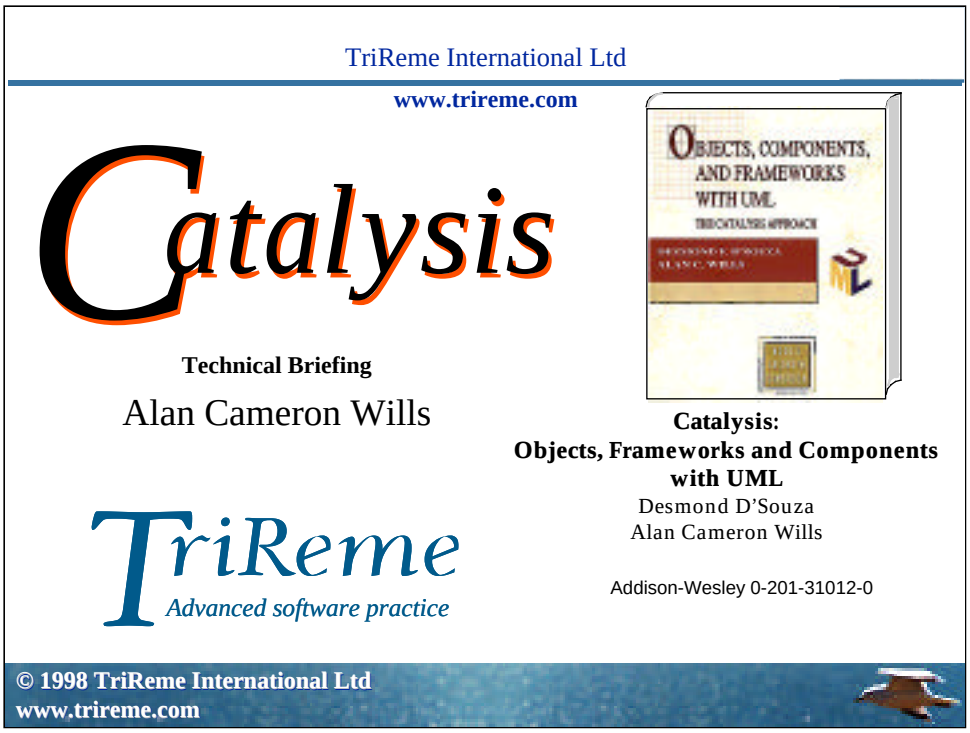
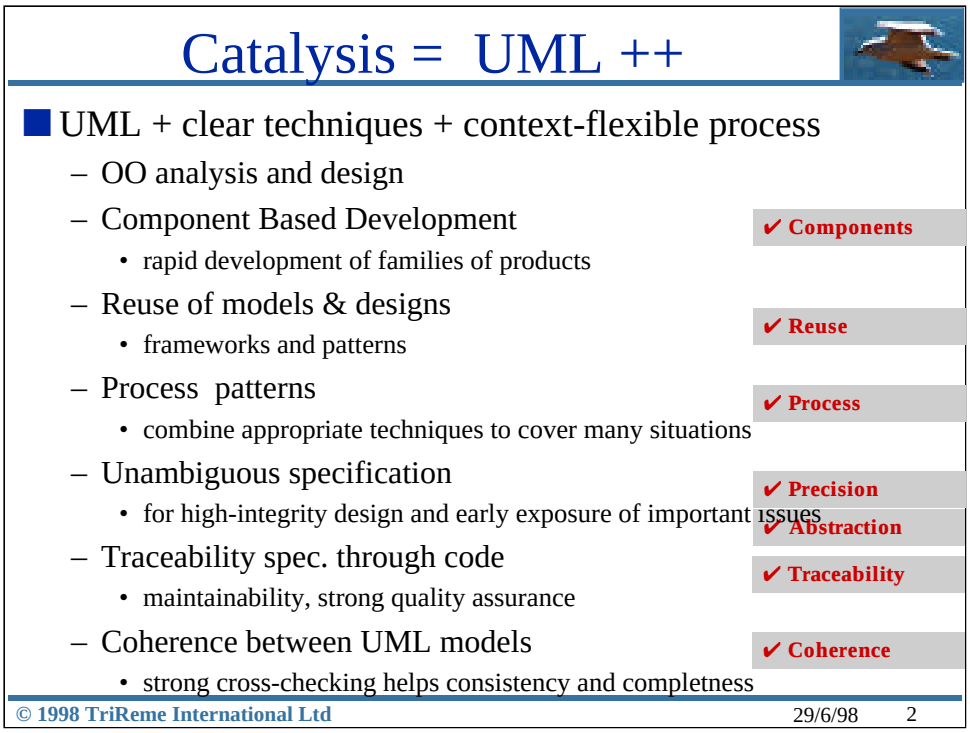


Catalysis is a method for component based and object oriented software development. It represents the culmination of several years' work by its authors Desmond D'Souza and Alan Cameron Wills, each of whom is a consultant and trainer of many years' experience. Catalysis is the result of their experience in consultancy with a wide variety of clients in diverse application areas including embedded, telecoms, and financial systems.

Alan Cameron Wills is technical director of TriReme, which provides mentoring and training in technical, management, and strategic issues in software development. TriReme's people are all widely-acknowledged experts in their fields.



This frame introduces the features and benefits that Catalysis adds to UML. The flags at the side introduce keywords that will appear in the rest of the presentation to highlight where we are seeing these features.

Catalysis provides a strongly coherent set of techniques for business analysis and system development using **UML**. It is now endorsed by a number of tool vendors, including Platinum and Sterling software (COOL enterprise-level development tools).

Catalysis provides a coherent method for **object oriented analysis and design**. The degree of detail at which the techniques are applied is variable, so it can be applied to both small and large projects.

Catalysis is specifically targeted as a method for **component based development**, in which families of products are assembled from kits of components. We also provide for **reuse** of other artifacts of the design process: frameworks of collaboration between objects are first-class units of development; many design patterns can be expressed this way.

Patterns also form the basis for a very flexible approach to the design process: rather than lay down one series of steps, we provide '**process patterns**, which guide the development planner in different sizes and shapes of project.

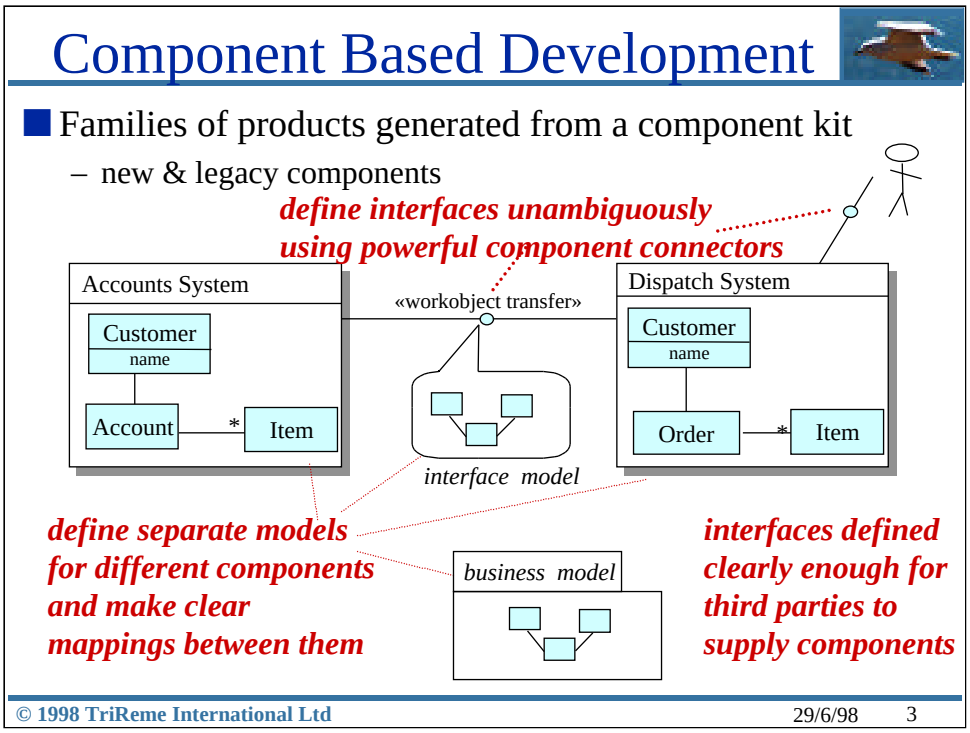
Catalysis provides a way to be as **unambiguous** about requirements as programming language is about an implementation. This has two big benefits:

• important issues are exposed early, that might otherwise be glossed over until coding;

• fewer misunderstandings about the meaning of a requirement or high-level design;

These are especially important in component-based design, where the readers and writers of an interface spec may have no other contact with each other; and for the design of high-integrity systems. For these systems, **traceability** is particularly important, and Catalysis includes techniques to map a system design to an analysis model.

The above features demand a strong **coherence** between the different models of UML. The models are intercoupled in a way that helps reduce gaps and inconsistencies.



Let's first see where we're heading for; then we can look at some of the techniques Catalysis includes, and afterwards come back to see how they are applied to get these benefits.

CBD is about building a family of software products from a kit of components. Some of the components may be adapted from existing systems.

Scenario 1: planning to market a range of applications. Instead of building one product from scratch, we do what car manufacturers do: plan a kit of components that can be assembled in different ways to form a family of products. Som e of the components may be reused from an earlier family.

Scenario 2: setting up a distributed system for your company that can be reconfigured as often and as fast as the business is reorganised (which is pretty often!)

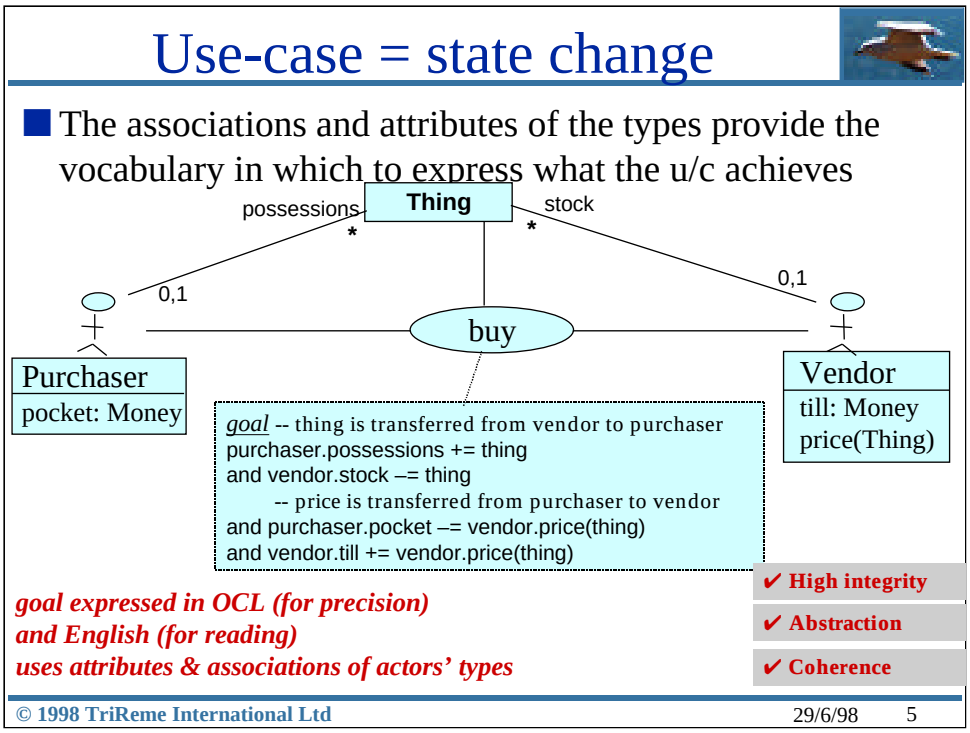
The trick of making a family of many products (or a system with many configurations) from one set of components is to make it possible to plug them together in different ways, rather like hardware components (think of chips, circuit boards, hi-fi boxes). This requires a small number of standard interfaces compared to the number of components: each component has one or more interfaces that can be connected to any other. To improve the range of interpluggability, the interfaces should allow some degree of negociation over capabilities (like faxes do, and cut-&-paste interfaces in OLE).

Catalysis provides:

- unambigous interface specification — allowing a chief architect to specify an interface standard, and third parties (who don't know each other) to make components that will interoperate with each other
- techniques to define powerful component 'connectors' abstracting above the level of OO messages (sort of like multi-pin plugs);
- 'retrieval' techniques for relating the differing models that different components (especially bought-in or legacy components) usually have — e.g. different notions of what a customer is.

Use case definition and refinement

- Use-cases are defined by what they achieve
- Use-cases and objects can represent collaborations of smaller use-cases and objects



Putting a little more flesh on the goal, we can write it more precisely by looking at the types of the actors.

This example shows how we can state exactly what the transaction achieves by modeling the cash and goods assets of the actors. The goal is a postcondition, a boolean expression relating the states immediately before and after any occurrence of the use-case. The formal parts refer to the attributes and associations of the actors, and can be type-checked.

Notice how we are using the type model to provide the vocabulary for defining the use-cases. This is an example of how Catalysis provides coherence between UML models.

Notice also how the goal is precisely defined, even though we have said nothing about how the effect is achieved. It encompasses mail-order, private transactions, shop sales, by cash or card or barter: the stated goal includes all these varieties and more. But by stating the goal precisely, it allows us to consider questions like the relationship between the amount of money and the goods transferred.

This example may be a domain or 'business' model, describing interactions between people or companies; but we can use exactly the same techniques in software design, to describe interactions between large or small pieces of software, or between software and people.

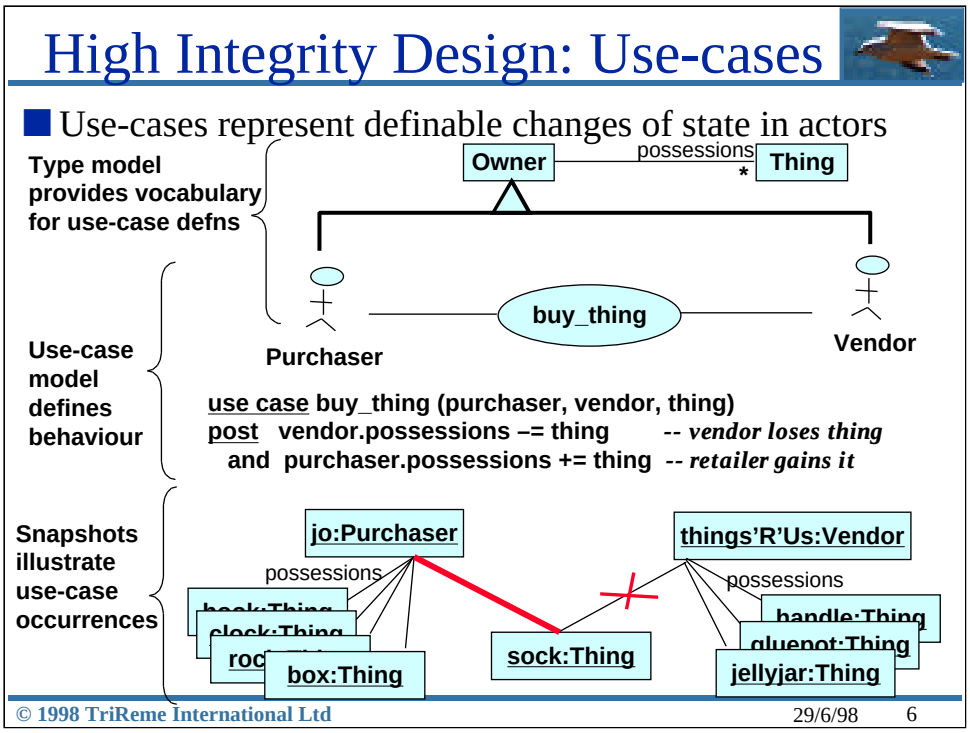
If this is part of a software specification, the goal can be used as the basis for a test harness. Quality Assurance people like the idea of test plans being laid early.

Notes:

* The goal is a postcondition, not an imperative statement. The idoiom x += y is here short for x == x@pre + y, where @pre is a UML keyword referring to the previous state.

* Other forms of goal include guarantees, which state what an ongoing process achieves for the duration it is in place.

* In UML, actors are types used in a specific way; those types can be drawn in a type diagram. In several of the current tools, attributes, associations and use-cases can all be presented on the same diagram like this; but that isn't essential to the technique.



(Here we've imporved the example type model a little.)

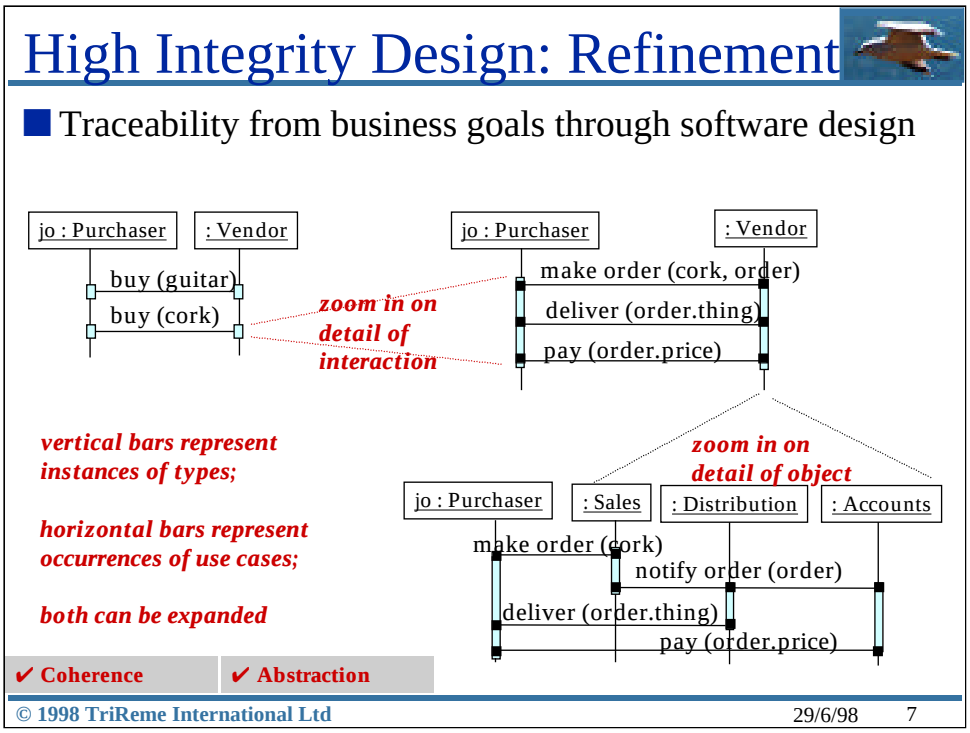
Snapshots are used to illustrate, animate, and help visualise goals and formal constraints.

A snapshot is an instance diagram that shows examples of the states immediately before and after a typical occurrence of a use-case. (The 'after' state is usually shown in red or in bolder lines.)

A snapshot must conform to the type diagram associated with the use case it is illustrating. Purchasers and Vendors in this example are varieties of Owner; every Owner has a set of possessions, each of which is a Thing: so the snapshot shows some owners each with a sheaf of possessions. The snapshot illustrates buy(jo, things'R'Us, sock), so we see a new link from the sock to jo, and the link to things'R'Us is deleted. Snapshots also typically show new objects; attributes written inside the instances may be crossed out and substituted.

(In a tools that doesn't support drawing lines and boxes in different colors, attach annotations "new" and "deleted".)

When a snapshot is used to illustrate a goal, it doesn't include messages between the objects: these are added in a later phase of design.



Sequence charts in Catalysis show occurrences of use-cases. The vertical bars are objects, and the horizontal bars are interactions between them — that is, use-case occurrences. (Messages between objects within a program are an elementary form of use-case; most methods use these diagrams at least for that purpose.)

The objects in the diagram should be instances of the types in the type model; and the horizontal bars should be instances of use-cases. (This is one of the ways in which Catalysis interrelates UML models.)

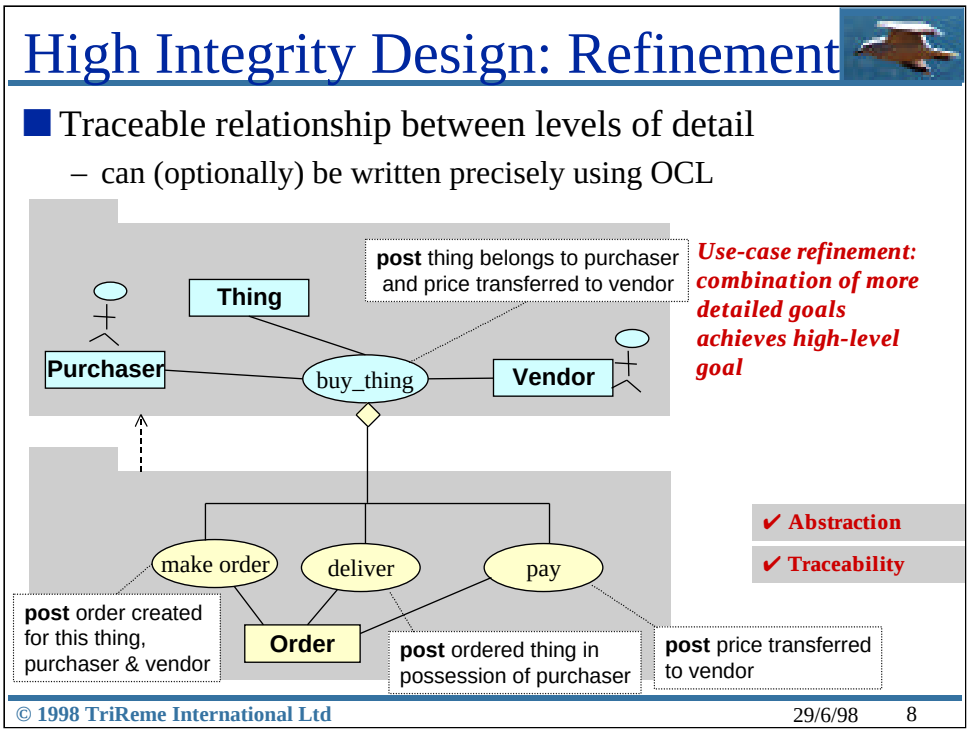
Refinement can be thought of as expanding a vertical and/or horizonal bar: ‘buy’ becomes a sequence of smaller interactions. We have also refined Vendor into different departments, and shown how the Purchaser’s interactions are actually with each of them.

Similarly, when I say “the ATM asked for my ID and I typed 4 at it” I could actually go into more detail: “the ATM’s screen asked for my ID and I typed 4 at its keyboard”.

It’s worth emphasising that *all these views are simultaneously true:* it’s just that some of them contain more detail about the truth than others.

The sequence charts show the two dimensions of decomposition fairly clearly. You can imagine continuing the process, breaking each department into people and support software, and including interactions between the people and the software; and also breaking the interactions into smaller details of protocol. At first we treat a piece of software (application, component, distributed system) as a single object; then we can break it into its major components, and break them into individual programming-language objects. With each decomposition, we show how the objects interact to fulfill the goals of the more abstract use-cases.

(However, we don’t usually design software in this top-down fashion! That’s just a convenient way of understanding the refinement relationship.)



The interactions that occur in the real world and within software are very complex. Any of them can be looked at in great detail ("I inserted my card in the machine; it asked for my PIN; I typed a 4; I typed a 2; … it asked which service I wanted; …) or in more abstract terms ("I got £50 from the ATM"). The same is true within software: we can talk about 'the GUI observes the Core' or 'there is a nightly transfer of accounts from the Boston to Zürich databases' rather than dealing with the fine details of the protocols of function calls or wire messages. At the same time, we sometimes *do* want to talk about the fine detail, so that we can work out how the interaction is achieved, or define what a component must do to be coupled to this interface. And we want to inter-relate the different levels of detail, so that we can say which detailed stories are implementations of which overall goals.

The principle of 'refinement' is that interactions and/or objects can be looked at in different levels of detail. The less detail we give, the more instances are included: there are many ways of buying a Thing, some of which involve some combination of making an order, taking delivery, and paying.

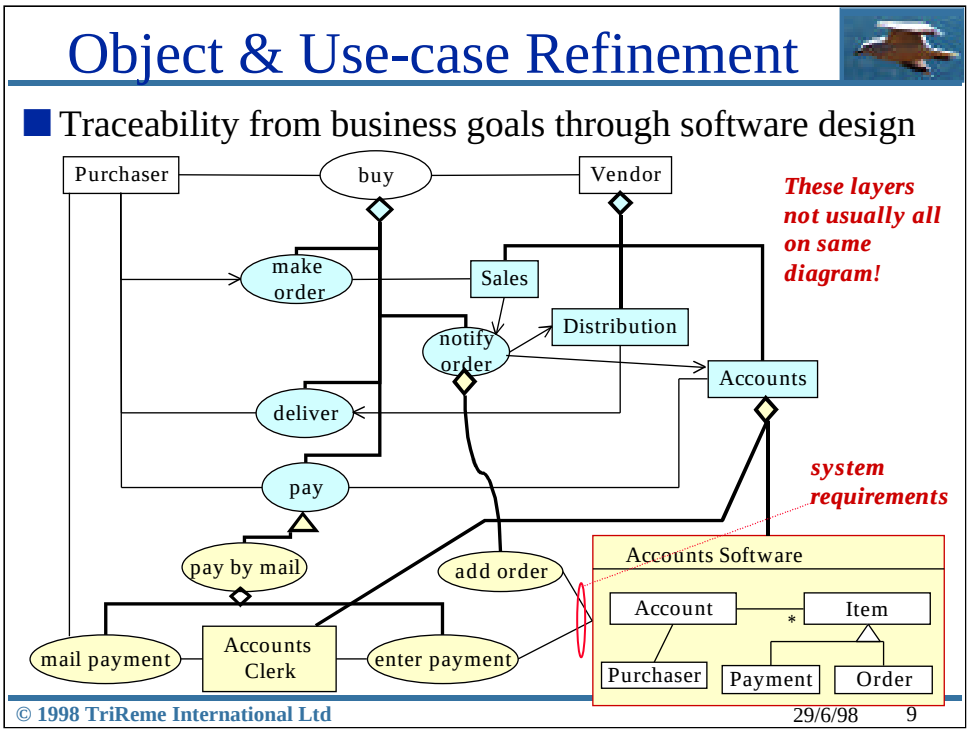
In Catalysis we use the assembly symbol to show that the more abstract use-case is made up of the more detailed ones. (Our convention is that wherever the assembly symbol is used, it should be meaningful just to look at what is above it.)

Each of the constituent use-cases can itself be defined with a goal and further refined to smaller interactions.

The finer level of detail always introduces new objects. In this case, the idea of an Order emerges: it is the link between the different stages of buying.

Typically, the different levels of refinement will be defined in different packages. Another package might describe a different way of buying.

The assembly symbol highlights the refinement relationship, but it doesn't say in what way to combine the constituent use-cases. Do they happen in sequence, or are they repeated, or do they work concurrently? We can activity diagrams to document this, or in straightforward cases, sequence charts. (Their goals often imply an ordering too.)



Sequence charts always show one particular example. Type diagrams summarise the relationships between obejcts and use-cases. Here we've shown the successive breakdown of both use-cases and the actors that take part in them. Notice how we can trace from business goals down to software requirements; and from there into software design; however, the layers wouldn't normally be all on one diagram.

The slide shows:

• Two layers of refinement of use-cases.

• An associated refinement of the actors, some constituents of which are software components; we've shown the Accounts Software package as an example.

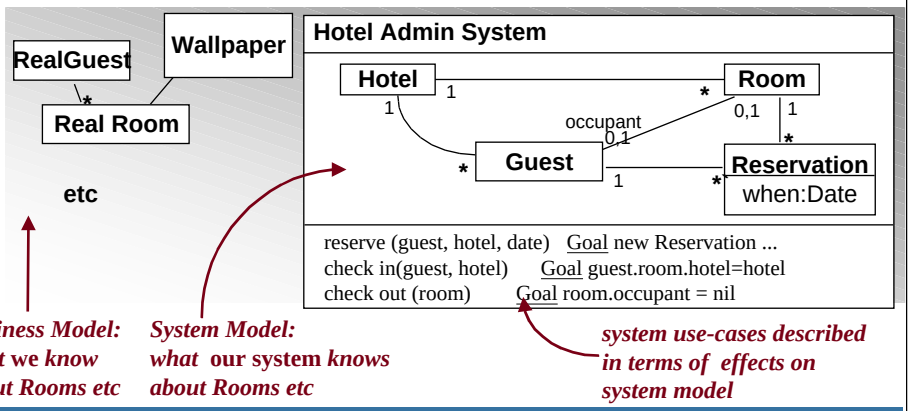
• The Accounts Software knows about Accounts, but probably not much about the sizes and weights of things — which the Distribution software probably knows more about. The types box labeled Accounts Software represents the fact that we're treating it as a single object at this level of refinement, and not discussing its internal design; the types drawn inside that box represent a model of what it understands about the world outside it. (However, they don't represent a design for its inner workings: there may be several different potential designs for an Accounts System, that all exhibit the same external behavior.)

• The Accounts Software component takes part in two use-cases ("enter payment" and "add order"): that is, it has two interactions with a person or another software component.

Each of these is part of a larger business use-case; and each of them represents some dialog of keystrokes, mouse clicks, API messages, etc that we haven't detailed yet. Each of them of course has some definable effect on the state of each of its participating actors, including the Accounts Software component.

Business model becomes system model

- Business model was about world surrounding our system
- System model is what our system knows about the world
  - just the stuff relevant to its responsibilities



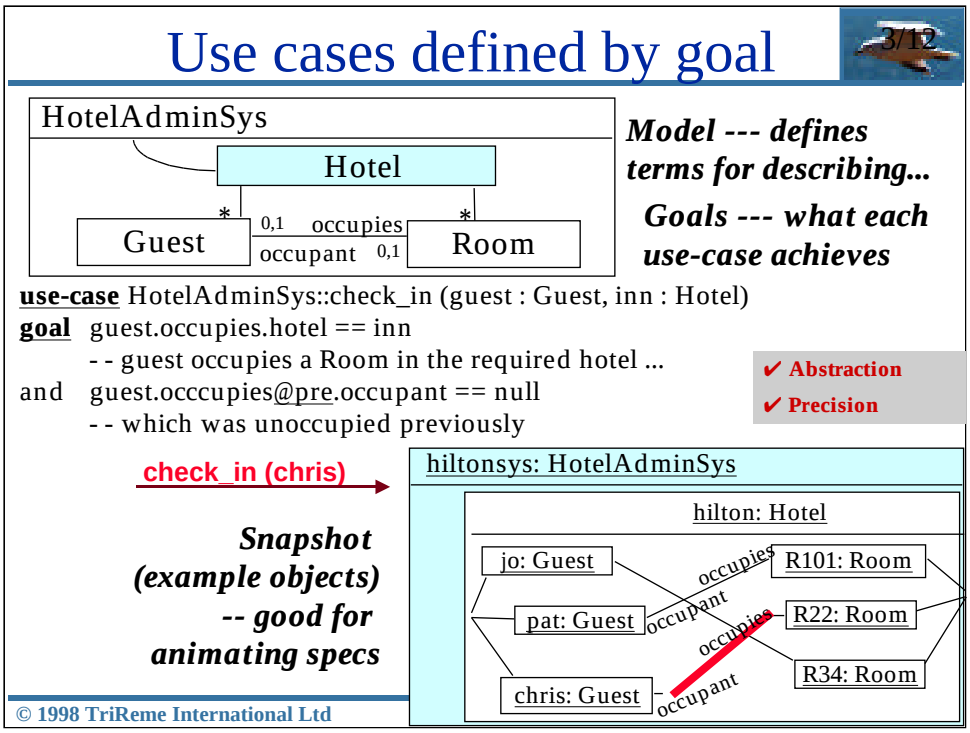
Business Model: what we know about Rooms etc

System Model: what our system knows about Rooms etc

system use-cases described in terms of effects on system model

There are two levels of types here: the outer objects like Wallpaper and Hotel Admin System, representing objects in the world of interest; and contained objects like Guest, which represent what the Hotel Admin System knows about Guests.

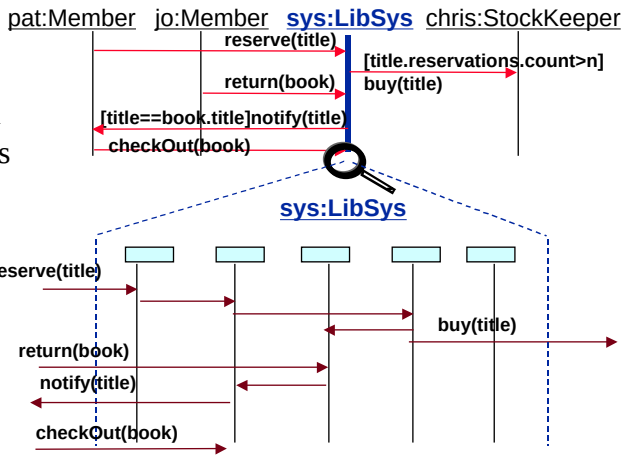
The contained types represent a model, rather than a mandate for the implementor. Any design is acceptable that produces the behavior expected from the use-case specifications; the contained model merely provides a vocabulary for expressing them. Only the use-cases in the bottom section are required. (Operations can be attached to contained types, but they are interpreted as 'factored' specifications: for example, we could attach 'reallocateRoom' to Guest; this would mean that the Hotel Admin System provides a facility of that name taking a Guest as a parameter; but still says nothing about how the system as a whole might implement that operation.



Here we see snapshots again, this time used to show the effect of a use-case on the state of a system to be designed.

Specification --> Design

- We now understand what our component is supposed to do
  - Scenarios of how our system is used within the business
  - Use-cases
    - Context diagram
    - Goals
- Next: look inside system
  - Scenarios
  - Use-cases



So far, we've:

— Made a model of the domain, to understand:

- what objects there are
- the goals of the major use-cases the business is about

— Drawn scenarios of how our component will help realise the major use-cases, breaking them down into individual steps

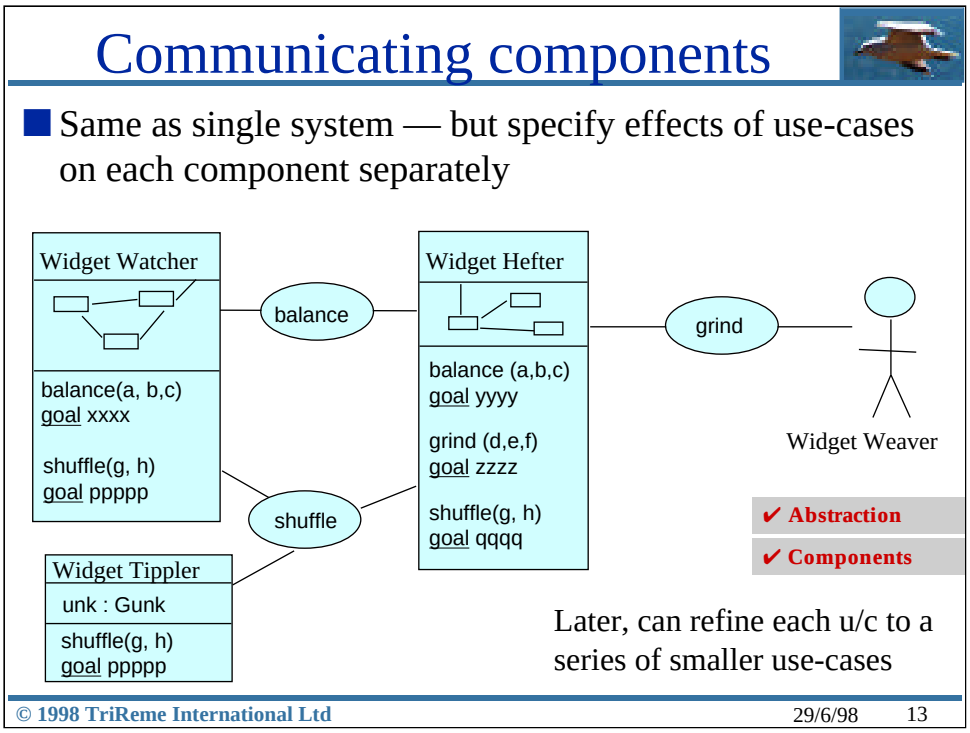
— Collected the individual use-cases in which our component is involved, at the level of discrete transactions between it and the actors surrounding it

So now we know what our component is supposed to do, and are able to envisage it as a spec with contracts, just as for an individual object. The component has some defined responsibilities, and defined collaborations with the actors around it.

We have, in effect, done some design: the goals of the business have been met by designing a business process with our component as one object within it. Whether the other components are people, hardware, or other software objects, the principle is no different. (In some projects, the external design will already be done, and you'll be starting at this point — perhaps after clarifying some of the goals.)

The next task is to distribute our component's responsibilities between objects inside it. This is like zooming in on the vertical bar in the event charts that represents our component. The horizontal interactions coming from outside should be the same: but now we can see more detail about the objects inside our system and the interactions between them. Notice in our illustration here, that the messages coming in and out of the component as a whole are unchanged.

The notations we'll use are essentially the same, since we're just doing more of what we've done already.



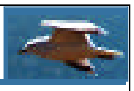
One of the benefits of this rigorous view of use-cases is that they can be used to define the interactions between components at a high level. At this level, we care neither about the internal workings of the components (just the effect of the use-cases on them); nor about the fine details of the interaction protocol (just the overall effects of each interaction).

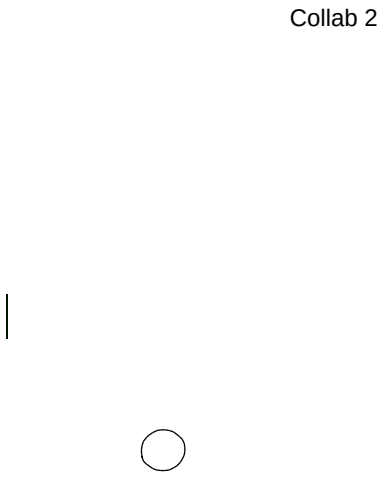
However, this view is specific to a given set of components. We'd like to define generic interactions between components, so that we can make them 'pluggable'. To get there, we must look at models and template models.

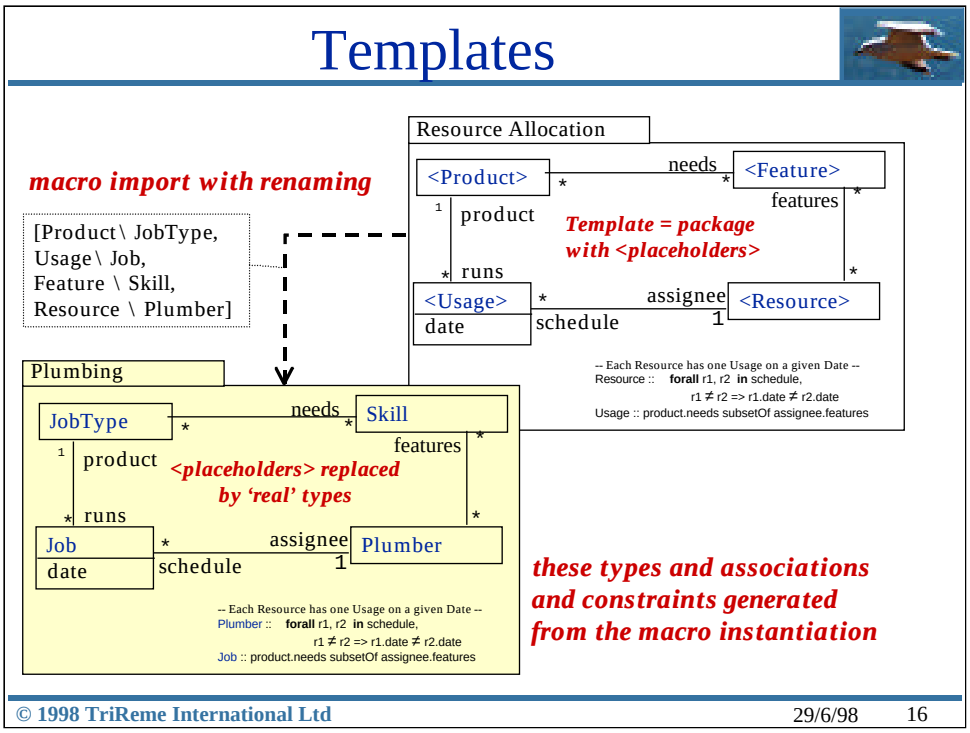
Model frameworks

- Frameworks: collaboration schemes as design units
  - including pluggable component connectors

Collaborations as reusable chunks







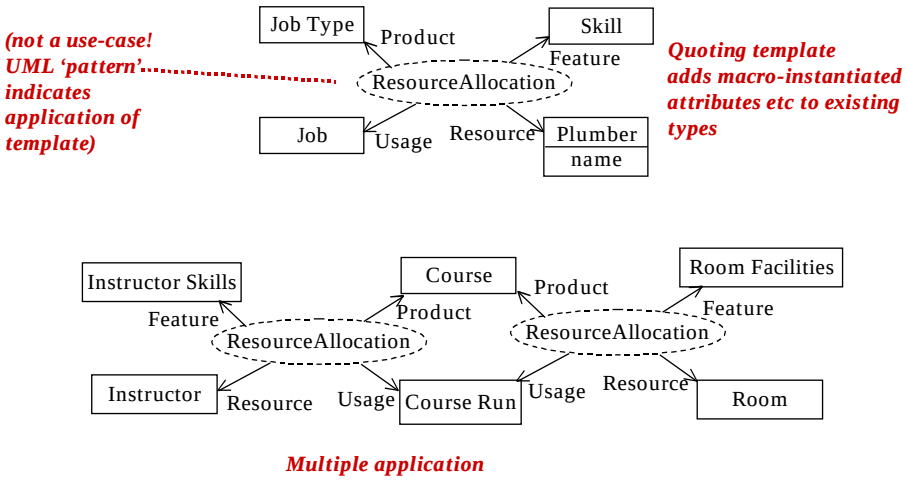
A template is a chunk of model (in a package) with some 'placeholder' names that can be substituted in order to make a model. This example is limited to static associations and attributes, but shows the substitution at work. The example is a general description of the relationships between any scheduled resource and the features it provides.

In the generated model, no plumber may be allocated to more than one job per day, and the plumber allocated to a job must have the necessary skills.

(JobType – things like 'install dishwasher' or 'clear drain'; Job — an occurrence of a JobType on a particular day.)

Template application

- Pictorial notation

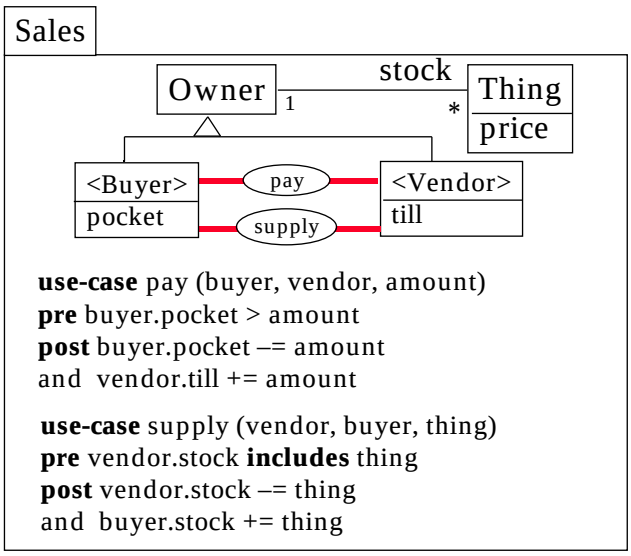


✔ Reuse

This notation shows the substitutions graphically. It also makes it easy to add extra features to the types, as well as those imposed by applying the template.

A template can be applied several times, to the same group of types. The second example shows allocation of both Instructors and Rooms to courses. (Instructor Skills: things like C++, UML, …; RoomFacilities: things like OHP, PCs).

Collaborations

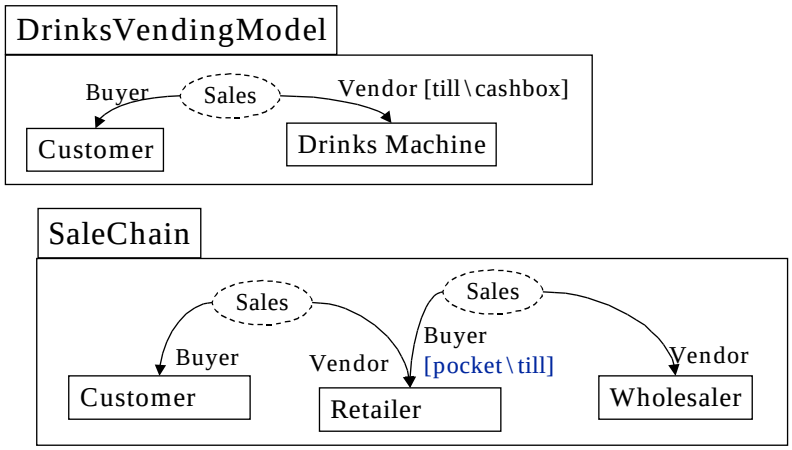


 29/6/98 18

This example shows a template used to define use-cases. (Although in this case, they are probably use-cases in the domain, the same technique applies to component and object interaction use-cases.)

Instantiating Generic Models

- Templates provide protocols

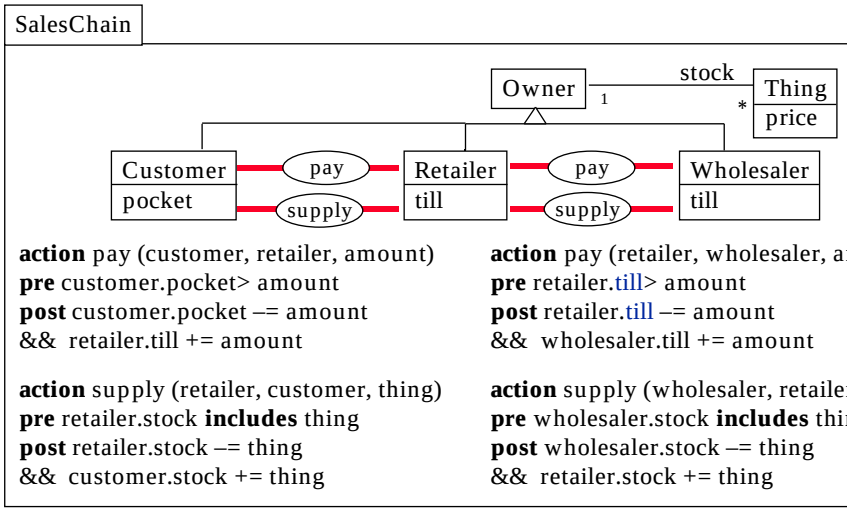


✔ Abstraction

✔ Components

Applying the sales use case.

Unfolded composite



action pay (customer, retailer, amount)
pre customer.pocket> amount
post customer.pocket –= amount
&& retailer.till += amount

action pay (retailer, wholesaler, amount)
pre retailer.till> amount
post retailer.till –= amount
&& wholesaler.till += amount

action supply (retailer, customer, thing)
pre retailer.stock **includes** thing
post retailer.stock –= thing
&& customer.stock += thing

action supply (wholesaler, retailer, thing)
pre wholesaler.stock **includes** thing
post wholesaler.stock –= thing
&& retailer.stock += thing

 29/6/98 20

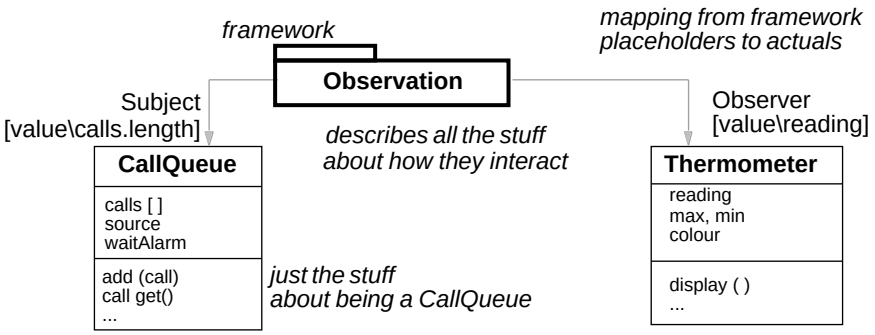
The resulting model (“unfolding”).

Notice how the till of the Retailer is now affected by both sets of actions: as we said at the beginning of this section, the state of an object is where the collaborations it takes part in interact.

Notice that these are specifications: it is still up to the designer of each type to produce a class matching the resulting spec.

Framework application

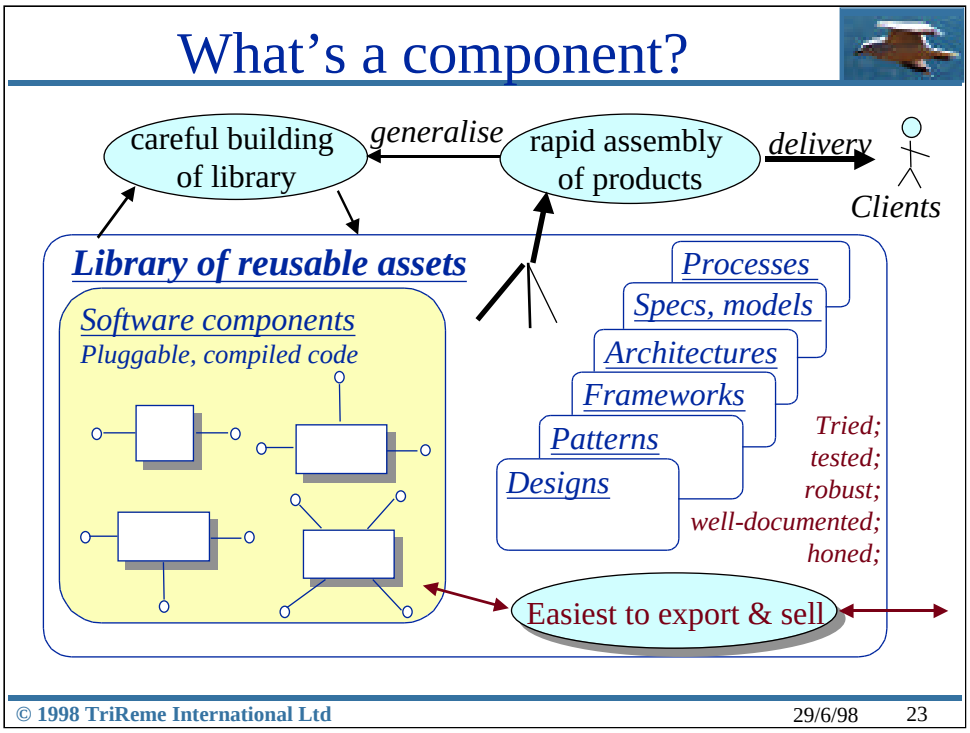
You draw an application of a framework:



Tool works out complete model by applying framework(s) t

Components

- Component kits need architecture
- Interfaces can be defined with framework use-cases
- Component models need mapping to common business model



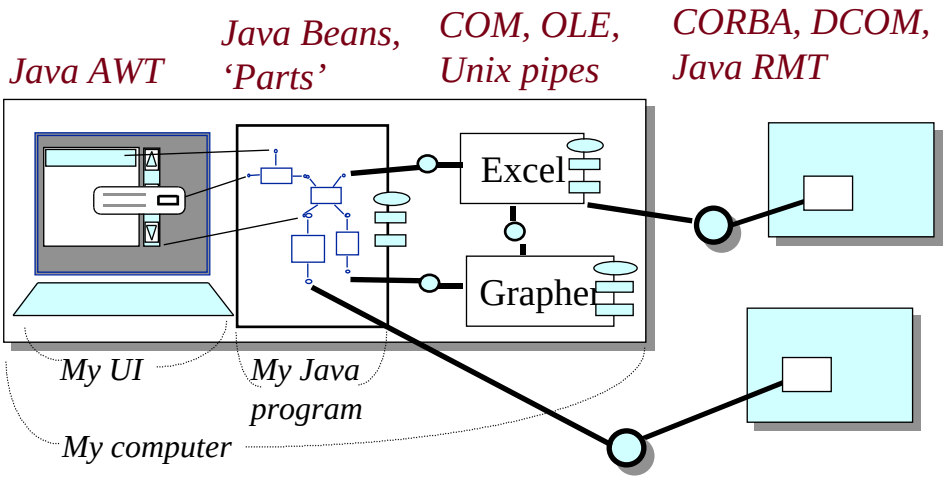
- Generic chunk of software* with robust, well-defined interfaces
  - –can be used with other components to make a variety of end-products
  - –hardware analogies
- Software development --> two activities:
  - –Rapid assembly of end-products from components
  - –High-quality component development

* Other reusable assets include: designs, specs, patterns, frameworks, architectures, ...

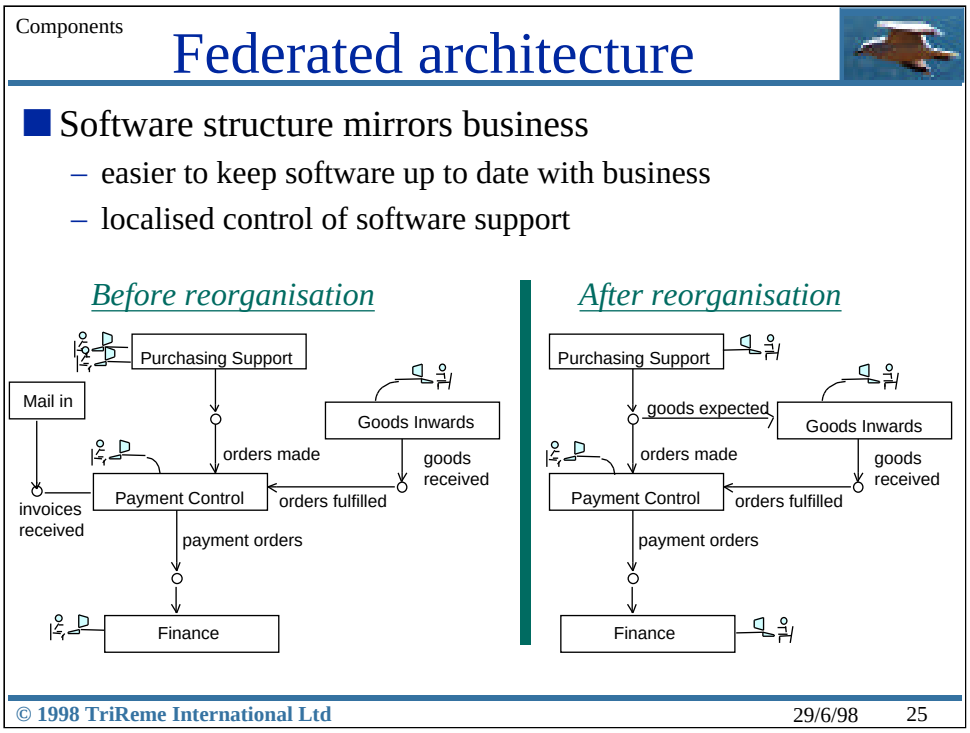
  - – Components are easiest to sell — black box

Where do we see components?

- different scales and boundaries ...



- **GUI builders**
  - Standardised visible widgets
- **'Beans', 'Parts'**
  - Java, C++, Smalltalk pieces — built for a given language
- **Object libraries**
  - –Infinity, ...
- **COM, OLE, Unix pipes, ...**
  - Cross the language barrier
- **Federated systems: CORBA, ActiveX, ...**
  - Cross the machine barrier

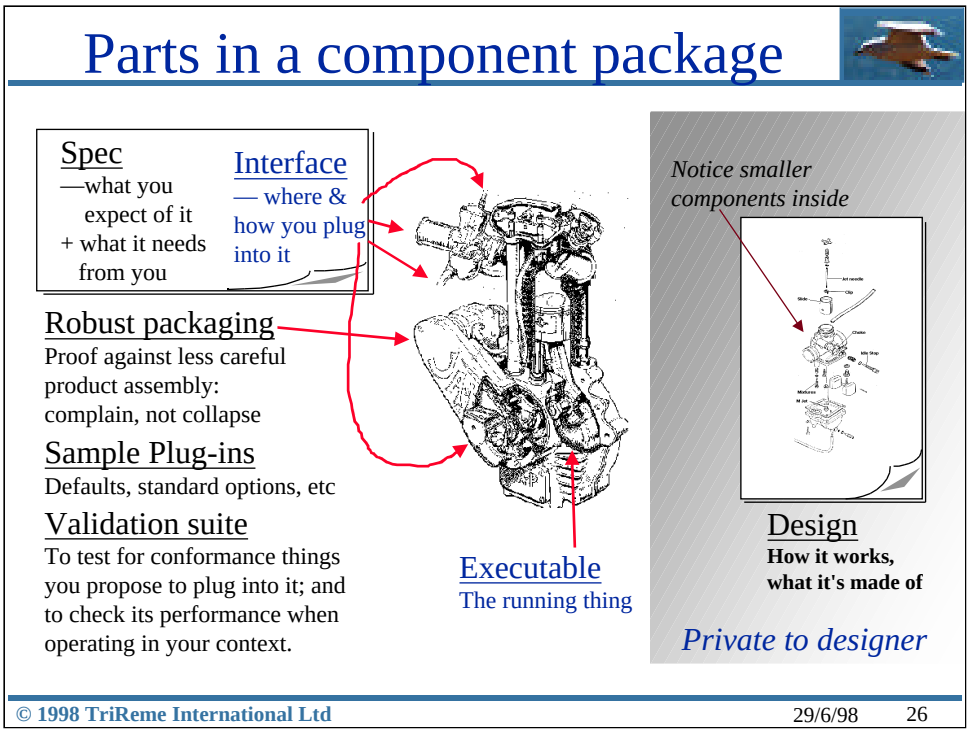


A federated architecture provides a subsystem that supports each business function. These machines (workstation or mainframe) should talk to each other in exactly the same pattern and pathways as do their users.

This brings several advantages over traditional central database systems:

- Each subsystem is under it's owner's control --- empowering the users to set it up with the support functions for the way they run their shop. This feels better and is more responsive to change, than the situation where everyone has to negotiate with the DP department.
- When the business is reorganised, it is easier to reorganise the software. For one thing, it's easier to see what will be affected; and also the changes just mirror the business changes.
  The illustration shows a firm that used to suffer from 'dumping' of unordered goods. They didn't find out about it until the invoices turned out not to match the purchase orders. They reorganised so that the Goods reception got a listing of everything expected. Are the diagrams of the business or the software? Both.
- Most system crashes only affect one department at a time.

The downside of federated architecture is a tendency to replication: Purchasing have their list of purchases, and Goods Inwards have their list of expectations. However, the advanced technology of replication and the cheapness of disk space make this much less of a concern than in the heyday of the big database.



Packaging components

A component is more packaged than any old object. The assumption is that it will be used in many contexts unknown to its own designers. It should be robust in respect of abuse from other components, complaining rather than collapsing.

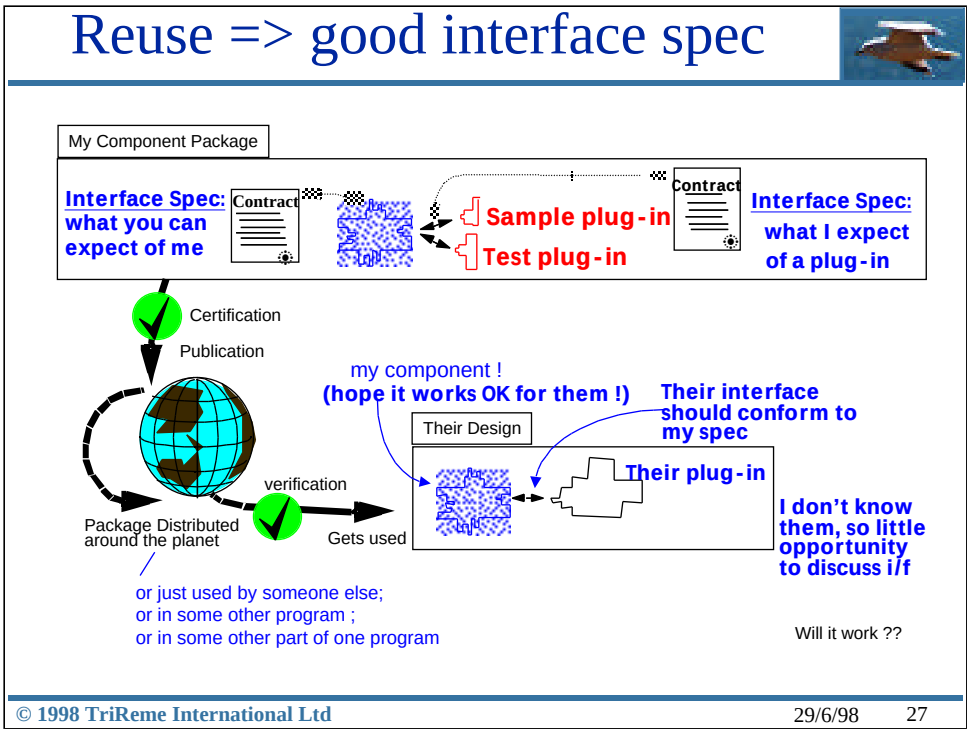
In addition to the executable code itself, there should be a specification documenting its behaviour unambiguously, using a suitable modelling and design notation. Since the average component will be used more than an individual product, it is, even more than usual, worth investing in good specification and design. The specification is essential because clients do not have access to the design, and should not have to waste time experimenting. A clear specification also tends to prolong the life of the designers' original vision, through many updates and enhancements.

Some component architectures require that each component should be able to answer at run time, queries about their connections.

The validation suite is a set of ancillary components for two purposes:

- One set help test the component once it is installed in a particular context, to provide test cases etc, and ensure it is running properly.
- A test version of our component exercises the components it is connected to, to make sure they behave as we require.

Not all of our client designers will wish to use all the customisation plug-points that we provide. We therefore provide default components for connecting in these cases, that provide standard behaviour.



Dynamic coupling of components is just one form of reuse.

Other forms include the import of a generic chunk of design into many other designs.

In this sense, "component" can include any piece of development work:

- code
- models
- high-level designs
- business rules
- design and analysis patterns
- plans
- etc ...

In every case, the essential thing is that the designer of the component does not have the opportunity to discuss the interface with every potential user. Nor can the component be designed to be dependent on any given feature of the other components with which it may work.

Therefore, it is much more necessary (than in a traditional, all-in-one room design) to define the interfaces clearly and unambiguously.

That is the topic of this talk.

Interfaces: just lists of operations

- Interfaces can be described by lists of operations

```
interface Vzrtpgk
{       void        shjhaa      (Grk);
        Grk         akfe        ( );
        int         bvbvb       ( );
        boolean     cmnn        ( );
}

interface NuclearPowerPlant
{       void        raise_rods  (int);
        int         temperature ( );
        void        lower_rods  (int);
}
```

```
class ZZNu2PL
    implements Vzrtpgk
{ int  shjhaa (Grk c) {…code…}
 ...

class Magnox
    implements NuclearPowerPlant
{ void raise_rods (int n)
   { …code...
```

 29/6/98 28

Looking at these examples, it's clear (for different reasons) that a plain list of operations is not enough as a behavioural spec.

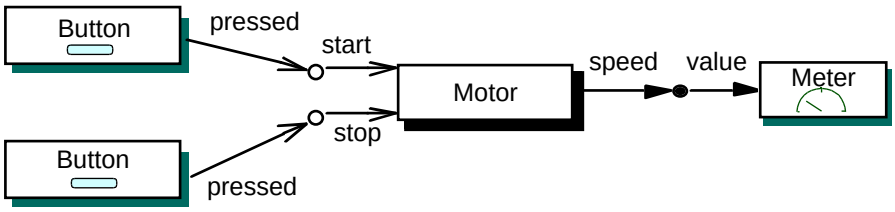
In one case, it's apparent that the only reason we accept lists of operations as specifications is that the names usually suggest the expected behaviour. But of course, people can interpret them differently.

The other example highlights that the need for proper specification can be a serious matter. Many software systems these days control systems upon which people's lives or livlihoods depend. We have to be like proper engineers rather than small-time joiners. (That needn't preclude us being both engineers and artists, just as the architects of buildings are.)

Therefore, we should write more substantial specifications, defining not only what the operations are, but also what they do. I see this as one of the biggest motivations for using rigorous modeling techniques.

But there's also another point to be made here: that in the real world of component-based software, the idea that an interface is a list of operations — specified or not — is not really powerful enough. We'll now look at some examples that use more abstract interfaces.

Component interfaces



- Important properties of component interfaces:
  - each interface general enough to be plugged with many others
  - each component well-specified enough to be able to use without looking inside it

 29/6/98 29

The idea of component-based development is to build software from parts that can be rapidly reconfigured. There are two development processes: one in which people rapidly plug components together to produce a bewildering stream of new products, vanquishing all competitors who are still developing things laboriously from scratch; and another process in which people are carefully building components that can be plugged together by the assemby team. This works best if the components are built as a kit.

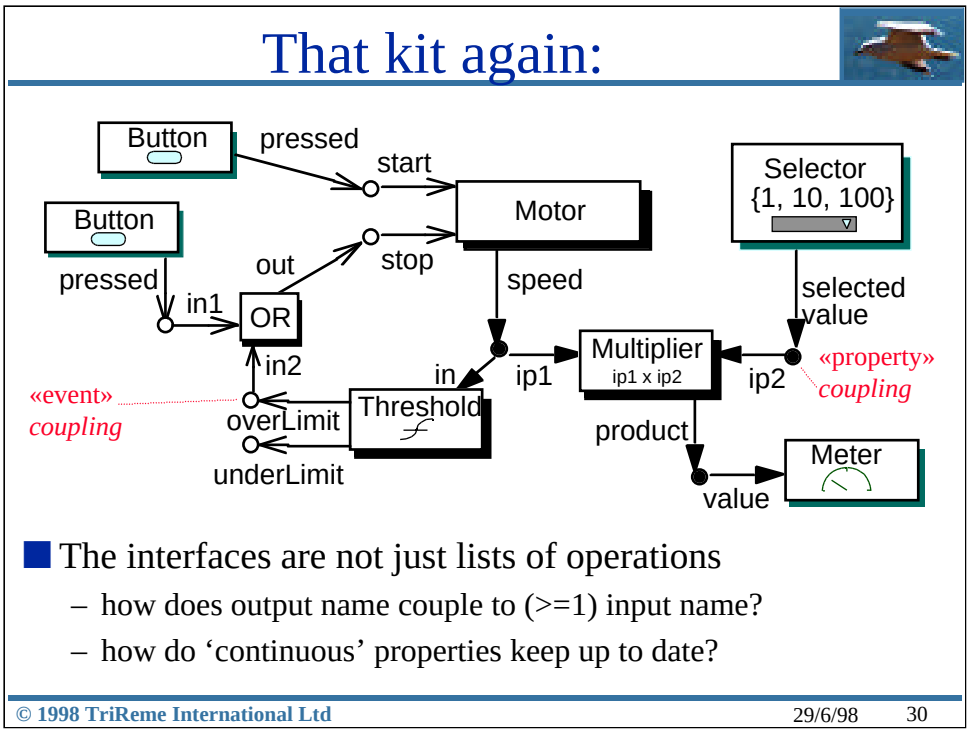
Hardware people have been doing this for years: here is an example intended to press that analogy — the components can be seen as either hardware or software. We'll discuss what principles can be drawn from this, and later go on to discuss how the same principles apply to large-scale components.

The idea of a Component Kit is that the pieces in the kit can be easily plugged together in many configurations --- just like electronic chips or mechanical components. You build products by picking components out of the bag, and plugging them together. Occasionally you build some specialised components yourself. Often a particular configuration you've created can itself be packaged up and sold or put into the library as a component.

The example shows a Motor controller (a software object that controls a piece of hardware), which comes out of the kit bag with three labelled connections: two inputs and an output. These connections have types (just as conventional variables or subroutines do) and can be coupled to any connection with a matching type. Here we've instantiated a couple of user interface Buttons, and a Meter, a user interface widget that displays values. The interesting thing about the speed/value connection is that, like a piece of electrical wiring, it continuously conveys the current value of the Motor's speed.

A Kit can work only if there is a standard set of mechanisms for the couplings: defining how, for example, the two Buttons are linked up to the two differently-labelled inputs of the Motor; and how the Meter's value is kept up to dat e with the Motor's speed.

A definition of a set of couplings is called a Component Architecture.



Here we've pulled some more bits out of the bag, just to show how flexible this kind of kit can be. A Selector is a menu; a Multiplier is a component that does not correspond to any piece of user interface; a Threshold is a component that sends an output every time its value input goes above a certain value.

The two most interesting things about the "Kit" idea are:

- The notation we've used for components helps think about what the configuration does, without bothering with the detail of the couplings.
- The idea of a consistent Component Architecture that defines what kinds of coupling there are, and how they work.

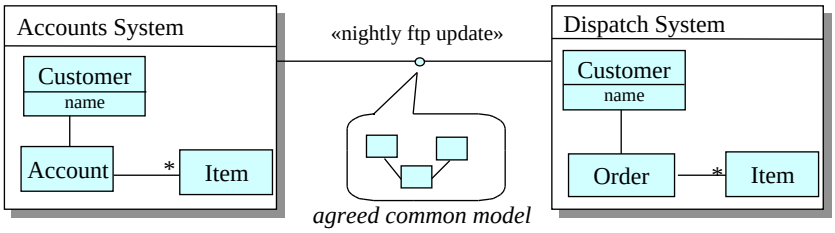
Members of different kits can be interconnected, but require some sort of adapter.

A common architecture means that tools can be help build systems by coupling components together. Digitalk's PARTS and IBM's VisualAge are have been doing this for a few years; both are based on Smalltalk (in which it's particularly easy to make dynamically flexible interconnections). Sun's Java Beans provides a Component Architecture, and the Beans Development Kit isthe corresponding visual builder tool. In C++, there are several visual builder tools, though mostly focussing just on GUI components.

Components in this sort of kit are typically written all in the same language, and execute within one machine. But they don't have to be, as we'll see next.

Large components

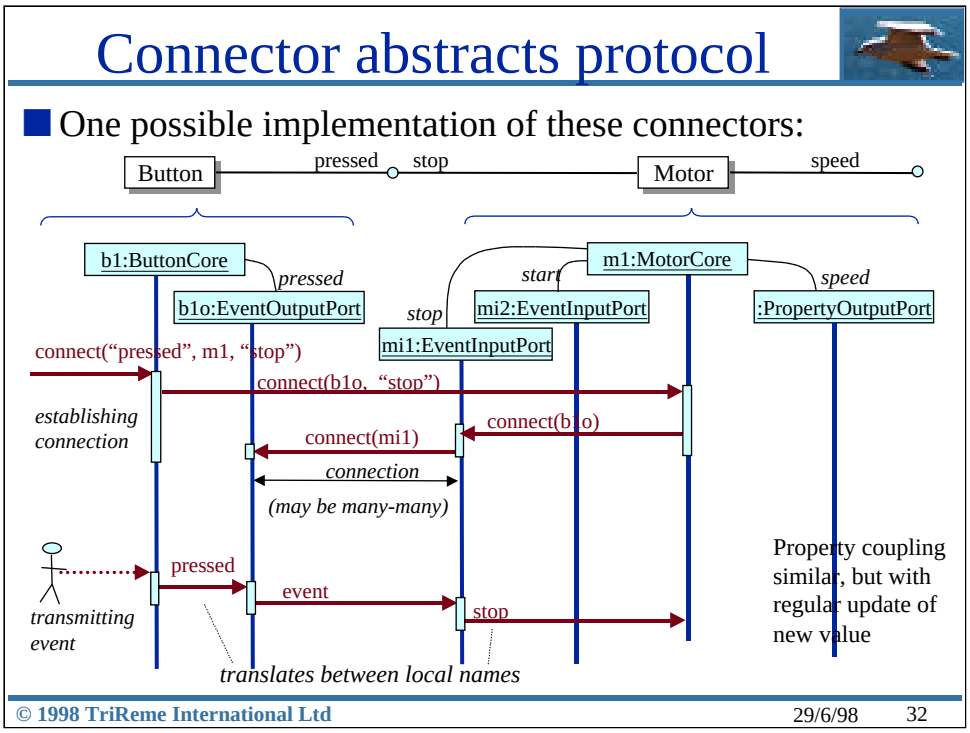
- Additional useful connector types
  - Workobject transfer
  - Transaction-based update
  - …
- Values identify business objects (not just numbers)
  - Need to define common models between components



The same principles apply to large components that make up enterprise-wide distributed systems. The nature of the connectors can be ‘bigger’ — for example, batches of information get moved in one go; and are transaction-conscious, with the possibility of rollback etc.

Another aspect that comes out at this scale is that the information passed in each interaction is substantial, about business concerns — customers, aircraft, phone calls, whatever is the subject matter of the domain. And as an extra complication, each system will tend to have its own internal model of each of these concepts; sometimes they will use the same word for different things.

Along with the definition of the connectors must therefore also go the definition of a common model and representation of the concepts dealt with on the interfaces. Like the technology of the connectors, this is an essential prerequisite to building a kit of reconfigurable components.



Each connector is an abstraction that hides considerable implementation detail: it is above the level of individual object-oriented messages. Furthermore, there are many ways of achieving the same functionality — so this just shows one possible scheme.

In this example, we first see that each component turns out to be made up of several objects: in these examples, a 'core' object responsible for the functions of the component, and a separate object for each port.

First we see some third party instructing the Button to connect its "pressed" output to the Motor's "stop" input. The Button knows the mapping between labels and actual port object, so it passes the message on to its "pressed" port object; which in turn applies to the Motor, which passes the request on to its own "stop" port. (We assume it's the core objects' business to know which objects are its ports; the rest of the world must just use their labels.) The two ports can then communicate with each other, and arrange to be connected. In particular, the output port keeps a register of all the input ports interested in anything it has to say.

Later, when the Button is pressed by a user, it gets its output port to send a message to all interested inputs; the message sent is one that is common to all ports of this type — which is what enables them to talk to one another, despite the difference in labelling between the connected ports.

This scheme achieves the componentware goal that reconfiguration is possible at run-time: that is, no recompilation or special programming is required to make the new connections. This means that each component can be a complete black box, with no source available.

There are of course other schemes that would have the same effect; but components can only be plugged into others that conform to the same scheme or "component architecture": we say they belong to the same kit.

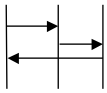
There are several types of connector. The «property» connectors are connected in the same way, but send messages every time some change in a given value occurs (such as the motor's speed). There are of course, different types of value, so each port has to check at run-time that it is being plugged into a compatible port.

Connectors and Ports

- Connector type:
  - An *effect* defined by postcondition or guarantee

«property»

- Connector class:
  - A *protocol* defined variously by CCS etc
  - Implements a given connector type



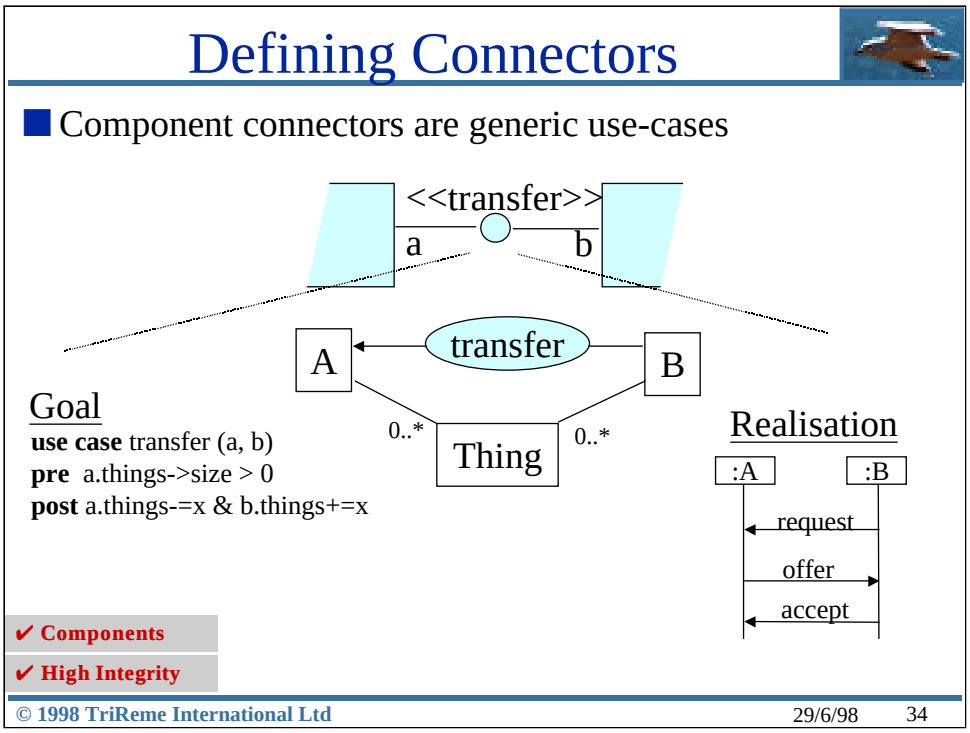
- Component:
  - An active element that can be connected to others

- Port:
  - The capacity of a component to take part in a given class of connector

The essential thing about a connector is that in defining it, you are not specifying any particular component: just defining how components interact to achieve a specific kind of interaction. (In Catalysisthese are called collaborations.)

Just like objects, connectors are encapsulated: the specification of what one achieves is independent of its implementation; and there can be subtypes etc.

To make an effective kit of components, you must first define the connectors; and define them carefully enough so that third parties that do not know each other can construct components that will work together. Again, hardware people have been doing this for many years.



The essential thing about a connector is that abstracts the details of an interaction.

So a framework can be used to define the connector.

In our previous examples of frameworks, we envisaged the framework being applied to a single model --- so that the Subject and the Observer or the Buyer and Vendor are all in the same design.

But in the component world, I write a component that only has one end of the interaction.

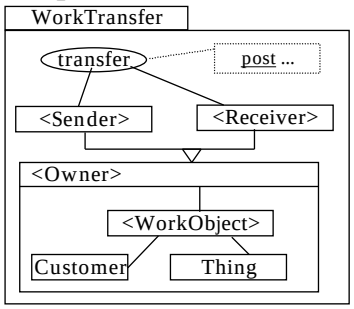
The framework tells me what messages I should send and what effect messages to me should send; and I can map the types and attributes in the framework into my own classes.

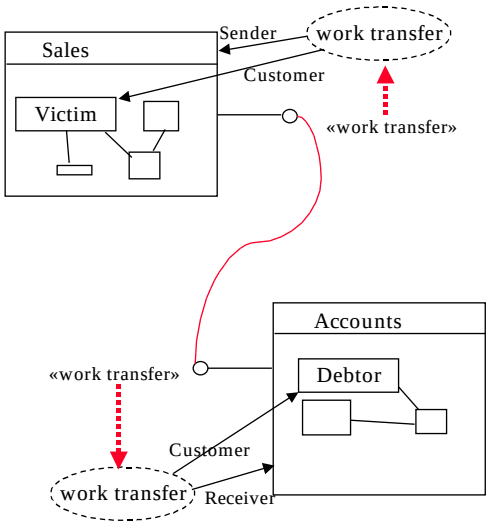
The framework also gives me a much simplified model of the other end, just enough to explain to me what I need to know about the aspect of their behavior that I see (Subjects have some sort of value; Vendors have a till they put the money in; Parents have heaps of money) without telling me anything about what else they do or are involved in. This has the usual decoupling benefit of ensuring that I design my end without unduly depending on any properties of their end that are none of my business.

We’re beginning to see at least three designers involved here: the architect who sets the definitions of the connectors in some evolving system; and the different designers who design components whose connectors conform to those specifications.

Model mapping

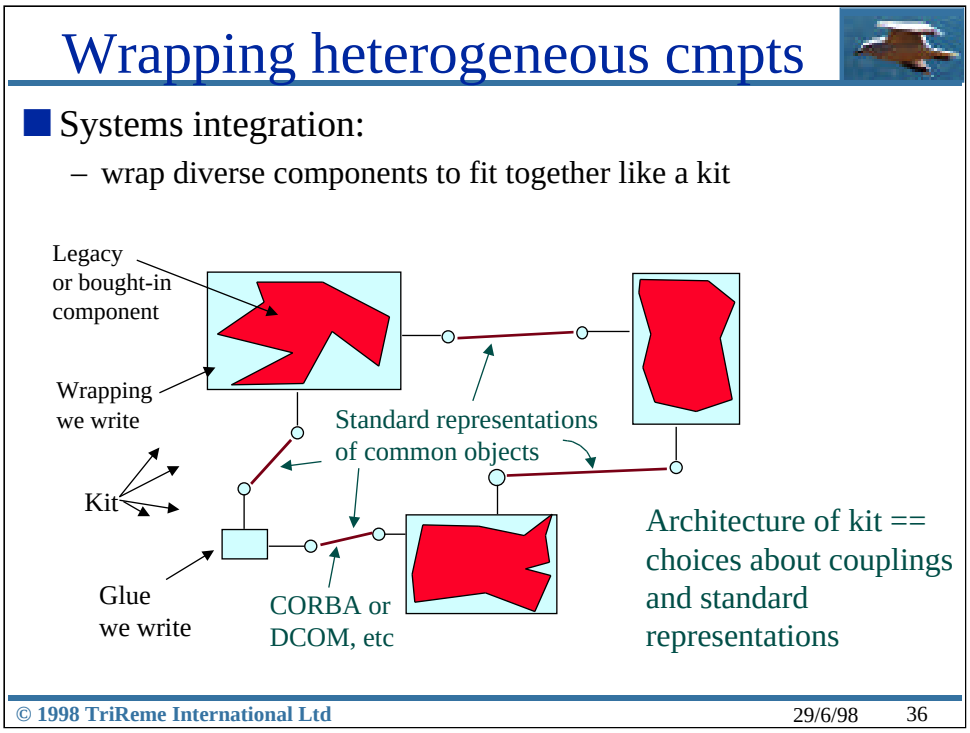
- Model in connector framework mapped to implementing component's model





allows the different components to have different models, and to have much bigger models than the connector framework is concerned with

✔ **Components**



In the light of the discussion about kits of components, we can get a better understanding of the systems integration job we started with.

Schematically, the task is to "wrap" each of the diverse components so that it forms part of a kit, with couplings that will work together.

There's a short-term/long-term choice here: either we can wrap them just sufficiently to work in *this* particular configuration; or we can put in a bit more effort, and make the couplings general enough so that they will work in a few different ways, and perhaps with some other related components we know about.

A Kit --- even if it's only going to be used in one system --- always has an Architecture: a set of choices about how the interconnections will work, and how the different internal representations of what (say) a Customer is, are converted into a mutually understandable representation for transmission through the couplings.

A well-though-out, clearly-defined architecture emerges as the most essential prerequisite when considering using components of any kind.

Further detail, and a substantial worked example, can be found in:

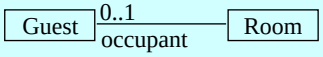
http://www.trireme.com/catalysis/book/C11.pdf

Other aspects

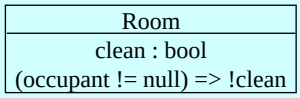
Documents

- Pure diagrams are write-only!
  - Good narrative should surround diagrams
  - Complement the readiability of natural language with the precision of UML

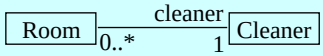
Each Room can be occupied by at most one Guest. (The occupant is the person responsible for payment. There may be others staying in the room, but we are not interested in them.)



The room may be clean or unclean; an occupied room is always considered unclean.



Each Room is assigned one Cleaner.



The Cleaner may have several Rooms.

Combine diagrams by giving each class all the attributes and operations from all its appearances.

requirements document, for example, should deal with different aspects of the model separately: each should have a short piece of description in natural language, followed by short formal diagrams or text (postconditions etc) to ensure the point being made is not misunderstood.

This illustration is dealing with the static aspects of Rooms, and will presumably deal with dynamic stuff later. Notice how the Room class appears several times. The interpretation is that each Room has all the attributes shown in the class's various appearances, including occupant, clean, and cleaner.

The expression "(occupant != null) => !clean" (meaning "if there is an occupant, then clean is false") is an *invariant* --- something that should always be true, at least while no operations are executing.

Invariants, like pre/postconditions, can be written on the class diagram, or within the narrative text, or in the Dictionary.

To 'scope' a postcondition or an invariant, write "AClass :: ". This means "the following applies to every member of AClass". (In some versions of OCL, you leave out the '::', but underline 'AClass'.)

Process Patterns

- "Process pattern" refers to the process of software or system development & maintenance.
- Of the form:

  Whenever your goal is XXX
  and your current situation is YYY
  then try doing ZZZ
  (but be aware of prerequisite PPP, risk RRR, side-effect SSS, timescale TTT, ...).

- The idea is to capture the wisdom of software strategy and tactics in digestible chunks
- See also “Organisational patterns” (strategic) www.bell-labs.com/cgi-user/OrgPatterns

 29/6/98 39

One of the problems with prescriptive software methodologies is that they tend to assume a fixed starting point --- for example, that the developer is designing a product from scratch. In practice, that’s a rare situation; a believable programming process should cater for many different starting points.

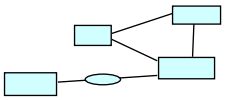
We havetried to cast our process advice into a set of discrete patterns. Each is in a ‘production rule’ form: trigger ---> action. The actions often set sub-goals, for which other patterns can be found.

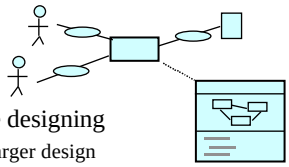
To use a process pattern, you find all the process patterns that apply to your current goals and situation: these form the basis of your plan. There may be several, that describe different aspects of the plan: for example, about building the team, estimating timescales, as well as the main outline of the plan. Because they are informally expressed, you need to apply your intelligence to choose and fit them together! It’s not a mechanical process. But by choosing a pattern (or perhaps combvining several), you start on the development of

Each pattern will yields some subgoals, which you can then hope to find some other patterns to match. Choosing and applying these, you put more detail into your plan.

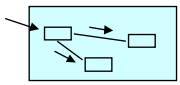
Layered design decisions

- Business ("Domain") model
  - understand what you're dealing with
    - Types: what objects there are
    - Use-cases: what they do





- System requirements spec
  - agree role & behaviour of what you're designing
    - System context: system as one object in larger design
    - System spec: goals of each use-case system takes part in

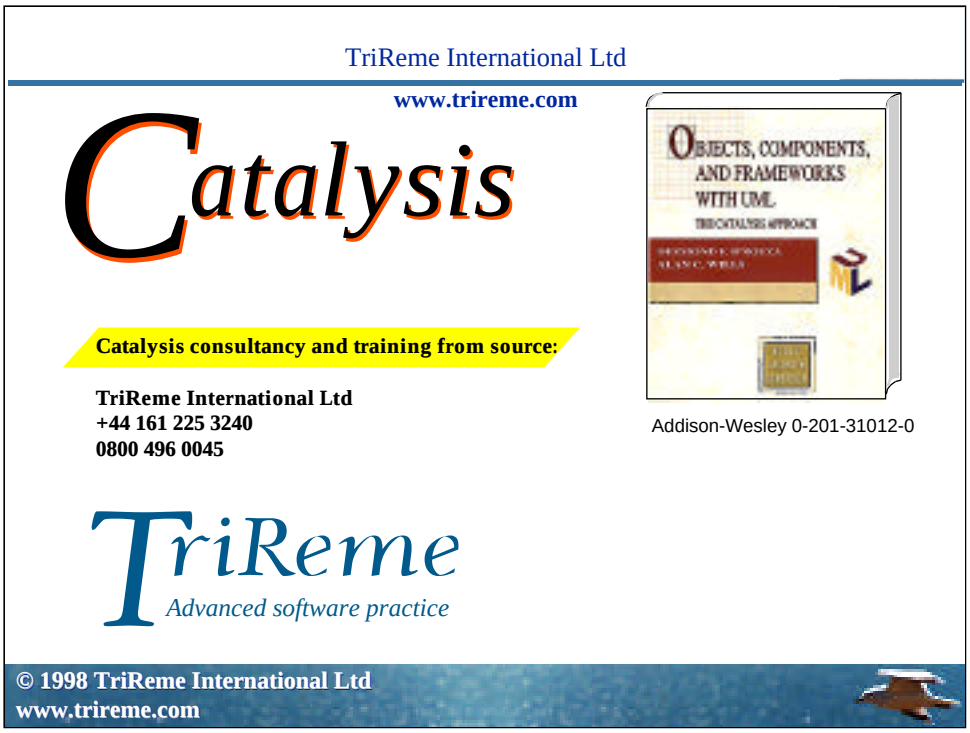


- System design
  - work out how it works inside
    - Collaborations and responsibilities of objects inside system

Catalysis: strengths

- Component Based Development ✔ Components
  - powerful interface abstractions, strongly specified ✔ Abstraction
  - mapping between differing component models ✔ Traceability
- High IntegrityDesign ✔ High Integrity
  - robust, reliable software
  - rigorous specifications define test harnesses ✔ Precision
  - clear semantics, coherence between UML models ✔ Coherence
  - traceable refinements ✔ Traceability
- OO Analysis & Design
  - precise models expose gaps & inconsistencies early ✔ Precision
  - abstractions good for whiteboard discussion and documentation ✔ Abstraction
  - fractal method: same techniques at all levels ✔ Process

If they are to be used in many situations, component interfaces need to be carefully defined. Careful specification and development are also needed for the increasing number of embedded and other systems on which money and lives directly depend. Catalysis couples rigorous specification techniques to UML, in an accessible and usable form that allows the designer to choose the degree of precision appropriate to



Catalysis is a method for component based and object oriented software development. It represents the culmination of several years' work by its authors Desmond D'Souza and Alan Cameron Wills, each of whom is a consultant and trainer of many years' experience. Catalysis is the result of their experience in consultancy with a wide variety of clients in diverse application areas including embedded, telecoms, and financial systems.

Alan Cameron Wills is technical director of TriReme, which provides mentoring and training in technical, management, and strategic issues in software development. TriReme's people are all widely-acknowledged experts in their fields.