Chapter 9 Model Frameworks and Template Packages

It isn't only 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 them model frameworks.

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

Pieces of design combine to produce only designs, and they 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; being able to put a design together rapidly from prefabricated parts is a valuable facility. Moreover, later we will outline (Pattern 11.1, *Role Delegation*) a uniform design and implementation style that also lets you plug code components together.

This chapter deals with model frameworks and explains how to build and compose them using template packages. It also discusses how the fundamentals of the entire Catalysis approach are themselves defined as such frameworks and shows how they can form the basis for a modeling language that is truly extensible.

9.1 *Model Framework Overview*

After 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. We call this set of relationships a *model framework*. Many popular design patterns boil down to a model framework combined with surrounding how-to, when-to, and whether-to advice.

A suitable tool should be able to support the building of models and designs by application of model frameworks. Suppose, for example, that our business model has a type Stock with a numeric attribute level; choosing a package of user-interface pieces, we find a type Meter for displaying numeric readings.

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, so we don't have to repeat all the stuff that has already been said about the two types. We need only define the extra attributes and operations needed to connect them.

This is where model frameworks are useful. Let's assume that because Observation is a common pattern, we have defined a model framework for it; using it gives us an abbreviated way of defining what we need (see Figure 9.1).

The vehicle for a framework is a generic, or *template*, form of package. Inside the template, some types and their features can be 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 for the Stock's level.

© ***framework*** A template package; a package that is designed to be imported with substitutions. It "unfolds" to provide a version of its contents that is specialized based on the substitutions made. (Note that our usage of *framework* is somewhat broader than its traditional usage as a collection of collaborating abstract classes.)

A framework can abstract the description of a generic type, a family of mutually dependent types, a collaboration, a refinement pattern, the modeling constructs themselves, and even a bundle of fundamental generic properties (associative, commutative, and so on). Frameworks are themselves built on other frameworks. At the most basic level, the structure of frameworks represents the basis for the organization of all models.

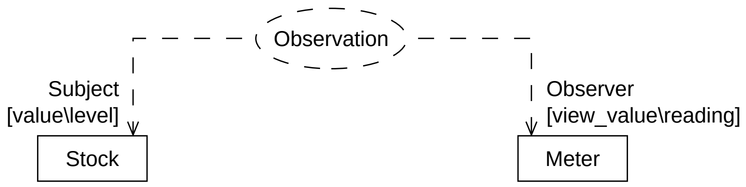


Figure 9.1 Example of applying a model framework.

Notice that what the Observation framework does is to represent a cluster of design decisions about how two types of objects should collaborate to provide the required effect. It does not discuss any other roles and collaborations of those objects, and it 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 9.11.

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.

Some programming languages have *class templates* or *generic classes*; UML has them, too. The notation is slightly different, but a class template is a model framework that contains only one class. We'll discuss class templates in more detail later.

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 11, Reuse and Pluggable Design: Frameworks in Code. This chapter is about frameworks of abstract models.

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. Ideally, the tool should keep the definitions of the framework, 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.

Among current popular tools, there is some support for templates in a restricted way. Typically, the template works more like a *script*: a series of operations that is applied once to a model, adding the necessary attributes and operations. This technique 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, because it is written in a scripting language.

In summary, templates provide a powerful way to capture reusable model frameworks, whether at an abstract specification level or down in the detailed design. In particular, templates are good for capturing collaborations. Even without tools, the template notation is a useful form of abbreviation even when 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 using only attributes and associations; then we will go on to deal with actions. Subsequent sections add further ideas, and a summary of concepts appears at the end of the chapter.

9.2 *Model Frameworks of Types and Attributes*

Suppose a plumbing company asks us to do an analysis of its business preparatory to getting some computerized support. After a day or two with them, we arrive at the central model shown in Figure 9.2. 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 are only a certain number of kinds of Job, and each is described by a JobDescription. Among other things, this model says which Skills are required for the job (electrical wiring, excavation, denial of responsibility, and so on). Each Plumber is qualified with a list of Skills. A key invariant is that no Plumber should be assigned to a Job for which he or she lacks 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.

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

So we generalize our model into a framework by creating a template package, as shown in Figure 9.3. (We will later drop the «framework» stereotype. 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* (see Figure 9.4). Notice that several of the names inside the framework definition are written within angle brackets (< >); they are *placeholders* that should be identified with actual type names when the framework is applied. This is the effect of the labeled arrows when the framework is applied.

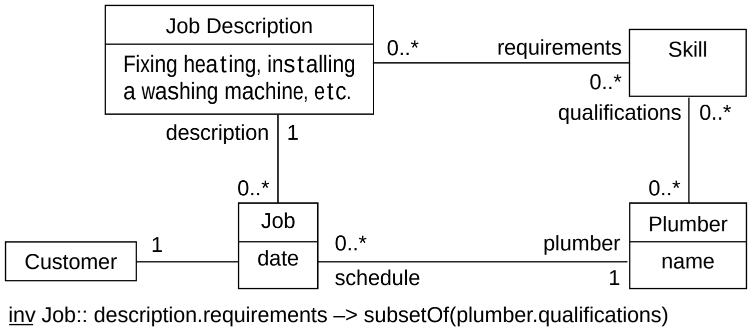


Figure 9.2 Model of allocating plumbers to jobs.

In the resulting model, each type has all the features given to it explicitly (such as, 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 can show an unfolded version on demand.

Turning again to seminar scheduling, we produce the model shown in Figure 9.5. 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 (projectors,

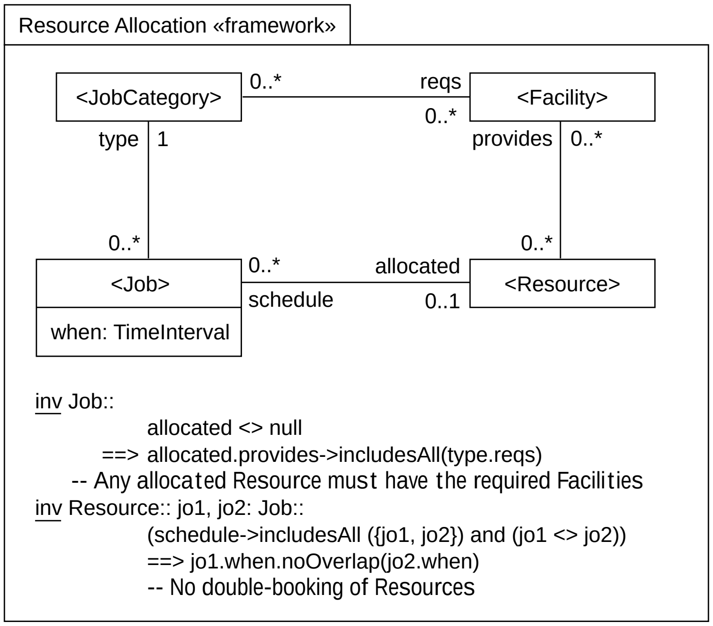

Resource Allocation «framework»
<JobCategory>
0..*
reqs
<Facility>
0..*
type
1
provides
0..*
0..*
0..*
<Job>
when: TimeInterval
0..*
schedule
allocated
0..1
<Resource>
inv Job::
allocated <> null
==> allocated.provides->includesAll(type.reqs)
-- Any allocated Resource must have the required Facilities
inv Resource:: jo1, jo2: Job::
(schedule->includesAll ({jo1, jo2}) and (jo1 <> jo2))
==> jo1.when.noOverlap(jo2.when)
-- No double-booking of Resources

Figure 9.3 Model of resource allocation framework.

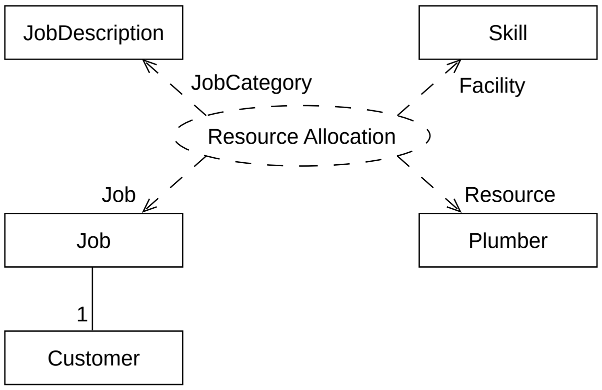


Figure 9.4 Application of resource allocation to plumbing.

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.[2]

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

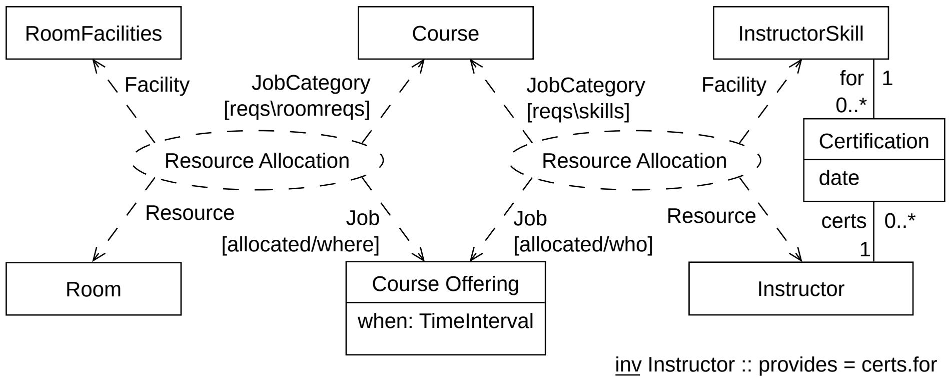


Figure 9.5 Double application of resource allocation to seminar scheduling.

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 ...]
```

When unfolded, statements from each framework application, after the necessary substitutions are made, are composed with each other and with any local definitions that are applicable. The unfolding is shown in Figure 9.6. Clearly, using frameworks 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.

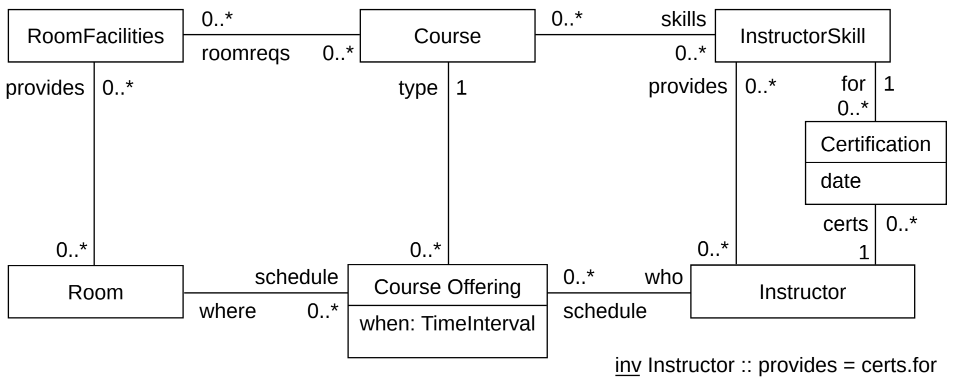
© ***unfolding*** Depicting the results of an import, possibly including substitutions, in the context of the importing package with the appropriate elements substituted.

© ***framework applicaton*** An import of a framework with substitutions; usually depicted graphically using a UML "pattern" symbol, with labeled lines for the type substitutions and text annotations for finer-grained substitutions (attributes, actions, and so on).

9.2.1 Framework Applications Are Not Subtypes

Could we have used subtyping to express the similarity between Courses and plumbers' JobDescriptions (Figure 9.7)? 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 specialized but rather the entire group of them along with their relationships and interactions.

2. The UML symbol for a pattern is a dashed use case, although pattern semantics have nothing to do with UML use cases. UML 1.1 does not have any semantics for patterns, only a notation; perhaps our semantics will be adopted.



CourseOffering:: who <> null ==> who.provides->includesAll(type.skills)
    -- Any allocated Instructor must have the required Skills
CourseOffering:: where <> null ==> where.provides->includesAll(type.roomreqs)
    -- Any allocated Room must have the required RoomFacilities
Instructor:: jo1, jo2: Job:: (schedule->includesAll ({jo1,jo2}) & jo1 <> jo2) ==> jo1.when.noOverlap(jo2.when)
    -- No double-booking of Instructors
Room:: jo1, jo2: Job:: (schedule->includesAll ({jo1,jo2}) & jo1 <> jo2) ==> jo1.when.noOverlap(jo2.when)
    -- No double-booking of Rooms

Figure 9.6 Unfolded view after applying frameworks.

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

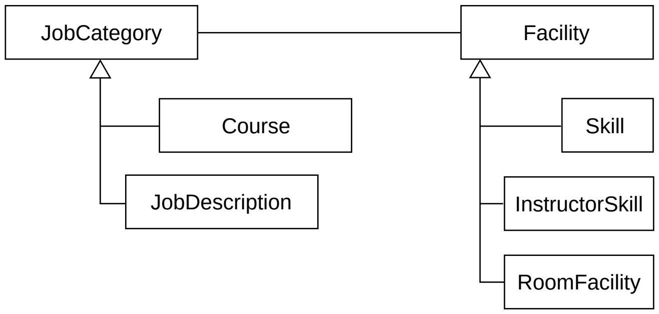


Figure 9.7 Why subtyping does not correctly reflect frameworks.

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 as shown in Figure 9.8.

Now we can explicitly apply the framework to those pairs that are acceptable (see Figure 9.9). The association eats might represent the assignment of food items to specific ani-

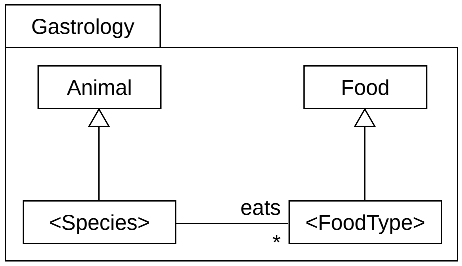


Figure 9.8 A gastrology framework.

mals 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.[3]

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

9.3 *Collaboration Frameworks*

A collaboration describes the interactions between a group of objects that are designed to work together. They send one another messages intended to attain a goal that they are designed to achieve jointly. Much of the skill of object-oriented design is about designing collaborations. The CRC technique (classes, responsibilities, collaborations) is basic to

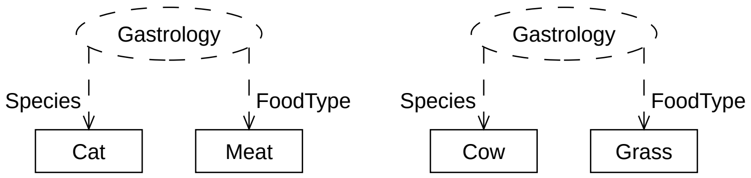


Figure 9.9 Application of gastrology framework.

OO design and is all about dividing the responsibilities for a task among 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 that is flexible and extensible. Doing it badly leads to an unmanageable mess.

3. Would that it were so.

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. Some tools explicitly provide a way to define collaborations and then compose them into bigger designs.

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

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 one another. 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.

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 must 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.

The Subject-Observer collaboration is a more technical example. In Figure 9.10, we try to show each object's external interface as split into different roles in different collaborations, whereas the internal attributes may be shared. 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. An update action is initiated by the Subject to keep the Observer up-to-date.

Of course, any pair of classes that conforms to this relationship in some chunk of program code will probably not be called Subject and Observer. They'll have bigger, more interesting roles in their program, perhaps as pieces of a GUI or proxies in a distributed system. But this is exactly what frameworks are about: We can define only 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 9.10, we really know only 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. We know only that it can happen and that when it does, the Observer must be updated.

9.3.1 Using Invariant Effects in Collaboration Frameworks

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

- The update action between the Subject and Observer

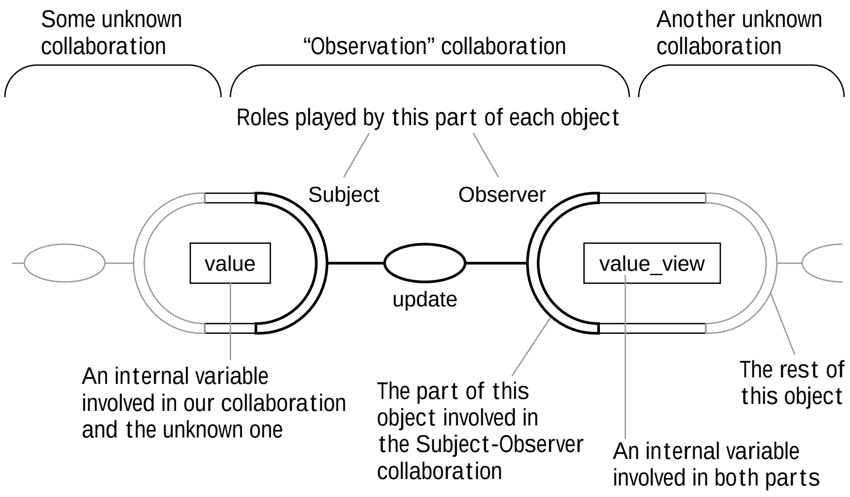

Some unknown collaboration
"Observation" collaboration
Another unknown collaboration
Roles played by this part of each object
Subject
Observer
value
update
value_view
An internal variable involved in our collaboration and the unknown one
The part of this object involved in the Subject-Observer collaboration
An internal variable involved in both parts
The rest of this object

Figure 9.10 Collaborations are about parts of objects.

- All the other actions we don't know about, which might change the Subject's value

  Specifying the first one is easy:

```
action Observer :: 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.)

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 (see Section 3.5.4, Effect Invariants) on the *external* section of the collaboration (see Section 4.8.1, External Actions). It is a postcondition without an action name or signature:

```
inv effect Subject ::
      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).

9.3.1.1 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.

Figure 9.11 on the next page 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 applies only to Observers that have a current Subject. 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; we have specified only that there should be such an action, with responsibility for executing it distributed somehow between Subject and Observer. When designers apply the framework to a particular pair of types, it tells them to provide this facility.

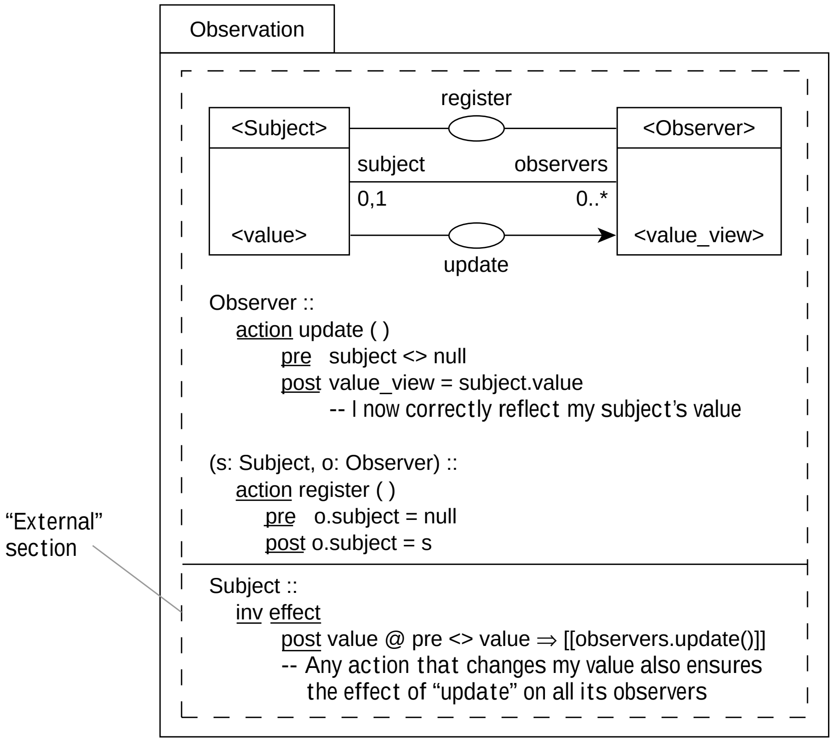

Observation
register
<Subject>
<Observer>
subject
observers
0,1
0..*
<value>
<value_view>
update
Observer ::
action update ( )
pre subject <> null
post value_view = subject.value
-- I now correctly reflect my subject's value
(s: Subject, o: Observer) ::
action register ( )
pre o.subject = null
post o.subject = s
Subject ::
inv effect
post value @ pre <> value ⇒ [[observers.update()]]
-- Any action that changes my value also ensures
the effect of "update" on all its observers
"External" section

Figure 9.11 Collaboration template.

9.3.1.2 More Abstract Models

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:

```
inv effect Subject::
    post    value@pre <>  value ==>
                observer.value_view = value
            -- Any action that causes a change in value
               must ensure that the observer's latest 'value_view'
               is the same as our latest 'value'
```

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

```
inv Observer :: subject.value = value_view
```

9.3.2 Applying a Collaboration Framework

One of the authors' clients designed a call management system for telephone sales teams. One of the types in the model was a CallQueue: the list of calls waiting for a particular group of operators. Let's suppose we have that type defined in one package; in another package, we have a kit of GUI widgets, one of which is a Thermometer—a useful display for numeric values. Figure 9.12 shows parts of their models.

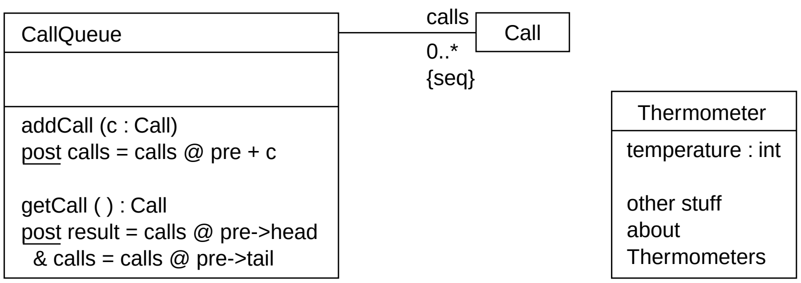


Figure 9.12 Target type model for using observation.

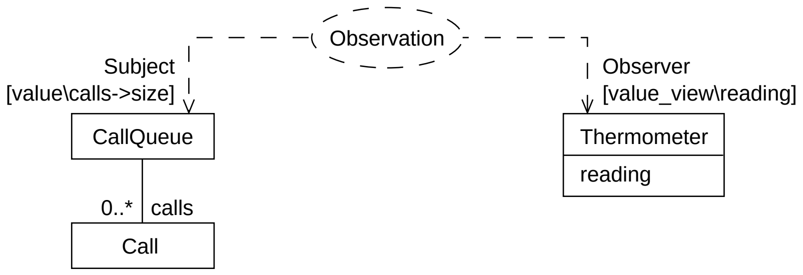
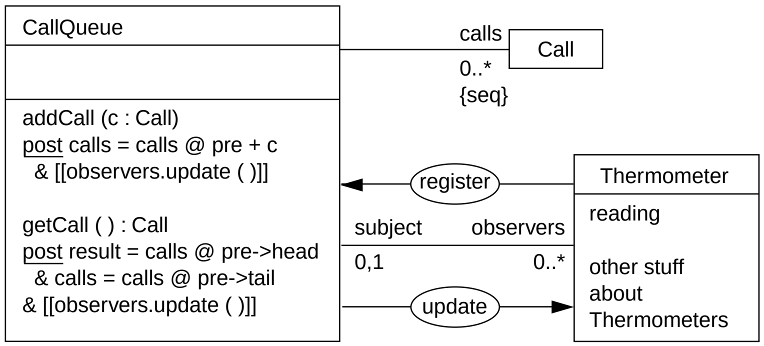


Figure 9.13 Framework application and substitutions.

Now we are designing the bridge between business logic and GUI, and we 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 as shown in Figure 9.13. 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 the result of applying the framework and gives the spec (see Figure 9.14).



```
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 9.14 Unfolded result of framework application.

Notice that 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.)

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

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

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

9.3.3 Using One Framework to Build Another

The idea of registering and unregistering is common to any situation in which each of two objects needs to know about the other. So we could separate this scheme into its own framework (see Figure 9.15). Any instance of A and B can be linked; aa and bb are the links in each direction. (The arrows indicate that 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.

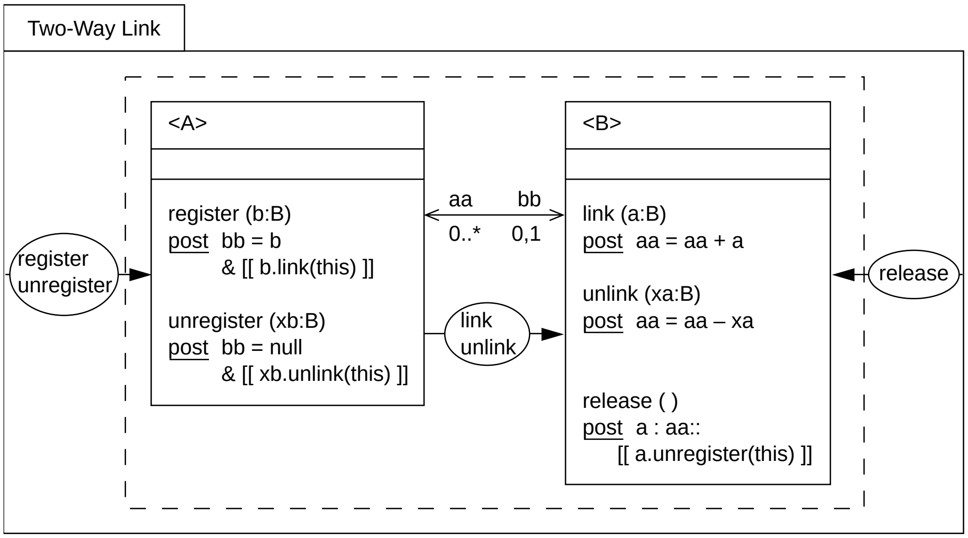


Figure 9.15 Low-level framework: Two-Way Link.

Now we see that we could have used this framework to help define the Observation framework (see Figure 9.16). (In fact, it can say more than before, because it now tells how the register action works rather than just calls for one.) Again, comparing Figure 9.16 with Figure 9.11, we can see that the use of a framework 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.

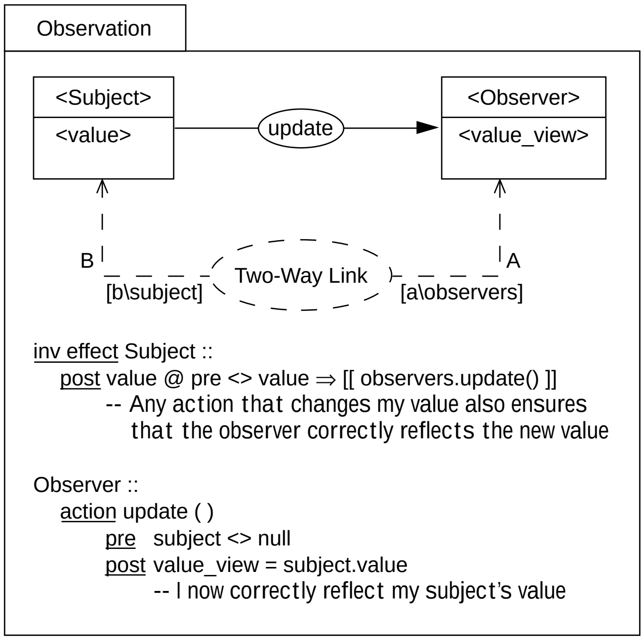

Observation
<Subject>
<value>
update
<Observer>
<value_view>
B
A
Two-Way Link
[b\subject]
[a\observers]
inv effect Subject ::
post value @ pre <> value ⇒ [[observers.update()]]
-- Any action that changes my value also ensures that the observer correctly reflects the new value
Observer ::
action update ( )
pre subject <> null
post value_view = subject.value
-- I now correctly reflect my subject's value

Figure 9.16 Observation framework using Two-Way Link framework.

9.4 *Refining Frameworks*

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

Fortunately, our frameworks have an additional dimension of flexibility: *refinement* (see Figure 9.17). 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 other models are. Furthermore, some of these refinements are themselves defined as frameworks.

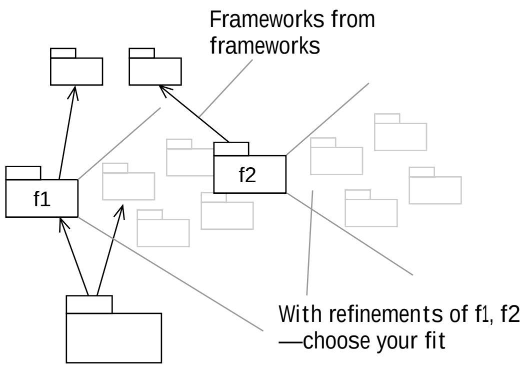


Figure 9.17 Frameworks are subject to refinement.

9.4.1 A Requirement

The framework in Figure 9.18 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 acquires it. We have shown this as a degenerate collaboration, as it will next be refined as a unit. (Notice that 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.)

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

9.4.2 A Collaboration Refining the Requirement

According to the Trade Supply framework (see Figure 9.19), 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.

There are different kinds of Traders in the world, and they get rid of their stocks in different ways. Some only sell them; others cook them up and serve them; still 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.

In an earlier example, we used only 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. Second, 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.

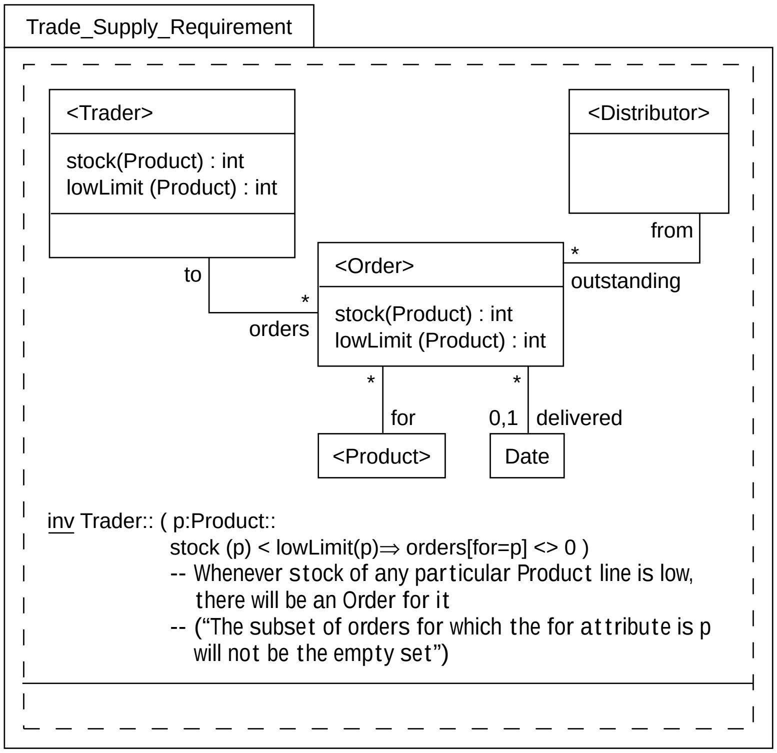

Trade_Supply_Requirement
<Trader>
stock(Product) : int
lowLimit (Product) : int
<Distributor>
from
*
outstanding
to
*
orders
<Order>
stock(Product) : int
lowLimit (Product) : int
*
*
for
0,1
delivered
<Product>
Date
inv Trader:: (p:Product::
stock (p) < lowLimit(p)⇒ orders[for=p] <> 0)
-- Whenever stock of any particular Product line is low, there will be an Order for it
-- ("The subset of orders for which the for attribute is p will not be the empty set")

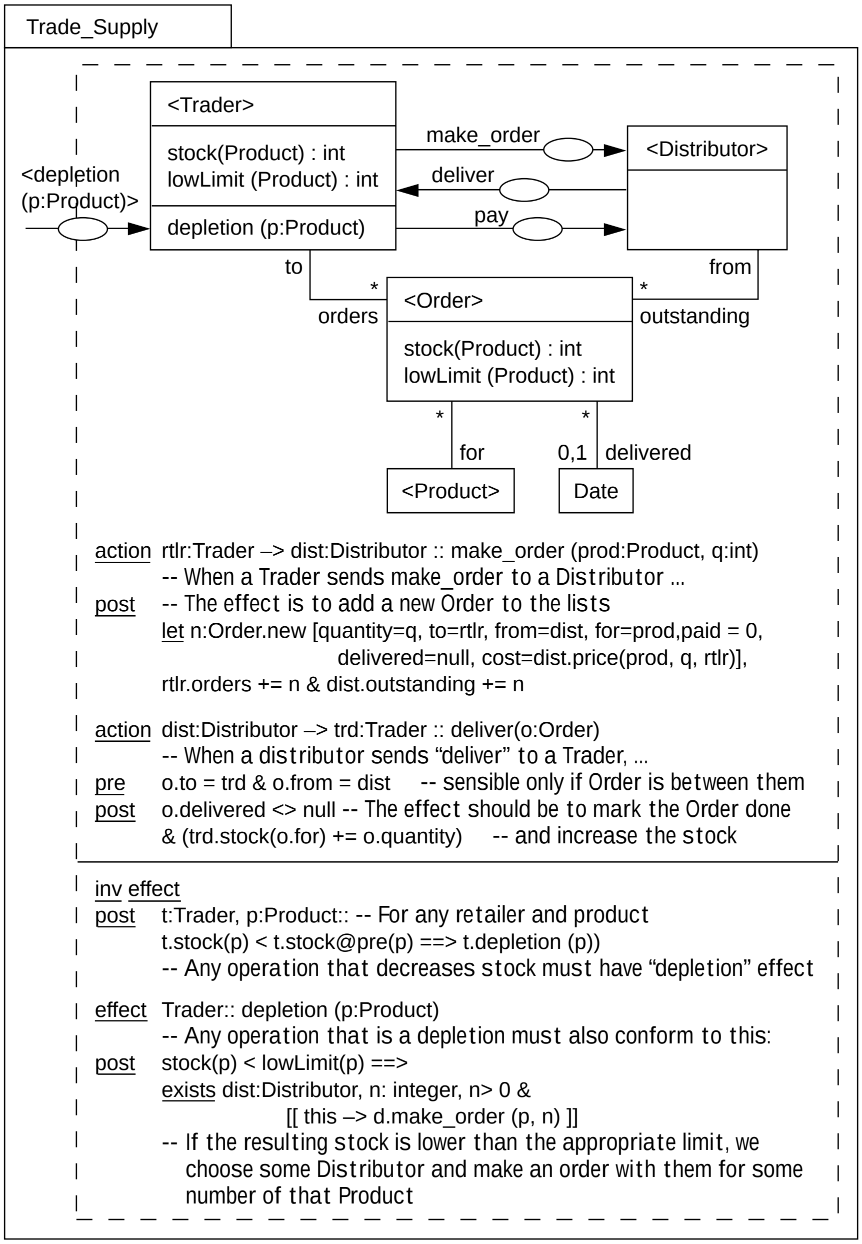
Figure 9.18 A framework for trading: stock maintenance.

9.4.3 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 must 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 its statements (pictorial or otherwise) already imply those in Trade Supply Requirements.

The general rules are the same as those discussed in Chapter 6, Abstraction, Refinement, and Testing. 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 9.20. (Actually, a more rigorous treatment of this argument reveals a hole regarding when Orders get removed. You might like to tighten the reasoning and the spec of Trade Supply.)



```
action  rtlr:Trader –> dist:Distributor :: make_order (prod:Product, q:int)
        -- When a Trader sends make_order to a Distributor ...
post    -- The effect is to add a new Order to the lists
        let n:Order.new [quantity=q, to=rtlr, from=dist, for=prod,paid = 0,
                    delivered=null, cost=dist.price(prod, q, rtlr)],
        rtlr.orders += n & dist.outstanding += n

action  dist:Distributor –> trd:Trader :: deliver(o:Order)
        -- When a distributor sends "deliver" to a Trader, ...
pre     o.to = trd & o.from = dist     -- sensible only if Order is between them
post    o.delivered <> null -- The effect should be to mark the Order done
        & (trd.stock(o.for) += o.quantity)     -- and increase the stock

inv effect
post    t:Trader, p:Product:: -- For any retailer and product
        t.stock(p) < t.stock@pre(p) ==> t.depletion (p))
        -- Any operation that decreases stock must have "depletion" effect

effect  Trader:: depletion (p:Product)
        -- Any operation that is a depletion must also conform to this:
post    stock(p) < lowLimit(p) ==>
        exists dist:Distributor, n: integer, n> 0 &
                    [[ this –> d.make_order (p, n) ]]
        -- If the resulting stock is lower than the appropriate limit, we
           choose some Distributor and make an order with them for some
           number of that Product
```

Figure 9.19 Trade Supply collaboration.

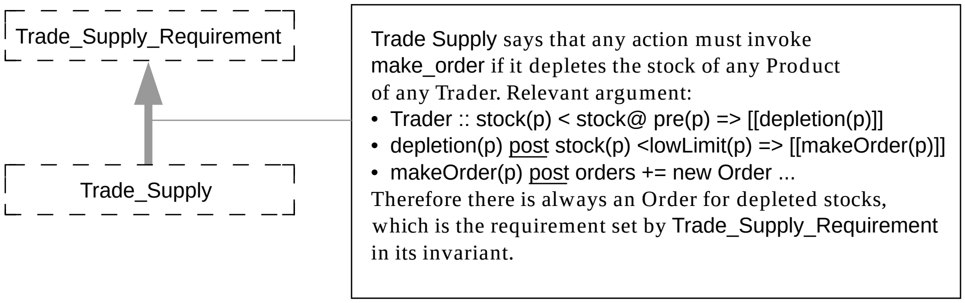

Trade_Supply_Requirement
Trade_Supply
Trade Supply says that any action must invoke
make_order if it depletes the stock of any Product
of any Trader. Relevant argument:
• Trader :: stock(p) < stock@ pre(p) => [[depletion(p)]]
• depletion(p) post stock(p) <lowLimit(p) => [[makeOrder(p)]]
• makeOrder(p) 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 9.20 Documenting a framework refinement.

9.5 *Composing Frameworks*

A commercial retailer plays many roles, participating in many collaborations. One such collaboration is its interactions with customers (see Figure 9.21). The collaboration shows

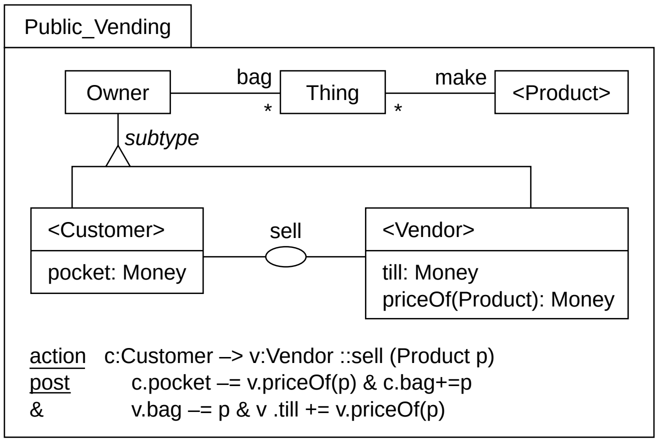

Public_Vending
Owner
bag
*
Thing
*
make
<Product>
subtype
<Customer>
pocket: Money
sell
<Vendor>
till: Money
priceOf(Product): Money
action c:Customer –> v:Vendor ::sell (Product p)
post c.pocket –= v.priceOf(p) & c.bag+=p
& v.bag –= p & v .till += v.priceOf(p)

Figure 9.21 Another view of the vendor: retail sales.

the relationship of Customers to Vendors. In the sell operation, cash and Products are transferred in opposite directions.

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 (see Figure 9.22). 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, using an invariant, to the Trader's stock that had been modeled as an integer for any Product.

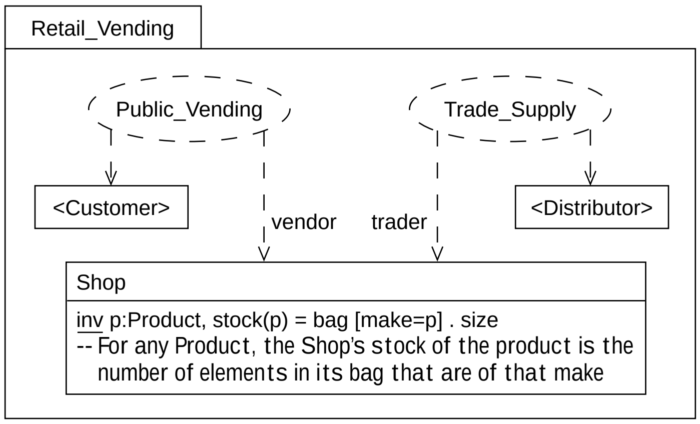

Retail_Vending
Public_Vending
Trade_Supply
<Customer>
<Distributor>
vendor
trader
Shop
inv p:Product, stock(p) = bag [make=p] . size
-- For any Product, the Shop's stock of the product is the number of elements in its bag that are of that make

Figure 9.22 Joining roles by applying two frameworks.

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

9.5.1 Building Systems from Collaborations

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.

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 organization). 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.

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 designs that are more flexible.

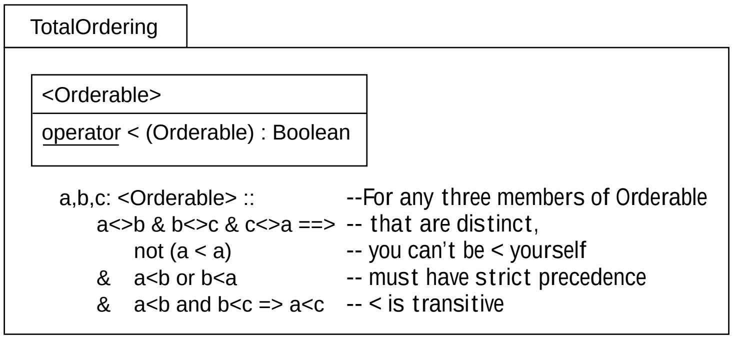
9.6 *Templates as Packages of Properties*

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

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

How do we avoid repeating these rules every time we state them? It is not a solution to say that 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. (For the same reason, we didn’t use subtypes for the Jobs and Skills in Section 9.2.1.)

Treating operators as functions (as in C++), we instead make a template package (see Figure 9.23). TotalOrdering is being used as a convenient package for a set of assertions or properties that we can apply to different types. Because we’re going to use the template



```
a,b,c: <Orderable> ::             --For any three members of Orderable
      a<>b & b<>c & c<>a ==>      -- that are distinct,
            not (a < a)           -- you can't be < yourself
      &     a<b or b<a            -- must have strict precedence
      &     a<b and b<c => a<c    -- < is transitive
```

Figure 9.23 Framework for TotalOrdering.

many times, we take the trouble to set out the rules precisely. Groups of useful properties such as TotalOrdering are sometimes called *traits*. With a rich enough library of traits,[4] you can make a wide variety of type definitions by combining several traits in “mix and match” style.

Not all operators have the TotalOrdering properties. For example, when you make a project plan, the tasks have only a partial ordering with respect to the must-precede operator. Task A must be finished before B and C, and both B and C must be finished before D

4. Including the obvious ones—associative, commutative, and idempotent—and many not-so-obvious ones that help factor fundamental commonality in structure and behavior.

is started; but the order of finishing B and C might not matter. Figure 9.24 shows some types whose properties can be defined in part with the help of this template.

9.6.1 A Template Can Have Provisions

A sorted list keeps its items in a uniquely determined linear order. You can make sorted lists of almost any type of object provided that it has a comparison operator with the Tota-

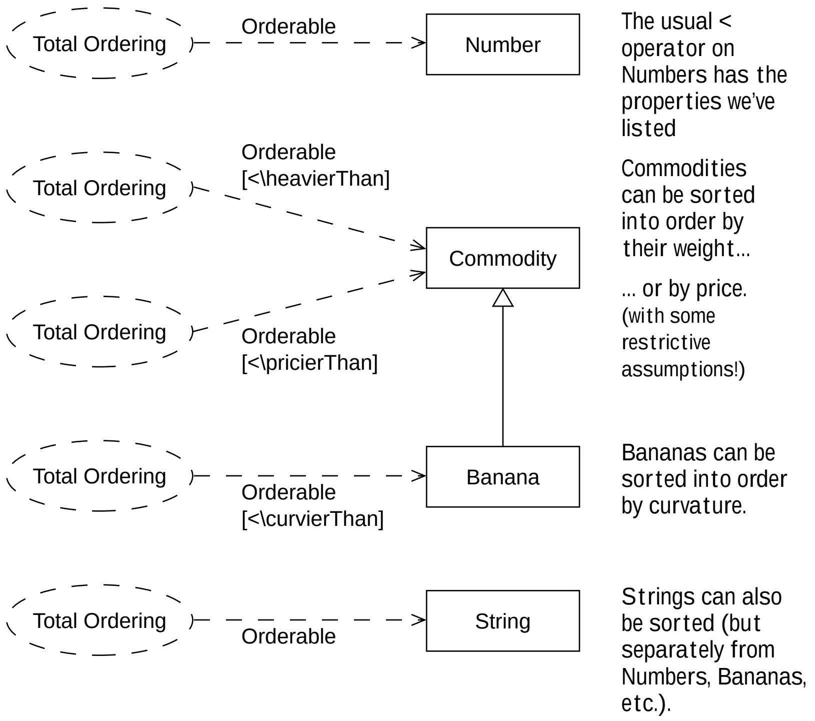

Total Ordering
Orderable
Number
The usual < operator on Numbers has the properties we've listed
Total Ordering
Orderable [<\heavierThan]
Commodity
Commodities can be sorted into order by their weight...
Total Ordering
Orderable [<\pricierThan]
... or by price. (with some restrictive assumptions!)
Total Ordering
Banana
Orderable [<\curvierThan]
Bananas can be sorted into order by curvature.
Total Ordering
Orderable
String
Strings can also be sorted (but separately from Numbers, Bananas, etc.).

Figure 9.24 Many different total-ordered items.

lOrdering properties. The template in Figure 9.25 defines what a SortedList means and includes the TotalOrdering properties on its items.

A Sorted List Template can be applied to any type; when applied, the imported TotalOrdering template will impose its properties on the type substituted for <Item>. The fragments in Figure 9.26 define different types that represent sorted lists with different content types.

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 TotalOrdering. If you put a bunch of Bananas into a

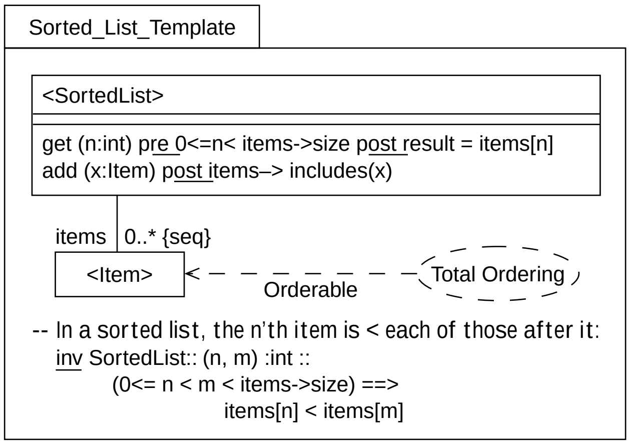

Sorted_List_Template
<SortedList>
get (n:int) pre 0<=n< items->size post result = items[n]
add (x:Item) post items–> includes(x)
items
0..* {seq}
<Item>
Orderable
Total Ordering
-- In a sorted list, the n'th item is < each of those after it:
inv SortedList:: (n, m) :int ::
(0<= n < m < items->size) ==>
items[n] < items[m]

Figure 9.25 Sorted List framework: What kinds of items are OK?

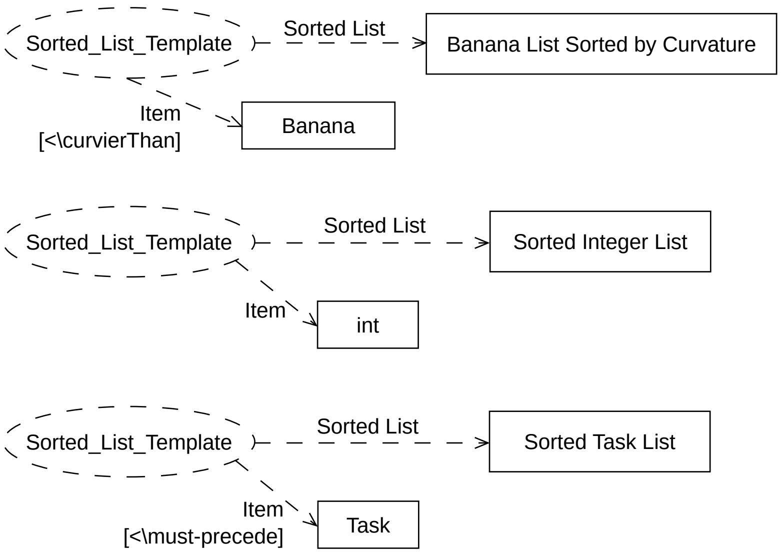


Figure 9.26 Applying the Sorted List framework with substitutions.

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, which should not really be totally ordered based on must-precede? The TotalOrdering properties would be imposed on Task, something that (1) is probably not what you intended, because it imposes a linear order on all tasks, and (2) could be inconsistent with the definitions of the must-precede operator itself.

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

In a separate section of the package we provide a way for the designer of a template to say, “This template should be applied only to things that you already intend to have certain properties.” The idea, called a *provision*, is a bit like a precondition, except that it typically works at design time.[5] Figure 9.27 shows an improvement of Sorted List Template. It says that if you have a type to which the TotalOrdering properties already apply, then it is OK to make Sorted Lists of it.

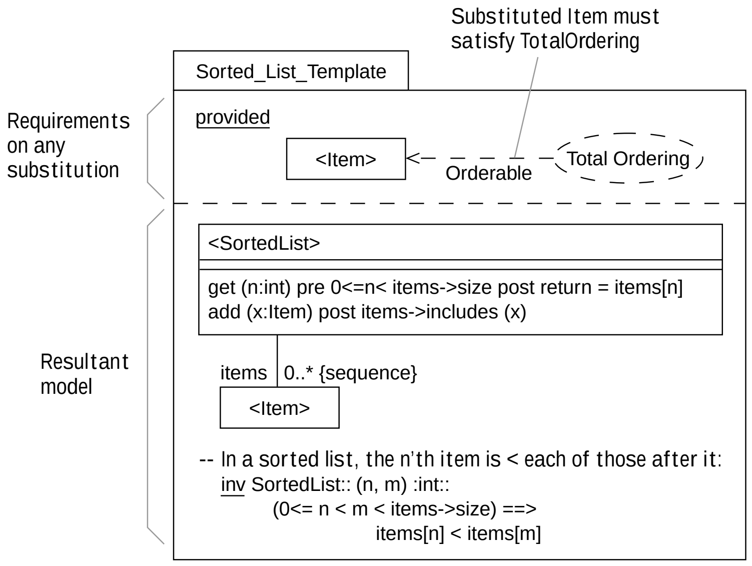

Substituted Item must satisfy TotalOrdering
Sorted_List_Template
Requirements on any substitution
provided
<Item>
Orderable
Total Ordering
<SortedList>
get (n:int) pre 0<=n< items->size post return = items[n]
add (x:Item) post items->includes (x)
Resultant model
items
0..* {sequence}
<Item>
-- In a sorted list, the n'th item is < each of those after it:
inv SortedList:: (n, m) :int::
(0<= n < m < items->size) ==>
items[n] < items[m]

Figure 9.27 Explicit substitution provisions in SortedList framework.

In the provisions section, you can put any model to which the substituted types must *already* conform.[6] Thus, you can require that one substituted type be a subtype of another; or that they have some relationship (see Section 9.2.1, Framework Applications Are Not Subtypes) or satisfy some predicate. The designer who applies the template must check, perhaps with help from a tool, that all the other parts of his or her model imply the properties laid down as provisions, and this should be documented much as a refinement is documented.

5. Using *reflection*, you can write generic code that does similar checks at runtime.

6. This lets us correctly describe C++ template design and usage, including the Standard Template Library.

To see exactly what this means, we must recall that all models are defined within a package and that models are usually structured into packages. The people who use 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 (see Figure 9.28). 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 9.20).

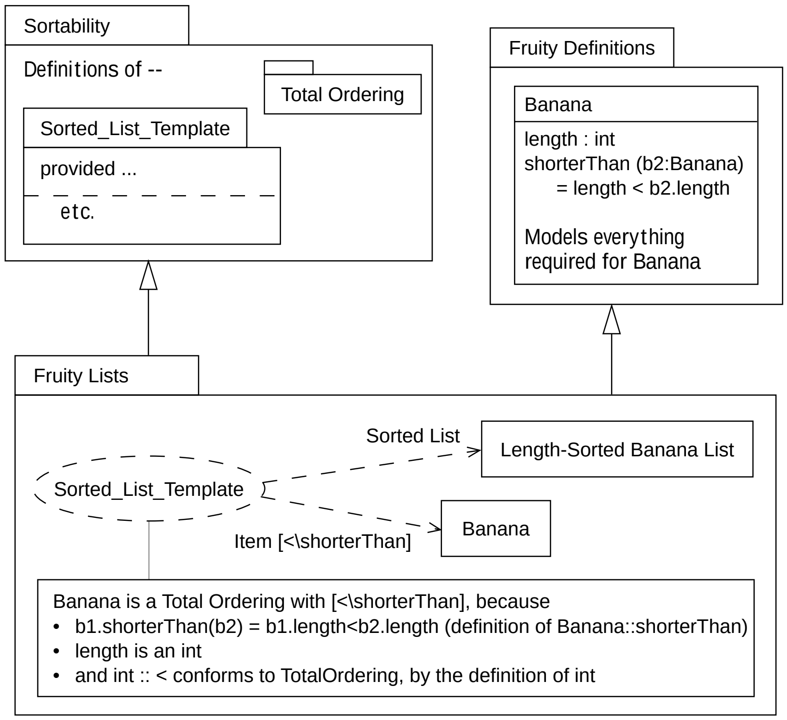


Figure 9.28 Applying templates that have provisions.

In general, when you build a framework you must 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.

© ***provisions*** A set of prerequisites associated with a framework; any elements substituted when applying this framework must meet those prerequesites in order for the framework to be applicable. Provisions are analogous to "design-time" preconditions.

9.6.2 Template as Generic Types and Classes

Many templates exist to define a single family of types, such as 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 to explicitly substitute that for the placeholder in the template. An abbreviated notation covers these cases.

Within the definition of the template package, you can use its name as one of the placeholder names—for example, the name of a type. Using UML conventions, add an inset dashed box at the corner of the type and list the placeholders of the template (see Figure 9.29).

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). Place the template parameters in the UML inset dashed box or show them as explicit textual substitutions[7] in the form shown in Figure 9.30. Either one is equivalent to drawing the syntax shown in Figure 9.31. (In C++, the equivalent would be roughly SortedList < Banana, shorterThan>.)

Nested packages are useful when they are generic; a standard package can contain definitions of several generics (see Figure 9.32). A user can import the container, making the names of the nested generic packages visible, and then apply the generics. Package provisions also work well with package nesting.

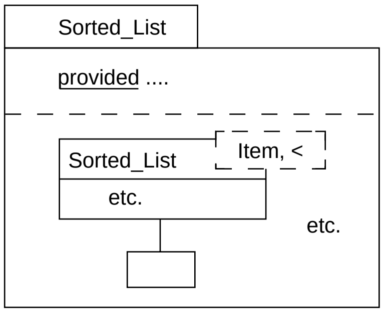


Figure 9.29 Defining a template with convenient syntax.

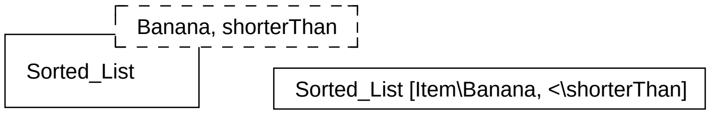


Figure 9.30 Alternative convenient syntax for applying a template.

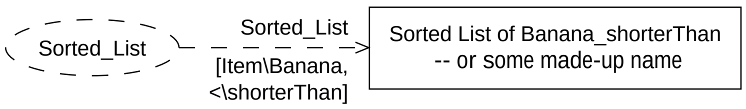


Figure 9.31 Equivalent "full" syntax.

7. The text version also works for nongraphical things such as attributes and inline declarations such as x: Sorted_List. UML would treat this as an uninterpreted string.

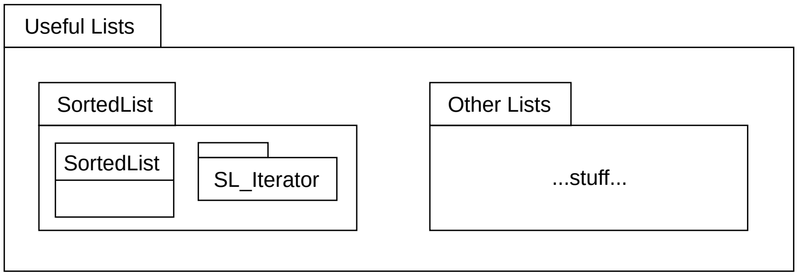


Figure 9.32 Templates and nested packages.

9.6.3 Using Substitution as Parameterization

You can substitute any name when you do an import—not only types and attributes but also variables, constants, and more. A substitution can be used to parameterize a spec. For example, if an integer constant MAX_SIZE is used within SortedList but is not set to any value, it can be substituted when SortedList is applied (see Figure 9.33). Figure 9.34 shows the graphical version.

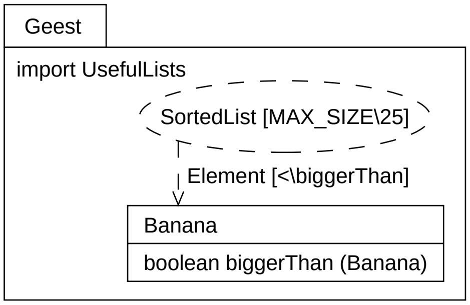


Figure 9.33 Substituting "values": textual version.

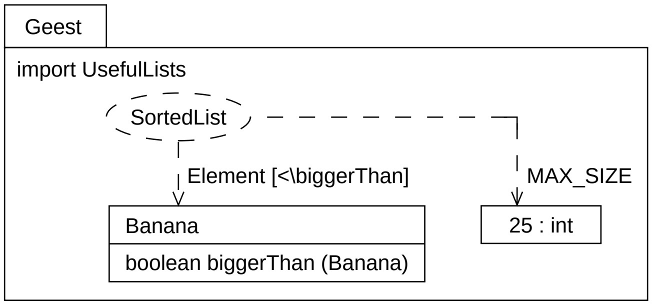


Figure 9.34 Substituting "values": graphical version.

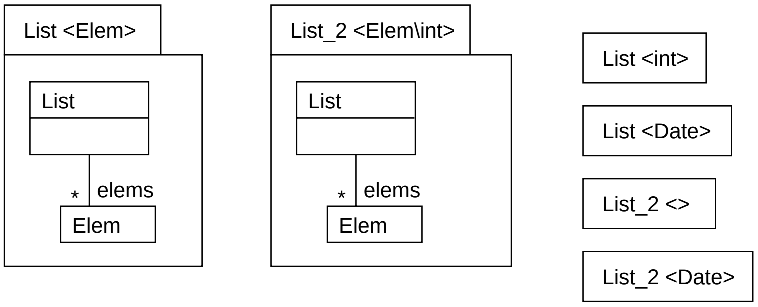


Figure 9.35 Explicit template parameters and parameter substitution.

9.6.4 Explicit Template Parameters

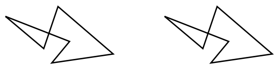
In general, we do not constrain the types that can be substituted, because we sometimes substitute just to avoid name clashes between multiple imports (see Figure 9.35). The "<..>" markers simply suggest places for substitution; but unmarked types can be substituted, and marked types can be left unsubstituted. Any substitutable type that isn't substituted when imported remains as a substitutable type in the importer.

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 also can be given default arguments.

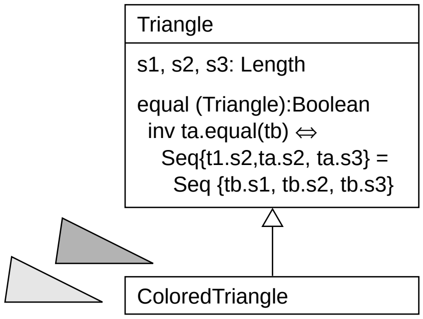
9.7 *Templates for Equality and Copying*

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 that is domain-independent. This section defines what they mean and provides standard template packages to use. These standard template packages can be used to define standard copy and equality as well as for features such as replication and caching.

9.7.1 Type-defined Equality



Are these two shapes equal? Some people might say yes, because all their lengths and angles are the same; others might say no; they are in different positions and hence different. It all depends on what exactly you mean by *equal*; sometimes there are different degrees and kinds of equality (such as "congruent" and "similar").



So whereas object identity is treated as an intrinsic property, equality must 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.

Equality is relative to type. Suppose that 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?

People who are aware of the type Triangle but not ColoredTriangle would say yes. For their purposes, the two Triangles are equal; they never ask about a Triangle's color. The fewer differences you're interested in, the more things look the same; The more you know, the more differences you can discern.

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 all of them. Equality on colored triangles could further discriminate on color. But the supertype has stated that as long as the sides are the same, two triangles are equal.

So the equality definition typically cannot simply be inherited, and we must do one or both of the following.

- 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 through 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) ⇒ ta.s1 = tb.s1 & ta.s2 = tb.s2 & ta.s3 = tb.s3

The same considerations apply to almost all comparison relations between members of the same type (for example, ≤ and ≥). An "equality-like" relation is one that conforms to the template shown in Figure 9.36. This principle can be applied to Triangles in a variety of ways (see Figure 9.37).

© ***equality*** A generic relation on a type. The relation must satisfy certain mathematical properties. Defined as a standard framework.

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) = --'self' and 's' are equivalent
    -- if there is some vector, offset (conceptually, the difference in their positions) for which
```

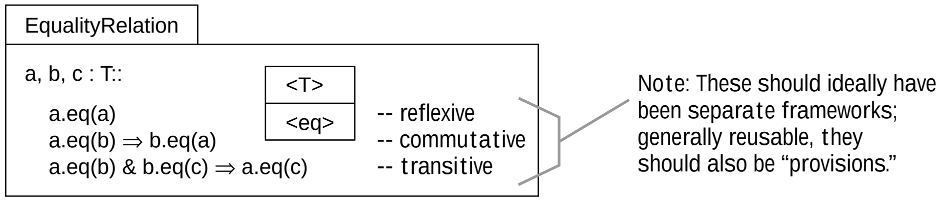

EqualityRelation
a, b, c : T::
a.eq(a)
a.eq(b) ⇒ b.eq(a)
a.eq(b) & b.eq(c) ⇒ a.eq(c)
<T>
<eq>
-- reflexive
-- commutative
-- transitive
Note: These should ideally have been separate frameworks; generally reusable, they should also be "provisions."

Figure 9.36 Template for equality relations.

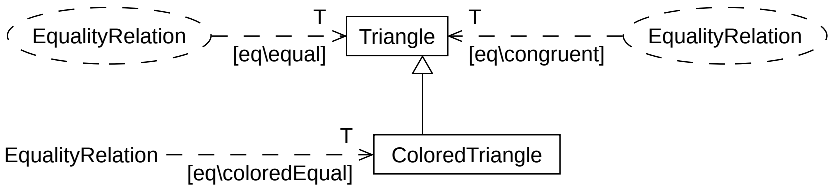
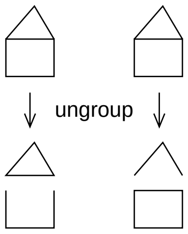


Figure 9.37 Defining different equality relations.

```
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)))
```

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



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 and so on. It's also possible to ungroup a Group, restoring the individual parts. This means that the two examples shown here before ungrouping, although equivalent by the preceding definition (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, if you stretch it horizontally and stretch a true square horizontally, very different things happen.

Here, equality-like relations need to be considered on a per-type basis; they should take into account dynamic and mutative behavior.

9.7.2 Copying an Object

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

You can use the template in Figure 9.38 to conveniently define a copy operation provided that your chosen comparison operator is a valid equality relation.

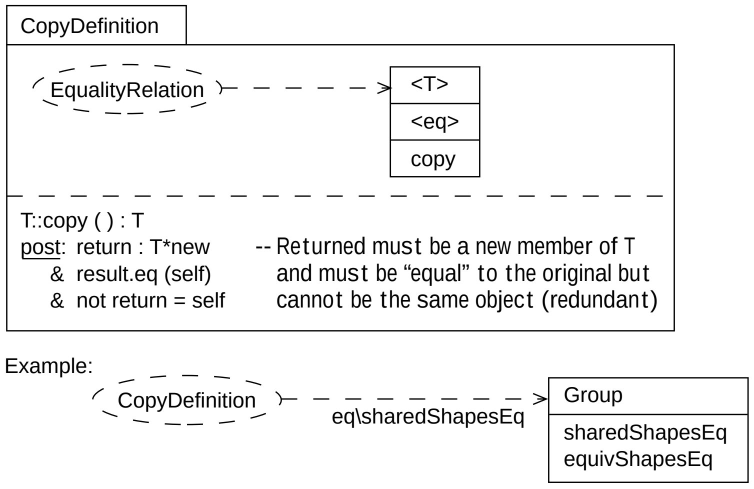

CopyDefinition
EqualityRelation
<T>
<eq>
copy
T::copy ( ) : T
post: return : T*new -- Returned must be a new member of T
& result.eq (self) and must be "equal" to the original but
& not return = self cannot be the same object (redundant)
Example:
CopyDefinition
eq\sharedShapesEq
Group
sharedShapesEq
equivShapesEq

Figure 9.38 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 that they share the same shapes) and equivShapesEq (the shapes themselves need not be shared but must be equivalent by some definition of equivalent).

In contrast to equality, the name of the operation doesn't need to be changed across subtypes for copy. If you know only that 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, because 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.

9.8 *Package Semantics*

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, both designers have the same intent except for the specific domain they work in; the same thing is true for using subtype arrows, a state transition, or superstates. And if they put the same stereotype on two elements, they mean the same thing (presumably).

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 them will have convenient syntactical forms, such as those UML provides. This section describes how new notations and semantic extensions can be defined precisely in Catalysis.

9.8.1 Interpreting Package Contents

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 in 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 diagrams can't express everything you want to say, so for details such as invariants and postconditions, we usually resort to text.[8] 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, and importing means only that you are including the statements from one package within another.

It's possible to write a precise set of rules (that is, a program) for converting each diagram element into text. And given any complex piece of text—such as an action specification, with its pre- and postconditions and odd constructs such as @pre and so on—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.

Books such as this one, which 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, although several projects are under way. Still, there is a wide range in how precisely their various visual notations are understood and how much reinterpretation 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, and that is why it would be nice to get an agreement on one of them.

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

9.8.2 Stereotypes and Dialects

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 invent only so many variants of boxes and lines and round things; if there are too many of them, newcomers soon despair of remembering what they mean.

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

8. Although Stuart Kent has shown how to move some of these assertions into the pictorial domain.

Stereotypes can be used on an individual model element as an alternative syntax to apply a framework (similar to Section 9.6.2, Template as Generic Types and Classes). The shorthand rules are as follows.

- If a type has the same name as its package, then using that name as a stereotype on a target type means to 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 to import the package, substituting the target attribute and its source type. It works similarly for other elements.
- Just as with other template shortcuts, stereotype application can use additional explicit substitutions: «name[x\a, y\b]». Or you can provide parameters defined on the template: «name⟨a,b⟩» (see Figure 9.39).

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 default set is a called a *dialect*. To specify which dialect a package should use, quote the dialect in the package tab. Naturally, the use of consistent dialects will simplify things; but if the dialects have a common underlying translation (see

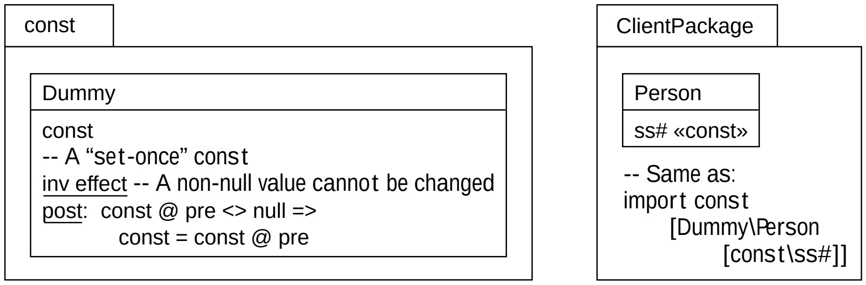


Figure 9.39 Defining the meaning of stereotypes using templates.

Section 9.8.3), you can even use custom dialects best suited to each portion of the problem (see Figure 9.40).

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 a pet variant on the basic ideas; all of them 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 neither possible nor appropriate to make it entirely fixed at the present time.

© ***stereotype*** A shorthand syntax for applying a framework; a stereotype is used by referring to its name as «name» attached to any model element. Frameworks provide an extensibility mechanism for the modeling language; stereotypes provide a syntax for using this mechanism.

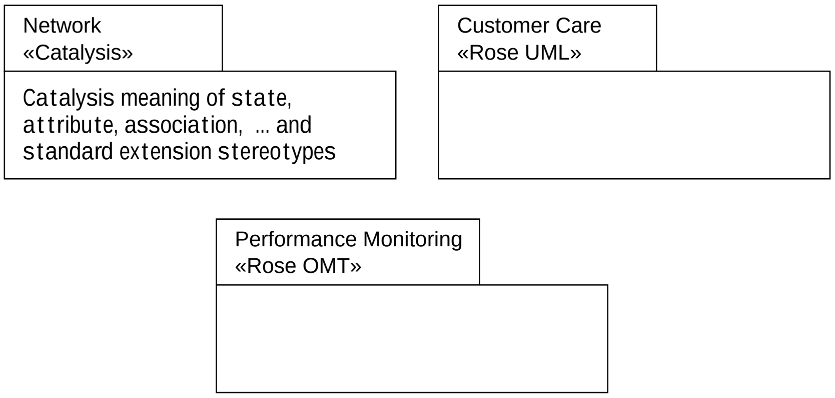

Network
«Catalysis»
Catalysis meaning of state, attribute, association, ... and standard extension stereotypes
Customer Care
«Rose UML»
Performance Monitoring
«Rose OMT»

Figure 9.40 Dialects of stereotypes and other notations in packages.

© ***dialect*** A package that contains a useful and agreed set of mutually consistent stereotypes, together defining a particular dialect of the modeling language. All modeling work is done in the context of selected dialect(s).

9.8.3 Examples of Semantic Rules for a Dialect

To close full circle, as perhaps you may have guessed by now, dialects and semantic rules themselves are defined within packages, although, as we've said, consider them virtual until further notice. But here is a short example to show the idea.

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 that contains an explicit stereotype means the same as *inverse attributes* (see Figure 9.41). So what should an association line mean if it has no stereotype tag? To define a default, you identify the untagged feature with the appropriate tag[9] (see Figure 9.42).

Figure 9.43 on the next page shows some equivalent ways to define an association. The top part (a) uses the highest-level notation: a line. In (b), you see the pattern notation for applying a template; in (c), a straight textual form, and in (d), the expanded result of any of the previous forms.

9.9 *Down to Basics with Templates*

We have seen that templates can be used to define domain-specific patterns, providing a higher-level notation for describing problems. The same templates can be used to define the modeling language itself, down to its formal basics.

9. A template could also introduce customized textual and graphical syntax for the application of that template; we do not generalize to such visual grammars in this book.

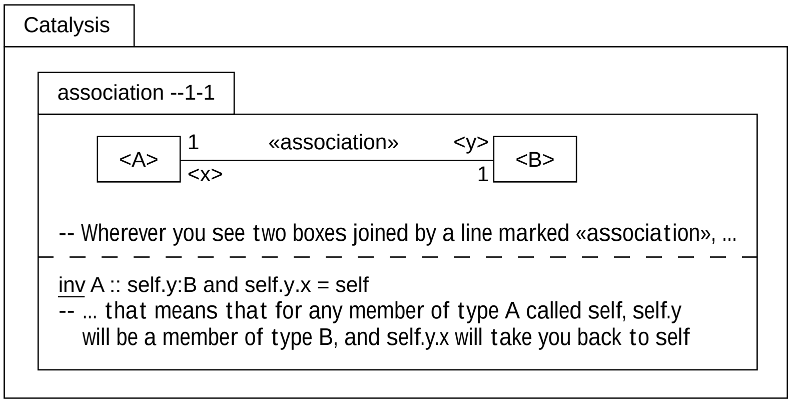

Catalysis
association --1-1
<A>
1
«association»
<y>
<B>
<x>
1
-- Wherever you see two boxes joined by a line marked «association», ...
inv A :: self.y:B and self.y.x = self
-- ... that means that for any member of type A called self, self.y will be a member of type B, and self.y.x will take you back to self

Figure 9.41 Template for what an association is.

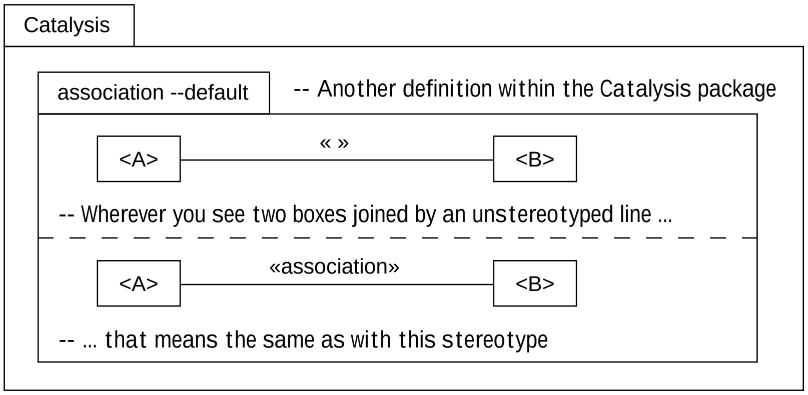

Catalysis
association --default
-- Another definition within the Catalysis package
<A>
« »
<B>
-- Wherever you see two boxes joined by an unstereotyped line ...
<A>
«association»
<B>
-- ... that means the same as with this stereotype

Figure 9.42 Template defining the meaning of default notation.

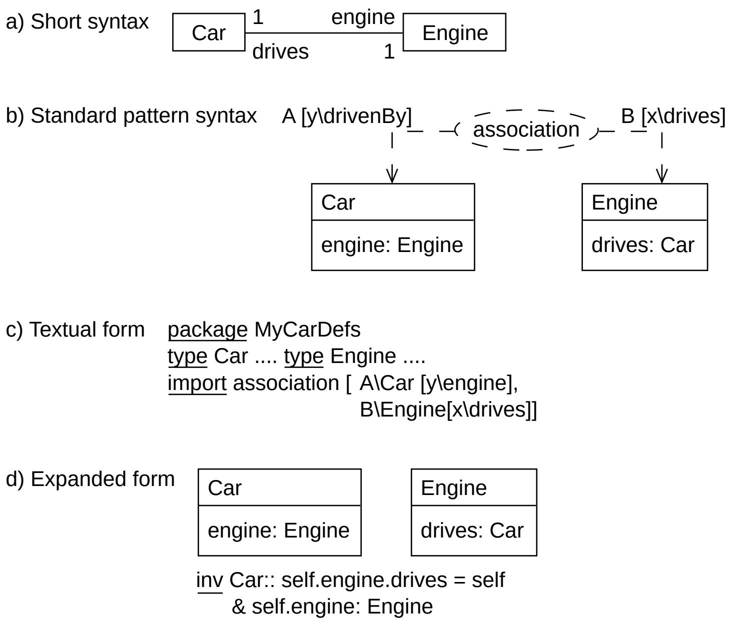
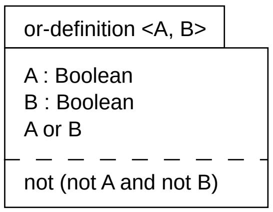


Figure 9.43 Interpreting association via template definitions.

9.9.1 Template Packages to Represent Inference Rules



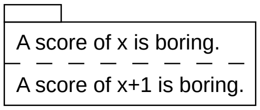
Templates can be used to represent general facts that are useful in understanding or reasoning about types. They can be presented as diagrams or as Boolean expressions. For example, at the very basic level, we can write things such as or-definition. It means that, if you happen to find an expression 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. Figure 9.44 shows a few others.

It is sometimes useful, with these kinds of rules, to use placeholders that themselves take arguments (see Figure 9.45). This may sound mind-bending at first, but this is as bad as it gets. We're now outside the realm of practical daily application for most software developers. But briefly, in case you're interested, the induction rule can be used to verify statements involving a progression. For example, here's how to prove that all cricket scores are boring.

- A score of zero is clearly boring, because 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, and clearly that is incrementally boring. So if x was boring, then x+1 definitely also will be boring. We can write this as



an inference of which we have satisfied ourselves; we don't need to give it a name, because we won't be needing it for long. Clearly, cricket is boring.

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

    200 : Integer and 200 >= 0

- 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.

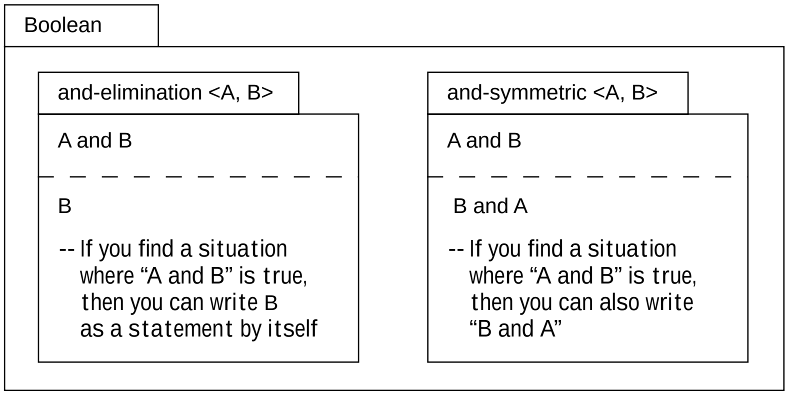

Boolean
and-elimination <A, B>
A and B
B
-- If you find a situation where "A and B" is true, then you can write B as a statement by itself
and-symmetric <A, B>
A and B
B and A
-- If you find a situation where "A and B" is true, then you can also write "B and A"

Figure 9.44 Templates for typical inference rules.

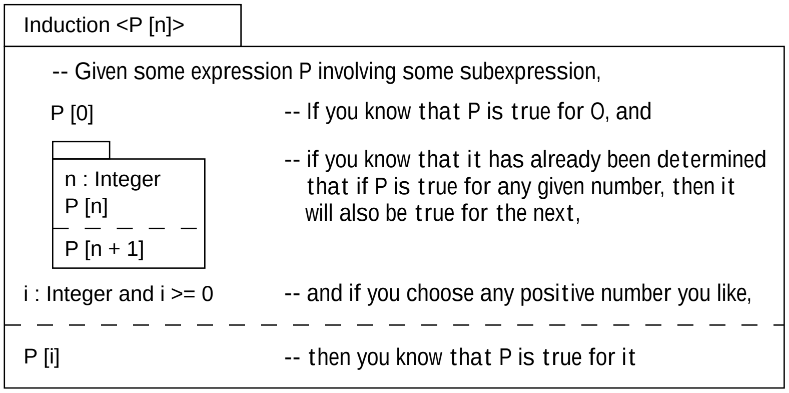

Induction <P [n]>
-- Given some expression P involving some subexpression,
P [0]
-- If you know that P is true for 0, and
n : Integer
P [n]
P [n + 1]
-- if you know that it has already been determined that if P is true for any given number, then it will also be true for the next,
i : Integer and i >= 0
-- and if you choose any positive number you like,
P [i]
-- then you know that P is true for it

Figure 9.45 Induction template.

9.9.2 Template Packages for Primitive Types

The primitive types that we use in every model and design—Boolean, arithmetic, sets, lists, and dictionaries—can be defined in basic packages that are 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 the types, from which other facts follow. For example, the package defining Boolean operators contains the propositions shown in Figure 9.46.

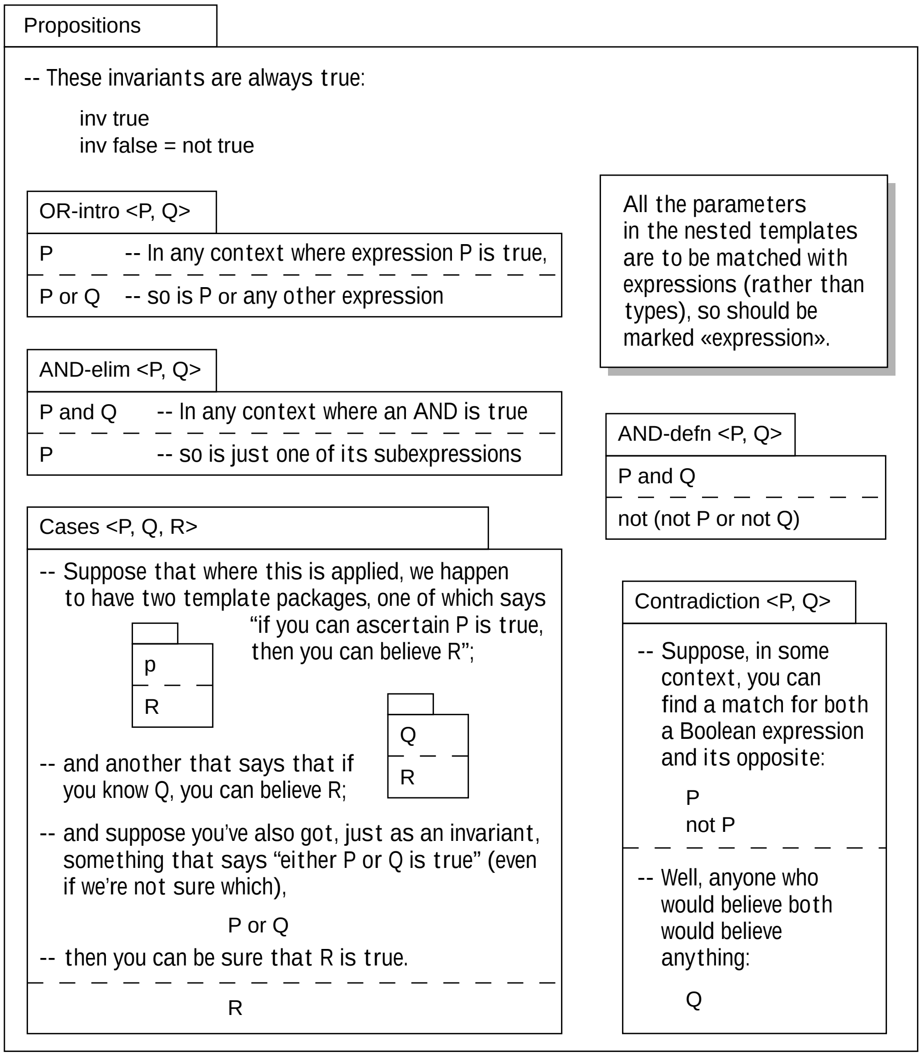

Propositions
-- These invariants are always true:
inv true
inv false = not true
OR-intro <P, Q>
P -- In any context where expression P is true,
P or Q -- so is P or any other expression
AND-elim <P, Q>
P and Q -- In any context where an AND is true
P -- so is just one of its subexpressions
Cases <P, Q, R>
-- Suppose that where this is applied, we happen to have two template packages, one of which says "if you can ascertain P is true, then you can believe R";
p
R
-- and another that says that if you know Q, you can believe R;
Q
R
-- and suppose you've also got, just as an invariant, something that says "either P or Q is true" (even if we're not sure which),
P or Q
-- then you can be sure that R is true.
R
All the parameters in the nested templates are to be matched with expressions (rather than types), so should be marked «expression».
AND-defn <P, Q>
P and Q
not (not P or not Q)
Contradiction <P, Q>
-- Suppose, in some context, you can find a match for both a Boolean expression and its opposite:
P
not P
-- Well, anyone who would believe both would believe anything:
Q

Figure 9.46 Propositions: a packaging of Boolean operators.

We can create another package that imports this one, defining Predicate Logic, which gives the meaning of the quantifiers "for all x, [some 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 collections (lists and dictionaries or maps) can also be defined with the help of sets and predicates.

9.9.3 Layered Semantics

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, and so on) imposed by the type definition.

In this fashion, we can build up a hierarchy of basic types and operators—not only the syntactic definitions but also their meanings! And because the basics can be different for different modeling and programming languages—for example, not all have exactly the same idea of the passage of time—different packages can be supplied for users of different dialects and can be referenced as stereotypes.

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, and choose the Java package to be able to check that your code matches your spec. A typical hierarchy for modeling is shown in Figure 9.47. But for most users, it is not necessary to know about the details of these basics, any more than you bother with the formal semantics of your programming language. Still, it is nice to know that this foundation can be made explicit and that the details can be made a matter of choice.

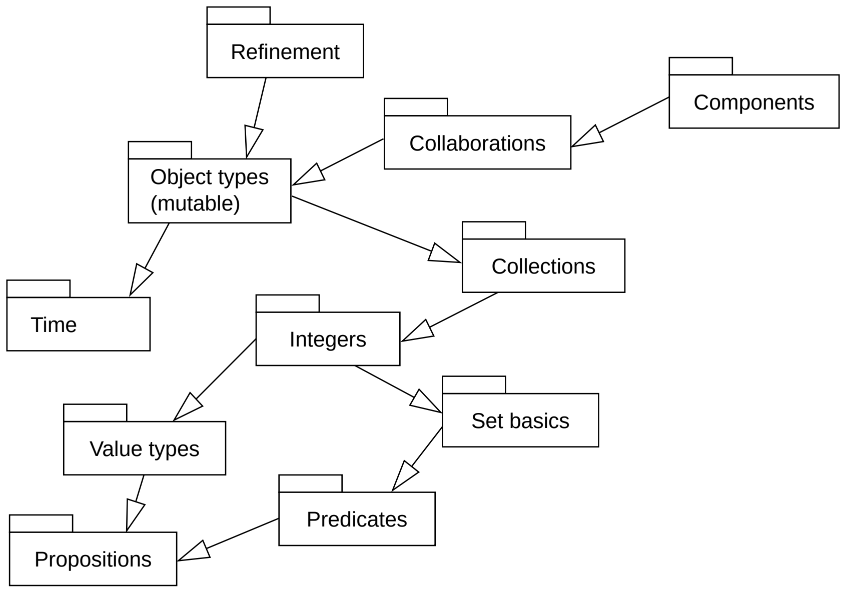


Figure 9.47 No primitives: full layering via packages and templates.

And, of course, most of these packages—especially the complex semantics ones—are virtual at present. But some research projects have indeed built up the packages of primitives. (The example given here comes from [Mural91].)

One interesting feature of these packages is that they define many "if you already know that this fact is true, then you can also assume that" rules. These packages talk about the fundamental properties of the expressions we can write: In the arithmetic package, for example, there is something that tells us that x+y is the same as y+x, a likely fact to use in programming. In the extreme case of safety-critical systems, designers can use these rules (and similar ones dealing with programming language statements) to check that their programs fulfill robustness criteria and indeed meet their specs. Designers can also build higher-level rules around them. The rules are checked once and then institutionalized as templates.

For the writers of support tools, these basic packages are a way of discussing and defining the exact details of the languages they support. This should have the benefit of making the languages more interoperable and should allow these writers 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.

9.9.4 Standard Packages

We have seen that nested packages can be used to define a related set of stereotypes, such as those needed for a particular method or language (see Figure 9.48). 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.

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.

9.10 *Summary of Model Framework Concepts*

A model framework is a generic package containing both normal and placeholder definitions. A placeholder is a name that 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.)

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

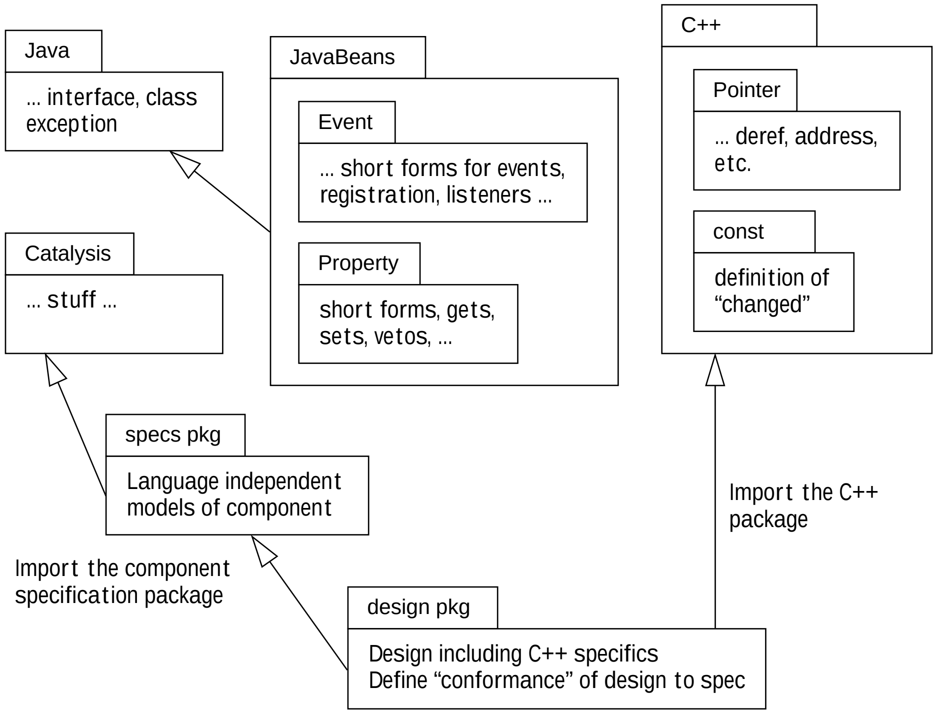

Java
... interface, class exception
JavaBeans
Event
... short forms for events, registration, listeners ...
Property
short forms, gets, sets, vetos, ...
C++
Pointer
... deref, address, etc.
const
definition of "changed"
Catalysis
... stuff ...
specs pkg
Language independent models of component
Import the component specification package
Import the C++ package
design pkg
Design including C++ specifics
Define "conformance" of design to spec

Figure 9.48 What can be achieved with standard packages.

9.10.1 Composition of Definitions

Recall that every model can be regarded as a list of individual statements: All the pictures can 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.

Figure 9.49 summarizes the model framework concepts.

9.10.2 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.

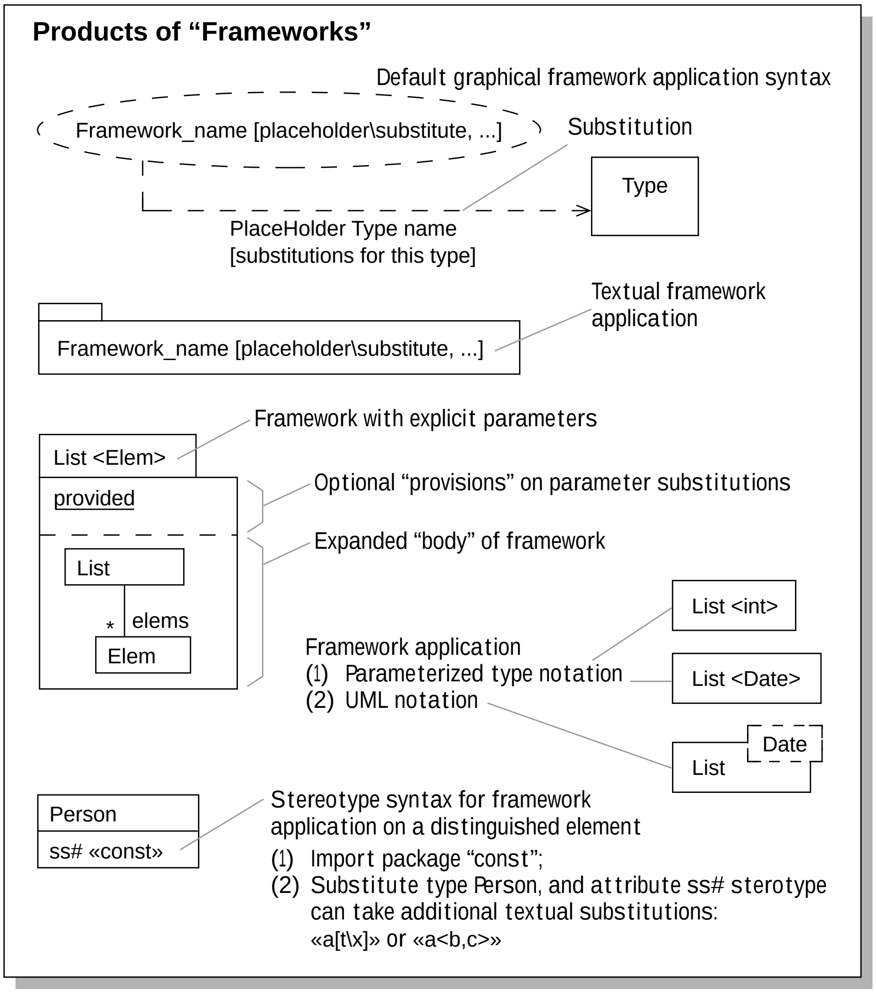

Products of “Frameworks”
Default graphical framework application syntax
Framework_name [placeholder\substitute, ...]
Substitution
Type
PlaceHolder Type name
[substitutions for this type]
Textual framework application
Framework_name [placeholder\substitute, ...]
Framework with explicit parameters
List <Elem>
provided
Optional “provisions” on parameter substitutions
Expanded “body” of framework
List
* elems
Elem
Framework application
(1) Parameterized type notation
(2) UML notation
List <int>
List <Date>
List
Date
Person
ss# «const»
Stereotype syntax for framework application on a distinguished element
(1) Import package “const”;
(2) Substitute type Person, and attribute ss# sterotype can take additional textual substitutions: «a[t\x]» or «a<b,c>»

Figure 9.49 Frameworks: patterns, generics, and stereotypes.