Chapter 9 Composing Models and Specs

Combining modeling elements packages and structuring them

Outline

Why composition rules are important

When building models and specifications, it is very important to be able to compose them with a clearly defined and intuitive meaning. This makes re-use and learning easier, since the whole can understood by understanding the parts, and then re-combined in a predictable way. All descriptions in Catalysis can be composed, from attributes and actions on a type, to entire packages.

A package is always built upon others that it imports; they provide the definitions of more basic concepts that it uses — all the way down to utter primitives like numbers and booleans.

A package augments and extends the material it imports. It may import several packages, so that their different definitions are combined. In many cases, the different sources of material will deal with some of the same things.

There must therefore be clear rules whereby the definitions that can be found within a package are ‘joined’ with others.

This chapter deals primarily with how to use packages in the building of others, and how to interpret the resulting compositions; and discusses some nuances of specifying and composing specifications of operations in the presence of exceptions.

9.1 *Sticking pieces together*

Every method of development must be good at building its artefacts — models, designs, plans, or whatever — from smaller pieces. This is because (a) we can only get our heads around a small chunk at a time, andcan only build big things by sticking small ones together; (b) parts are more likely to be reuseable if they can be stuck together in different ways.

Much of the rest of this book is about building from parts. We'll discuss building models from frameworks, and building software from components. The software has its own intricate 'plugging' mechanisms; this chapter is about the much simpler matter of combining models, as used in type specifications and collaborations. What we are putting together here is specifications and high-level designs — so this is a design activity, rather than a systems integration one.

There are three specific situations for composing models:

- The documentation chapter (Chapter 6, p.237) says that we can present a large model as a series of smaller diagrams. That's great for a guided tour through the model; but an implementor needs to see everything relevant to each action or type. So we need some exact rules by which the diagrams recombine to make the big picture.
- The packages chapter (Chapter 8, p.331) talks about one package importing others. We need some rules governing how to combine the definitions from the different sources. (The frameworks chapter (Chapter 10, p.389) takes this idea even further and combines generic models.)
- Our components (Chapter 11, p.437) can have multiple interfaces: that is, they must satisfy the expectations of several different clients, who might or might not know about each other. Each interface is defined by a type; therefore we need a way of working out what it means to satisfy two or more types at the same time.

This chapter deals with the basic mechanisms of combining the pieces, used in different ways by all of these.

9.2 *Joining and subtyping*

In Catalysis, we use two different ways of composing specifications:

Joining — combines two views of one type; each view may impose its own restrictions on what the designer of the type is expected to achieve.

Joining happens when two views of the same type are presented in different diagrams — they might be in the same package or imported from different ones.

Joining is about building the text and drawings of a specification from various partial descriptions.

Type intersection — combines two specifications, each of which must be observed by an object that belongs to the type.

Subtyping happens when a component or object must satisfy different clients. Each specification must be satisfied independently of the other.

A subclass should generally conform to any type(s) its superclasses claim to implement (except C++ 'private' subclasses).

A type specification defines a set of instances — the objects that satisfy that spec. Subtyping is about forming the intersection of two sets: those objects that happen to satisfy both specifications.

We know that the behavior of anything from a very simple object to a large distributed system can be specified with a type definition: which has actions specified in terms of a model.

The rules for Joining and Subtyping can be summarised as operations that you perform on the model:

Join

1. Collect the attributes and associations from both views into one picture.

2. AND invariants from the two views.

3. Collect the action specs from both views into one list.
For any action that has specs from both views:

4. AND any preconditions

5. AND the postconditions

Type intersection

1. In each type specification, AND its invariants with each precondition in each action-spec; and also with each postcondition.

2. Form a model with the attributes from both specs, and with actions from both specs.

3. For any action that is defined in both specs, rewrite its spec as
pre pre1 or pre2 post (pre1=>post1) and (pre2=>post2)
(using the versions after ANDing with invariants).

This chapter focuses mostly on the Joining operation.

9.3 *Type intersection — combining views*

Type intersection means conforming to the expectations of more than one client, each of whom has a self-contained type specification that they are expecting you to conform to. Any system that has more than one class of user has this problem — each of them has their own view.

In software, the pluggable components that we want to build need the same ability.

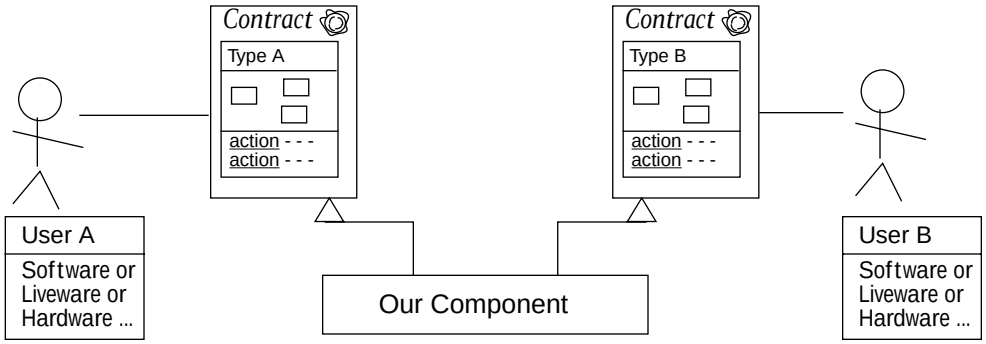


Figure 175: Component must satisfy multiple views

From User A's point of view, no matter what else Our Component does, it must always conform to the expectations set by Type specification A; and similarly for B.

Supposing we want to design such a component, how do we go about it? In the most basic form of OO design, we use our type specifications as the basis for the models. But here there are two; and two sets of actions.

Here are the basic steps for combining small types whose models are a few attributes each:

1. Combine the two lists of attributes.

- If any attributes from the two types have the same name, determine whether it's accidental, or because the name is inherited from a common supertype. If the former, rename one of them — it won't make any difference to the meaning of the type; but if inherited, then they really do mean the same thing.

2. Merge the two lists of operations. If the actions are all of different names, that's easy.

   For any operation that has specifications from both types:

- Take the precondition from each type and AND it with the invariant from the same type;
- Take the postcondition from each type and AND it with the invariant from the same type;
- Write the new operation specification as:

  pre pre1 or pre2
  post (pre1=>post1) and (pre2=>post2)

Why do the invariants have to be absorbed into the action specs? Because it's the pre/postconditions that are the real behavioral spec: an invariant is just a way of factoring out common assumptions made by all those within its own context. Outside that context, the same common assumptions might not apply, so we have to make them specific in any pre/posts we want to move out of the context. Once all the actions have been combined, it will probably be possible to factor out a common invariant that occurs in all the combined operation specs.

Combining larger types. If we are talking about two three-hundred-page documents, this may take a little longer. This is covered properly in the components chapter (Chapter 11, p.437), but the essence is the same on a larger scale:

- Build a new model sufficient to include the state information from both types; verify that this is so (and assist the testing team), by writing two sets of abstraction functions (§7.4, p.290) that will retrieve any piece of information from your component back into the language of each type.
- Then build your component with façades, each of which is dedicated to dealing with just one of these clients (§7.7, p.315), and supplies the actions it expects, translated from your component's internal model.

9.4 *Combining packages and their definitions*

Combining importees and importer

So far, we've rather glibly talked about the way definitions are augmented and combined when packages are imported. This section looks in more detail at how the materials from imported packages are combined with each other, and with the additional material already in the importer.

9.4.1 Definitions and joins

Multiple appearances...

What exactly does a type box in a diagram mean? What does it mean if boxes claiming to represent the same type appear in different diagrams in a package? Or are imported from different packages into one?

... within the same and imported packages...

Within one package, boxes headed with the same type name may appear in different parts of the same diagram, and in different diagrams in the package, and in textual statements in the Dictionary or the documentary narrative. Some of the diagrams in the (unfolded) package may have been imported from other packages, but that makes no difference. Whatever the source of the multiple appearances, it is always possible to take them all and join them into a single type definition.

... observe same join rules

The rules are the same for all kinds of multiple appearance, whether multiple diagrams in one package, or definitions from multiple packages. The operation of combining them is called 'joining'. Frequently, the individual splinters don't denote any type, taken on their own: it's only when you put them together that they define a set of objects characterised by a particular behavior.

Different kinds of definition that you can find in a package each have their own joining rules. This section describes the rules for each kind.

9.4.2 Joining packages

Join packages by joining all their contents

The join of two packages is formed by forming a bag of all the statements and definitions from both packages, and then joining those definitions with the same names (after any renaming, as discussed in Section 9.6, "Name mapping on import," on page 380). This is applied to any nested packages, along with everything else.

A package's full unfolded meaning is got by joining its imports to its own contents. There are specific rules for joining different kinds of definitions.

Type specifications are joined by joining their static models and their action specs.

9.4.3 Joining static models

Join a type

To join multiple appearances of a type into a single type definition:

and invariants

- and all invariants. The type has an effective invariant which is the conjunction of all the separate ones.

- Take all the attribute names from the appearances and put them into a set; the completed definition should have the same set of attribute names. Include association names and parameterised attributes in the same way.

- For each attribute (or association) name in an appearance, consider any type constraint to be an invariant. So "count:Integer" just means "for any object x of this type, x.count always belongs to the type Integer". The completed type should contain an invariant that ands these together. So: Combine attribute definitions

  count : Integer and count : Number => count : Integer
  -- all Integers are Numbers anyway
  thing : Boolean and thing : Elephant
  — contradiction – can't join these definitions, unsatisfiable spec
  connection : Trasmitter and connection : Receiver
  => connection:Transceiver — which is a subtype of both of these types

- If the attribute has parameters (similarly, if the association has qualifiers), you can just write the attributes as overloaded functions, provided the parameter types don't overlap. So price (CandyBar):¢ and price(FuelGrade):$ can remain as two attributes, since there is no CandyBar that is also a FuelGrade (yet). Include parameterized attributes

  The general rule is that you first convert the attribute types into invariants of the form:

  (param1 : ParamType1 and param2 : ParamType2) => attribute : ResultType

  Anding expressions like this together gives a result that says "if you start with parameters like these, you get this kind of result; if you start with parameters like those, you get that kind". If an argument ever belongs to both parameter types, then the result should belong to both result types.

  (All of which conforms to the usual contravariant rules.)

9.4.4 Joining action specifications

Action specifications are joined according to a covariant rule, that permits any appearance of a type to reinforce preconditions and invariants. The rules apply both to operations or actions localized to particular types, and joint actions. Join actions specs by separating anding pre / post

Treat actions with different signatures (names and parameter lists) separately.

For each action signature, take the individual pre and postconditions, and:

- and the preconditions;
- and the postconditions;
- and the rely conditions;
- and the guarantee conditions.

ANDing preconditions means that, in different packages or diagrams (or in different parts of your narrative) you can use preconditions to deal with different restrictions, confident that these restrictions will apply regardless of what is analyses in other packages. Under Fault Management, we can say that a call can be made only if the Each spec imposes a restrictive precondition

line is not under maintenance; under Billing, we can say that a call can be made only if the Account associated with the Line is not in default. Each package has no comprehension of the other's constraints.

...and a guaranteed consequence

ANDing postconditions means that, in different packages, you can use postconditions to deal with different consequences of an action: under Billing, you can say that a charge is added to the associated Account; under Fault Management, we can say that the count of successful calls is incremented.

Similarly concurrency: rely and guarantee

ANDing rely conditions means that in different packages, you can define different invariants the designer should be able to rely on while the action is in progress; ANDing guarantee conditions allows you do separate invariants your action will preserve.

Example

Thus, the two separate specs could be:

```
action Agent::sell_life_insurance ()        action Agent::sell_life_insurance ()
pre:    c.isAcceptableRisk                  pre: self.territory->includes (c.home)
post:   c.isInsured                         post: territory statistics updated
```

The combined spec as the result of joining the two would be:

```
action Agent::sell_life_insurance (c: Customer)
pre:    c.isAcceptableRisk and self.territory->includes(c.home)
post:   c.isInsured and territory statistics updated
```

Advanced Topic

4.4.1 Two styles of writing and composing action specs

Rationale for the join rule

The reason for using these join rules for actions is that each action specification could have been written without knowledge of the other specification, or of its attributes. When I write my preconditions I do not know what other preconditions you may want to impose on that action; we both should be confident that, when our separate specifications are joined together, each can rely on its restrictions still being in force. Similarly for postconditions.

Sometimes you want conditional guarantee, not restriction

Sometimes, however, you want to write a specification that makes guarantees for certain cases, where those cases may overlap with others. In this case you can use an alternate style for writing the spec: do not use an explicit precondition, but describe the case within the postcondition itself. Using the same composition rules:

Omit the explicit precondition

```
action Stack::push (in x, out error: Boolean)
post: self.notFull@pre => self.top = x and error = false
        -- provided I was not full beforehand, x will now be my top element
```

This spec tells us what happens in the preferable case, and might on its own, be all we need. But perhaps, in another part of your spec, you want to write down what would happen in the other case:

```
action Stack::push (in x, out error: Boolean)
post: self.full@pre => error = true
        -- provided I was full beforehand, you will get an error flag.
        -- I'm not telling you what might happen to my contents!
```

The joined spec would make one guarantee for one case, and another guarantee for the second case (the two cases happen to be disjoint in this example).

The effect of the join is now different

```
action Stack::push (in x, out error: Boolean)
post:       (self.notFull@pre => self.top = x and error = false)
    and     (self.full@pre => error = true)
```

It is possible to compute a resultant precondition from this specification:

```
action Stack::push (in x, out error: Boolean)
pre:    self.notFull or self.full
post:       self.notFull@pre => self.top = x and error = false
    and     self.full@pre => error = true
```

Hence, these two different styles of writing specs can be used to accomplish these two different goals of composing separate specifications. Some more details on dealing with exception conditions in specifications appears in Section 9.7, “Action Exceptions and Composing Specs,” on page 382.

9.4.5 Joining type specifications is not subtyping

Advanced Topic

A type specification denotes a set of objects — all objects that meet that specification are members of that type. Some ways of combining type specifications correspond directly to operating on the corresponding sets of objects; ’join’ does not.

Types are sets of objects denoted by a spec

When you join type specifications, you are combining the descriptions themselves, not directly the types (sets of objects) they specify. In the usual case two type specifications are joined when a package, P1, imports two other packages, P2, P3, which both provide separate specifications (T_{s1} and T_{s2}) for the *same* type, T. Within package P1, the resulting specification of type T is the specification that results from a join:

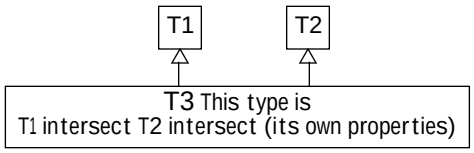
A join combines two type specs, not two types

T_{s1} join T_{s2}

In contrast, when you define a subtype, you are defining a subset of objects; when you combine multiple supertypes, you are intersecting the corresponding sets. The rule for intersecting action specs is quite different than the rules for ’join’. You can write a different expression,

Subtype is set intersection of objects

T1 * T2



which represents the type of objects that conform both to T1 and also to T2 — that is, the intersection of the two sets. Type intersection or subtyping is a “no surprises” combination. Anything you’re guaranteed by one spec can’t possibly be taken away by the other. It is what happens when you have multiple supertypes which both provide specs for the same action; or when you combine a supertype action spec with a corresponding spec in the subtype. If we had the following explicitly declared specs:

```
T1::m pre: A post: X
T2::m pre: B post: Y
T3::m pre: C post: Z
```

Each complete action spec is ANDed with the others

Then the resulting equivalent spec, after combining with the supertype specs, on the type T3 is obtained by ANDing all three pre / post *pairs*[1]:

T3::m (pre: A post: X) and (pre: B post: Y) and (pre: C post: Z)

Which is equivalent to:

T3::m pre: (A or B or C) post: (A => X and B => Y and C => Z)

Subtype means 'no surprises'

An implementation of the resulting operation spec is guaranteed to meet the expectation of anyone who expected either T1 or T2. An invocation of 'm' is valid anytime A is true (since that would make (A or B or C) true), and is guaranteed to result in X (and perhaps *also* Y, Z; depending on whether B or C were also true).

While 'join' can impose restrictions

When you join two type definitions, you are not usually intersecting the types, depending on how the action specs were written. For example, if I ask you for an object that conforms to Billing's idea of a Line, I would expect to be able to make calls whenever the Account is in order. If you give me some thing that conformed to Billing::Line join Fault_Management::Line, then I will find to my dismay that sometimes my Account is OK, but I still can't make calls.

In fact, many partial specifications that you find in models don't constitute a complete type specification at all — they have attributes but no actions. Strictly speaking, any object would satisfy such a type spec, because it states no behavioral requirements. Types that are only attributes (and associations) are only meaningful as part of the models of larger types.

9.4.6 Joining action implementations

How to 'join' implemented methods?

An action's specification can be implemented by designing a refinement into a smaller set of actions — ultimately, in software, messages. Program code is the most detailed kind of action implementation.

Tricky: AND of two sequential procedures?

Implementations cannot be joined in the same way as specifications. Firstly, it isn't clear what ANDing two programs together would mean; the machine has to follow one list of instructions or the other. Therefore your support tool should complain if you try to provide code for two operations with the same name in the same class; or two refinements of the same action into different sets of smaller steps.

Just select one implementation?

A second complication is that any implementation of the Call action provided by, say, the Billing package, isn't likely to satisfy the requirements specified by Fault Management — since neither world understands the concepts of the other. So we cannot always accept an imported implementation, even if there is no competing implementation from the other packages.

1. In a join, you separately AND the precondition; then the postcondition;

The only circumstance under which an implementation can be imported is when there is no difference between the pre/post specification in the importee and the corresponding specification in the unfolded importer — that is, when there is no extra material about this action from the other imports, and no extra material specified here. In that case, we know that the designer was working to the same spec. [1]

Only if it's spec = the combined spec

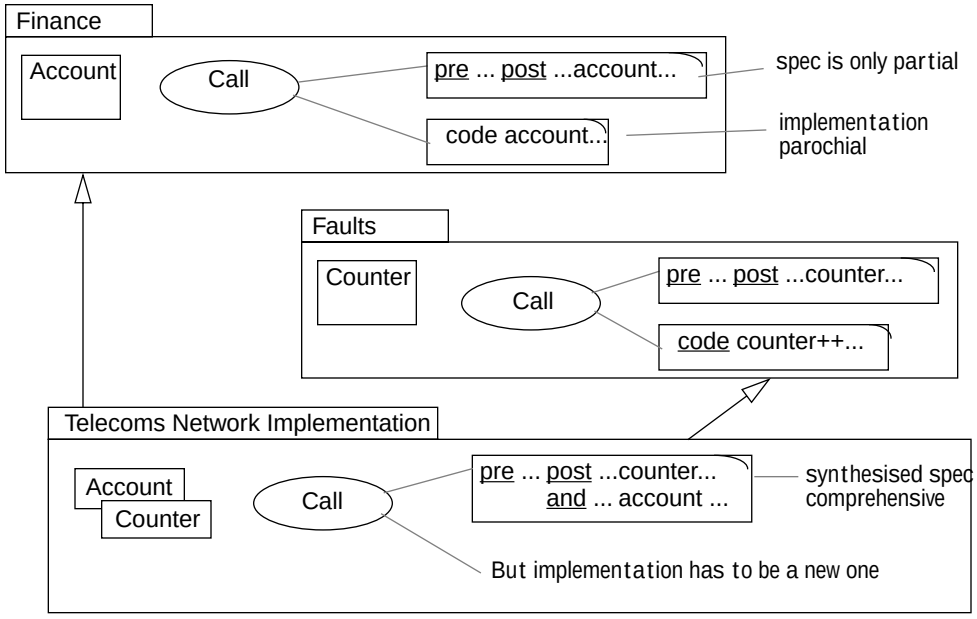


Figure 176: Importing and joining specs vs. code

Does this mean we cannot bring together code written in different packages? Of course we can, but they have to be routines that can be separately referred to; and as designer of the importing package, you have to design the implementation that invokes each of them in the right way. Some languages allow you to invoke super.method(); others permit Super1::method() and Super2::method().

Hence, designer must explicitly invoke to combine them

Each of the packages for the telecoms network is a *view* of the whole system, constructed from the point of view of one department or business function. Knowing this, the package designers should not presume to provide their own implementations of overall actions. Instead, they should provide auxiliary routines that help implement their concerns. So Faults could provide successfulCallLog and Finance could provide callCharge. The designers of Telecoms Network Implementation can then choose to invoke these where appropriate in their own implementation of Call.

For partial views, code methods as partial, auxiliary routines

1. The rules here are derived from [Wills 92], in which it was allowed to inherit an implementation if you could also document a proof that the existing implementation would meet the combined spec.

9.4.7 Joining classes

A class is an implementation template

A class defines an implementation or partial implementation of an object, with program code for localized actions, and variables for storage of its state. A class can also be documented with invariants over its variables, and pre / post specifications for each operation signature. Some programming languages support these features — notably Eiffel, which provides a testing facility that uses them.

Join of classes is

The idea of joining class definitions isn't something you find in a programming language: by the time you get to compilation, it's assumed you've chosen your program code; no need to automatically compose it with any other code.

...but quite useful

But some programming environments, such as Envy, do allow a class to be synthesized from partial definitions imported from different packages ("applications" in Envy). There are restrictions that help prevent ad hoc modification of the behavior of the instances. Among other things, this permits a very useful form of structuring development work: since an object typically plays multiple roles, and each role is meaningful in the context of interactions with objects playing other roles, the class is not the best unit of development work to assign to a person or team; instead, the collaboration between roles should be an implementation unit.

Other interesting work has been done in the area of *subject-oriented programming*, which strives to compose implementation classes that define different views, or roles, of some objects.

Catalysis rules:

The rules in Catalysis are:

- The joined class has all the variables in the partial definitions. Types must be the same. (If they were widened, the preconditions of some methods might fail, finding values in the variables they couldn't cope with; if they were narrowed, some methods might find they cannot store the values they need to.)
- The joined class has all the methods from the partial definitions. Methods are joined according to the rules for actions: only one method for each signature is allowed (within this class), and not even that if there is a pre / post spec attached to that signature in one of the other joinees.
  Pre / post specs may be inherited from superclasses.
- Invariants over class variables, and pre / postconditions attached to message signatures, are treated as for joining types: they are ANDed.

9.4.8 Joining narrative

Can this be usefully defined?

How shall we combine the narrative documentation of a package with those of the packages being imported? Perhaps the best that can be done automatically is to stick them end to end.

Perhaps, with hypertext

But decent support tools provide for a hypertext structure. Importing means that the reference hooks in the imported text are available for linking from the importer's text. Also, the importer's text can include references into the imported text.

Type names are qualified with package names

Does that make the type CustomerCare::LineCircuit different from Network::LineCircuit? In a sense, yes. A type is just a list of facts you know about an object, and so the CustomerCare version is more fully defined one. There are many implementations of the Network version that would not satisfy the more stringent requirements of the more developed one. So should we perhaps really give them different names? In Catalysis the answer is that "PackageName::TypeName" is really the full name of a type. But we use the same type name from one package to another, so that we can easily add on properties in each importing package, as we've done here.

(An alternative convention would be to disallow additional statements to be made about a type by importers, allowing only that subtypes can be defined. This would be needlessly restrictive, and make it very complex to define multiple views that can be re-combined. Also, it is difficult to avoid seemingly innocent invariants on a new type actually constraining an existing one: so the policy of using subtyping would be difficult to enforce in practice, since such constraints would break subtyping rules.)

Importing contradictions

What happens if you import two packages that impose contradictory constraints? For example, one package asserts that some attribute x has to be > 0, and another says x< 0. The result is just that you've modeled something that can't exist: something you can talk about but won't find any implementations of, like orange dollar notes, dry rain, or honest politicians. If it's a model of a requirement, you'll find out when you try to implement it. Moreover, you can construct empty models within a single package: whether importing is involved is irrelevant.

9.6 *Name mapping on import*

Name conflicts

But a complication arises where two packages are imported from different authors, who just happen to have used the same name for different things. BluePhone might find it useful to import a ready-made accountancy model; but what that package calls an Account (of our company with the bank) might be different to what the Billing department calls an Account (of one of our customers with us). In that case, we really don't want the two types to be confused.

Renaming

On the other hand, sometimes we do want types with the same name to be identified: maybe the accountancy package's idea of a Customer is consistent with ours. Or sometimes we want two differently named types to be identified: perhaps what we call a Customer, an imported package calls a Client.

Tools flag conflicts

The first defence against unwanted results is that your support tools should raise a flag when you import a package that contains name conflicts. This is probably not necessary where the first definition of the name can be traced back to a common ancestor package (as with the LineCircuit example).

Renaming

What do we do if we want the two definitions to be separate? In the importing package, one of them should be given a different name. We can show this on a diagram like this:

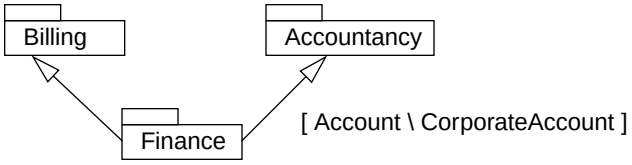


Figure 177: Import with re-naming

[old name \ new name]

The annotation " [X \ Y] " means "rename X to Y in the importing package (the tail end of the arrow)".[1] It doesn't mean there is any change in the original package from which the definition comes.

This means that in the unfolded Finance package, there is an Account type that means what it does in Billing (perhaps with some more information added in Finance); and a CorporateAccount type that means what Accountancy::Account did (again, perhaps Finance adds some more to it).

Tools and name conflicts

A tool should attach to each import a record of these mappings, at least for each name conflict (that does not arise from a common ancestor). In those cases where the designer opts to confirm that two types with the same name should be identified, that decision can be recorded with a mapping like [Customer\Customer]. An interactive tool will also distinguish between a name, and the string representation of that name.

1. Sorry about the orientation of the arrows. We're following UML's dependency convention: the source of the arrow imports the target.

To deal with the case where different types should be considered the same, we can either write within the importing package an extra invariant like Client=Customer; or we can use renaming. The former ends up with two names for the same thing, which could potentially lead to confusion; so the latter is preferred. A typical package diagram with renaming might be:

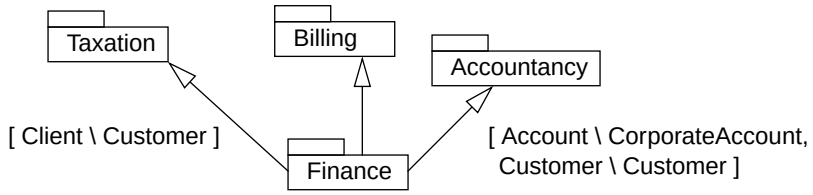


Figure 178: Using re-naming to combine types

Import schemes can also be written in text form:

Textual versions

```
package Finance    imports Billing
                   and     Taxation [Client \ Customer ]
                   and     Accountancy    [Account \ CorporateAccount,
                                           Customer \ Customer ];
```

It's possible to write package expressions:

Package expressions

Taxation [Client\Customer] represents a package that is exactly the same as Taxation, but with the names changed.

P1 and P2 represents a package that has all the statements of packages P1 and P2, and nothing further.

P1 imports P2 is a boolean expression stating that every statement of P1 can also be found in P2. You can also write P1 => P2.

Types and packages are just two of the many kinds of elements that can be renamed. Most other named elements are also subject to renaming. This facility is used to help create template packages in Chapter 10, *Model Frameworks*.

9.6.1 Selective Hiding

It is sometimes useful to explicitly render a name syntactically invisible to its importer. Rename to an empty substitute:

Rename to empty

```
import SomePackage [ someName \ , anotherName\ ]
```

Hiding a package hides all the names defined within it.

Exceptions can add much complexity

Exception handling adds complexity to any application. Even if the normal behavior of some component would be easily understood, the presence of exceptions often dramatically complicates things. We want to be able to separate exception specification — to keep the normal behavior specs simple; but we also want to address the specific characteristics of exceptions.

9.7.1 Required Exceptions vs. Undefined Behavior

Distinguish *normal*, *exception*, and *undefined* behavior

The outcome of invoking an operation with some inputs and initial state will be either *defined*: the operation is required to complete and the outcome must satisfy some specification; or *undefined*: the specification does not constrain the outcome for those cases. Any *defined* behavior could, in turn, be considered as: (i) *normal*: the operation performed the required task; or (ii) *exception*: the operation did not perform the required task due to some anomaly, and signalled the failure as required.

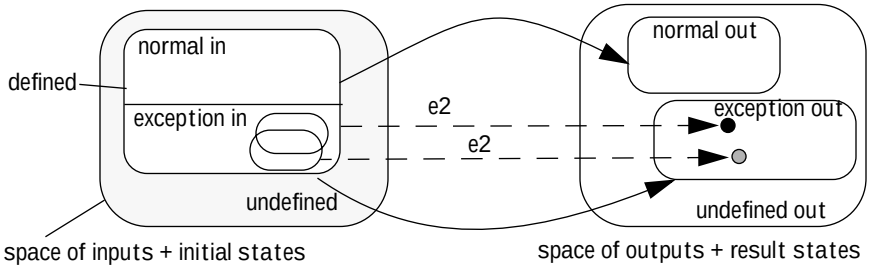


Figure 179: The spaces of normal and exception behavior

Guaranteed exceptions must be specified

It is important to distinguish the *exception* case, in which the operation has met its specification with an exceptional outcome; from the *undefined* case: the operation has no specified behavior. The figure shows that normal and exception outcomes are disjoint, because there should be unambiguous checks to distinguish required success from required failures; the caller should not be guessing *"hmm...wonder if that last call succeeded"*.

The figure also shows that different inputs can give rise to different exception outputs (e1, e2); in some situations there may be more than one exception condition true.

9.7.2 Design by contract vs. Defensive Programming

Defensive programming is paranoia in code

Over the years, there has evolved an approach to programming called "defensive programming". In essence, when you implement any operation you do the following:

- consider the normal invocation of your code; implement it.
- consider the countless abnormal invocations of your code; implement checks for those conditions, and take some kind of "defensive" actions e.g. return a null, raise an exception.

This approach to programming can be quite damaging. Each implementer provides those defensive checks that she thinks of *in the code*, but none of the interfaces document what is guaranteed to be checked and by who; or, what outcome is guaranteed in the event of those errors. Responsibilities become very blurred, and the code becomes littered with disorganized, redundant, and inadequate checking and handling of exception cases.

...with interfaces unclear about responsibilities

Instead, be clear about the separation of responsibilities in the design itself. The specification of each operation should clearly state what assumptions the implementer makes about the invocations — the caller must ensure those are met; and what corresponding guarantees the implementer will provide. This includes a specification of what failure conditions or paths the implementer *guarantees* to check, and the corresponding outcomes. Then, implement to that contract; allow for the "defensive programming" mode when debugging the code, and when running tests (Section 7.1.4, "Operation abstraction," on page 268).

Interfaces should spell out guaranteed exception handling

By all means, employ "defensive specification" at appropriate interfaces in your system; but make sure that the checking and exception handling is specified and documented as part of the interfaces, not just in the code.

9.7.3 Specifying Exceptions

Advanced Topic

We want to separate normal and exception conditions, both within one spec, and across multiple specs. However, we still want our descriptions to compose with predictable and intuitive results.

In Catalysis, the approach of specifying using pre/postconditions simplifies matters, since you can have multiple specifications for an action that compose following clear rules. However, exceptions pose some unique requirements; our approach is influenced by the work on ADL (http://www.sunlabs.com/research/adl).

Multiple pre/post is a start

We introduce two special names, normal and exception. These names can be used in two ways:

Special names: *normal*, *exception*

1. As boolean variables names that can be bound before the pre/post specification section; define normal and exception in terms of the success and failure indicators that operation will use. They are treated as special names, as opposed to names introduced locally within a "let...", because their binding must be shared across all specifications of that action.

   ...define success and failure indicators

   ```
   action Shop::order (c: Card, p: Product, a: Address, out success: Integer )
       normal = ( success=0 ) ... -- success indication to the caller
       exception = ( success < 0) ... -- failure indication to the caller
   post: ...
   ```

2. As boolean variables that can be used within a postcondition e.g.

   ...and are usable in postconditions

   ```
   action Shop::order (c: Card, p: Product, a: Address, out success: Integer )
   post:    c.isOk => normal -- success indication if card is OK
            if (....) then (exception and ....) -- failure indication must be raised if ....
            if ( exception ) then ( ..... ) -- any failure must guarantee ....
   ```

We can now require the operation to never have an undefined outcome:

```
post:   ( normal or exception ) = true
        -- must indicate success or failure; returning +2 would be an implementation bug
```

Or, to never raise any exception besides a particular set:

```
post: exception => ( success = -1 or success = -2 )
```

Can now specify success effects

We can now write the successful outcome, assuming that the success indicator will be defined someplace. Effects guaranteed with success indicator:

```
action Shop::order (....)
post:   normal => (     -- if success is returned, then caller is assured the following
            Order*new≠0 and
            c.charged (...) and
            ( p.noInventory => RestockOrder*new [...]≠0)
```

or, conditions under which success must be indicated:

```
action Shop::order (....)
post:   c.OK => normal -- if the card is OK, definite success indicator
```

..and exceptions

We can write the exception outcomes in the same manner.

```
action Shop::order (....)
post:   not c.OK => ( success = -1 ) -- specific indicator for bad card

action Shop::order (....)
post:   not a.OK => ( success = -2 ) -- specific indicator for bad address

action Shop::order (....)
post:   exception => ( Order*new–>isEmpty )
        -- if failure signalled, guaranteed that no new order was created
```

With multiple exceptions, don't comit to a specific one

Typically, however, you want to deal with multiple possible exception outcomes in a slightly flexibly manner. If you place an on-line order, given the preceding spec, what should happen if credit card number and address are both invalid? Which exception should be raised? It is best to leave the choice of *which* exception to signal to the implementer, as long as failure indication is guaranteed. This helps with composition of specifications, each with their own exception conditions; as is the case of failures in distributed systems. Hence:

```
action Shop::order (....)
post:   -- bad card means some failure indication
        c.bad => exception
        -- a failure indication, with code -1, will only happen if the card was bad
        (exception and success = -1) =>  c.bad
```

Use "isException" for flexibility

This is a very common form of specification for exceptions, so we introduce a convenient query isException on the pre-defined type Boolean, and re-write it as:

```
post:   (c.bad) . isException (exception, success = -1)
```

which is exactly equivalent to the longer form. Our definition of isException:

```
-- a given trigger condition is an exception means...
```

```
Boolean::isException ( generalFailure: Boolean, specificIndication: Boolean ) =
        -- if the trigger condition was true, then some failure has been signalled, and
    (   (self = true) => generalFailure ) and
        -- the specific Indication will not be raised unless the trigger was true
        ( generalFailure & specificIndication => (self = true ) ) )
```

One final point: suppose you write a specification which simply says: One subtle point

Success indicator = a; Failure indicator = b;

If a, then x is guaranteed to have happened;

If b, then y is guaranteed to have happened.

You would, strictly speaking, have to admit an implementation that simply failed every time, as long as it met the failure indication. You should either be more strict about the kinds of exceptions that can be raised, and when; or, assume a reasonable convention where the implementor is obliged to try to meet the success goals, and should only raise an exception if that turns out to be impossible.

7.3.1 Exception Indication Mechanisms

Different languages have different mechanisms to indicate exceptions: return values, exception objects thrown, signals raised, etc. For specification purposes you can work with any one of these, including some language-neutral mechanisms (like return values; remember that the signature of an abstract action specification is itself always subject to valid refinements, Chapter 12, *Refinement*). Includes language specific mechanisms

```
action T::m (....., out success: Boolean)
post:    not success =>  (...guarantees about every indicated failure...)

action T::n (...., throws (Object))
     exception = thrown (Object) -- no using throw except to indicate failure

action T::n (.....)
post:    (self.wrongState) . isException ( thrown (WrongState.new) )
```

Or, to make some guarantee on any exception thrown, not necessarily by self:

```
action T::n (....)
post:    thrown ( Object ) => (...e.g. all state cleaned up ...)
```

The meaning of the language specific mechanisms, such as throw, is provided by working in a context where the appropriate packages have been imported (Chapter 15, *Frameworks*).

7.3.2 Exceptions with general Actions

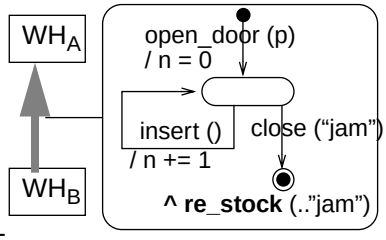
This approach extends to general actions. An exception in an abstract action can be traced across action refinements: specific traces or sequences of detailed actions can be specified as raising an exception on the abstract action (rather than having to invent a new abstract action for it, or having to ignore it at the abstract level). It can be traced through action refinements down to the level of exceptions in program code. Abstract actions can also specify exceptions

..that are meaningful at that level

Of course, when specifying an abstract action you should only describe those exceptions that have meaning at that leve l of abstraction, particular exceptions that arise in the problem domain; not every form of software exception is meaningful there!

The refinement then maps failure sequences as well

Let us re-visit the example in Section 5.5.3, “Refining an Action or Use Case,” on page 218, on re-stocking of a vending machine. Suppose the warehouse inlet door for a product can jam when closed. If this outcome is an interesting exception at the abstract level, it could have been specified as such on the joint action. In the refinement, the appropriate sequence can be mapped to this abstract ‘exception’ action.



This provides a precise basis for exceptions in traditional use case approaches.

7.3.3 Exceptions and use case templates

Just as we introduced a narrative-styled template for defining use cases, it is useful to incorporate exceptions into use cases narrative as well.

use case	sale
participants	retailer, wholesaler
parameters	set of items
pre	the items must be in stock, retailer must be registered, retailer must have cash to pay
post (normal)	retailer has received items and paid cash wholesaler has received cash and given items
normal indicator	confirmation to retailer
exception indicator	no sale confirmation to retailer from wholesaler
on exception	neither cash nor items transferred

Similarly, it is useful to document those sequences that might give rise to an exception, as part of the use case documentation. The mapping from the formal refinement description (e.g. a state chart) to this narrative is straightforward:

use case	telephone sale by distributor
refines	use case **sale**
refinement	1. retailer calls wholesaler and is connected to rep 2. rep gets distributor memberhip information from retailer 3. rep collects order information from retailer, totalling the cost 4. rep confirms items, total, and shipping date with wholesales 5. both parties hang up 6. shipment arrives at retailer 7. wholesaler invoices retailer 8. retailer pays invoice
abstract result	**sale** was effectively conducted with amount of the order total, and items as ordered
exception	retailer cancelled order before it was shipped (step 5, use case **sale**)
exception outcome	confirmed cancellation -- implicit: non sales confirmation; no cash or items transfer

9.8 *Summary*

All elements should be composable

All modeling elements should be composable, so that specifications and designs can be factored into smaller parts; and re-combined in predictable and intuitive ways.

Packages build upon others

Packages usually import other packages, those on which their definitions are based. We have looked at the rules whereby imported definitions are combined with new material, and material from other imports.

Packages extend definitions from others

Each package has a notional ‘unfolded’ form, in which all the definitions from the imports and their imports are visible. New facts and definitions in a package can constraining its own declared names, and those which are imported.

Exceptions need some special techniques

Specifying exceptions so they can be composed, and so abstract actions with exceptions can still be refined, needs special care.