go further into the Handover. The effect is to transfer the Mobile's communication to a different station. we can represent the Mobile-Station links with connectors, and draw a before/after snapshot:



A type diagram:



and an action spec:

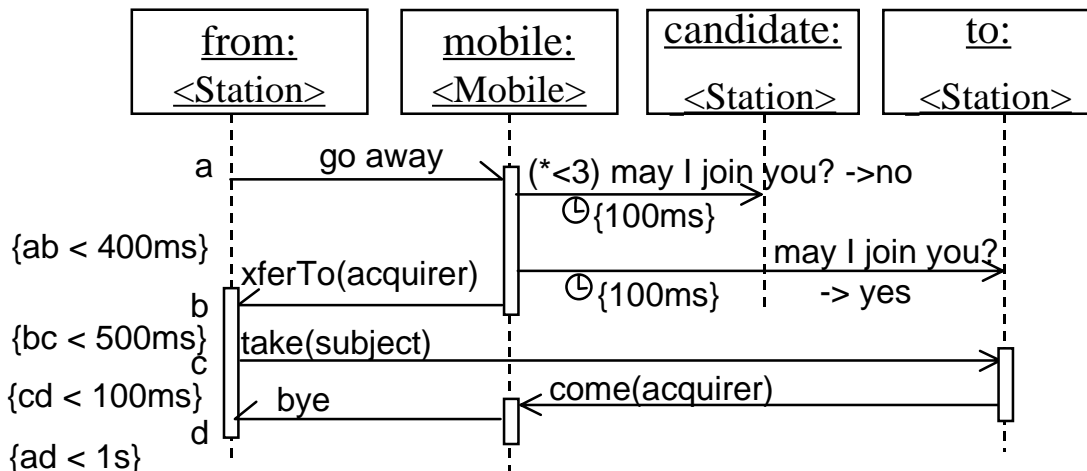
```

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'

```

That tells us what the handover achieves, 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

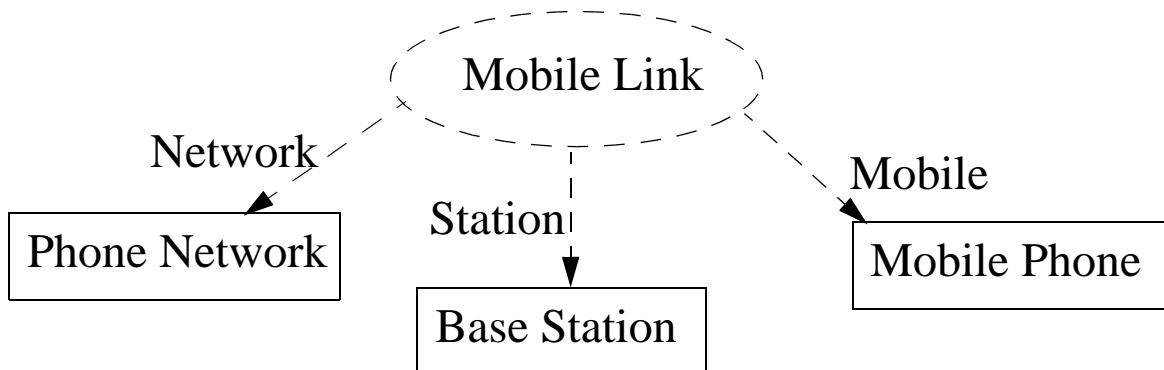
sequence chart --- let's choose the latter on this occasion; this version includes some timing constraints too:



(The Mobile keeps track of up to three Stations' signal strengths, and when the signal from the current airlink gets poor, 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:



This is like a macro application: it effectively creates a copy of all the diagrams etc we've 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.

Summary. 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.

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 an-

other, 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 aren’t 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. [Neierstrasz] 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. (Systems like Notes are an example.)

Processes for Component Based Development

The key features:

- 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.

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, 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 specs can be applied to it.

Catalysis techniques are particularly valuable both for component based development and high-integrity systems:

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

[Catalysis] Desmond F. D'Souza and Alan Cameron Wills, *Objects, Components and Frameworks in UML: the Catalysis approach*, Addison-Wesley, 1998.

[Meyer] Bertrand Meyer, *Object Oriented Program Construction*, 1988.

[Neierstrasz] Oscar Neierstrasz, paper in OOPSLA 98.

[UML] Martin Fowler, *UML Distilled*, Addison Wesley, 1997. See also <http://www.omg.org>

[UML-RT] Bran Selic et al: <http://www.objecttime.com>

[Warmer] Jos Warmer and Anneke Kleppe, *The Object Constraint Language*, Addison Wesley, 1998.