Chapter 10 Model Frameworks and Template Packages

Building specifications and designs from generic reusable parts.

Outline

It isn't just chunks of code that can be made into reusable assets. Designs and specifications too can be separated into parts, which can be kept in a library and subsequently combined in many different configurations. We call these model frameworks.

The basic tool for representing and combining frameworks is a generic form of package, called a template (or template package).

Of course, while program-code components just need to be plugged together to produce executable code, pieces of design combine to produce only designs, that still need to be implemented. But if you've read the book this far, you'll agree that a design represents much of the major decision-making that goes into finished code: so that being able to put a design together rapidly from prefabricated parts is a very valuable facility.

This chapter deals with model frameworks, and how to build and compose them using template packages.

10.1 *Model framework overview*

Recurring pattern = model framework

Once you've been modeling and designing object systems for a while, you start noticing certain patterns recurring. We can see the same set of relationships, constraints, or design transformation in different designs. This set of relationships, we call a 'model framework'. Many design patterns boil down to a model framework, combined with surrounding how-to, when-to, and whether-to advice.

Can build models by applying patterns

A suitable tool should be able to support building models and designs by application of model frameworks. Suppose, for instance, we have already some type 'Stock' in our business model, with a numeric attribute 'level'; and choosing a package of user interface pieces, we find another type 'Meter' for displaying numeric 'readings'.

e.g. Observation framework

Now we want to specify that Meters can be used to display Stock levels, using the well-known Observation[1] pattern. As usual, we can focus on different aspects of a model in different diagrams, and so we don't have to repeat all the stuff that has already been said about the two types. All we need do is define the extra attributes and operations needed to connect them.

"Application" is easy

This is where model frameworks are useful. Let's assume that because Observation is such a common pattern, we have defined a model framework for it; using this gives an abbreviated way of defining what we need:

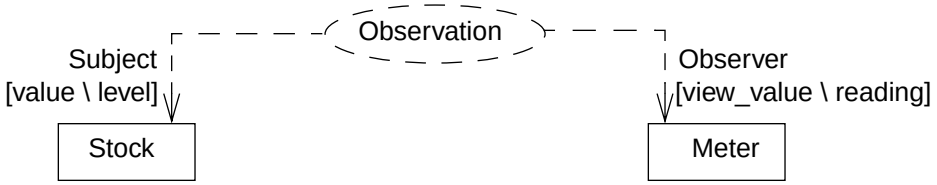


Figure 180: Example of applying a model framework

'Framework' is template package; 'apply' is import with substitution

The vehicle for a framework is a generic or 'template' form of package. Inside the template, some types and their features are defined using placeholder names. Looking at its definition in the library, we find that the Observation template has two type placeholders 'Subject' and 'Observer'; we have imported that package, substituting Stock and Meter. The sustituted definition becomes part of the model. In other words, whatever attributes and operations are defined for Subject within the template's definition, are now defined for Stock. Other names can be substituted too. The template uses an attribute called 'value' for the aspect of the Subject that we want observed; so we substitute it with the Stock's 'level'.

It is very effective for roles in collaborations

Collaborations. Notice that what the Observation framework does is to represent a cluster of design decisions about how two types of object should collaborate to provide the required effect; it does not discuss any other roles and collaborations of those objects, and decouples this design work from any specific domain. This is one of the most effective uses of model frameworks.

1. State in one object tracks state changes in another; see Figure 190.

Patterns. A pattern is a set of ideas that can be applied to many situations. A framework is at the heart of many patterns; but a pattern usually also includes less formal material about alternative strategies, advice on when to use it, and so on. When you keep a framework in a library, it should be packaged with this documentation.

.. many patterns

Class Templates. Some programming languages have 'class templates' or 'generic classes'; UML has these too. The notation is slightly different, but a class template is a model framework with just one class in it. We'll discuss them in more detail later.

..generic classes

Class frameworks. A variety of techniques can be used to build executable frameworks, from which programs can be quickly generated by subclassing, by 'plugging in' new components, and by interpreting purpose-built languages. We will look at these kinds of frameworks in Chapter 12, *Reuse and Pluggable Design — Frameworks in Code* (p.493); this chapter is about frameworks of abstract models.

..code frameworks

Tool support. Tools that support model frameworks and templates should allow you to 'unfold' each application of a model framework, so as to see the full resulting model, with all the substitutions made.

Tool support

Ideally, the tool should keep the definitions of the framework and the original definitions of the types to which it is applied, and each diagram in which the framework is applied. If the user changes any of these, the resulting unfolded model should change in step. Furthermore, the tool should allow you to define your own frameworks, in the same notation as the models themselves.

Incremental synchronization, and user-defined frameworks

Among current tools, there is some support for templates in a restricted way. Typically, the template works more like a script, a series of operations which is applied once to a model, adding the necessary attributes and operations. This has the disadvantage that the simpler original definitions are lost, and changes are less easy to make. It is also less easy to see what the template is about, since it is written in a scripting language.

Not the same as a Basic script

Summary. Templates provide a powerful way to capture reusable model frameworks — whether at a very abstract specification level, or down in the detailed design. In particular, they are good for capturing collaborations. Even without tools, the template notation is a very useful form of abbreviation — even where the template is not very rigorously defined. It's an easy way to say on a diagram 'this, this, and this type have such-and-such a relationship.'

The rest of this chapter begins by looking at how frameworks work to help build static models with just attributes and associations; then we will go on to deal with actions. Subsequent sections add further ideas, and there is a summary of concepts at the end.

A simple plumbing service model

Suppose a plumbing company asks us to do an analysis of their business, preparatory to getting some computerised support. After day or two with them, we arrive at this central model:

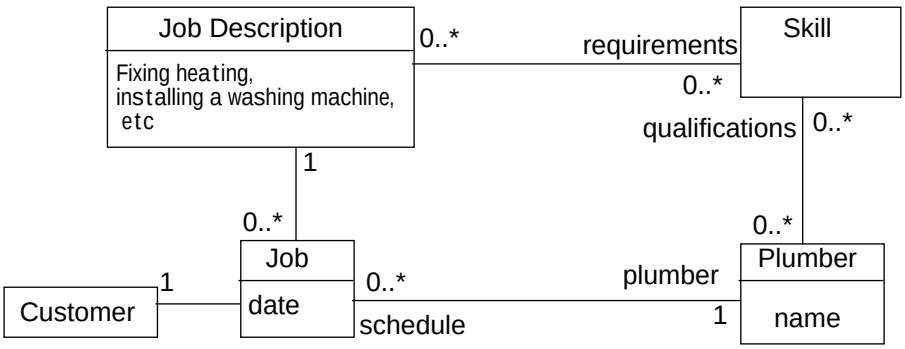


Figure 181: Model of allocating plumbers to jobs

Key types and invariants

Each Plumber is at any one time scheduled to do a list of Jobs, each of which takes place on some date and is for a particular Customer. There is only a certain number of kinds of Job, and each is described by a Job Description. Among other things, this says what Skills are required for the job (electrical wiring, excavation, denial of responsibility, etc). Each Plumber is qualified with a list of Skills; and a key invariant is that no Plumber should be assigned to a Job for which he or she has not the appropriate skills — or, as we've written it in the invariant, every Job's description's requirements must be a subset of the qualifications of the assigned Plumber.

A seminar services model looks similar

There's more to the model than this, of course, and work continues. Meanwhile, our consultancy gets involved in another modeling contract, with a commercial teaching organisation. We soon realise that we can make some savings here: Course Offerings, which happen on particular dates, are occurrences of Courses — just like Jobs and JobDescriptions; and Courses call for certain Instructor Skills (arm-waving, blustering, hypnosis, and the like).

So we generalise our model into a framework by creating a template package, like this. (We will later drop the «framework» stereotype):

Extract a framework

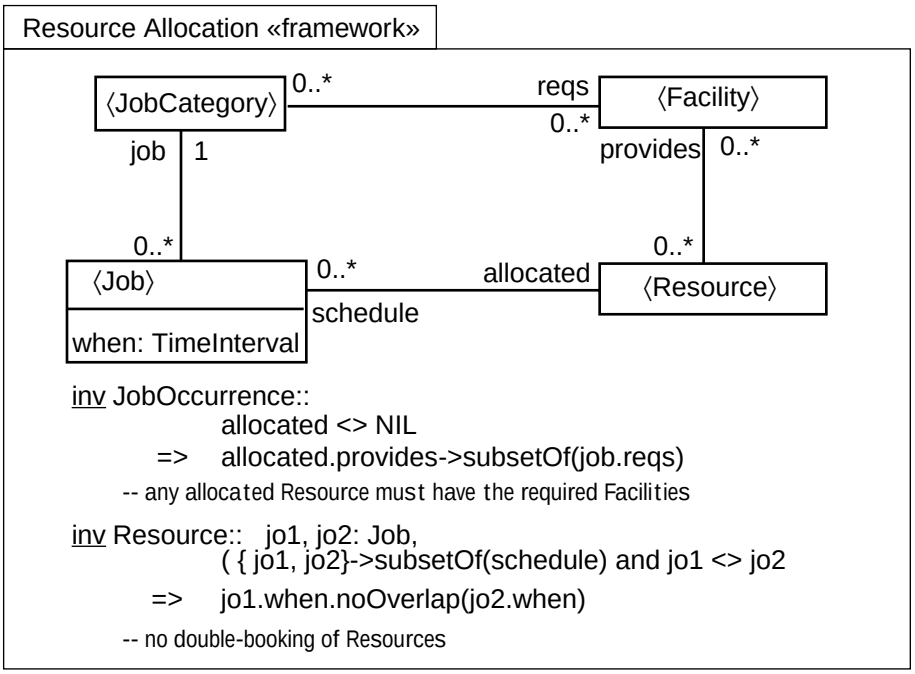


Figure 182: Model of resource allocation framework

We've taken the opportunity to add more details, particularly about Resources not being double-booked. (TimeInterval will have to be defined somewhere; we've assumed it has a boolean function noOverlap, that compares two TimeIntervals.)

Now our plumbing model can easily be generated from a 'framework application':

Generate the original model by application

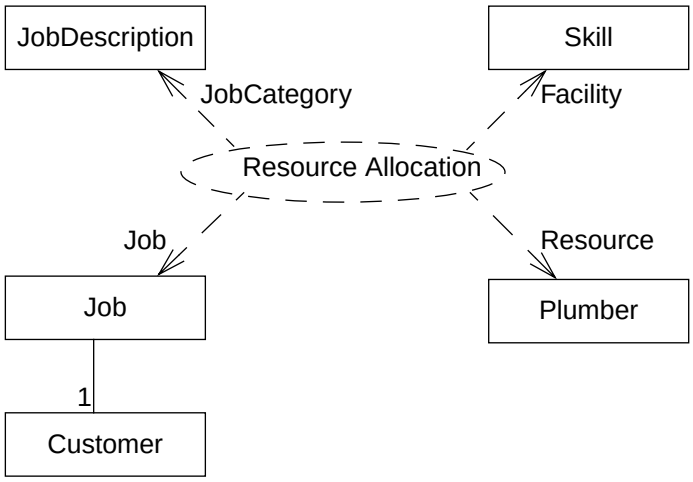


Figure 183: Application of resource allocation to plumbing

<Placeholders> are substituted in application

Notice that several of the names inside the framework definition are written with ⟨angle brackets⟩ : these are 'placeholders' which should be identified with actual type names when the framework is applied. This is the effect of the labelled arrows when the framework is applied.

Models are composed, with substitution

In the resulting model, each type has all the features given to it explicitly (like the Job's Customer); and also all the features defined by the framework, as name-substituted by the application. Working out the complete model is called 'unfolding'. A good tool will be able to show an unfolded version on demand.

Applied to seminars...

Turning again to the teaching organisation, we produce this model:

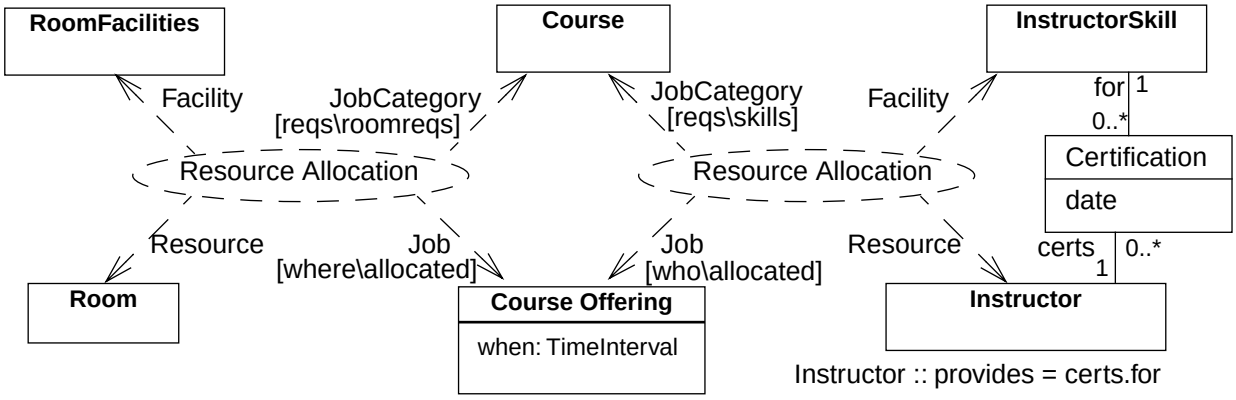


Figure 184: Double application of resource allocation to seminar scheduling

In this case, same framework applied twice

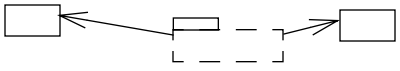
This happens to apply the same framework twice. Both Rooms and Instructors are constrained to provide the right stuff: Instructors have skills, Rooms have various facilities (OHP, whiteboards); and neither must be overbooked. We have also added an extra idea, that Instructors' skills, determined by dated certifications, define the 'provides' association from the framework: we have modeled this explicitly and tied it into the provides association with an invariant.[1]

Additional attribute substitution

This example also shows name substitution in the form [framework-name \ applied-name]. We have used it to rename some of the associations, to avoid Courses having different attributes with the same name. This text form and the arrows are equivalent: it is sometimes convenient to write instead of drawing pictures:

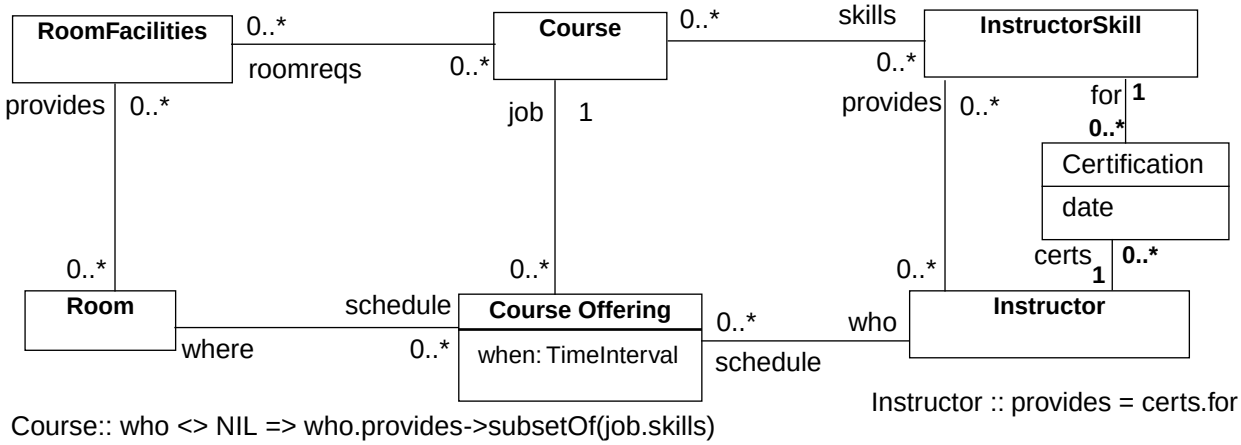
```
ResourceAllocation [JobCategory \ Course
                        [ reqs \ roomreqs ],
                    Resource \ Instructor ...]
```

1. The UML symbol for a pattern is a dashed "use-case", though pattern semantics have nothing to do with UML "use-cases", and the arrow direction has nothing to do with UML dependencies. We would have preferred the dashed package below. UML 1.1 does not have any semantics for patterns, just a notation; perhaps our semantics will be used.



When unfolder, statements from each framework application, after making the necessary substitutions, are composed with each other, and with any 'local' definitions that are applicable. The unfolding looks like this:

An "unfolded" view




```
Course:: who <> NIL => who.provides->subsetOf(job.skills)
        -- any allocated Instructor must have the required Skills
Course:: where <> NIL => where.provides->subsetOf(job.roomreqs)
        -- any allocated Room must have the required RoomFacilities
Instructor:: jo1, jo2: Job, ( { jo1, jo2}->subsetOf(schedule)) & jo1 <> jo2 => jo1.when.noOverlap(jo2.when)
                                        -- no double-booking of Instructors
Room:: jo1, jo2: Job, ( { jo1, jo2}->subsetOf(schedule)) & jo1 <> jo2 => jo1.when.noOverlap(jo2.when)
                                        -- no double-booking of Rooms
```

Figure 185: 'Unfolded' view after applying frameworks

Using frameworks clearly reduces complexity and duplication. It also provides a higher-level view of the model, making it clear that each loop of four associations forms part of a single relationship, the one we've called Resource Allocation. So frameworks are a useful kind of abstraction.

A clearer view

10.2.1 Framework applications are not Subtypes

Could we have used subtyping to express the similarity between Courses and plumbers' JobDescriptions?

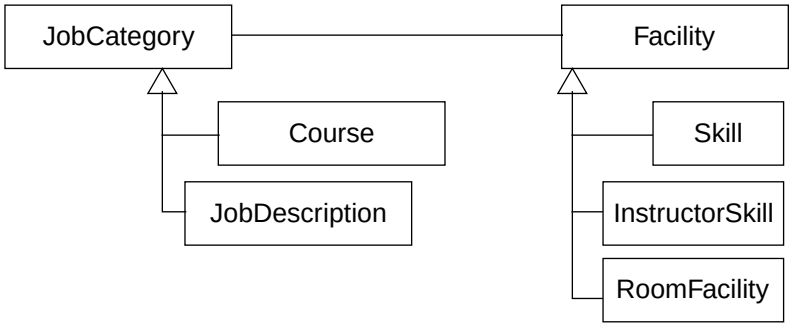


Figure 186: Why subtyping does not correctly reflect frameworks

Specialize groups of types, not individual ones

Not really. This would imply that plumbing Jobs might require (or could be used with) overhead projectors, and other mix-ups. It isn't the individual types that are specialised, but the entire group of them, along with their relationships and interactions.

Animals eat food

Another example is the faulty old syllogism "Animals eat Food; Cows are Animals; Beefburgers are Food; hence Cows eat Beefburgers." The mistake is that the first statement should not be taken to mean that every object conforming to the type Animal can eat every instance of the type Food. A more explicit statement would be "For every subtype A of Animal, there is a subtype F of Food such that all members of A can eat any member of F". Using frameworks, this can be written:

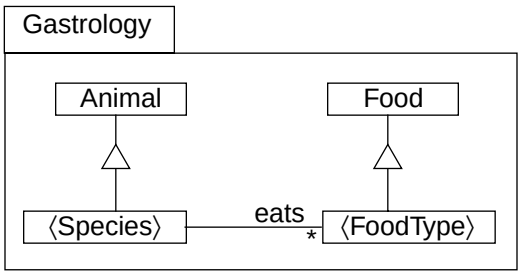


Figure 187: A gastrology framework

Now we can explicitly apply the framework to those pairs that are acceptable:

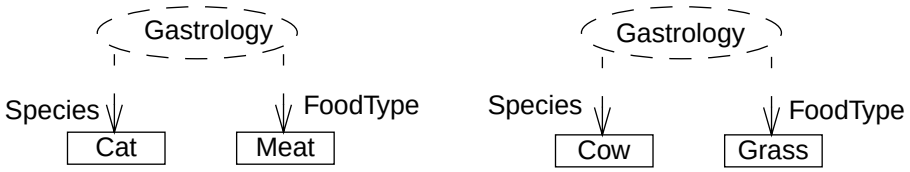


Figure 188: Application of gastrology framework

The association eats might represent the assignment of food-items to specific animals in an automatic feeding system. We thus ensure that instances of Grass will be the only members of Food proffered as fodder to any Cow-instance.[1]

A framework can use non-placeholder types, such as Animal and Food. So by applying the framework to Cat and to Cow, we are asserting that they are both subtypes of Animal.

1. Would that it were so.

10.3 *Collaboration frameworks*

A ‘collaboration’ describes the interactions between a group of objects that are designed to work together. They send each other messages intended to attain some goal which they are designed to achieve jointly.

Much of OOD is about collaboration design

Much of the skill of object-oriented design is about designing collaborations. The CRC technique (Classes, Responsibilities, Collaborations [Beck et al]) is basic to OO design, and is all about dividing the responsibilities for a task between collaborating objects. These design decisions distinguish OO programming from merely structured design, in which all the work is lumped into one program. In return for the extra decision-making (if you do it well), you get a decoupled design, flexible and extensible. Doing it badly leads to an unmanageable mess.

Good collaboration designs should be reused

The careful design of collaborations is of such value that, when you have done it well, it is worth recording the ideas and using them again. This is the motivation of many patterns (for example in Gamma et al). Some tools explicitly provide a way to define collaborations and then compose them into bigger designs (notably Taskon’s OORAM).

We use frameworks

In Catalysis, frameworks are our reusable pieces of design; so let’s now use them for reusable pieces of collaboration.

Collabs focus on just one set of related roles

The interesting thing about a collaboration is that it defines the interactive relationship between two or more objects; but when you define it by itself, you avoid saying anything about the other relationships each role-player might have.

Parenting and working

As a real-world example, if you describe what it means to be a parent, you’re talking largely about your interactions with your children, and the effects you have on each other. When you describe what it means to be an employee, that’s a different role with a different set of interactions with an object described in different terms. But although the collaborations can be described separately, the fact is that every object usually plays a role in several collaborations: perhaps you are both a parent and an employee. Each object conforms to the spec of its roles in the various collaborations it takes part in.

Collabs always affect each other

Separate collaborations can have effects on the same object attributes. Parenthood affects the bank balance; fortunately, that’s the same attribute that is improved by employment. So when we combine roles in one object, we usually have to take into consideration this interference between the different roles. Indeed if two roles didn’t interfere in this way, it wouldn’t make any difference whether they were assigned to the same object or not.

Subject-observer

The Subject-Observer collaboration [from Gamma et al] is a more technical example. In this illustration, we try to show each object's external interface as split into different roles in different collaborations; while the internal attributes may be shared:

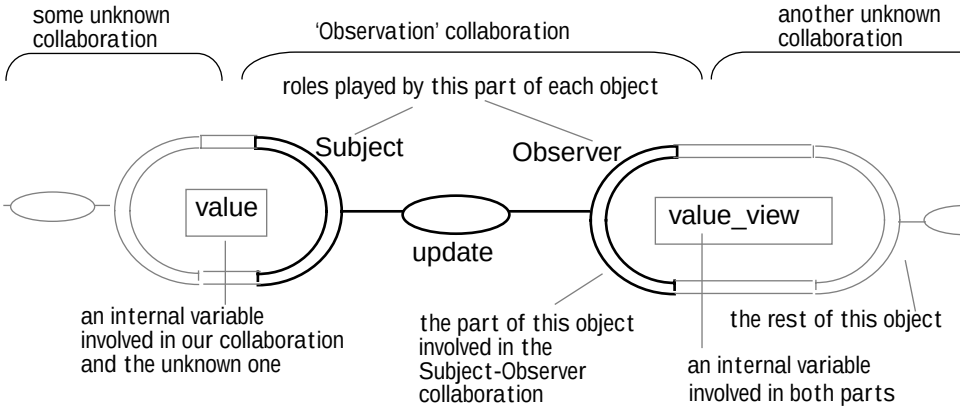


Figure 189: Collaborations are about parts of objects

The collaboration governs two roles, Subject and Observer; the Subject has some sort of value, and the Observer has another, let's call it its value_view. There is an update action initiated by the Subject in order to keep the Observer up to date.

Just two roles of some objects

Of course, any pair of classes that conform to this relationship in some chunk of program code will probably not be called 'Subject' and 'Observer': they'll have some bigger more interesting roles in their program, such as pieces of a GUI, or proxies in a distributed system. But this is exactly what frameworks are about: we can define just the aspects about which we have something to say, and then allow users to use other names and extend the definitions when they apply our framework.

So, as we've shown in Figure 189, we really only know about part of each object we're describing: the rest depends on whoever chooses to use the framework. In this example, we don't know how or why the Subject's value gets changed — only that it can happen, and that when it does, the Observer will have to be updated.

10.3.1 Using Invariant Effects in Collaboration Frameworks

We now specify framework actions

The big difference between this framework and the ones we have discussed so far, is that it has actions. In fact, there are several:

- The update action between the Subject and Observer
- All the other actions we don't know about, which might change the Subject's value.

Specifying the first one is easy:

```
Observer ::
    action update ( )
        post    value_view = subject.value
                - - I now correctly reflect my subject's value
```

(As always, an action might abstract a sequence of smaller messages; but we'll leave it to someone else to refine it.)

And constrain 'external' role behaviors

But the crucial thing we have to say in this framework is that the update action occurs as part of *any* other action that changes the Subject's value. To say this formally, we use an 'effect invariant' (Section 4.8.5, "Effect Invariants," on page 171) on the *external* section of the collaboration (Section 5.8.1, "External actions," on page 229). It is a post-condition without an action name or signature:

Invariant effect

```
Subject ::
    inv effect
        post    value@pre <> value ⟹ [[ observers.update() ]]
                - - Any action that changes my value also ensures
                - - that the observers correctly reflect the new value.
```

(Recall that [[*anAction*]] means that the postcondition of *anAction* is achieved as part of this action. If there are several observers, the same applies to all of them.)

The idea is that this postcondition applies to every other action performed by a Subject, no matter where the rest of the spec of that action comes from. So when we use this framework, we must AND the effect to all the postconditions of each of the other operations. When we finally come to program the Subject class (or rather, the class playing the Subject role when we've applied the Observation framework) we'll find that wherever the spec tells us to change its value, we must also update the corresponding Observer (or whatever the user has changed the names to.)

Add supporting attributes

Completing this example. Our framework can be applied to any pair of types, and will add to them the necessary specification to say that one of these types can be an observer of the other. But there must be some way of telling which Subject-instances are observed by which Observer-instances. We can define that as an association, and add another action for making links that belong to it.

Resulting framework

Figure 190 shows the collaboration framework as we would normally draw it. The subject-observers association links particular instances of the two types, and the update action only applies to Observers that have a current Subject.

Still defer details with joint 'register' action

The register action links a particular Subject to a particular Observer. It is a 'joint' action: we have not specified here how it happens, or even to whom you send the message to make it happen — just that there ought to be such an action, with responsibility for executing it distributed somehow between Subject and Observer. When a designer applies the framework to a particular pair of types, it tells him to provide this facility.

Could defer further with 'external' invariant

More abstract models. Just to illustrate how Catalysis lets you choose how detailed or abstract to be: we could have written the overall requirement without mentioning the update action at all, with an even less detailed external effect invariant:

```
Subject::
    inv effect
        post    value@pre <>  value ⟹
                    observer.value_view = value
                - - Any action that causes a change in value
```

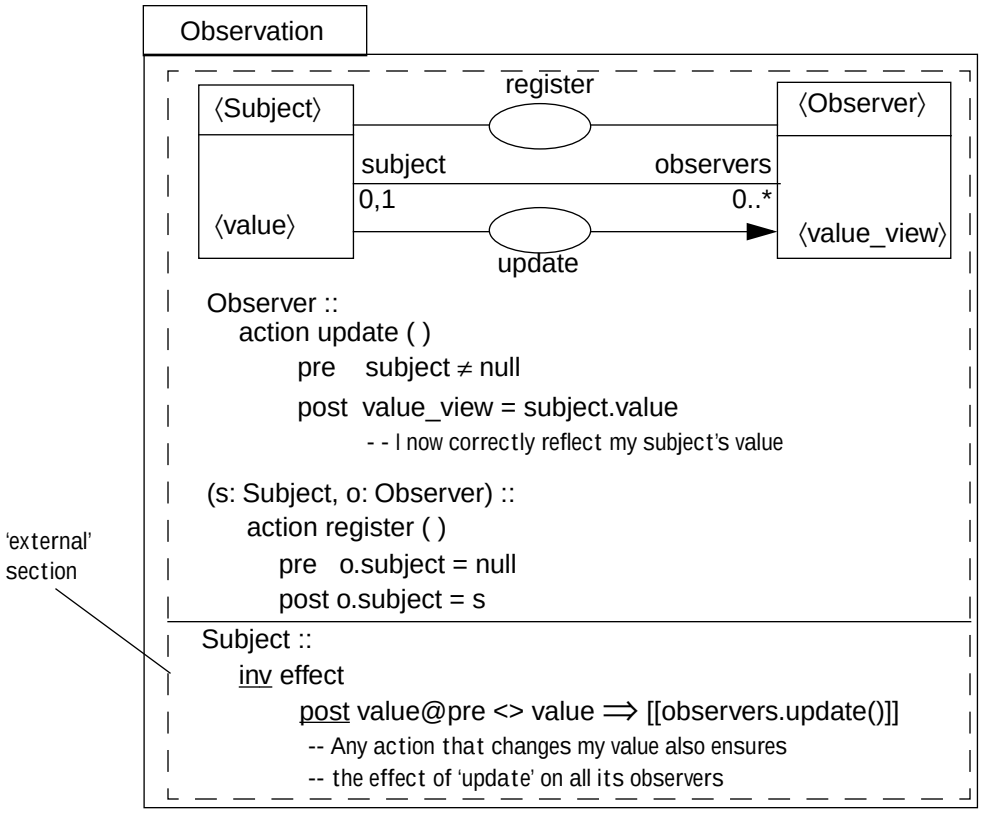


Figure 190: Collaboration template

- - must ensure that the observer's latest 'value_view'
- - is the same as our latest 'value'

Or instead, we could have just written an invariant external to the collaboration, defining the overall goal, saying nothing about how it is achieved:

inv Observer :: subject.value = value_view

10.3.2 Applying a collaboration framework

To apply, start with your model

One of the authors' clients designed a call management system for telephone sales teams. One of the types in the model was a Call Queue — the list of calls waiting for a particular group of operators. Let's suppose we have that type defined in one package; while in another, we have a kit of GUI widgets, one of which is a Thermometer — a useful display for numeric values. Here are parts of their models:

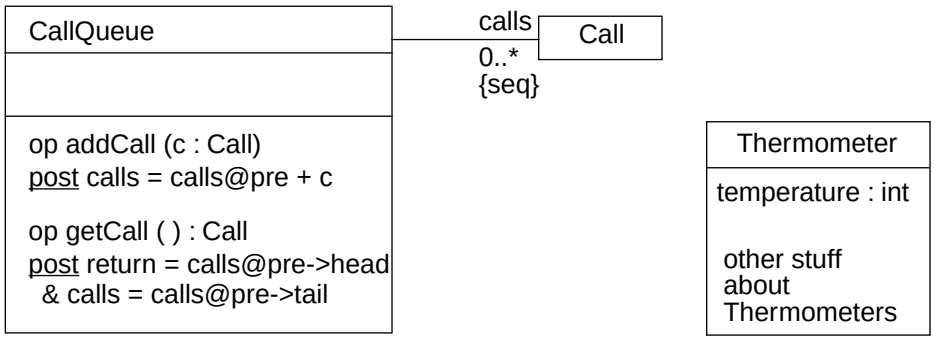


Figure 191: Target type model for using observation

Then apply the framework

Now we are designing the bridge between business logic and GUI, andwill make a package into which we import (among other things) the CallQueue and Thermometer. We'd like to make it possible for the 'temperature' of a Thermometer (the number it displays) to be used to show the number of Calls on a particular CallQueue. So we apply our Observation framework:

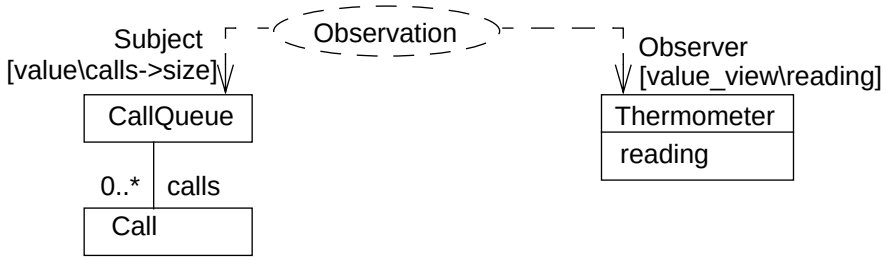
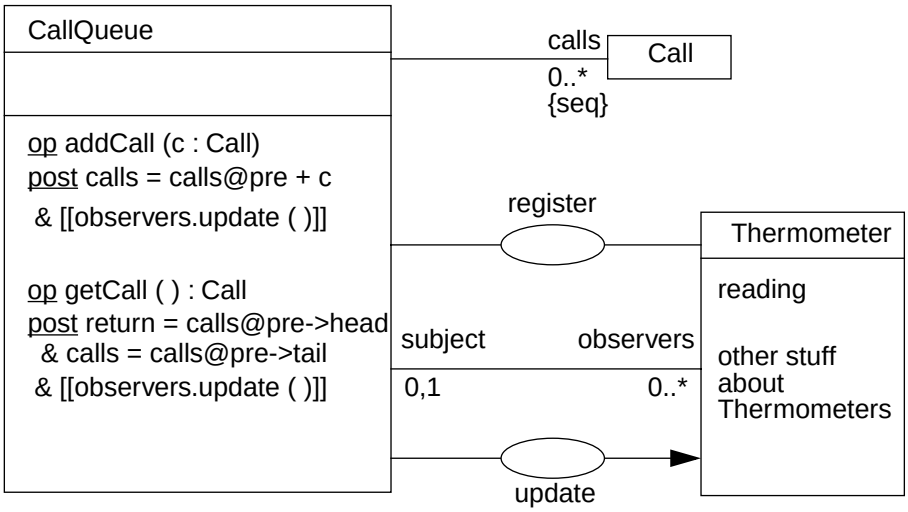


Figure 192: Framework application and substitutions

Unfold, but only if you disbelieve

This has the effect of adding the necessary specifications. Now let's suppose we have a tool that can display the unfolded model if we wish; it shows that the result of applying the framework and gives this spec:




```
Thermometer ::
   action update ( )
         pre    subject <> null
         post  reading = subject.calls->size
               - - I now correctly reflect my subject's value

(s: CallQueue, o: Thermometer) ::
      action register ( )
         pre   o.subject = null
         post o.subject = s
```

Figure 193: Unfolded result of framework application

Notice how the Observer and Subject names have been replaced. (We've actually made a slight abbreviation here: the addCall and getCall operations will always change the calls size every time, so we can drop the "value <> value@pre =>" from their postconditions.)

Still more design to do: joint 'register' action

Designing from the resulting model. So far, we have used frameworks for building models: what we end up with is a specification, which still has to be implemented. In this particular example, there is some work left to the framework's user, because we have not been told how to realise the register and update actions as specific operations on the objects. (Some other framework might choose to provide more.)

'register' design should be reusable

The update action could be realised as a single notify(newValue) message; or, in the Smalltalk MVC style, could consist of an update( ) message to the Thermometer, which then has to come back to the CallQueue asking for details of the changes.

The register action would typically be initiated by some supervisiing object telling a particular Thermometer to observe a particular CallQueue; then the Thermometer has to introduce itself to the CallQueue, so that each knows about the other.

10.3.3 Using one Framework to build Another

The idea of registering and unregistering is common to any situation where each of two objects needs to know about the other. So we could separate this scheme into its own framework:

Define 'register' framework

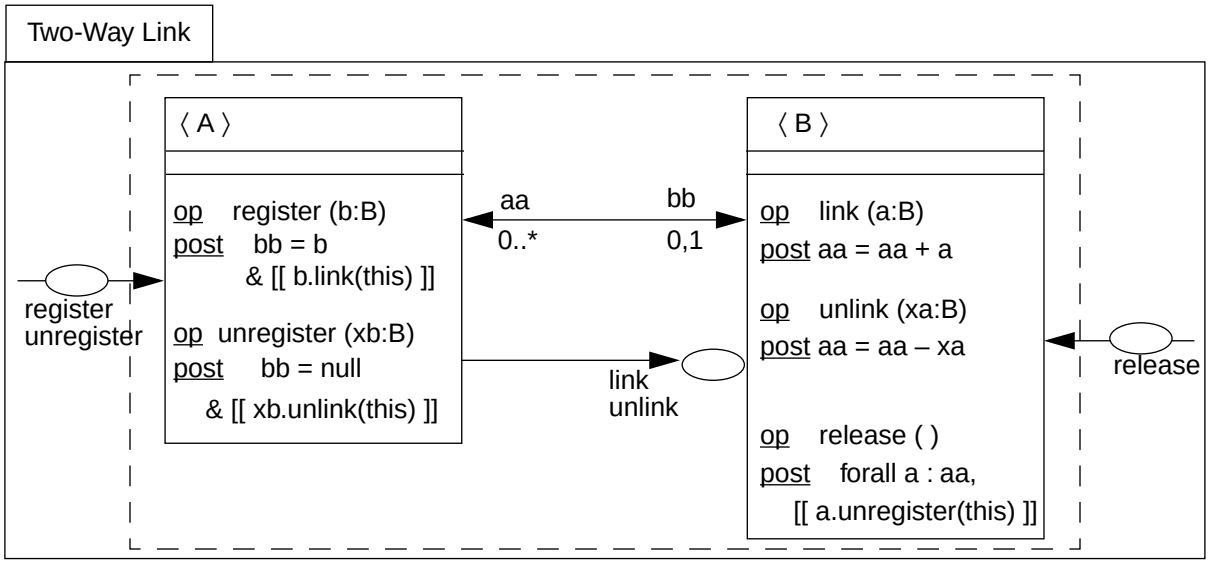


Figure 194: Low-level framework: Two-Way Link

Any instance of A and B can be linked together; aa and bb are the links in each direction. (The arrows indicate we've definitely decided to make the link navigable in each direction.) The operations intended for use from outside this collaboration are register, sent to an A to link it to a particular B; unregister; and release, sent to a B to unlink it from everything. The other operations on Bs, link and unlink, are intended only as an internal part of the design of this collaboration. We've written the specifications so that they are quite explicit about how the two-way links are maintained.

External register / unregister / release; internal link / unlink

Apply this framework to complete the design

Now we see that we could have used this framework to help define the Observation framework. (In fact, it can say more than before, because it now tells how the register action works, rather than just calling for one.)

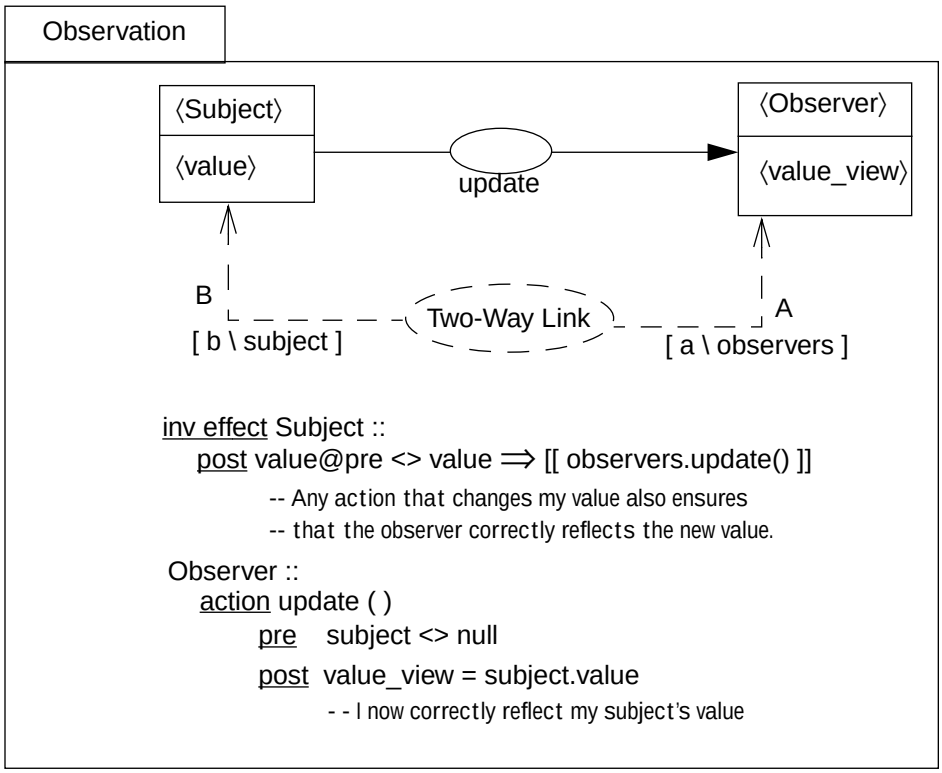


Figure 195: Observation framework using 2-way link framework

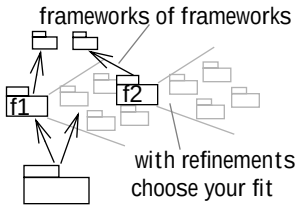
Again, simpler view

Once again, comparing with Figure 190, we can see how the use of a template imposes a higher-level order on the appearance of the model, substituting a meaningful single pattern on the diagram for a variety of links and operations.

10.4 *Refining frameworks*

Frameworks are an expressive abstraction tool, and are used throughout Catalysis, even in the definition of very basic modeling constructs. Still, as a template-like mechanism, they can only be used in situations where the problem at hand is suited to the parameterization and the level of granularity of the framework.

Frameworks are expressive as-is



Fortunately, our frameworks have an additional dimension of flexibility — *refinement*. There is no restriction that a framework be defined at a fixed level of detail; frameworks themselves are subject to refinement, abstraction, and composition in exactly the same ways as normal models. Further, some of these refinements are themselves defined as frameworks.

An extra dimension — refinement

10.4.1 A requirement

This framework illustrates a relationship between a Trader, who makes Orders from a Distributor. In the framework, we don't care how the Trader gets rid of stock, nor how the Distributor aquires it. We have shown this as a degenerate collaboration, as it will next be refined as a unit.

Abstract trade requirement

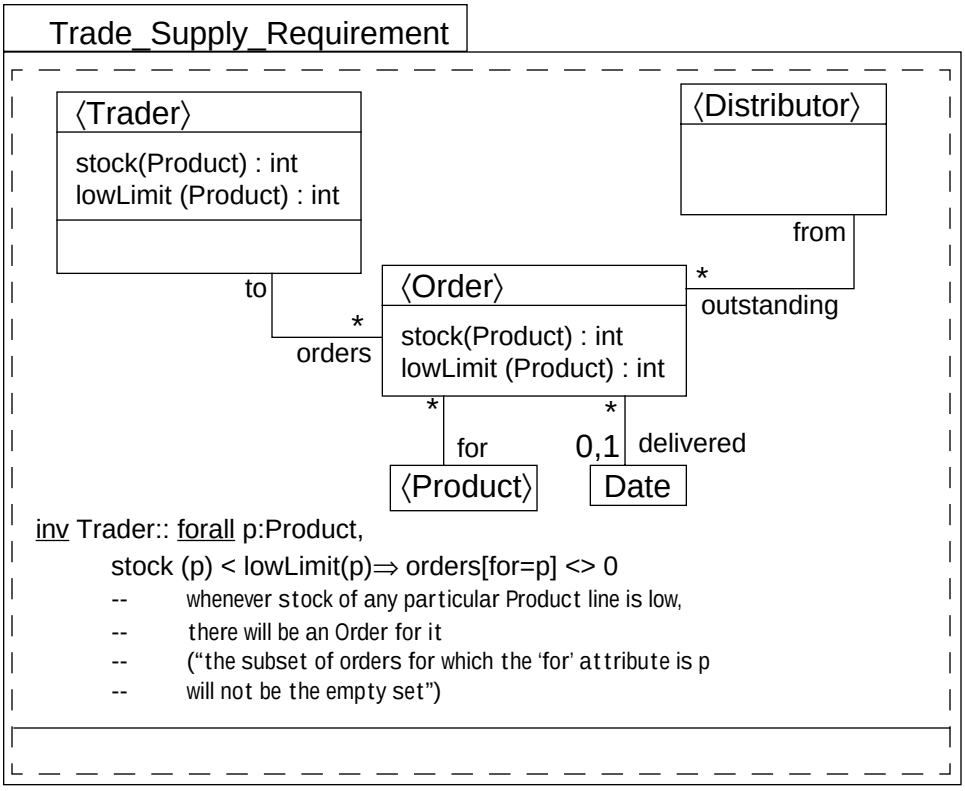


Figure 196: A framework for trading: stock maintenance

(Notice how we've made all the types substitutable except Date. So we would likely have Date defined as an actual type, somewhere in the package in which this framework definition appears, or in the packages it imports.)

Can be used as-is

This tells us that a Trader must always have an Order pending for low stock: that way, we can hope to avoid outages. Designers might find it convenient to use this framework by itself, and then go on to define how Orders are made, within their own models. Or we could go on to define another framework that includes that information.

10.4.2 A collaboration refining the requirement

Can also have a refined framework

According to the Trade Supply framework (Figure 197), when the stock of any Product gets low, the Trader makes an Order with the Distributor using the make_order action. Once an Order is established, the Distributor can deliver the goods, and the Trader can pay.

It is completely generic, yet more designed

There are different kinds of Traders in the world, that get rid of their stocks in different ways. Some just sell them; others cook them up and serve them; others build things from them. But no matter how stock depletion happens, the Trade Supply framework still manages to tell us that make_order should happen when stocks get low.

Good style of using 'effect invariant'

In an earlier example, we just used one effect clause. Here, we have split the cause from the effect. First, we've invented an effect name depletion (p) as a placeholder name for any action (no matter where defined) that causes stocks of p to be reduced: an invariant effect clause tells which (unknown) actions are considered to have the depletion effect. Secondly, we have defined a postcondition for depletion in the usual way for an effect: this says that we require to perform the make_order action.

Separating cause and effect in this way is useful when they have a many-many relationship, and are the essential reason for the effect construct.

10.4.2.1 Documenting the refinement

We want to claim that anyone who uses the Trade_Supply framework, using the same placeholder-substitutions, will achieve the goals set by Trade_Supply_Requirement. More precisely, we need to document a reason for believing that if we combined the two models, we'd end up saying no more than we've already said in Trade-Supply: that all of its statements (pictorial or otherwise) already imply those in Trade Supply Requirements.

The general rules are the same as those discussed in Chapter 7, *Abstraction, Refinement, and Testing* (p.265). In this case, the static models are the same; the only thing we have to worry about is that invariant in the Requirement.

We can write it as shown in Figure 198. (Actually, a more rigorous treatment of this argument reveals a hole to do with when Orders get removed. You might like to tighten the reasoning, and the spec of Trade Supply.)

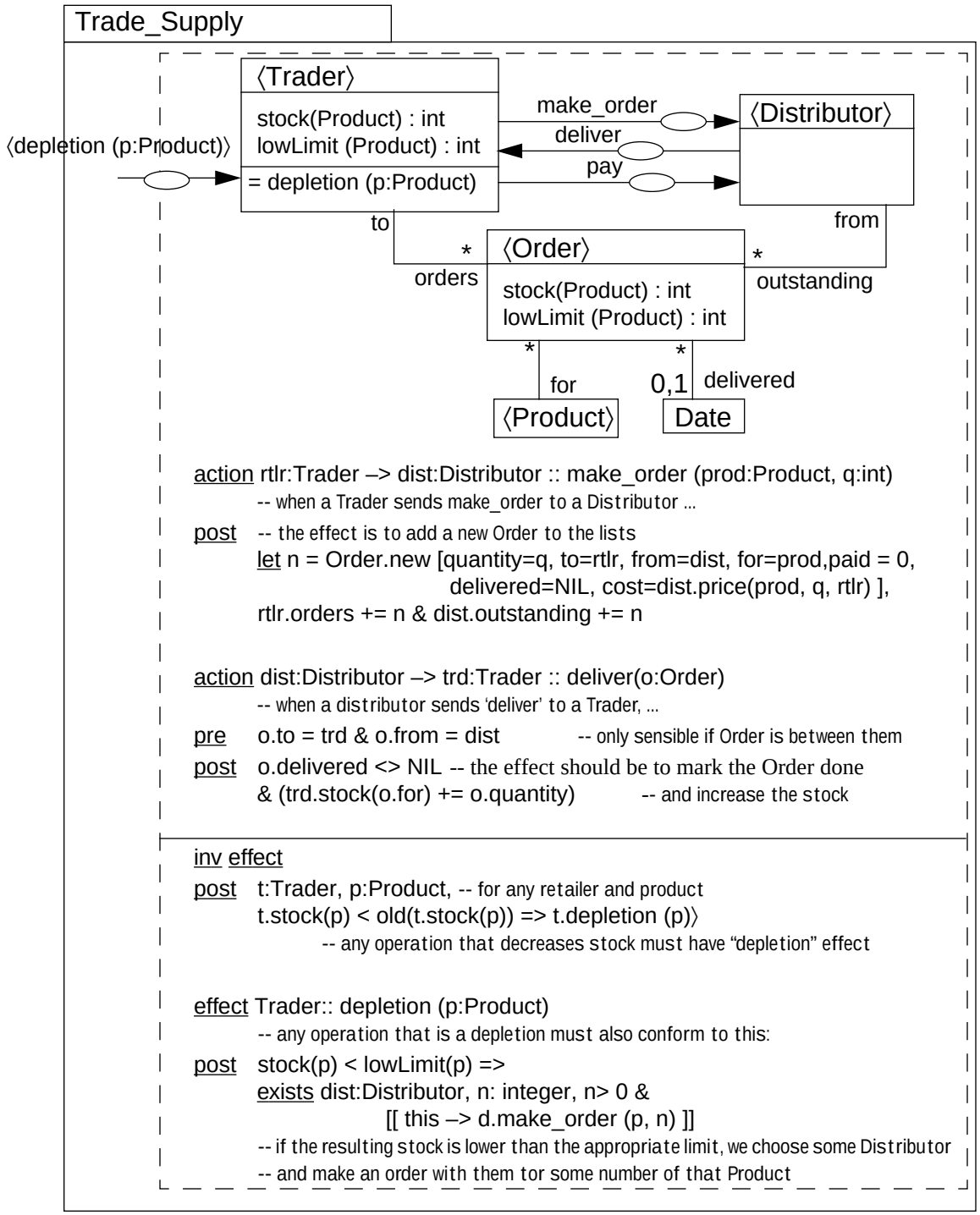
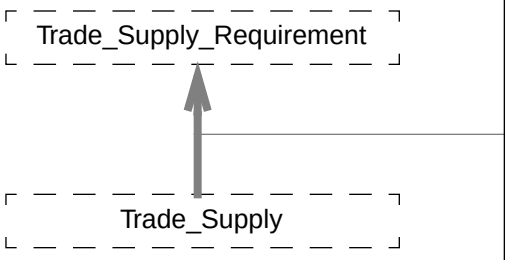


Figure 197: Trade Supply Collaboration



Trade Supply says that any action must invoke make_order if it depletes the stock of any Product of any Trader. Relevant quotes:

- Trader :: stock(p) < stock@pre(p) => [[depletion(p)]]
- depletion(p)
  post stock(p) <lowLimit(p) => [[makeOrer(p)]]
- makeOrder(p) post orders += new Order ...

Therefore there is always an Order for depleted stocks, which is the requirement set by Trade Supply Requirement in its invariant.

Figure 198: Documenting a framework refinement

10.5 *Composing frameworks*

retail-customer roles

A commercial retailer will play many roles, participating in many collaborations. One such collaboration will be its interactions with customers. The next collaboration shows the relationship of Customers to Vendors. In the sell operation, cash and Products are transferred in opposite directions.

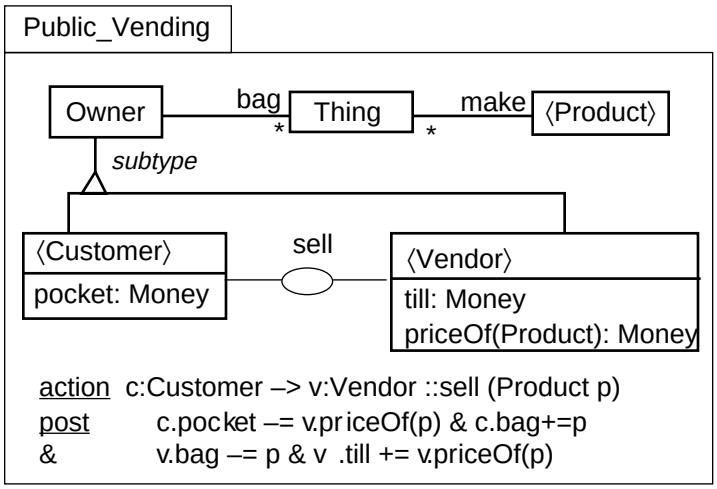


Figure 199: Another view of the vendor: retail sales

Compose with the earlier trade-supply roles, tie attributes together

A Shop is a type of object that plays the roles of both Trader and Vendor. So now we compose the two frameworks into a single picture, with Customer, Shop, and Distributor as the key players. In Public_Vending, the Vendor's stock was represented as a set of Things, each of which is an example of a Product; so this model must be tied up, using an invariant, with the Trader's stock which had been modelled as an integer for any Product.

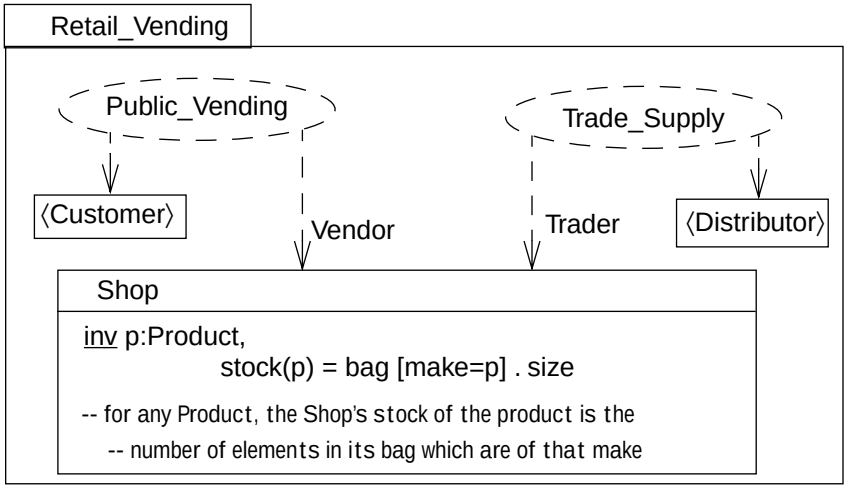


Figure 200: Joining roles by applying two frameworks

This gives the detailed design to be coded

The final step is to implement the types with classes. Supposing that Shop:sell is not refined further, but implemented as a single message, the designer will have to observe Trade_Supply::depletion whenever stocks get depleted – so a call to make_Order will sometimes have to be part of executing sell. Because the design has been so fully thought through, the class implementation will be simple.

10.5.1 Building systems from collaborations

Synthesize objects by composing roles

Shop is a synthesis that plays two roles. Each role is about the interaction with another type of object — or rather, a role of another object. The Shop functions by having enough roles to make a coherent unit: in this case, ensuring the throughput of stock.

Independent collaborations affect each other via shared objects

Given a variety of different collaborations, it is possible to construct many different role-playing objects. Collaborations are plugged together by making objects that play roles in each (and sometimes more than one role in one collaboration, just as a person may wear more than one hat in an organisation). For each object, it is necessary to state how participation in one role affects the other by tying together their vocabulary of state changes — as is done with the Shop's 'bag' and 'stock' above.

The collabs represent the key design units

But the main work of the design resides in the collaborations themselves, and plugging them together is relatively straightforward. Collaborations are the best focus for design, and objects are secondary. Following this principle results in more flexible designs.

10.6 *Templates as packages of properties*

Unrelated properties of objects can have much in common

Suppose you frequently find yourself modelling bananas, with a keen interest in their *curvier-than* relation; elsewhere, you trade commodities, with a *pricier-than* relation; in a class library you model strings, with a *dictionary-precedes*. Some have several such relations — physical objects can be compared separately for weight, size, price.

Many properties like '<'

All these objects have a comparison operation that works largely like '<' on numbers, in that they observe certain rules — a banana can't be curvier than itself; it is either less curvy or not than any other banana; and if mine is less bendy than yours and yours is less so than your friend's, then mine must be less bent than hers. These properties are quite important in some contexts, for example if they are to be sorted into a unique linear order.

Subtype of 'Magnitude' would be incorrect

How do we avoid repeating these rules every time we need to state them? It is not a solution to say those types (or their attributes) are all subtypes of, say, Magnitude, which packages operator < (Magnitude) with all the rules. That would mean that any Magnitude could be compared with any other, and a String's dictionary-position could be compared with a Banana's curvature. (The same reason we didn't use subtypes for the Jobs and Skills in Section 10.2.1 on page 395.)

Use a framework

Treating operators like functions (as in C++), we instead make a template package:

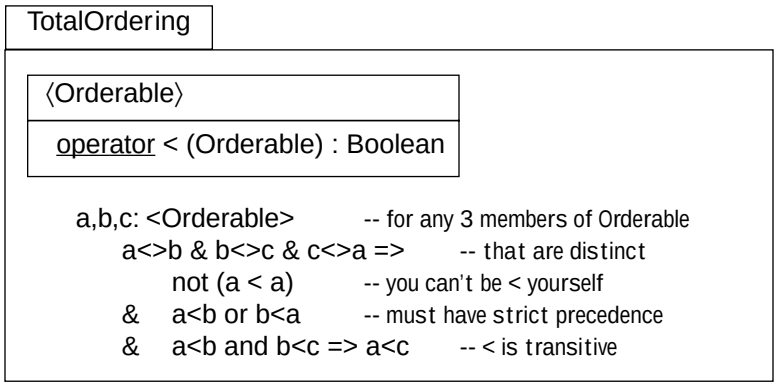


Figure 201: Framework for Total Ordering

Describe precise meaning of Ordering 'trait'

TotalOrdering is being used as a convenient package for a set of assertions that we can apply to different types. Since we're going to use the template many times, we take the trouble to set out the rules precisely. Groups of useful properties like TotalOrdering are sometimes called 'traits'. With a rich enough library of traits, you can make a wide variety of type definitions by combining several traits in 'mix and match' style.

Not all operators have the 'Total Ordering' properties. For example, when you make a project plan, the tasks only have a partial ordering in respect of the *must-precede* operator. Task A must be finished before B and C, which both have to be finished before starting D; but it might not matter which order B and C are done in. Here are some types whose properties can be partly defined with the help of this template:

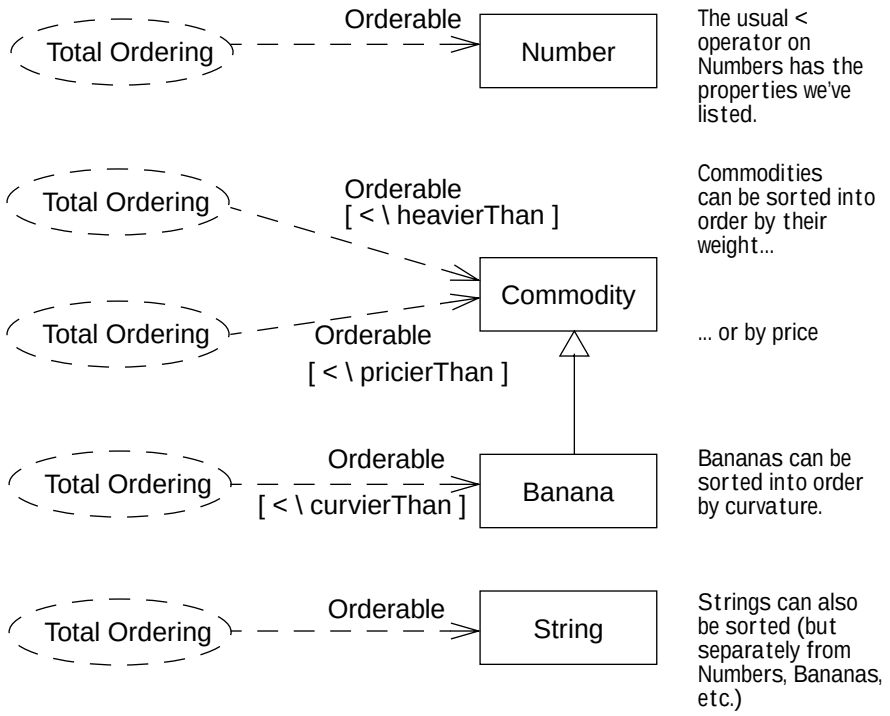


Figure 202: Many different total-ordered items

10.6.1 Template can have Provisions

Sorted List can handle any type with Total Ordering properties

A Sorted List keeps its items in a uniquely determined linear order. You can make sorted lists of just about any type of object, provided it has a comparison operator with the Total Ordering properties. This template defines what a Sorted List means, and includes the Total Ordering properties on its items:

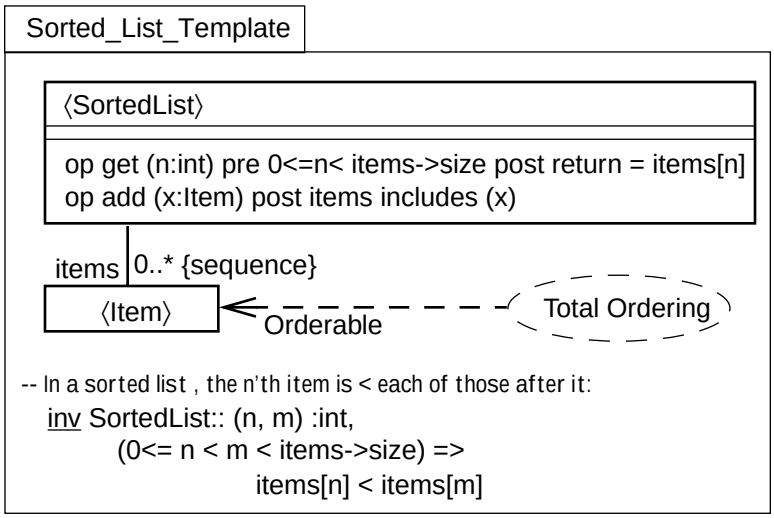


Figure 203: Sorted list framework: what kinds of items are OK?

Now, a Sorted List Template can be applied to any type; when applied, the imported Total Ordering template will impose its properties on the type substituted for <Item>. These fragments define different types which represent sorted lists with different content-types:

This template will *impose* the ordering properties on its item type

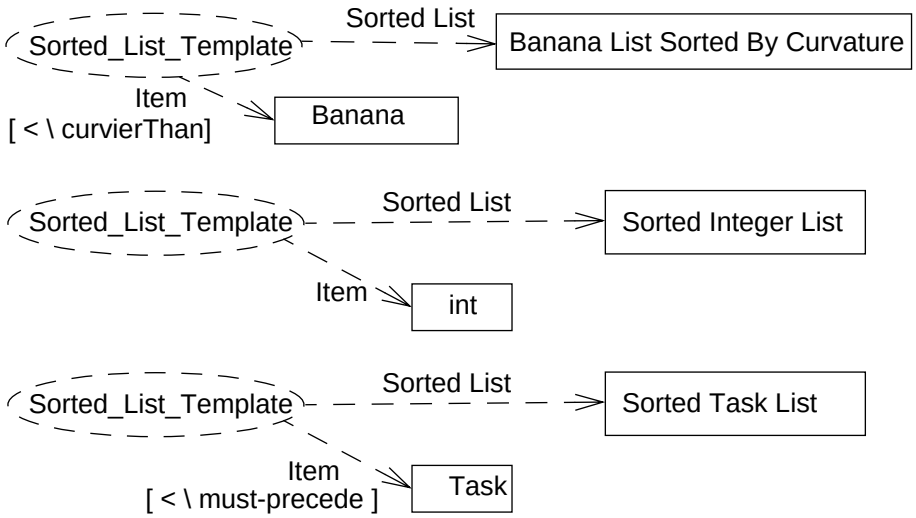


Figure 204: Applying the sorted list framework with substitutions

We have explicitly identified a separate new type for sorted lists of each item-type (you can't put a Banana in a Sorted Integer List). The template *imposes* on the item types the properties of the relevant operators. Notice that we also substitute "<", which the Sorted List Template has imported from Total Ordering. If you put a bunch of Bananas into a BananaListSortedByCurvature, they may end up in different relative positions than in a BananaListSortedByPrice.

What happens if you try to model a sorted list for something like project Tasks, that should not really be totally ordered based on *must-precede*? The result would be that the TotalOrdering properties would be imposed on Task — which (i) is probably not what you intended, since it imposes a linear order on all tasks, and (ii) could be inconsistent with the definitions of the *must-precede* operator itself.

But, instead of imposing ordering properties,

What we really want to do it to state that, as a prerequisite, the type substituted for <Item> should independently have the properties described by the Total Ordering template; if not, it is not suited to the Sorted List template.

we want to require that substitutions conform

We provide a way in which the designer of a template can say "this template should only be applied to things which you already intend to have certain properties", in a distinguished section of the package. The idea, called a provision, is a bit like a pre-

Define a provision

condition, except it typically works at design time[1]. Here's an improvement of Sorted List Template. It says that, if you have some type to which the Total Ordering properties already apply, then it is OK to make Sorted Lists of it:

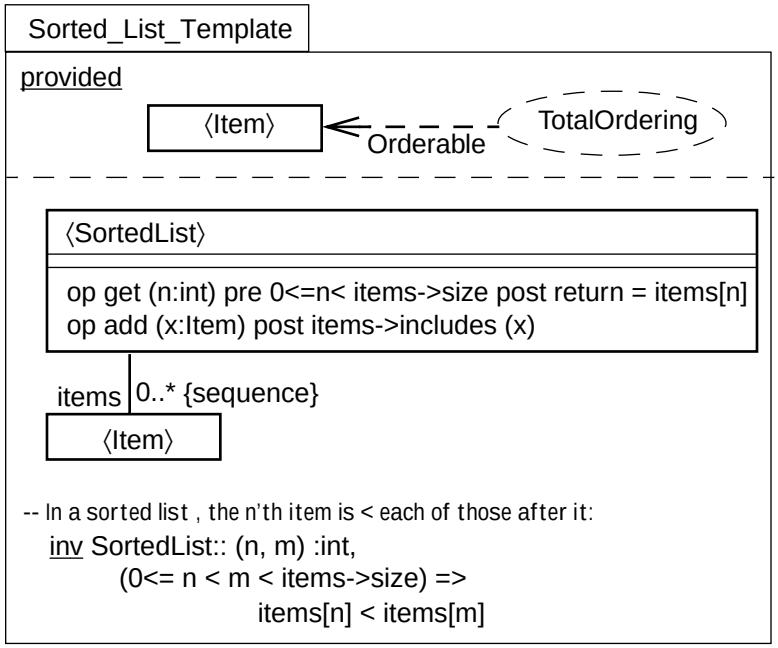


Figure 205: Explicit substitution provisions in Sorted List framework

Document how provisions are met

In the provisions section, you can put any model to which the substituted types must *already* conform[2]; thus, you can require that one substituted type is a subtype of another; or that they have some relationship (Section 10.2.1, "Framework applications are not Subtypes," on page 395); or satisfy some predicate. The designer who applies the template must check, perhaps with help from a tool, that all the other parts of her model imply the properties laid down as provisions, and this should be documented much like a refinement.

...for each application of that template

To see exactly what this means, we must recall that all models are defined within some package, and that models are usually structured into packages. Whoever uses our template will probably apply it in a separate package, into which they will have to import both the package defining our template, and the packages in which their item-types are defined. The imported packages must provide definitions that imply everything given in the provisions clause; and a conformance justification should be attached to the application of the template (Figure 198).

In general, when you build a framework, you will need to make certain assumptions about the things that are substituted for your placeholders in order for that application of the framework to work as intended. Use the provisions section to document these requirements.

1. Using *Reflection*, you can write generic code that does similar checks at run-time
2. This lets us correctly describe C++ template design and usage, including the STL

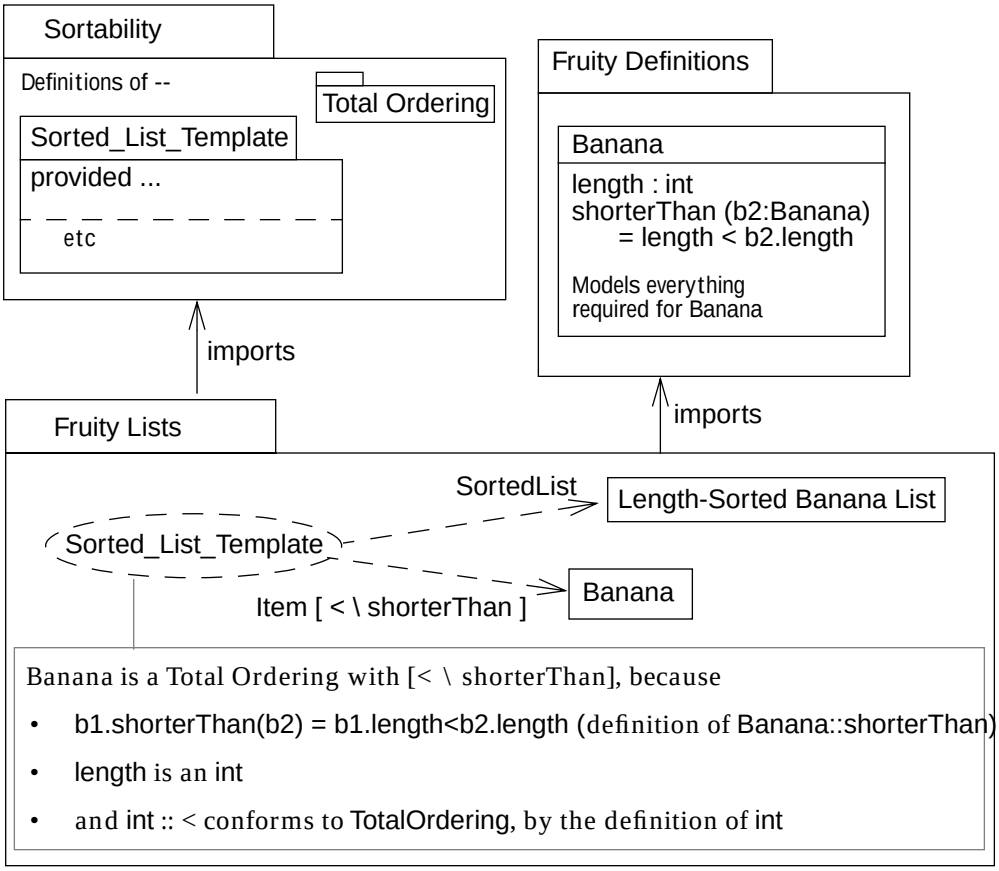


Figure 206: Applying templates that have provisions

10.6.2 Template as generic types and classes

A short syntax for template types

Many templates exist to define a single family of types, like the Sorted Lists. It is inconvenient to explicitly invent a new type name every time we want to make a new sorted list of something, and then explicitly substitute that for the placeholder in the template. An abbreviated notation covers these cases.

Placeholder name = package name

Within the definition of the template, you can use its name as the one of the placeholder names — e.g. the name of a type. Using UML conventions, add a inset dashed box at the corner of the type, and list the placeholders of the template.

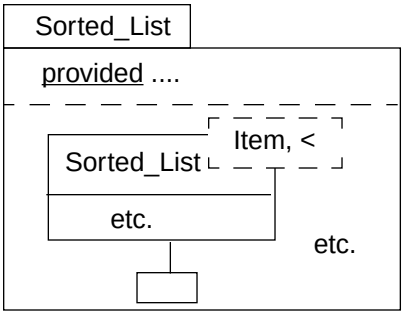


Figure 207: Defining template with convennient syntax

Implicit import and substitution

To use the template, draw a type using the name of the template package (this is also the name of the primary template type, which the type drawn implicitly substitutes), with the template parameters in the UML inset dashed box, or shown as explicit textual substitutions[1], in this form:

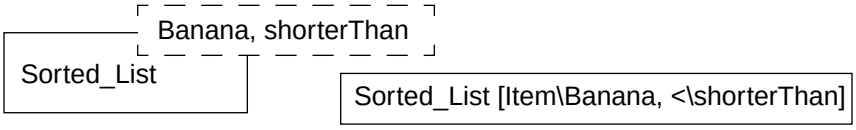


Figure 208: Alternate convenient syntax for applying template

Either one is equivalent to drawing:

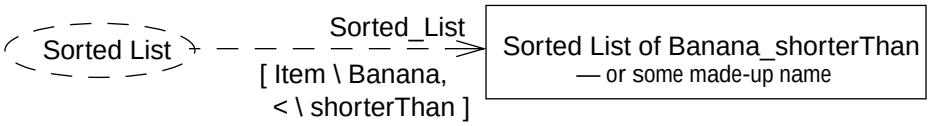


Figure 209: Equivalent “full” syntax

(In C++, the equivalent would be roughly SortedList < Banana, shorterThan>.)

1. The text version also works for non-graphical things like attributes; and inline declarations like: x: List[T\int]; UML would treat this as an uninterpreted string.

Nested packages are quite useful when they are generic: a standard package can contain definitions of several generics. A user can import the container, making the names of the nested packages visible, enabling him to apply the generics. Package provisions also work well with package nesting.

Nesting packages helps further

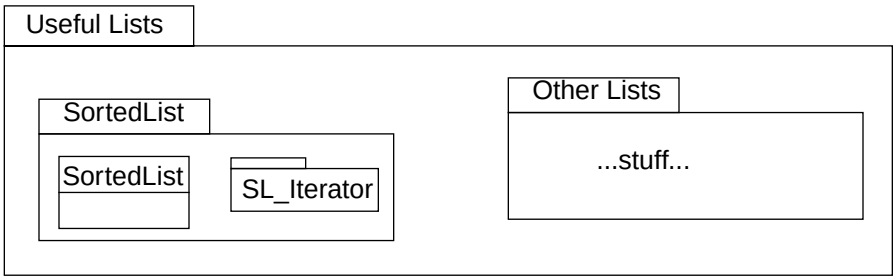


Figure 210: Templates and nested packages

10.6.3 Substitution for parameterisation

You can substitute any name when you do an import; not just types and attributes, but variables, constants, and more. A substitution can be used to parameterise a spec. For example, if an integer constant MAX_SIZE is used within SortedList, but not set to any value, it can be substituted when SortedList is applied:

Substitute for variables and constants

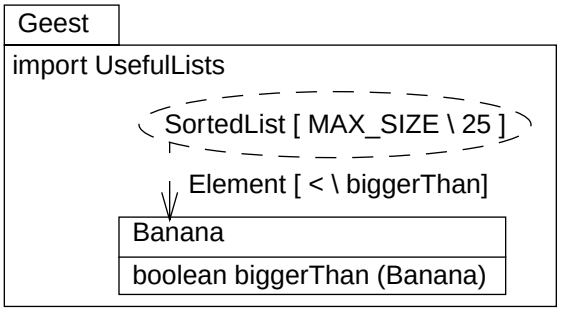


Figure 211: Substituting “values”: textual version

or even:

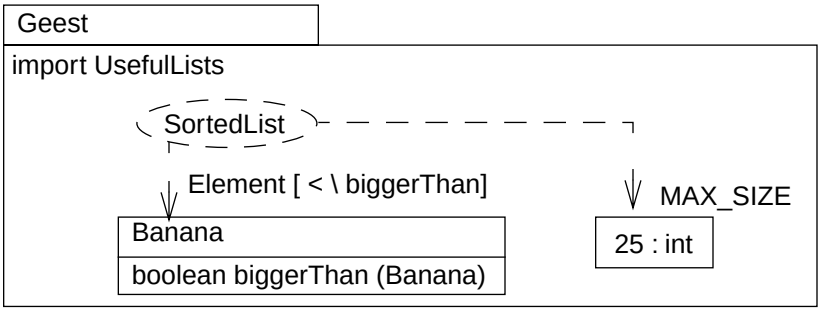


Figure 212: Substituting “values”: graphical version

10.6.4 Explicit template parameters

In general, any element can be substituted

In general, we do not constrain which types may be substituted, since we sometimes substitution just to avoid name clashes between multiple imports. The “<..>” markers simply suggests places for substitution; but unmarked types can be substituted, and marked types left unsubstituted. Any substitutable type that isn’t substituted when imported just remains as a substitutable type in the importer.

unless explicit parameters declared

Packages can be given explicit parameters. These are substitutable elements that must be explicitly substituted by the importer (even if only by one of its own parameters). They can be given default arguments as well.

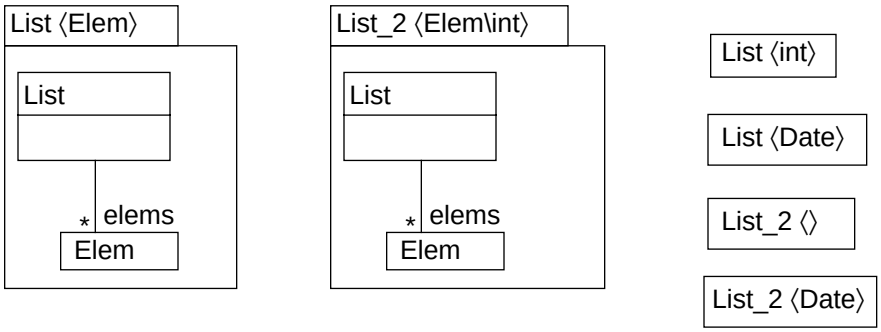


Figure 213: Explicit templates parameters and parameter substitution

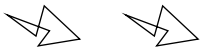
10.7 *Templates for Equality and Copying*

A common dilemma

What does it mean for two objects to be equal? For one to be a copy of the other? These questions arise often in different forms, and have an answer which is domain independent. This section defines what they mean, and provides standard template packages to use. These can be used to define standard copy and equality, as well as features such as replication and caching.

10.7.1 Type-defined equality

Are these two shapes equal?



No single definition of 'equal'

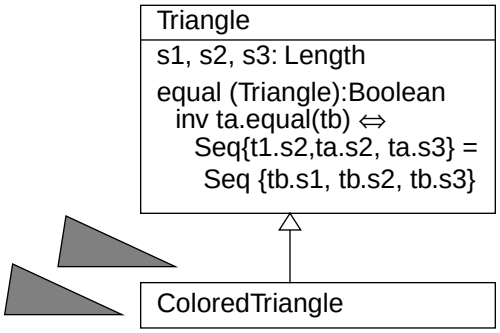
Some might say Yes, because their lengths and angles are all the same; some might say No, they are in different positions, hence different. It all depends on what exactly you mean by 'equal'; and sometimes there are different degrees and kinds of it (like 'congruent' and 'similar').

Identity is predefined

So, while object identity is treated as an intrinsic property, equality has to be defined separately for each type: there's no automatic meaning for it. There may be several useful equality-like relations, or none, and it's up to the inventor of a type to define them.

Triangles vs. colored triangles

Equality is relative to type. Supposing you define that two Triangles are equal if the lengths of their sides are equal. Then someone produces two members of the Triangle type that happen to be colored. One is blue and the other red, but their sides are the same. Are these objects equal?



Someone who is aware of the type Triangle but not ColoredTriangle would say yes — for her purposes, they are: she never asks about a Triangle's color. The less you're interested in, the more things look the same; the more you know, the more you can discern differences.

Supertype equality gets tricky with subtypes

Moreover, it would be wrong to contradict our definition of equality in a subtype. Triangle is the set of all triangles, colored or not, and it should be the place where you put statements that are true about them all. Equality on colored traiangles could further discriminate on color. But the supertype has stated that that as long as the sides are the same, two triangles are 'equal'.

So the equality definition typically cannot simply be inherited, and we have to do one or both of these:

- Have a differently-named equality-like relation for every type. This isn't as bad as it might first sound — it forces you to think out the differences.

- Have a single notion of ‘equal’, but be more careful about what we promise about it. For example, we could say that for two triangles to be equal they must have equal sides — but not necessarily the reverse:

  ta . equal (tb) ⇒ Seq{ta.s1, ta.s2, ta.s3} = Seq{tb.s1, tb.s2, tb.s3}

A template for equality

The same considerations apply to almost all comparison relations between members of the same type. (≤ and ≥, for example.) An ‘equality-like relation’ is one to which conforms to this template:

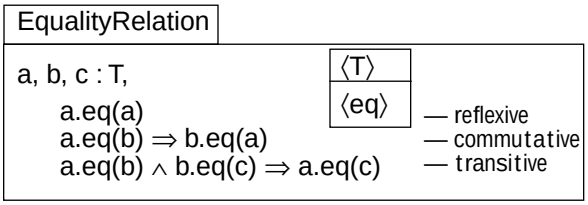


Figure 214: Template for equality relations

This can be applied to Triangles in a variety of ways:

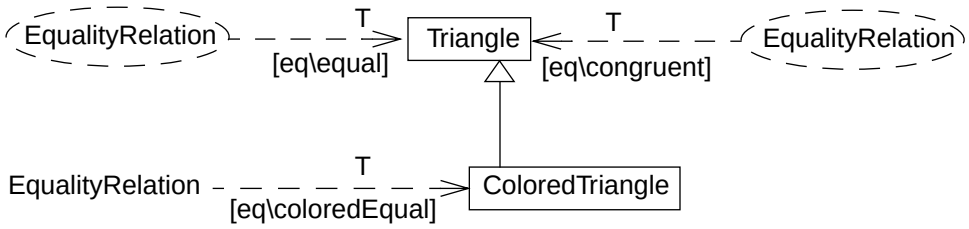


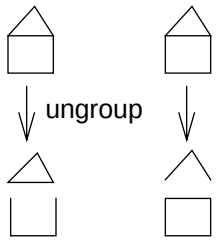
Figure 215: Defining different equality relations

Sometimes you can make an equality-like relation that seems reasonable for all subtypes. For example, we could define equality for shapes of any form by assuming they all have some boolean contains(p:Point), as follows:

```
Shape::equivalent (s: Shape) =
    -- if there is some vector, offset (conceptually, the difference in their positions) for which
    Vector->exists ( offset |
        -- any point is in “self” exactly when (point - offset) is in s
        Point->exists ( p | self.contains(p) = s.contains (p-offset)))
```

While this is more general, it may still fail to adequately address colored points.

Cannot just look at current apparent attributes



A user of a graphical editor can make a Group of other Shapes, which then behaves as one Shape, for the purposes of moving it around etc. It’s also possible to ungroup a Group, restoring the individual parts. That means that the two examples shown here prior to ungrouping, though equivalent by the above definition (i.e. looking the same), are “significantly unequal” — that is, unequal in a sense that is likely to be important to their users. Similarly, a rectangle may happen to be temporarily shaped like a square; however, stretch it horizontally, and stretch a true square horizontally, and very different things happen.

Equality-like relations need to be considered on a per-type basis; they should take into account dynamic and mutative behaviour.

Equality should consider dynamic behavior

10.7.2 Copy

It's often necessary in an action spec to require that a new copy of an object is made — that is, one that is equal (by some definition) but not identical. Just as equality must be defined separately per type, so must copying. To copy a Triangle means copying its three sides; to copy a Grouped collection of Shapes means copying the constituents of the Group.

Meaning of copy depends on meaning of equality

A copy operation can conveniently be defined *provided* your chosen comparison operator is a valid equality relation, by using this template:

Template for copy

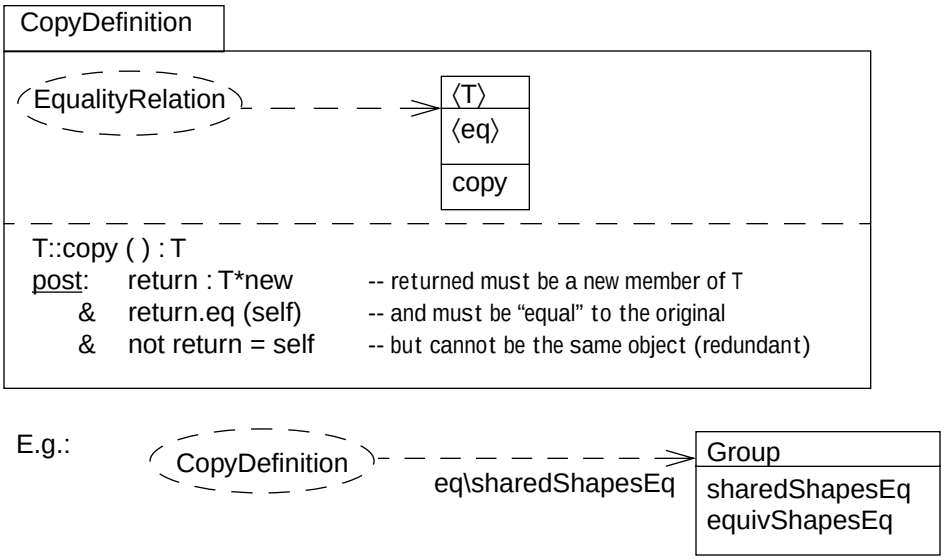


Figure 216: Template for "copy" with prerequisite of "equality"

This copy definition could be used on Groups with either of two different equality operators: sharedShapesEq — two group objects are considered equal provided they share the same shapes; and equivShapesEq — the shapes themselves need not be shared, but have to be equivalent (for some definition of equivalent).

e.g. 2 deep / shallow copy of a group of shapes

Unlike equality, the name of the operation doesn't need to be changed across subtypes for copy. If you only know you've been given a Shape, you know that getting a copy will give you the same visible result; you have no expectations about anything more, since you have no information on type-specific operations. However, a subtype of shape would have to copy in accordance with its specialized definition of the equality operator; so, colored shapes would have to copy the color as well.

10.8 *Package semantics*

Patterns from domain-specific to modeling constructs themselves

Template packages define the meaning of recurring patterns of models and designs; but the idea extends to the basic modeling constructs themselves. If two designers draw a pair of type boxes, with a 1-1 line between them, they both have the same meaning — except for the specific domain they work in; similarly for using subtype arrows; or a state transition; or superstates. And, if they put the same stereotype on two elements, they mean the same thing (presumably).

All can be done with templates

Templates can be used to define fundamental modeling constructs, as well as any extensions, in Catalyisis. This includes associations, associative classes, qualifiers, and even types and subtypes. Of course, most of these will have convenient syntactical forms, such as those the UML provides. This section describes how new notations and semantic extensions can be defined precisely in Catalysis.

10.8.1 Interpreting package contents

Value precise visual representations

We've already observed that the diagrams we draw in UML could as easily be written in the form of textual statements. The advantage of the diagrams is that they are easier to find your way around, and to grasp as a whole. Presumably they put your visual processing capabilities to work on the problem, leaving your linguistic processor free to mouth punchy-sounding businessspeak.

..but some things must be in text, and all should have text equivalents

But diagrams can't express everything you want to say, and so for details such as invariants and postconditions, we usually resort to text.[1] The pictures themselves can be converted to textual statements in a similar style. So the entirety of a model can be thought of as a collection of assertions. A package is a chosen set of such statements; while importing just means that you are including the statements from one package within another.

Semantics means being able to reducing nice notations to basics

It's possible to write down a precise set of rules (that is, a program) for converting each diagram element into text. And given any complex piece of text like an action specification, with its pre and postconditions and odd constructs like @pre etc, it is possible to write a set of rules for converting it into a longer set of statements in terms of a much more basic set of ideas. These sets of rules are called the *semantics* of the language.

Users don't need that usually; tools do

Books like this one, that explain a notation and how to use it, are informal versions of the semantics: informal in the sense that they aren't written as an executable program, and include ambiguities and inconsistencies. No-one has yet written a full formal semantics for UML, Catalysis, or Objectory, though several projects are under way. Still, there is a wide range in how precisely their various visual notations are understood, and how much re-interpretation will be required by practitioners. The closest things most people see in practice are the consistency-checking facilities of various support tools. Unfortunately, the different tools have slightly different ideas about the semantics, which is why it would be nice to get one agreed.

1. Though Stuart Kent has shown how to move these assertions into the pictorial domain.

What has been written is a description of the abstract syntax: that is, setting out what constructs there are and some of the constraints on them. These are sometimes called ‘metamodels’. However, they are far from being a full semantics.

10.8.2 Stereotypes and dialects

Too many visual symbols get confusing

Another disadvantage of a pictorial notation is that there aren’t enough symbols to cover all the subtly different things we want to say. You can only invent so many variants of boxes and lines and round things; and if there are too many of them, newcomers soon despair of remembering what they all mean.

Stereotype tag tells meaning

We use UML *stereotypes* for this reason. A stereotype is a tag that you can attach to any box, arrow, or other pictorial construct, that tells you exactly which meaning is intended. In other words, it tells you which translation rule to use from the semantics (assuming there is one).

You can stereotype individual elements

Stereotypes can be used on an individual model element as an alternate syntax to apply a framework (similar to Section 10.6.2, “Template as generic types and classes,” on page 415). The shorthand rules are:

- If a type has the same name as its package, then using that name as a stereotype on a target type means: import the package, substituting the target type.
- If an attribute has the same name as its package, then using that name as a stereotype on a target attribute means: import the package, substitute the target attribute and its source type; similarly for other elements.
- Just as with other template shortcute, sterotype application can use additional explicit substitutions: «name[x\a, y\b]», or provide parameters defined on the template: «name⟨a,b⟩».

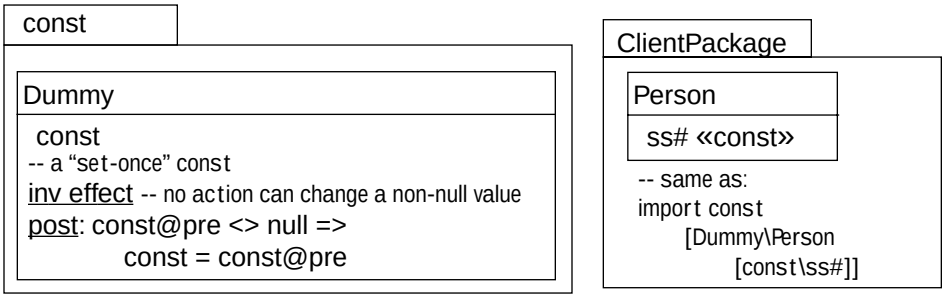


Figure 217: Defining meaning of stereotypes using templates

Better to to package default meanings, or a family of stereotypes

However, freely adding individual stereotypes leads to inconsistent models. Rather than attach stereotypes to every construct in the picture, we establish a set of defaults: a particular default meaning for each pictorial element without stereotypes, or a consistent family of stereotypes. This mapping is a called a *dialect*. You can specify which dialect a package should use, by quoting the dialect in the package tab. Naturally, con-

sistent dialects will simplify things; but, if the dialects have a common underlying translation (see Section 10.8.3), then you can even use custom dialects best suited to each portion of the problem:

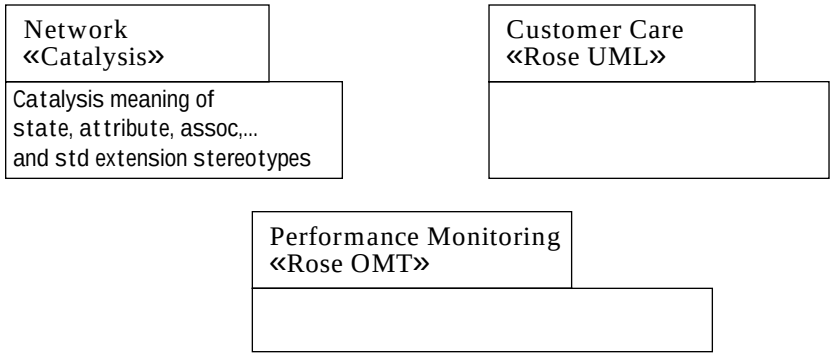


Figure 218: "Dialects" of stereotypes and other notations in packages

Use this facility conservatively

Stereotypes make the language extensible. This can be a disadvantage or an advantage, depending on whether you make your money by using the notation, or by pontificating about it. Every self-styled expert has their own pet variants on the basic ideas; all of which are, of course, improvements. It is widely agreed, though, that UML is by no means the last word on modeling languages, and that it would be not possible nor appropriate to make it entirely fixed at the present time.

10.8.3 Examples of semantic rules for dialect

A dialect is a package

Just as a final complication (or, perhaps, as you may have guessed by now): the dialects and the semantic rules themselves are of course defined within packages — though as we've said, we'll have to consider them virtual until further notice. But here is a short example, just to show the idea.

Its semantic rules are nested templates

Semantic rules are expressed as templates; a dialect contains nested packages for its semantic rules. Each rule translates a slightly higher-level notation into its equivalent lower-level one. Here, any line between two type boxes, with an explicit stereotype on it, means the same as “inverse attributes”:

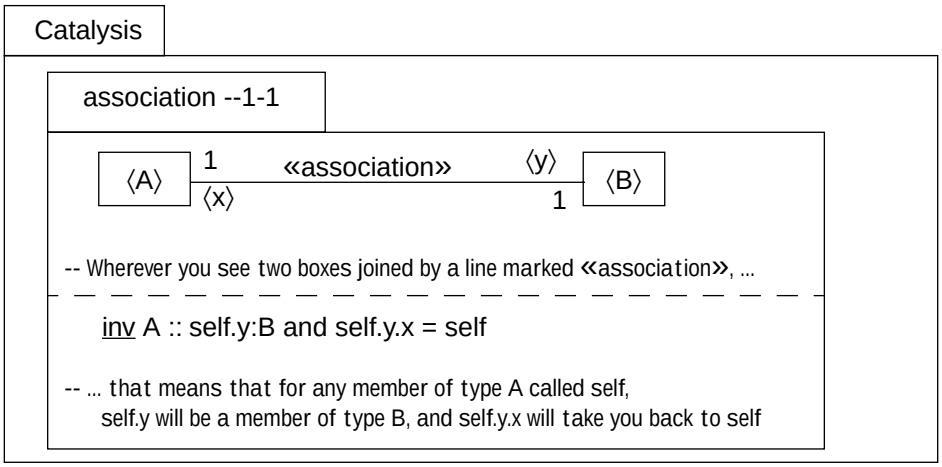


Figure 219: Template for: what is an “associaton”

Unadorned notations are also defined

So what should an association line mean if it has no stereotype tag? To define a default, just identify the untagged feature with the appropriate tag[1]:

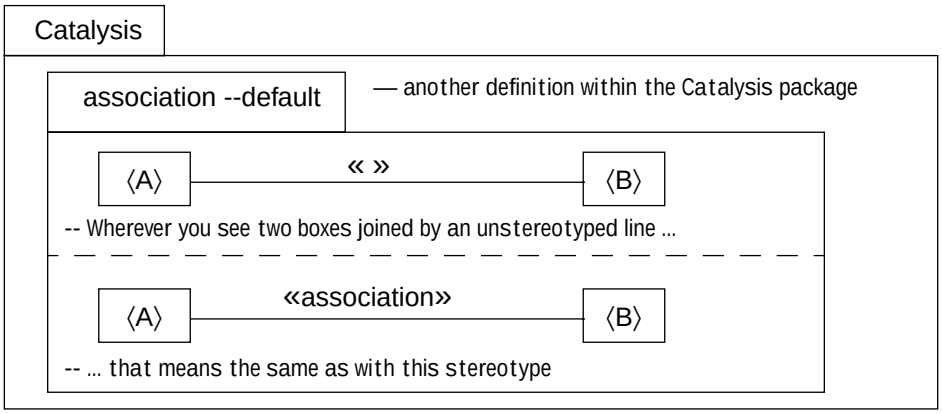
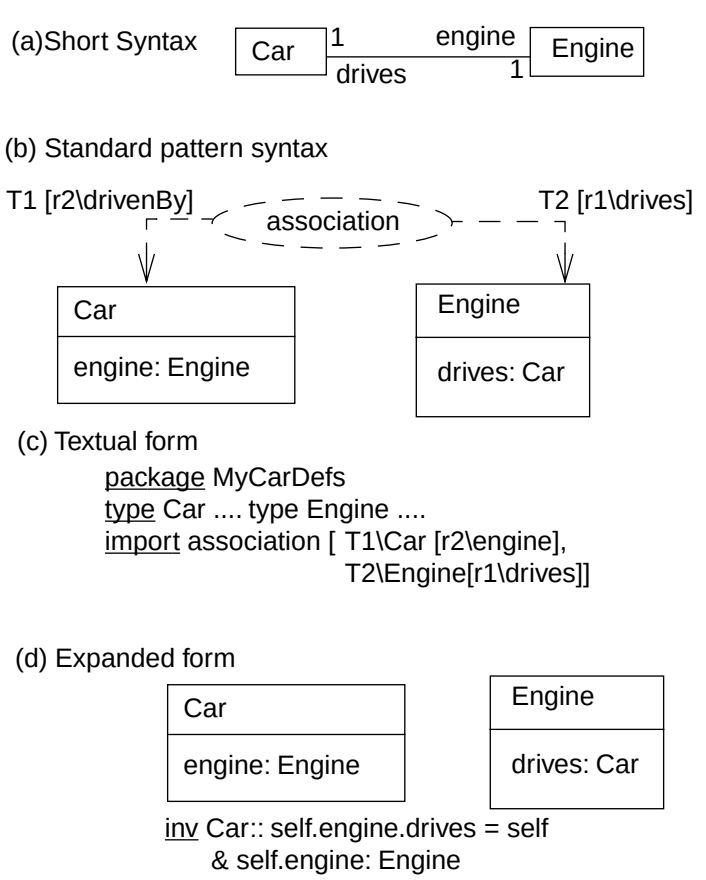


Figure 220: Template defining meaning of default notation

1. We will not generalize to visual grammars, ambiguities, etc.



Here, then, are some equivalent ways to define an association. The first uses the highest level notation: a line. Second, the pattern notation for applying a template. Third, a straight textual form; and lastly, the "expanded" result of any of the previous forms.

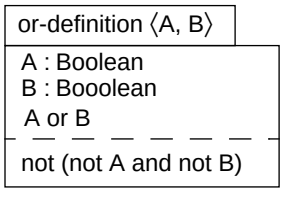
Figure 221: Interpreting association via template definitio

Seriously advanced topic

10.9 *Down to Basics with Templates*

10.9.1 Template packages to represent Inference Rules

Inference rules can be packaged



Templates can be used to represent general facts that are useful in understanding or reasoning about types. They can be presented as diagrams or as boolean expressions. For example, down at the very basic level, we can write things like or-definition. It means that, if ever you happen to find an expression involving 'or' and two Boolean expressions on either side of it, you can match them to <A> and <B>, and rewrite the whole thing using not and and.

Here are a few others:

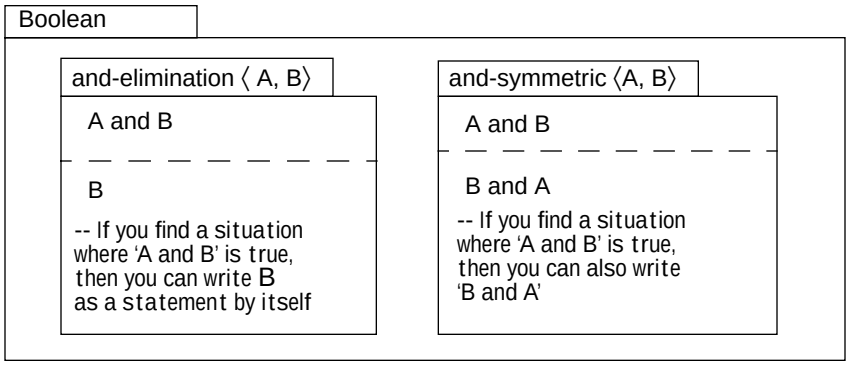


Figure 222: Templates for typical inference rules

Even induction

It is sometimes useful, with these kinds of rules, to use placeholders that themselves take arguments (which may sound mind-bending at first; but this is as bad as it gets):

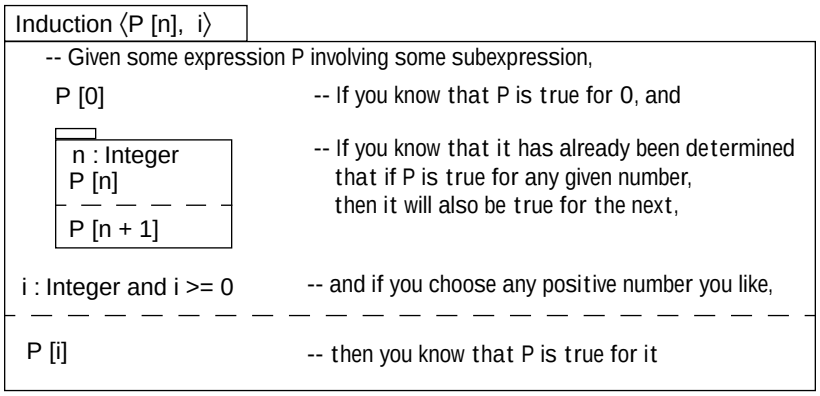


Figure 223: "Induction" template

To show cricket is boring...

We're now outside the realm of practical daily application, for most software developers. But just briefly, in case you're interested: the induction rule can be used to verify statements involving a progression. For example: to prove that all cricket scores are boring:

- A score of zero is clearly boring, since nothing has happened:

  A score of 0 is boring.

- Given any score, whether it is 42 or 103 or anything — call it x — then a score of x+1 is bound to be more boring than x. This is because x+1 can be achieved only after more cricket has occurred, which is clearly incrementally boring. So if x was boring, then x+1 definitely will be boring too. We can write this as an inference of

which we have satisfied ourselves; we don't need to give it a name, since we won't be needing it long:

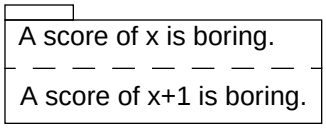


Figure 224: Clearly, cricket is boring

- Let's choose a particular score, say 200. You will agree that:

  200 : Integer and 200 >= 0

- Now at this stage, all the requirements of the Induction template have been met, with these substitutions:

  P [<x>] --> A score of <x> is boring.
  i --> 200

- The Induction rule tells us that we can now safely conclude:

  A score of 200 is boring.

Note that negative cricket scores might yet prove interesting.

10.9.2 Template packages for Primitive Types

No real 'primitive'; some may be 'pre-defined'

The primitive types that we use in every model and design — boolean, arithmetic, sets, lists, and dictionaries — can be defined in basic packages, imported by all others. These types are most easily defined in an axiomatic style — that is, by simply stating a

number of fundamental facts (‘axioms’, mathematicians call them) that are true about them, from which other facts follow. For example, the package defining boolean operators contains:

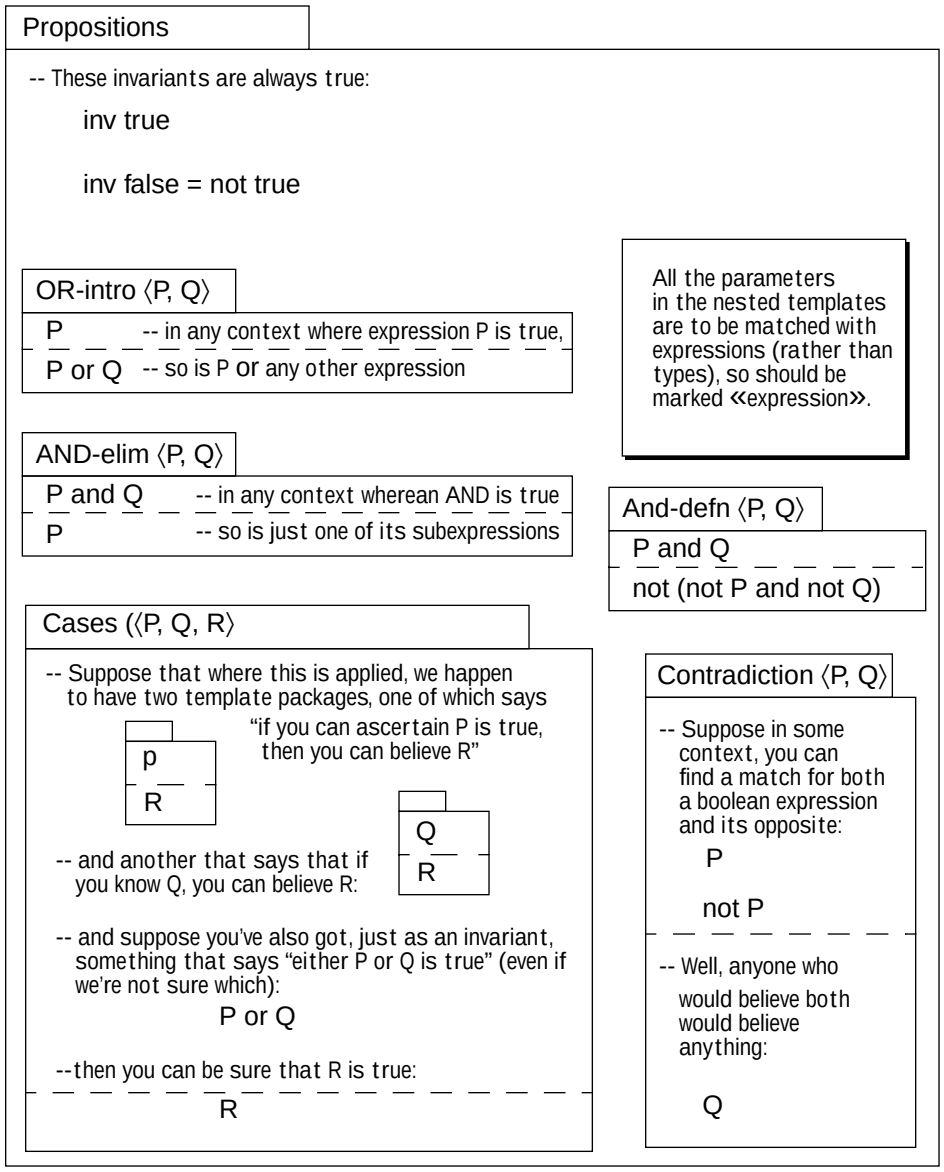


Figure 225: Propositions: a packaging of boolean operators

Predicate logic defined in terms of basic boolean operators

We can create another package that imports this one, defining ‘Predicate Logic’, which gives the meaning of the quantifiers “for all x, [som e expression about x]” and “there is an x such that ...”. A package about Sets comes next, and together with Predicates is imported to help define the rules of arithmetic. Other kinds of collection (lists and dictionaries or maps) can also be defined with the help of sets and predicates.

10.9.3 Layered Semantics

Then objects, types

The ideas of objects and types can also be defined in this style. Membership of a type is implied by observance of all the constraints (invariants, postconditions, etc) imposed by the type definition.

Can support different modeling languages, domain-specific extension, ...

In this fashion, we can build up a hierarchy of basic types and operators; not just the syntactic definitions, but their meanings too! And, since the basics can be different for different modeling and programming languages — not all have exactly the same idea of what the modulo operator means, for example — different packages can be supplied for users of different dialects, and referenced as stereotypes.

Example of layered packages

In fact, the entire semantics of the modeling and programming languages that you use can be defined in this way: choose your basic modeling package on which to build your specification; choose the Java package to be able to check that your code matches your spec. A typical hierarchy for modeling might be:

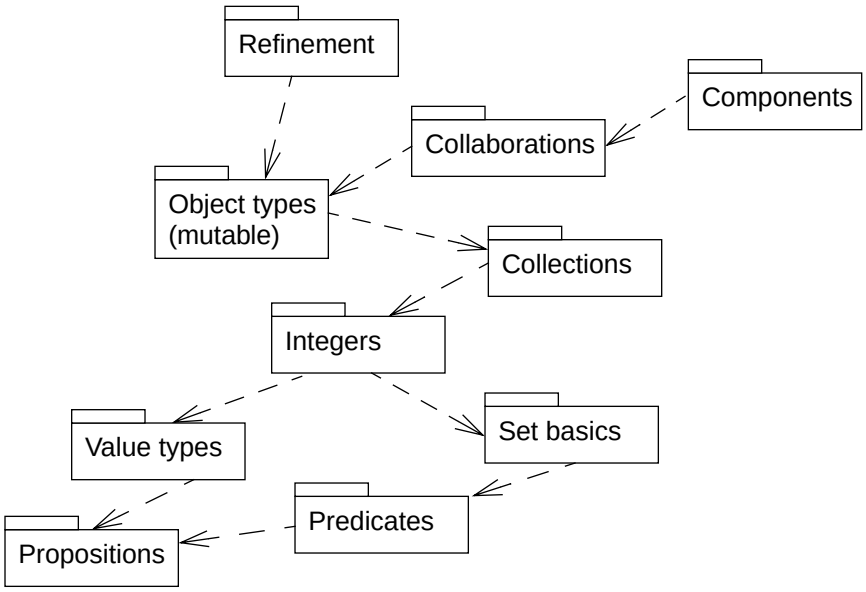


Figure 226: No primitives: full layering via packages and templates

Most users should not care

But for most users, it will not be necessary to know about the details of these basics, any more than you bother with the formal semantics of your programming language. But it is nice to know that this foundation can be made explicit, and that the details can be made a matter of choice.

And of course, most of these packages — especially the complex semantics ones — are virtual at present. But some research projects, notably Mural [Jones et al 1991] and Larch [Leavens et al] have indeed built up the packages of primitives. (The example given here comes from Mural.) [Wills 1992] contains work on the definitions of object types in this way.

One interesting feature of these packages is that they define a lot of “if you already know that this fact is true, then you can also assume that” rules. They are talking about the fundamental properties of the expressions we can write: in the arithmetic package for example, there will be something that tells us that x+y is the same as y+x, a fact that it wouldn’t be unusual to use in programming. In the extreme case of safety critical systems, designers can use these rules (and others like them dealing with programming language statements) to check that their programs fulfill robustness criteria, and indeed meet their specs; and build higher-level rules around them, checked once and then institutionalized as templates.

A few will want the reuse of specifications and proofs

For the writers of support tools, these basic packages are a way of discussing and defining the exact details of the languages they support mean. This should have the benefit of making them more interoperable, and should allow them to define more sensible consistency checks. But for most of us, the importance of this level of detail is secondary, arising from its relevance for tool designers.

Quite helpful for tools

10.9.4 Standard packages

We have seen that nested packages can be used to define a related set of stereotypes e.g. those needed for a particular method or language.

Modeling and programming languages

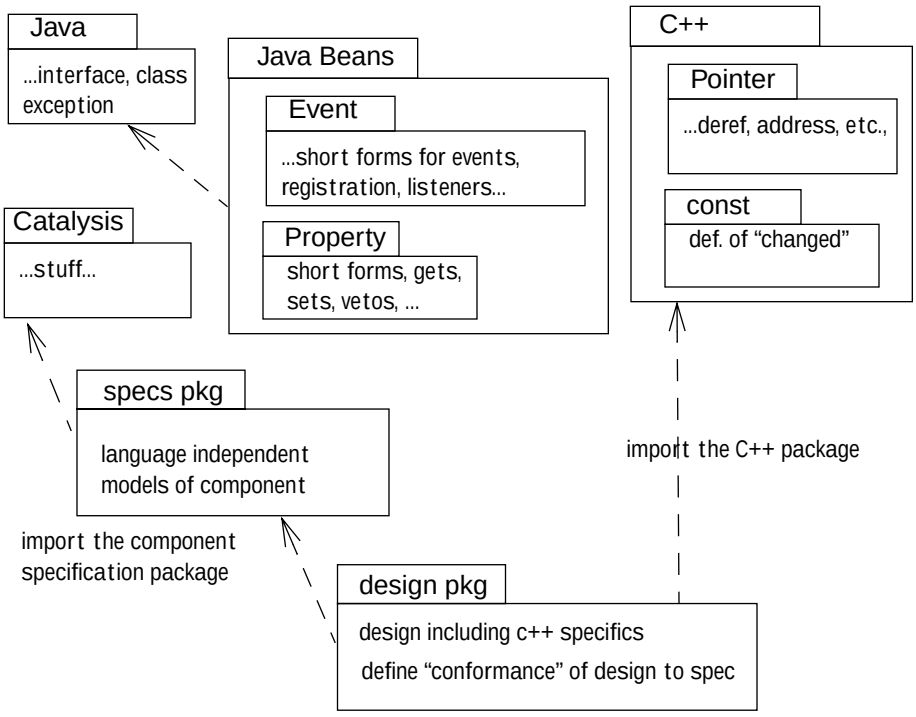


Figure 227: What can be achieved with standard packages

There is a Standard Catalysis package that is imported automatically into all others. It defines numbers, logic, and other basics. It is called catalysis.spec.lang. If you explicitly import any other *.spec.lang packages, catalysis.spec.lang is no longer automatically imported.

catalysis.spec.lang

catalysis.java.lang, catalysis.javabeans.lang

There are standard packages for various programming languages, enabling you to embed code in Catalysis designs. They are called catalysis.java.lang, catalysis.cpp.lang, catalysis.eiffel.lang and catalysis.smalltalk.lang. These packages define the valid syntax and semantics of programming constructs in these languages.

10.10 *Summary of model framework concepts*

10.10.1 Model framework definition and application

A model framework is a generic package containing both normal and placeholder definitions. A placeholder is a name which can be substituted when the framework is used. Each use or 'application' of the framework provides its own substitutions of the placeholders. Placeholder names are distinguished with ⟨angle brackets⟩. The names of attributes and associations of placeholder types are themselves placeholders. (This is not automatically true of actions.)

A framework can be applied just by quoting it in a model:

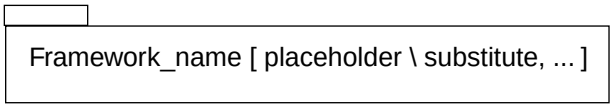


Figure 228: Textual framework application

In the substitution list, an attribute or action placeholder can be referred to by the qualified form TypeName :: attributeName. For placeholder types and actions, substitution can be shown pictorially with arrows:

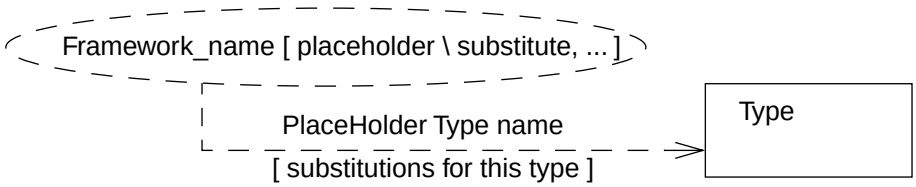


Figure 229: Graphical framework application

If the framework has a provisions section, an application is only considered meaningful if the requirements are already conformed to, prior to the applicaton, with the same substitutions. A justification should be attached to each application, documenting how the requirement is met.

10.10.2 Composition of definitions

Recall that every model can be regarded as a list of individual statements: the pictures can all be translated into formal text. A template package is a collection of statements; when it is applied, the statements (subject to substitutions) are added to the model.

Each type and action in the model is defined by all the statements made about it in its various appearances. Some, all, or none of these may come from template-applications. All these statements compose following the standard composition rules.

10.10.3 Usefulness of Model Frameworks

Model frameworks can be used to express relationships that straddle type boundaries, and to encapsulate relationships made up of a collection of types, associations, and actions. They are a powerful tool for abstraction, and a useful unit of reuse.