Chapter 7 Using Packages

A *package* is a container for any piece of development work you want to treat as a unit. Packaging is concerned with the dependencies between pieces of design work, which package diagrams help expose. Good packaging reduces dependencies and is fundamental to good reusable design.

The principal relationship between packages is *import.* When you start a new piece of work, you almost never work from scratch; there are at least all the primitive definitions (the numbers and so on). More usefully, you write an implementation against a corresponding specification package, and you write that specification by importing a corresponding domain model package.

This chapter discusses packages, the relationships between them, and how to use them effectively. Section 7.1 introduces packages and explains how a package contains names and formal definitions within an informal narrative structure.

Packages are built by importing other packages and by extending the imported definitions (Section 7.2). This import facility can be used to effectively partition development work in several useful ways (Section 7.3) as well as to decouple units of work from changes in others (Section 7.4).

Packages themselves are subject to packaging as nested packages (Section 7.5), and a special form of encapsulation and decoupling applies to them (Section 7.6). When packages have multiple imports, you must be careful about what it means to import like-named elements from two sources and sometimes must explicitly rename some elements (Section 7.7).

Packages also serve as encapsulated units of builds and can help with versioning and configuration management (Section 7.8). Section 7.9 compares packages to the package-like constructs found in some programming languages.

7.1 *What Is a Package?*

The package is the basic unit of development product—something you can separately create, maintain, deliver, sell, update, assign to a team, and generally manage as a unit.

A package might contain any logical unit of development artifact: types, classes, compiled code, frameworks, patterns, documentation, tests, and so on.

In the Catalysis model, all development work creates or modifies a package even if you don't give it a name. The package is the collection of all the names you define and all the statements you make about them, whether in diagrams or in text form. All the modeling work you do—drawing types, creating associations, specifying an operation—is done within a package.

Every package (except for some extremely basic ones) imports other packages. Import is analogous to Java's import or C's include. It means that all the definitions and statements in the imported package are usable within the importing package.

A package is its own "world": when working in a package, you can know or believe only what is within it—either introduced directly in the package or known in a package that it imports. You cannot refer to other things. In that sense, a package represents a unit of knowledge or belief. When you model a type with an integer attribute, you are using the imported package that defines what you know, and can say, about integers. When you write a C struct with an integer field within a package, you also import the package that defines what a C struct and its fields mean.

For example, it's common in large enterprises to make a model of the business domain—let's say, a telecommunications company with Calls, Circuits, and so on. When you are writing the requirements for a billing system for the company, you naturally use some of these terms, and you want to ensure that you use the same names for the same concepts and with the same relationships between them—as well as add extra material specifically about accounts. So your Billing Requirements package will import the Telecoms Business package; so will the Network Management package and others, thus ensuring that terms used throughout the company have the same meaning. (We have seen such companies in which wildly inconsistent vocabularies were used in various departments—until they got sensible and wrote a business model.)

Then when you design components of the system, you can import the relevant requirements model and use the types it defines.

But even a package that doesn't import anything locally defined uses names such as +, 42, true, integer, and and—all the primitives as well as the relationships they have with one another. These, too, are defined in packages.

User-defined models, such as design patterns, frameworks, and other architectural elements, are also defined in their own packages. The set of definitions composing each model is grouped into a package and can be used by many others. Even attribute definitions, associations, and refinement relations are defined within a package.

7.1.1 Notation

A package is drawn as a box with a tab containing the package name. Figure 7.1 shows one that defines the basics of a telecommunications business. Inside the box, you can draw some or all the definitions the package contains. However, packages can be quite large, so this approach is often impractical. Tools allow you to click on the package symbol and

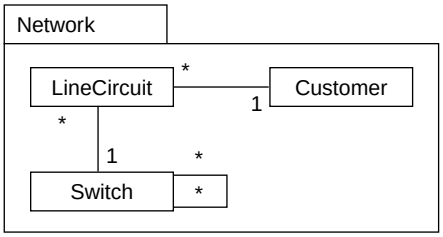
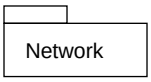


Figure 7.1 Basic telecomm package and contents.

browse the collection of type diagrams and so on that the package contains. In a document, you can make each major section, and the whole document, correspond to a package; you can structure subsections around actions, subject areas, nested or imported packages, or something else.

Within textual descriptions, it is sometimes useful to be explicit about which package originated the definitions you are talking about. Using the context symbol as follows means "What follows is part of the Network package" :

```
package Network :: ...
```



The package diagram symbol is often used without contents. In this case, many people (and tools) prefer to move the package name from the tab to the main box. This convention is inconsistent but looks nicer.

© ***package*** A named container for a unit of development work. All development artifacts—including types, classes, compiled code, refinements, diagrams, documentation, change requests, code patches, architectural rules and patterns, tests, and other packages—are in some package. A package is treated as a unit for versioning, configuration management, reuse, dependency tracking, and other purposes. It also provides a scope for unique names of its elements.

7.1.2 What Does a Package Contain?

In the early days of object-oriented programming, attention was focused on the class as the fundamental unit of modularity. But there are other equally valid units of design work. Think of all the different pieces of work you could usefully transmit to a colleague or store in a database; here's a list of things a package might contain.

- Types and classes—specifications and implementations of objects.
- Source or compiled program code.
- Collaborations—partial schemes for object interactions.
- Top-level names of constants, variables, and functions.
- Refinements—documented abstraction relationships between types, collaborations, requirements, classes, and so on. A refinement is often placed in a separate package from the elements it relates by importing those packages into its own, and adding documentation justifying the refinement.

- A group of types or classes that form a coherent set and don't make sense without one another.
- Partial definitions of types or classes described from one user's point of view, with potentially different aspects of each type captured in different packages.
- Useful information about an existing type or class—for example, an observation stating, "Class Bicycle conforms to type WheeledVehicle"—that has not been explicitly stated by the designer of class Bicycle in the package.
- Patterns and frameworks—recurring schemes of types and collaborations that are abstracted into a pattern, optionally including generic implementation code, and then used in many places.
- Requirements—initially informal and then formalized descriptions of what is expected of a system or component.
- Change requests—a common form of requirement.
- Bug reports—with symptoms; hypotheses on the causes; justifications; scenario or test data to reproduce the bug, and so on.
- Modifications to code, including bug fixes, that can override or patch existing code in another package (with certain restrictions ensuring backward compatibility).
- Narrative documentation (or preferably, hypertext) accompanying the model.
- Diagrams—presentations of model elements in various visual forms.
- Other (nested) packages—as a grouping and macro-visibility construct.
- Tests or verification documents—specifications of correct behavior, concrete initialization and test data, expected results, and actual results.

In addition, there are many project-management-related attributes associated with the elements in a package and with packages themselves. These attributes include effort, priority, status, authors, schedules, and so on.

It is a central tenet of our approach that every picture should be translatable into text-based statements, not because you'd ever want to but because that gives it a clear and unambiguous meaning. For the same reason, the text-based notation we use can be translated into a small core syntax. Notice that this is more than a claim to have a clear syntax: It is an unambiguous semantics. So the names, definitions, and statements of a package[1] really carry all the information we will discuss in the rest of this section.

A package can contain definitions of constants. It can also contain descriptions of prototypical instances as used in snapshots, scenarios, or interaction diagrams or can even contain a global declaration of a name that refers to a particular object.

In short, the answer to "What can be in a package?" is the same as "What can be in a file?" or "What can be in a document?" A Catalysis package is similar to a package in Java or a namespace in C++; a Catalysis package contains not only code but also specifications and designs and, unlike programming languages, allows for some forms of factoring and then reconnecting partial descriptions.

1. A package is a many-sorted theory, as in [Larch91] or [Mural91].

In short, a package can contain any formal or informal description, program code or model, and textual or pictorial statements (see Figure 7.2).

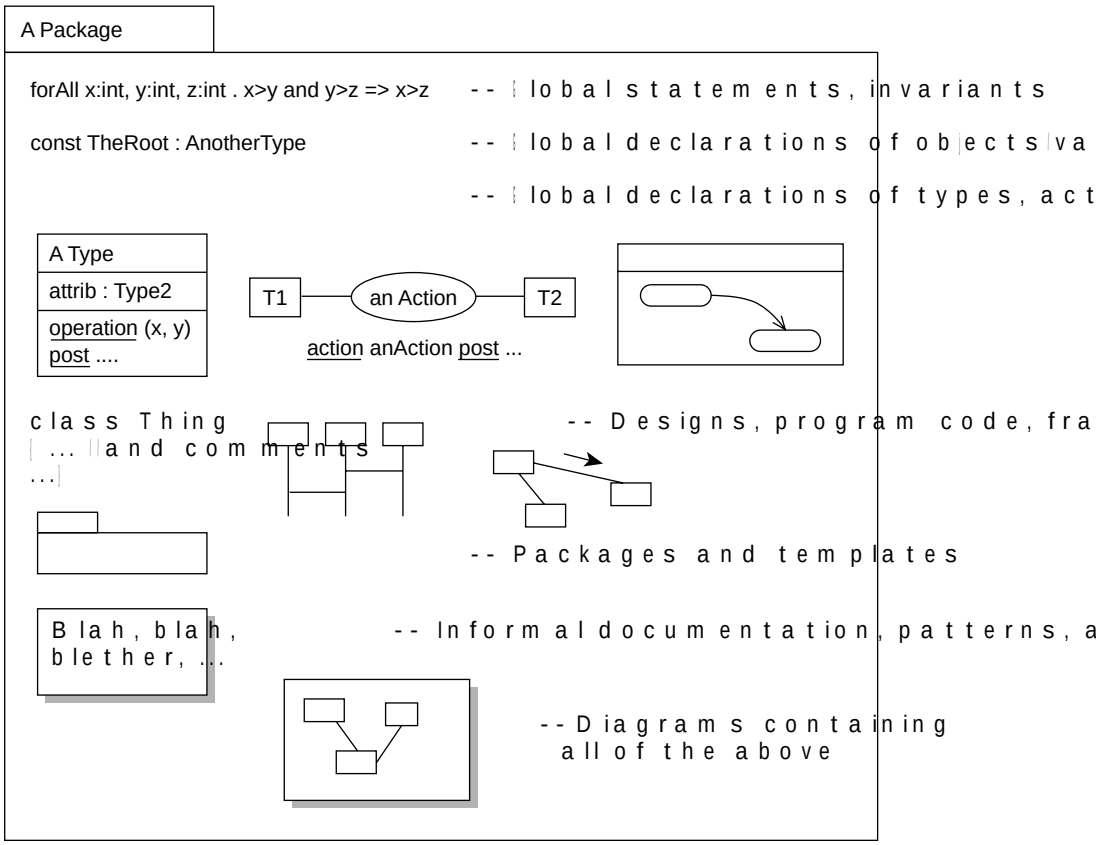


Figure 7.2 General package contents.

7.1.3 Model of a Package

A package consists of at least the following elements.

• *Name*: Its own name should be unique (within a given containing package) and may include a version identifier. A package has a planetwide unique effective name and in that respect is rather like a URL (the Internet addresses typically of the form http://someMachine/aDirectory/aFile). The name should not change once it is published. Conversely, no two published versions should have exactly the same name even if the package contains only a bug fix.

• *Administration*: Publication status, author, version, and derivation history.

• *Declarations*: Names that are declared in this package for types, actions, classes, invariants, constants, and nested packages.[2]

• *Facts and rules*: Statements involving the names—definitions of them (such as constant or type or class definitions) and constraints (such as invariants, pre- and postconditions, and cardinalities of associations). Facts can be stated pictorially—in the forms of all the static type models, action diagrams, and state charts that we have been discussing—or in text.

• *Dictionary*: With every declaration there should be a dictionary entry. Its purpose is to convey informally what the name represents and its relationship to its real world. Whether or not it is presented as a single table, the dictionary is notionally the holder of all the names and their definitions in a package even if the definitions are actually spread around narrative text or within other pieces of syntax.

• *Diagrams and appearances*: The total model defined by a package is split across a number of diagrams. Each model element can have multiple appearances in different diagrams or in fragments of formal text.

• *Notes*: Every diagram can have embedded informal notes. Any syntactical construct—for example, a type, an attribute, a postcondition, a subexpression, or a variable—can also have an informal note. A note can appear in addition to or instead of the more formal statement.

• *Narrative*: The formal contents of the package should be divided into digestible chunks and embedded with text and diagrams in a narrative document. This narrative need not be linear; it can be a web of information, including links that refer to imported packages. All documentation is also part of a package.

7.1.4 The Narrative and the Dictionary

It is an important principle of Catalysis that precise, formal notation be used to complement, structure, and render unambiguous the requirements, design, or other modeling documents that you write. The more informal, immediately readable material (such as what

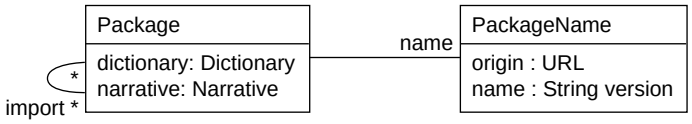
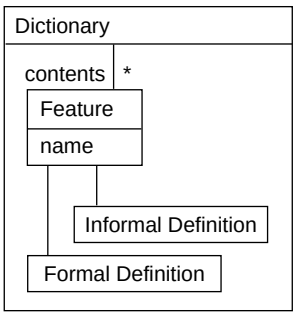


Figure 7.3 Every package has a dictionary and a narrative.

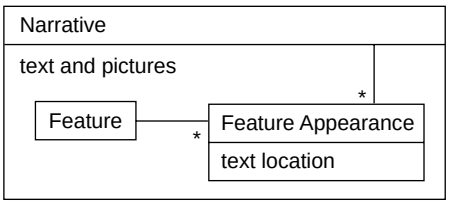
2. The very constructs "types," "class," "association," and so on are themselves defined in packages (see Section 9.9, Down to Basics with Templates).

you are reading) is called the *narrative*. Every package has a dictionary and a narrative, as shown in Figure 7.3.



The dictionary is the interface between the informal and the formal. It has an entry for the name of each named *feature* together with both its formal and its informal definitions. Whether or not the dictionary is presented as an actual table (either in printed matter or on screen—a good tool should be able to extract it and present it if you so require), the dictionary is the notional holder of all the features defined within the package.

Features include types, collaborations, actions, patterns, classes, or extensions to classes—anything that has a name and a definition (see Figure 7.4).



The narrative of a package is the important structure of text, pictures, illustrations, anecdotes, footnotes, and all the other apparatus of good explanation directed at helping human readers to understand what you are describing. Embedded in that material are the the formal descriptions of the features you wish to display, just as we have used formal diagrams to illustrate what we are also trying to explain in words. Neither the formal nor the informal would be very good without the other. Informal is too vague; formal is too inscrutable.

Features may have several *appearances* in a narrative, each as part of a different diagram at a location in the text. For example, the type Feature has just appeared in three different places. Rather than display a single wall-sized diagram, it is better to split it into smaller topics and discuss each one in turn, using the formal notation to back up the flowing, exciting, erudite prose. Each appearance contributes to the overall definition of that feature within the package.

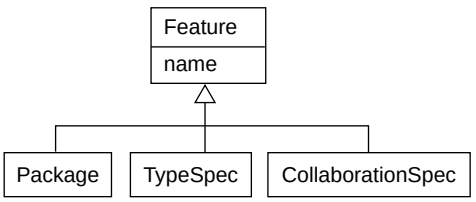


Figure 7.4 Features include anything that has a name and a definition.

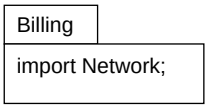
7.2 *Package Imports*

One package can import another package, and everything stated in the imported package is seen in the importer. It is as though the imported package contents were copied into the importer except that when you update the imported package, the importer also changes; and the importer cannot delete anything that was defined in the imported package; it can only add to it. The example in Figure 7.5 on the next page shows a structure of importing packages. Notice that the arrow goes from the importer to the package being imported. Thus, the Billing and Fault Management packages can both see the definitions of Line Circuit, Switch, and Customer that were defined in the Network package.

Furthermore, every definition is introduced within a package (even if we've forgotten to say which one!). The package identifies the context in which any diagram is drawn. If you draw a diagram on the tablecloth while having a project dinner, the drawing is in a package, and the imports are all the assumptions made by the people at the table about the terms you're using. You all know what a Thingamajig is, because it's in the project's business model (embodied, we hope, in a more conventional medium than the tablecloth).

If you take part in a different project that happens to use the term *Thingamajig* for something in the business, you recognize that you could be using different imports there. And if I write "2+3," you know what I mean because we both import the same package of arithmetic definitions we learned about in school.

Imports are shown as a dashed arrow from importer to exporter—strictly with the stereotype marker «import». An equivalent is to write an import statement inside the importing package (along with any other contents), as in this diagram. A formal text alternative to the import diagram is




```
package Billing imports Network;
package Fault Management imports Network;
package Network imports Telecoms Stds;
```

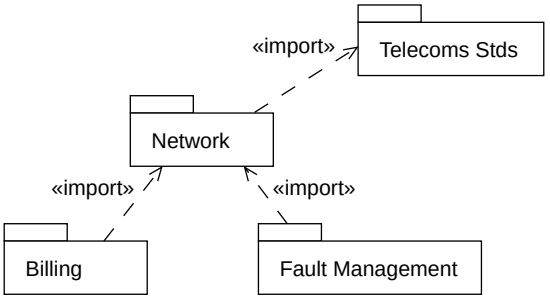


Figure 7.5 Package imports.

7.2.1 Importing to Extend or Specialize

One package can import another package and extend it. The extending package adds more definitions; its reader knows everything there is to be learned from the original package plus more that the extender has defined (see Figure 7.6). Extension is drawn with a generalization or specialization arrow in UML; but our default meaning, whenever an import is drawn without other qualification, is extension.

In Catalysis you are allowed to make additional statements about an imported model

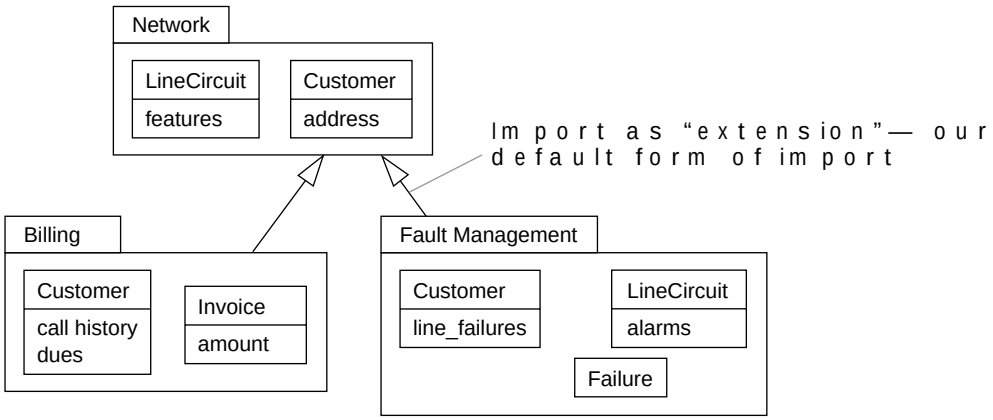


Figure 7.6 Package import for extension.

element in the importing package; examples are new attributes, new actions, or even new pre- and postcondition pairs for an imported action. In the Billing package you can define call-history and cost information with a Customer; in that case, the Fault Management package may find the history of failures reported by that Customer to be more interesting. This gives us a flexible mechanism[3] for defining multiple views or subject areas: Define a base package that introduces shared terms, such as Circuit and Customer; then define each view as an importing package, extending the definitions of Circuit and Customer for that view. Each view describes somewhat different aspects of an overlapping set of types and actions. Extension corresponds roughly to C++ include, Envy prerequisite, and the mathematical idea of theory extension.

You can draw an approximate analogy between package import and class inheritance. In both cases definitions from one package or class become available to another package or class. With packages, imported types become available; with classes, inherited methods become available. In addition, with packages, the importing package can define additional properties about the same type; with classes, the inheriting class can extend the definition

3. The commonly used solution of subtyping does not address this problem.

of a method by adding its own specific processing. One difference is that you can override a method in languages such as C++ and Java and thereby entirely replace the superclass method.

© ***import, extension*** A relation between packages whereby all names and definitions exported by the imported package are accessible in the importer together with any new elements and added statements about the imported elements that the importer may introduce. A package exports all introduced elements as well as all elements accessible via extension imports. Import by extension is the default rule for import.

Extension has a number of typical uses. It is used for defining a component spec from a business model and for defining variations. In the first case, the component model builds on the definitions of the business model. So once you've read the Networking requirements, you know not only what a LineCircuit is but also how to point at one in your software system.

In the second case, the imported model contains partial definitions; the extenders add the missing pieces in different ways. For example, you might have two slightly different networking requirements: one for small offices and a second one for large offices. They could perhaps contain different collaboration or type definitions at a detailed level but be based on a common underlying abstract model.

It is sometimes useful to prevent a model element—a type, an operation, and so on—from being extended in another package. That element can be marked as «final»: Any importing packages can use it as is but cannot extend it. Specifically, an importer cannot add *new* information pertaining to that element, but new specifications that are *derived* are still permitted (see Section 3.5.2, Redundant Specifications Can Be Useful).

7.2.2 Import for Private Usage

An import for private usage is a private relationship between packages. If a package is used by another package, the user of the first package can see its definitions; but the user's users can't. This arrangement lets the importing package make use of definitions while keeping them hidden from other importers; it is used most commonly to document internal design relationships that should not be visible to others (see Figure 7.7). Malik's Accounts Receivable System can use definitions made in Pat's Billing Design as well as those in Billing Spec but cannot automatically use the contents of Jean's Generic Invoicing Component Spec. To do the latter, another import would have to be declared.

For packages that contain designs, this makes sense: Jean's . . . is used to help implement Pat's . . ., which Malik doesn't need to know about. It's the usual argument for encapsulation at the level of a complete package. For large systems, encapsulation at this level is more useful than it is at the level of individual objects or classes.

By contrast, we generally use extension on the level of models and specifications because each package in the chain defines concepts that are used all the way down the line. The basic concept of a Line Circuit will be used in nearly all the packages in a telecommunication business, and it will have been defined up at the level of the Network

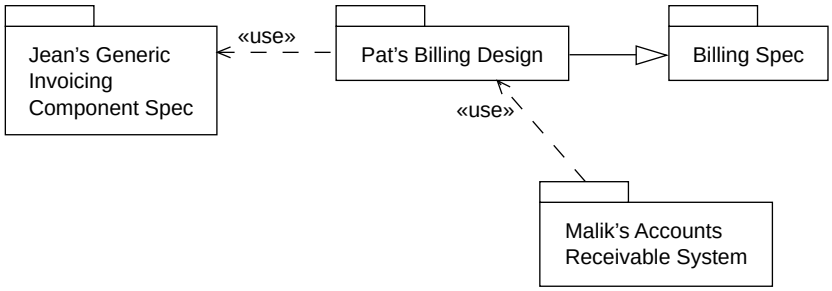


Figure 7.7 Import for private usage.

package; or perhaps ClockBuster extends an industry-standard package that defines a Line Circuit.

Some OO analysis and design tools automatically infer dependencies when you use a class from one package within another package. We prefer to state imports explicitly: You should use the contents of a package only if you explicitly have decided to use it. This policy ensures that we keep conscious control of the coupling between packages.

© ***import, usage*** A relation between packages whereby all names and definitions exported by the imported package are accessible in the importer together with any new elements and added statements about the imported elements that the importer may introduce. However, elements accessible only via usage imports are not exported by a package.

7.2.3 Rules on Imports

The first rule is that import defines visibility. Every name or assertion you can use either is defined within the package you're working in or comes from the packages it imports. If you want to use something defined in another package, import it. If you do not want to import that entire package, you must refactor it into smaller parts that can be imported separately.

You can think of a package as a complete "world" of information: names, terms, definitions, and rules. The package within which you are working defines all the things you know about and all that is known about them.

Visibility is transitive for extension imports; your children's descendents can see your definitions. It is not transitive for usage imports; only a direct usage importer can see your definitions.

The primary reason for requiring transitivity is that individual types and classes almost never stand on their own. If you depend on a particular class, then you also depend on at least the type definitions of all its method input and output parameters, which could often be in its imported packages; if these definitions change or if a new version is released, your work will need examination.

The second rule is that circular extensions are forbidden. Extensions cannot form a cycle. Because visibility is transitive for extensions, a cycle in imports means that all elements are mutually visible. The entire cycle might as well be in a single package.

Note that there are always some elements that are intrinsically mutually dependent even for their definitions. It is futile to try to break these cycles; just put them into one package.

Usage imports, in contrast, can be circular. It is permitted, but not good practice, for usage imports of designs and implementations to be circular; if they are, it means that no one can use one in the circle without bringing all the others. Often the key to breaking circles is that an implementor should refer to someone else's specifications and abstract type definitions rather than implementations or classes. For example, A's reference to B's implementation is suspect: It would be better to depend only on the requirements spec that B has implemented. (See Section 7.4.1.)

Fourth, there can be no contradictions. An extender can add type or action definitions. It can also add new information about imported types and actions but should not contradict what is already known.

For example, in the early school years we learn a package of arithmetic in which the numbers have well-known operations. In college, we may learn another package that talks about statistical operators. The package does not define any new types. We are still talking about the numbers we have always known; it's just that we now have new information about the same types.

An example of a contradiction would be for one package to give a postcondition x > y and an extender to give the postcondition x < y for the same action.

The rules also allow multiple import. You can import multiple packages; it just brings in all the definitions and statements expressed in all the imported packages. (For more about this, see Section 7.7, Multiple Imports and Name Conflicts.) However, a package should not import very many others. The more packages it depends on, the more likely it is to be affected by any changes.

In general, there is no point in giving someone a package unless he or she also has access to the packages it imports: The meaning of the package is in part contained in the exporters.

7.2.4 Catalysis Import and Dependencies

UML has a relationship between packages called *dependency*, which can be annotated as an import; another relationship is called *generalization*. In Catalysis you must permit and track dependencies, by explicitly deciding to import a package, before you can use any of its contents. This approach is very much like the import and include schemes in programming languages. A general dependency is inferred after the fact, whereas an import is decided beforehand.

Two Catalysis packages may have different definitions (of types, classes, and so on) that happen to have the same name. This is because they may come from different sources. Moreover, using *extension*, an importing package can "say more" about the terms that

were imported from another package. UML has another modeling element called *model*; it seems superfluous once you have packages.

7.2.5 Virtual and Concrete Packages and Virtual Imports

A *concrete* package is one documented in the UML syntax; *concrete import* is the relationship whereby statements of one concrete package are adopted as part of another. However, we sometimes work from a common set of terms that was never documented as a concrete UML package—for example, some telecommunications standards.

A *virtual* package is a coherent set of ideas, whether written or not, that has not been explicitly documented as a UML package but that we still wish to reference in our work. It might be a consensus in your business or in a standards organization about the meaning of different terms and the properties of the things they refer to. Or it might be a document written in informal language.

For example, one of our clients in the telecommunications industry recently built a business model. In that world, all the manufacturers and operators adhere to several standards. So some of the client's model was determined by what the standards say and will have to be updated whenever the standards change; in other words, it imports the industry-standard virtual packages. So our client's business model contains definitions of the standardized concepts (or at least its interpretations of them) along with all its own definitions. Because there were not yet any UML versions of these standards, we simply consider the standards documents to be part of an implicit virtual package and proceed to "virtually" import it.

So a virtual package is a set of ideas that is represented in some part of your concrete package (see Figure 7.8). And to virtually import means to write the definitions into your own model because they were never formally defined in the package you import. But at least you have documented the set of things you assume. We can also talk about an informal import, in which some other document is referred to in an ad hoc way.

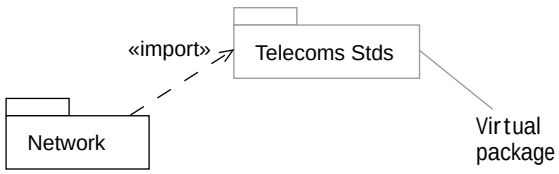


Figure 7.8 Import of virtual package.

It might have been better to have put the standards into concrete packages and thereby clarify which definitions were the client's own and which were imported. Also, should the telecomm standards organizations ever get around to writing UML packages with precise models, it would become easier all around to adopt them.

Virtual imports always raise the question, "Should we make a concrete package of this?" The answer is usually that it would be a good idea. But in some cases you may really use the external ideas only as a prompt but interpret and redefine them as you go

along. This is fine as long as you've consciously decided to do so. The disadvantage is that you may be compromising future interoperability with others in the same area; how will you connect to someone else's system that uses those standards? And if you use the same terms for different concepts, you risk confusing people. BluePhone's LineCircuits were subtly different from those of everyone else in the industry, a common stumbling block for "cross the street" recruits.

One of the benefits of using an import (virtual or otherwise) is that you are forced to use a different name for a different concept: Redefinitions aren't allowed, although extensions are fine. (Maybe they should have called them BlueLineCircuits or something.) And one of the benefits of talking about virtual packages is that it makes you consider what the sources of your ideas are or should be and whether you are or should be using the same or different terms for some wider body of ideas.[4]

© ***package, virtual*** A name that informally denotes (as opposed to actually containing) some set of terms and definitions that you want to refer to from other packages. A virtual package can be "virtually" imported by a "real" package.

7.3 *How to Use Packages and Imports*

Catalysis rules for packages and imports give you much flexibility in factoring your models into separate parts. There are some common idioms for using these facilities.

7.3.1 Separation of Concerns

Packages separate work into areas that can be dealt with individually, with explicit dependencies between them. This arrangement helps control the propagation of changes and the concomitant expense. Therefore, you should divide your work into manageable chunks and put them into different packages. The same principle applies at all levels of development, from business modeling to program code. The typical methods for partitioning contents of packages are as follows.

• *Vertical slices*: This approach reflects the fact that different users have different views of a business or system—for example, accounting, call processing, and network monitoring. They may use different software applications, performing different actions on possibly overlapping sets of objects.

• *Horizontal slices*: This technique separates business models, specifications, and designs down to the level of technical infrastructure and communication protocols. These are different levels of description of the same phenomenon and are related primarily by some form of refinement relation.

4. Critics may wish to consider how well we have adhered to this advice with respect to the metamodels of Catalysis and UML.

• *Different domains*: This partitioning is based on domains such as user interface, persistence, or the primary problem domain.

• *Templates and patterns*: After different concerns have been separated, you often find that the same general schemes can apply in a variety of situations. With careful structuring, they can be made to apply simply by changing names. This is the basis of *model frameworks* based on *template packages*, which allow you to write generic models. The main aspects of most analysis and design patterns can be expressed in this way. These generic schemes can be used to express, among other things, connection protocols between components. We will look at template packages in Chapter 9, Model Frameworks and Template Packages.

• *Translation schemes*: One particularly systematic approach to development encodes a selection of patterns as translation schemes. To move from requirements spec to detailed code, you choose a succession of translations. Template packages can be used to represent translation schemes. Another variety of translation scheme expresses the semantics of a language, such as UML, by defining how it translates to more basic terms. This technique is especially useful for stereotypes, the variable part of the UML notation.

• *Assertions and proofs*: Particularly when you're specifying safety-critical systems, it is important to document all critical information about the beast being built. Sometimes, redundant or derived facts are useful: If "the coolant valve is open whenever temp > 700" and "whenever the coolant valve opens, an alarm rings," it may be worthwhile to explicitly state that "whenever the temperature *rises* above 700, the alarm rings." There are many less obvious properties of a software component, and sometimes the original author is not the one who documents them in the package. A separate package can then be written, not to define anything but only to contain this extra information.

The following sections look at vertical and horizontal partitioning in more detail using models and software applications for a telecommunications business.

7.3.2 Vertical Slices

One way to separate concerns at the business and requirements levels is according to the point of view of different categories of users—a particular kind of subject area. This approach gives you different models of the same types and actions (they must be combined at some stage before you can build a system). You base your views on the high-level actions of those users. Each action needs a model to specify what it achieves and needs additional static and dynamic models to refine it into finer-grained actions.

For example, BluePhone's analysts begin by looking at the business activities of their company. They produce the high-level collaboration diagram in Figure 7.9, which they would like to partition. This is a typical high-level business model obtained by asking people, "What do you do? Whom do you talk to?" Each of the actions shown here refines to a series of many smaller concurrent actions: making a call, chasing a fault, changing service level, making a charge and sending a bill, and so on. The actors refine from departments and groups of people to individual customers, roles within departments, and so on.

These major actions partition the model into vertical slices so that different teams can interview the actors separately. They will get different perspectives on the same types. Finance and Billing will be interested in the customer's bank details; Service Provisioning will want the customer's line characteristics; Fault Management needs alarm information about lines and circuits. All of them may be interested in name and address.

It is useful to establish a central package of shared basic definitions that are known throughout the business—the vocabulary the staff use in their work. The *viewpoint* packages can import this package. Like our earlier Network package, it tends to be a mostly static model (types and attributes).

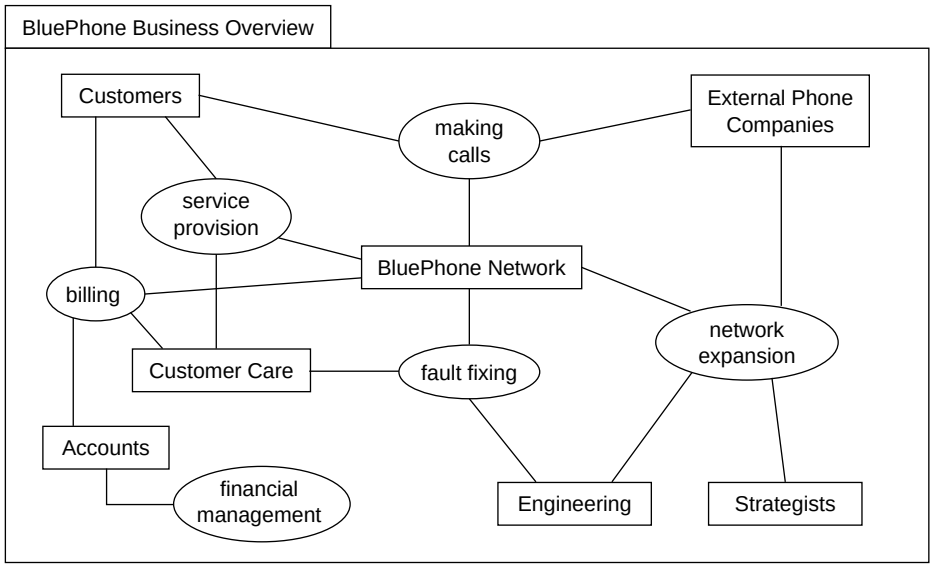


Figure 7.9 High-level business collaboration.

Sometimes another partitioning becomes apparent in a layer in between (see Figure 7.10). For example, BluePhone decided that many Fault Management and Network Expansion tasks are shared between the major activities and that they would have an intermediate layer of shared business model packages for different areas. Performance monitoring, for example, contains a number of actions concerned with gathering and processing connection statistics together with models of the performance figures produced. Strategists use the figures for planning network enhancements, whereas engineers look for unusual changes suggesting malfunctions.

All these packages are concerned with modeling business activities. Some of the actions at the detailed levels may be supported by a software application.

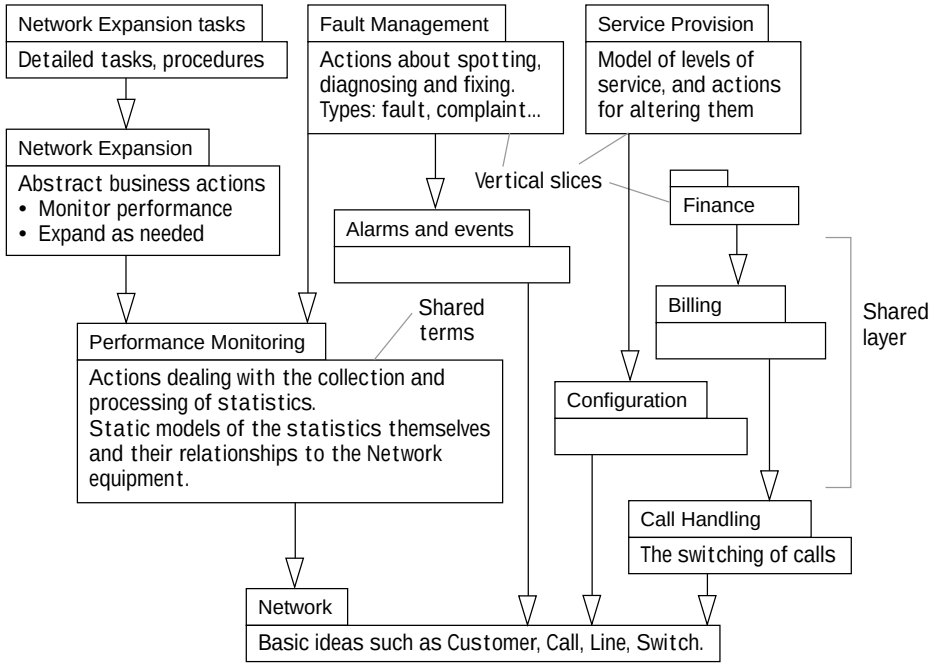


Figure 7.10 Packages segment vertical slices from a common base.

7.3.3 Horizontal Slices

In addition to separating different business areas, it is also useful to separate high-level business models, detailed business tasks, specifications of software supporting those tasks, and technology infrastructure in the software implementation. BluePhone decides to build a new software interface for the Customer Care role in Figure 7.9, integrating service and billing inquiries. Based on the package partition in Figure 7.10, we define a new software requirements package that imports the packages relating to abstract service and billing business activities; we separately design the application to meet the spec (see Figure 7.11).

BluePhone decides not to build specialized tools for Fault Management or Network Expansion. Instead, more basic tools for Performance Monitoring and Alarms will be used where necessary by the actors involved in the larger tasks of Fault Management and Network Expansion; they will, of course, need to do a larger portion of the abstract business tasks themselves because the tools are not specific to their needs.

The detailed tasks for Fault Management and Network Expansion are now rewritten to use these tools (see Figure 7.12). The old network expansion tasks are replaced by new procedures, but both the old and the new achieve the same high-level aims; the new tools provide a new refinement in which portions of the high-level tasks are performed by software tools. This is a good case for separating the detailed and abstract views of the business. We can then easily deal with the different realizations of the business processes, including those that involve software.

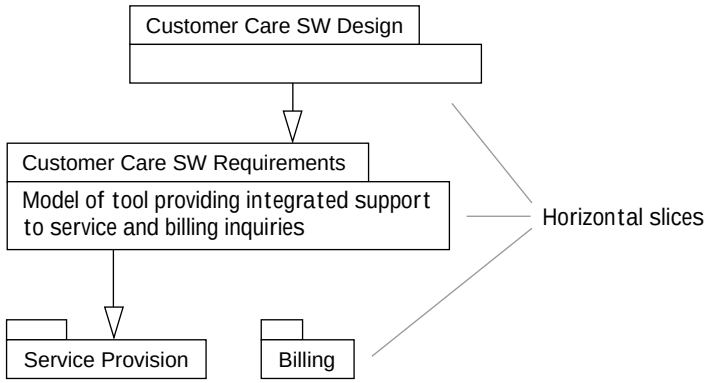


Figure 7.11 Horizontal packages slices: business, software spec, software implementation.

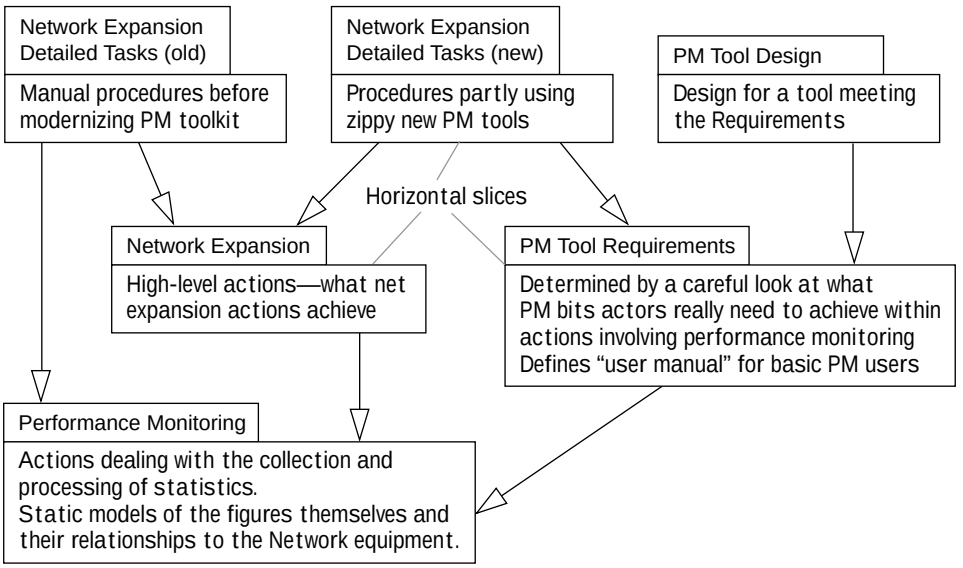


Figure 7.12 Abstract action with reengineered process and tools.

For example, the split_subnet abstract business action might be "Estimate network traffic and then purchase the cheapest equipment combination that can handle that load." In the new business, the worker may perform the estimation with the help of the new PM tools, and select the cheapest equipment combination using a Web browser and desktop calculator.

The two versions of Network Expansion Detailed Tasks import Network Expansion, because it specifies the overall abstract goals that each of them meets. Each includes refinement schemes showing how its detailed actions are composed to achieve these goals,

so they must import the package containing the goals. Perhaps the old way of estimating traffic involved watching the lights flicker on equipment and monitoring user complaints, whereas the new way uses the PM tools proactively for the same purpose.

The new Detailed Tasks package also imports the Tool Requirements so as to define how actions involving the tool are used; interactions and scenarios of using the tool to accomplish network expansion tasks are included in this package.

Finally, a Design for the new toolset also imports the Tool Requirements, again because it includes refinement schemes showing how the spec is met.

7.4 *Decoupling with Packages*

You could contain all the work of a project in a single package and import only whatever was determined before you started. But making smaller packages helps manage the work and, in particular, dependencies.[5]

For example, BluePhone's type Call has an understandable meaning without the idea of a Customer Invoice. On the other hand, it would be difficult to understand what an Invoice was about if you didn't know what a Call was. So we can put into one package some core ideas that must be understood by everyone in the company, and then we can build separate packages for different areas of activity. Fundamentals such as Call, Line, and Customer would go in the Phone_Basics package, and then package Billing and package Network_Maintenance would import that. Such separations make it easier to manage the development work and its products and to reduce the impact of changes in one area on another. In fact, this is the basis of the decoupling that gives OO methods their flexibility.

But the appropriate separations often become clear only when the development work has progressed to some extent. Typically, a modeling or programming project evolves a mass of classes and there comes a point when you realize that order should be imposed on it.

A useful task for a chief designer at this stage is to draw a map of dependencies and then partition them into packages with a good import structure. As we noted earlier, this means that the designer should allow no circularities and that each package should import not too many others. These principles usually imply reviewing the model and applying decoupling patterns to pry apart the pieces.

An effective tactic is to stare at the diagram and think, "Does the idea of a Hatstand make sense on its own without the idea of a Hat? Or vice versa?" and act on the results (substituting your own types for Hats). Beyond that, the tactics are a bit different for classes and types.

5. Good package design also helps with reuse and parallel development.

7.4.1 Role-based Decoupling of Classes

In a program (or an implementation model of it), the working elements are classes. Programmers have no difficulty writing a piece of software that is all classes and no types. Indeed, any program that is fairly directly based on a model of the business tends to end up with a class for every business type and not much else. This situation usually needs some remedy.

Remember that types are behavioral specifications of objects, pieces of documentation telling you only what can be seen of an object from the outside; any internal attributes are there only to help express the specification. Classes, on the other hand, are designs for objects, defining internal variables in which information is actually stored as well as program code for the operations. In a program, pure abstract classes (interfaces, in Java) represent types even though most programming languages do not provide for attributes, operation specs, and other apparatus for specification. Programs may also contain partial abstract classes, each of which may stand both to represent a type and to contain a partial implementation of it.

In a program, each object generally communicates with several others. Drawing a dependency diagram between the classes, you end up with many loops. For example, callbacks such as in an observer pattern result in loops. Simplistically, you may decide that each must know about the other, because it sends it a message (see Figure 7.13). In fact, this program is far more coupled than it need be. Each of these classes does not need to know about the "more operations" of the other class; they are used by other correspondents we've not shown here.

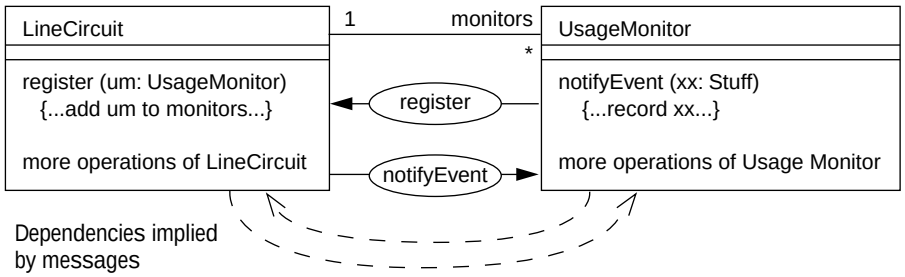


Figure 7.13 Overcoupled class dependencies.

The general rule is that, for every class, you can work out a set of types it uses by looking at the messages it sends. LineCircuit needs, not specifically a UsageMonitor, but simply something that understands the notifyEvent(Stuff) message; let's call it a CircuitListener. And UsageMonitor—well, maybe it is designed specifically to monitor LineCircuits, but maybe it could also handle a few other things. All it needs is something to which it can send the register(UsageMonitor) message that will send the right messages back.

We can do the same thing for all the other collaborations the two classes take part in. We end up with each class dependent on a set of types that represent only that behavior

each class expects of its neighbors. As shown in Figure 7.14, each class implements the types required by its correspondents (here we've drawn classes in bold, types in light). The dependencies are now from each class to the types it implements and the types it uses but not to other classes. At the object level at runtime, nothing looks different: There are still LineCircuit and UsageMonitor objects that send messages to each other. All we've done is to pull apart the design to decouple the code.

This approach enables each class to be in its own package, importing only other types and not other classes. The interface types can be further partitioned into packages with unidirectional dependencies; MonitorableElement is dependent on CircuitListener but not

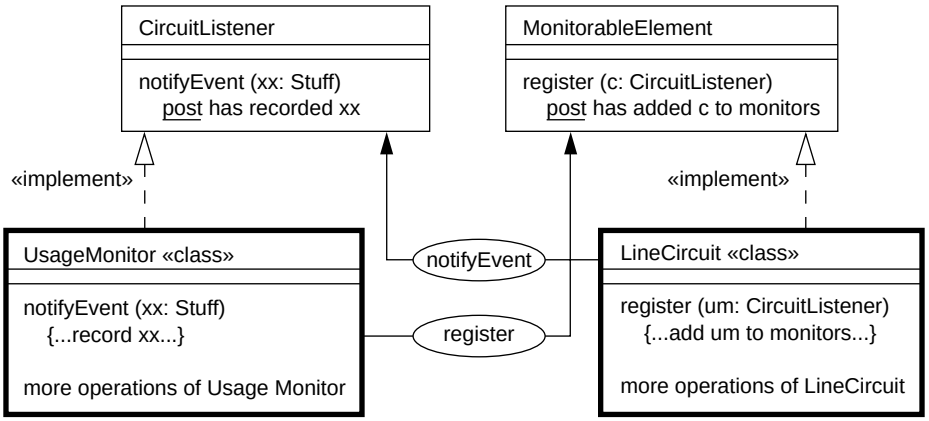


Figure 7.14 Decoupling via types.

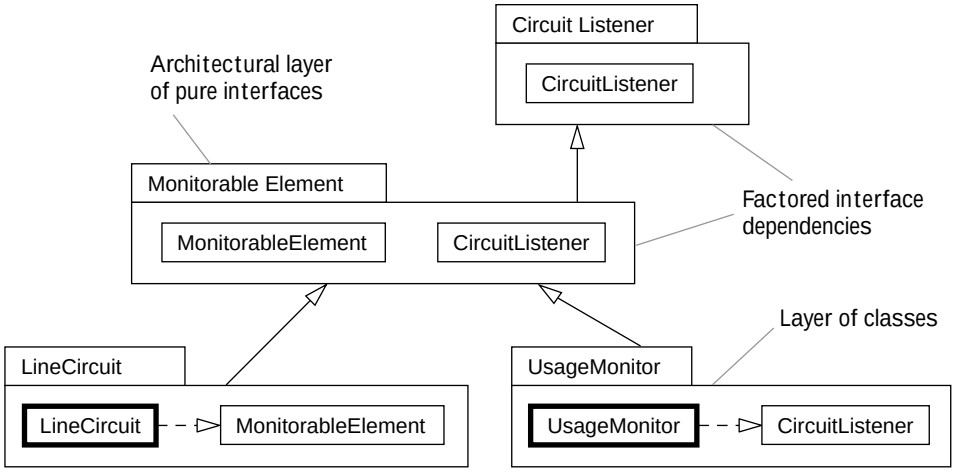


Figure 7.15 Horizontal separation: collaborating types versus classes.

vice versa. The prototypical structure is shown in Figure 7.15. (This is analogous to the usual policy with C++ header files, but stronger: A header should contain only interfaces—that is, pure abstract C++ classes.) However, it may not always be useful to go as far as one class per package. Many classes work closely with each other and will always be interdependent and modified together. Such classes should be in the same package.

To summarize this pattern, extract from each class a definition of the minimal specifications of each type of object it needs to work with. This separates the roles played by each class, and they can be packaged more easily.

It's been suggested that this pattern can form a general design policy, one that is enforced by some research programming languages: that all variables and parameters be declared as types and never as classes. The monitors variable in LineCircuit could not be declared to be a list of UsageMonitors; you would be forced to invent the supertype.

7.4.2 Decoupling in Business and Requirements Models

Let's move back to BluePhone's business model. Let's suppose that after some brainstorming, we arrive at an unstructured draft that includes the fragment shown in Figure 7.16. Although we will discuss everything in this section in terms of type definitions, the same rules apply to all definitions, including actions and nested packages. There are extra complications to deal with for programming classes.

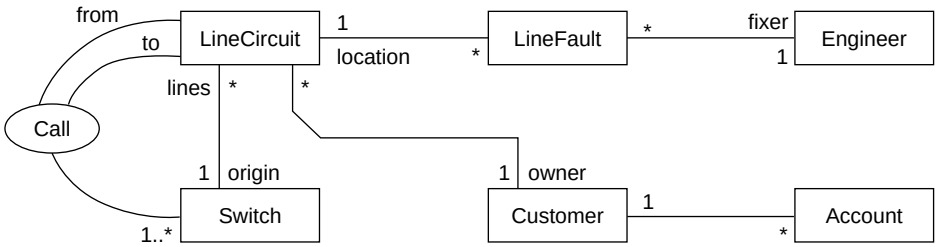


Figure 7.16 Desired overall model.

On reflection, we decide that it's possible to understand LineCircuits, Calls, and Switches without LineFaults and Engineers, but not the other way around. We want to separate the material into different packages (see Figure 7.17). As an additional benefit, these packages represent views of different departments in the company.

To see exactly what each package tells us, we can "unfold" the imports—that is, redraw everything as if it were in one package. For example, Billing unfolds as shown in Figure 7.18. In other words, within the Billing package, we can write postconditions and invariants that refer to LineCircuits, Calls, and Switches, as well as Customers and Accounts. But LineFaults and Engineers are not in scope here.

Within the Network package, we can write statements about LineCircuits and Calls and Switches but not about Customers, Accounts, LineFaults, or Engineers.

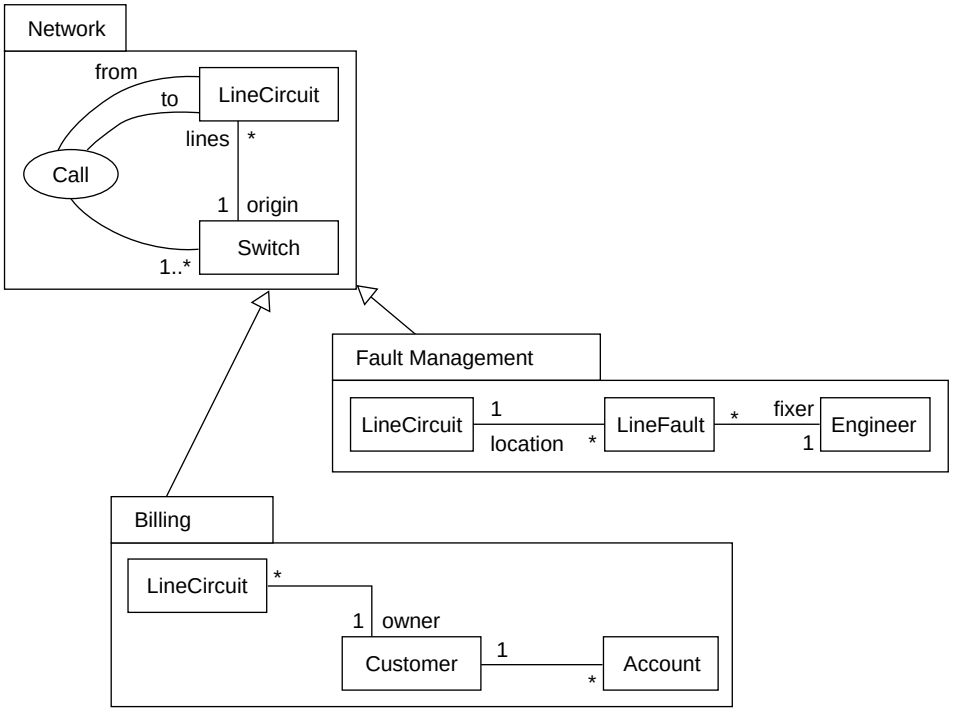


Figure 7.17 Model factored across packages with imports.

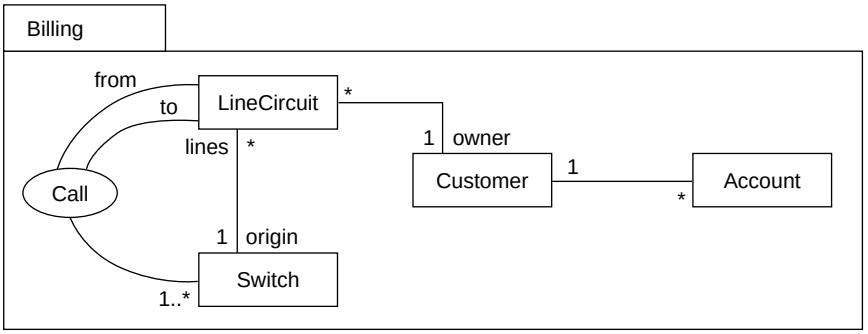


Figure 7.18 Unfolded model in Billing package.

7.5 *Nested Packages*

Any time you name an element (a type, an attribute, and so on) you must also know the scope within which that name uniquely refers to that element. A Cobol variable name

must be unique but only within one program. The Internet address systems have managed planetwide uniqueness. A Catalysis type name must also be unique but only within its package. Importing a package gives access to all names and definitions (of types, collaborations, and so on) visible within it.

What about package names themselves? A package can be nested inside one other package. A package name must be unique only within its containing package; each package has an expanded name, prefixed by the expanded name of its container package. This arrangement gives us a flexible scoping mechanism for package names; the nesting lets us deal with a group of packages as a single thing to import, move, version, and so on.

When you import a package, the short names of its nested packages become visible to you; but the names and definitions within those nested packages remain hidden until you explicitly import those packages. You can always refer explicitly to a nested package by qualifying its name with the name(s) of its parent package(s). Nested within a package, the rules about usage and extension between subpackages are the same as usual. A nested package automatically imports its parent package and hence can see its sibling package names and element names and definitions in the parent package (see Figure 7.19).

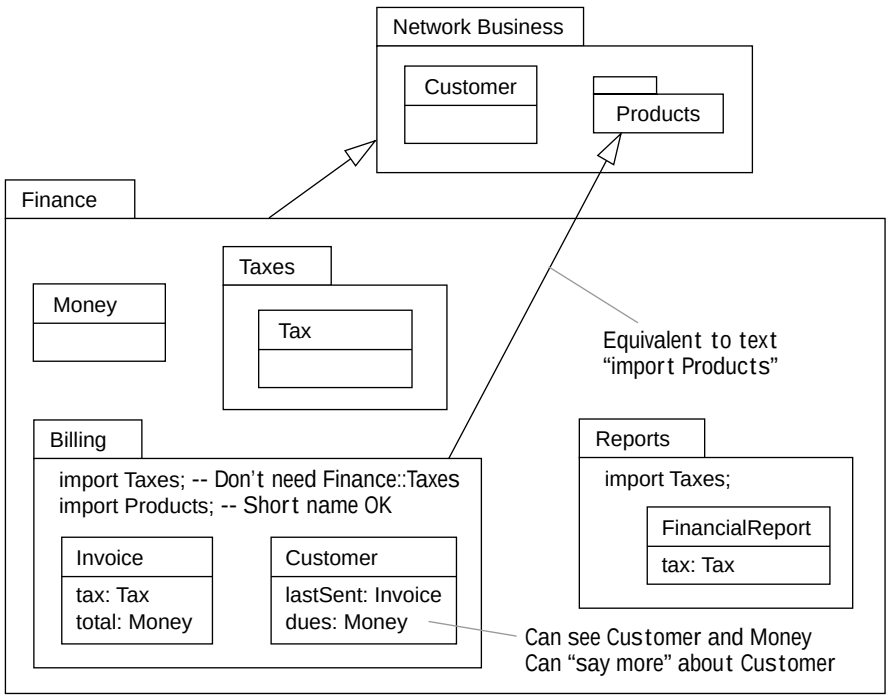


Figure 7.19 Nested packages and imports.

Importing the containing package gives you access to the names of nested packages; you can then directly refer to those you are interested in by their shorter local names, for example to import them. A package automatically imports its containing package; for example, Billing automatically imports Finance (and, by transitivity, Network Business as well).

Provided that noncircularity is observed, a package can explicitly import the following.

- Sibling packages (nested within the same container as itself; their names are directly visible, because it implicitly imports its container). For example, Billing can import Taxes.
- Packages nested within those it has imported. For example, Billing can import Products.

Notice that there is always some package containing everything, whether or not it is explicitly delineated. This illustration forms a package, a world of its own. Outside this book, there may well be definitions of things called "Finance," but we know there is no guarantee that what we and they mean by Finance are the same.

© ***package, nested*** A package whose name is itself scoped within a containing package. The contained package implicitly imports its container.

A nested package scheme could be understood as a convenient abbreviation for an unnested import structure, with additional scoping of nested package names. To translate to flattened form, qualify each package's name with that of its container and then import container packages (see Figure 7.20).

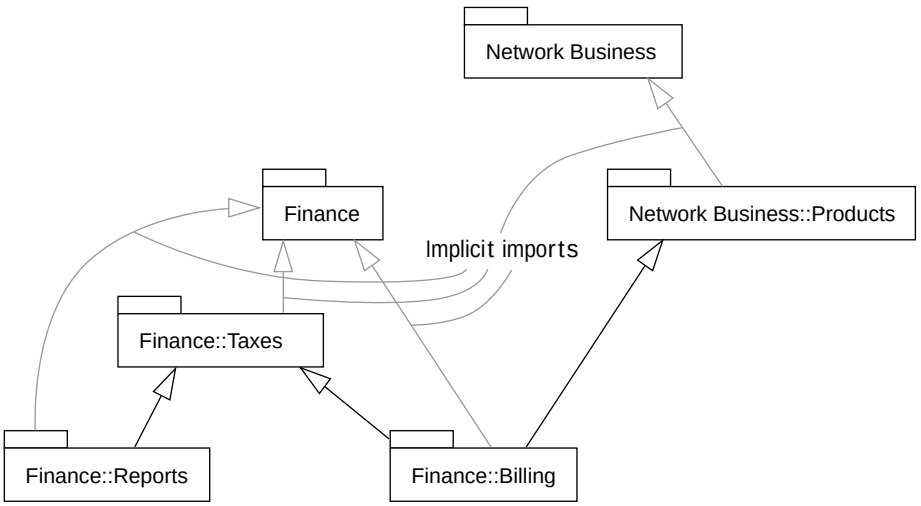


Figure 7.20 Equivalent flattening of nested packages.

7.6 *Encapsulation with Packages*

Let's look at two issues that arise with the use of encapsulation in packages: model decoupling with auxiliary definitions and the use of encapsulation in specifications.

7.6.1 Model Decoupling with Encapsulated Auxiliary Definitions

A type (or action or other definition) from a preexisting package may not directly provide all the definitions that you would like to associate with them.

Suppose in Corporate Management we issue annual reports and need an operation that compares two quarterly FinancialReports using a formula that includes the difference between taxes paid. It would have been nice if the comparison had been defined in the Finance::Reports package, but perhaps the package was written by someone else and we're stuck with it as it is. Hence, the issue of encapsulation for abstract models is similar to that for code except that we can make packages that add new material to any given type and thereby remedy an existing deficit. For example, a FinanceCompare package could import Finance::Reports and add a definition of the comparison (see Figure 7.21). Moreover, FinanceCompare could be a nested package, limiting its visibility and confining the dependency to a limited area.

7.6.2 Is Encapsulation Relevant for Specifications?

Encapsulation is not quite as important for abstract models as for implementations. Suppose we implement a position on a surface as a class with (x, y) coordinates and provide operations for moving it, finding the distance from another position, and so on. Later, we decide it would be better, instead of (x,y), to store the position as (distance from origin, angle from x-axis). We must rewrite my operation code; if I did not encapsulate my code carefully, clients that used the (x,y) variables directly would no longer work.

Contrast all this with a model of a position. First we use x, y. How important is it that other parts of the abstract model avoid using the x and y attributes? Suppose we next decide that we'd rather model positions using (r,w). First, this is less likely to happen than with an implementation: there are no issues of efficiency but only considerations of appropriateness for conveying the ideas. Second, we can just as easily leave the old model as it is while defining the new one. All it needs is an invariant to state how the two are related.[6] Now anyone can use whichever model is preferred.

If I get the model wrong, then I will have to really change it rather than only extend it; but in that case, my users would have to change anyway, and encapsulation is no protection. Encapsulation helps protect code changes when the spec remains the same; it is less effective at stopping the propagation of specification changes.

6. r*r = x*x + y*y and r*sin(w)=y and r*cos(w)=x, if we have our school math right.

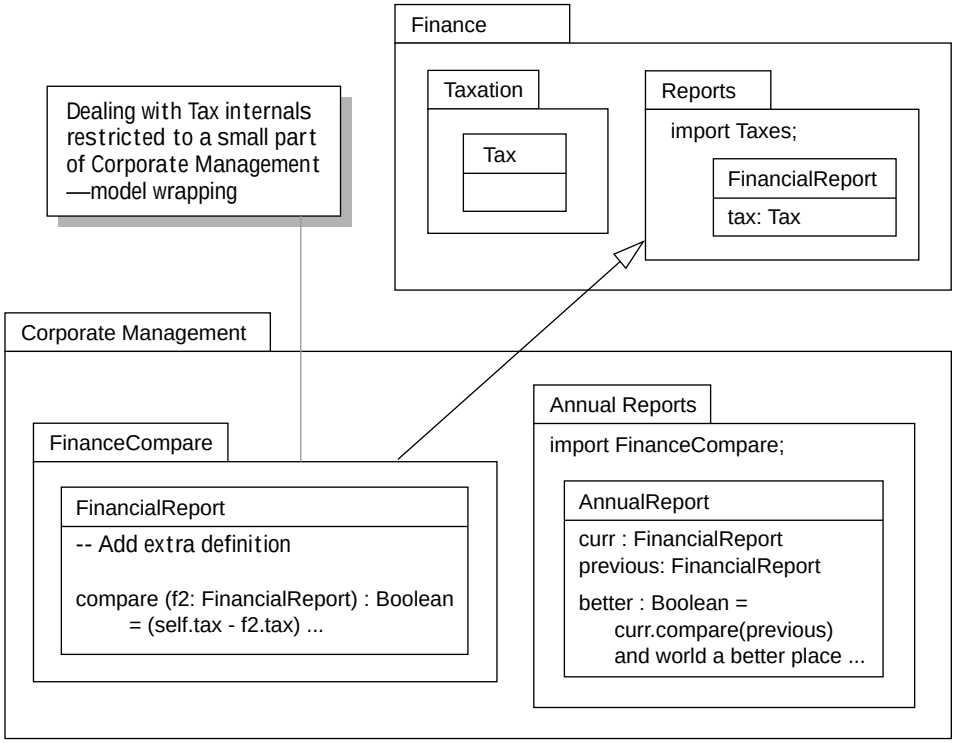


Figure 7.21 Using auxiliary definitions in packages for model encapsulation.

The same thing applies to larger pictorial models with associations. Some methodologists have argued that the tendency to draw models with several types and associations is antipathetic to encapsulation. This would be true if the boxes represented classes and if the lines represented their variables: Each class should be separately designed, independently of the others. But when the boxes are types and the lines merely abstract attributes, the issue of encapsulation is much weaker.

So we support the protest of these folks against those whose diagrams are merely pictures of their code, who simultaneously design parts that rightfully should be independent. We also support the argument against the blind use of tools that convert between pictures and code—and doubly so for people who do so in both directions and claim they have produced abstract analysis and design models.

Properly used, a type diagram is a model—and not an implementation—of a program component. In other words, it is a valid, retrievable abstraction of any correct implementation.

7.7 *Multiple Imports and Name Conflicts*

What happens when you import several packages? The unfolded importer has all the definitions of all the exporters. When the definitions in the different exporters are unrelated, this is simple.

Now suppose the BluePhone analysts extend their interest to CustomerCare and decide that the contents of that package need to refer to both faults and accounts (see Figure 7.22). If CustomerCare is unfolded, are there two copies of the definitions imported from Network? Not really.

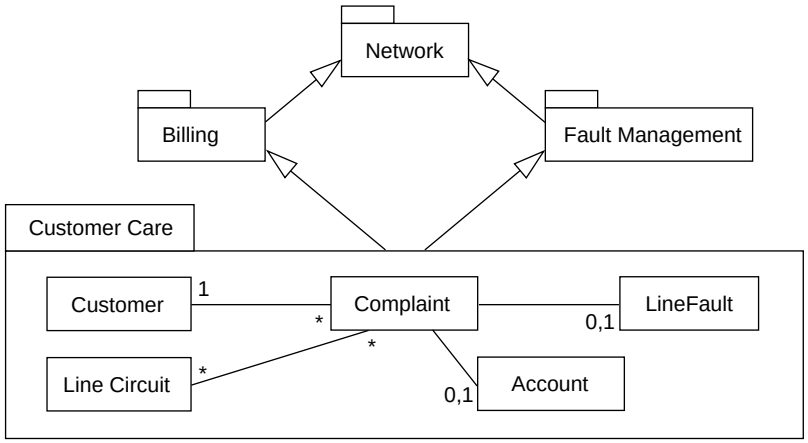


Figure 7.22 Dealing with multiple imports.

Think of each package as a set of logical statements about the various types. If it makes it any easier, remember that every diagram can be translated into a set of statements in text form. The fact that every LineCircuit has an origin that is a Switch is one of the facts stated in Network. By import, the same fact is known within both Billing and Fault Management. By import from them, it is also stated within CustomerCare. If you like, you can think of that fact being stated twice within CustomerCare; but that doesn't make it any truer, nor does it mean that there are two kinds of LineCircuit.

So packages are just lists of facts about named ideas and their relationships. A package can add new facts about a type, about which relatively little is known in an exporter. A student of CustomerCare knows that attached to a LineCircuit there is a Customer, a number of LineFaults, and any number of Complaints: all properties unknown to anyone who has read only the Network package.

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, so the CustomerCare version is a 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 give them different names? In Catalysis the answer is that "PackageName::TypeName" is the full name of a type. But we use the same type name from one package to another so that we can easily add 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 could be defined. This convention would be needlessly restrictive and would make it complex to define multiple views that could be recombined. 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, because such constraints would break subtyping rules.)

What happens if you import two packages that impose contradictory constraints? For example, one package asserts that attribute x must be > 0, and another says x< 0. The result is that you've modeled something that can't exist—something you can talk about but won't find any implementations of, such as 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 self-contradictory models within a single package: Whether importing is involved is irrelevant.

7.7.1 Name Mapping on Import

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

Conversely, sometimes we 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.

The first defense against unwanted results is that your support tools should raise a flag when you import a package that contains name conflicts (although this is not needed if the first definition of the name comes from a common ancestor package).

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 such as the one in Figure 7.23. The annotation [X \ Y] means "Rename X to Y in the importing package (the tail end of the arrow)." It doesn't mean that there is any change in the original package from which the definition comes. In the unfolded Finance package, there is an Account type that means what it does in Billing (perhaps with more information added in Finance) and a CorporateAccount type that means what Accountancy::Account did (again, perhaps Finance adds to it).

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). When the designer opts to confirm that two types with the same name should be identified, that decision can be recorded with a mapping such as [Customer\Customer]. An interactive tool will also distinguish between a name and the name's string representation.

To deal with the case when different types should be considered the same, we can either write within the importing package an extra invariant such as Client=Customer or we can use renaming. The former approach ends up with two names for the same thing, and that

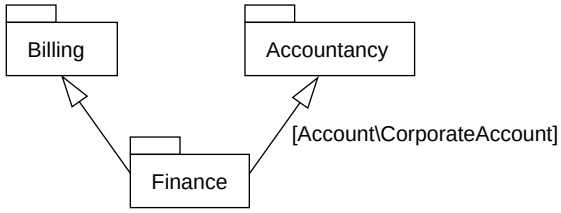


Figure 7.23 Import with renaming.

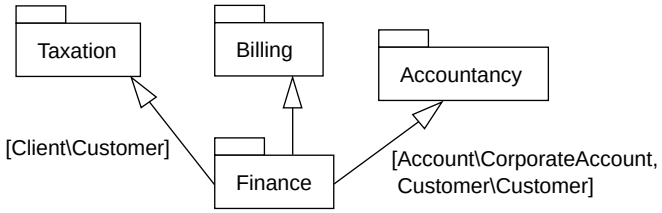


Figure 7.24 Using renaming to combine types.

could potentially lead to confusion; so the latter is preferred. A typical package diagram with renaming is shown in Figure 7.24. Import schemes can also be written in text form:

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

It is possible to write package expressions, directly composing large-grain units:

- 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 only two of the many kinds of elements that can be renamed; other named elements of a package, including nested packages, are also subject to renaming. This facility is used to help create template packages in Chapter 9, Model Frameworks and Template Packages.

7.7.2 Selective Hiding

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

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

Hiding a package hides all the names defined within it.

7.8 *Publication, Version Control, and Builds*

Packages are units that can be passed around and kept in repositories: They are the units of management of development work. It is therefore essential that different versions of a package not get confused.

The contents of a package are always evolving as long as it is being worked on. We will not discuss snapshots of that evolving package or the conveniences of a *sandbox* model that a repository-based tool may provide to insulate multiple users from the interim changes of others.

The Catalysis rule is that a package has an attribute publication state whose type at least includes Editable and Published. Once Published, a package cannot be altered and does not revert to Editable; all the packages it imports must also be Published. All you can do is release a new version or variant.

We also distinguish between versions and variants, two common relationships between packages that are revised after publication.

- A new *variant* may have any change compared with previous variants of packages of the same name. The contents might be completely different. Users know that a new variant may not be compatible with the old one.
- A new *version* is backward-compatible from a user's point of view. The new material is only an extension of the old. If the contents are program code, then the code may be different, but it should at least meet the previous version's spec.

These rules are fairly commonplace and are supported by good version and configuration management tools. Package-level versioning and configuration management can be used to structure the requirements or design for a beta version and a release version (see Figure 7.25). Nested packages (see Section 7.5, Nested Packages) make this approach even more useful, because an entire containing package can be treated as a single unit.

Packages contain everything including business models, change requests, component specifications, source code, bug reports and their test data, changes and bug fixes to code,

test specs, concrete initialization and test data, expected results, and even actual results. It becomes simple to write queries to extract what you need in a package.

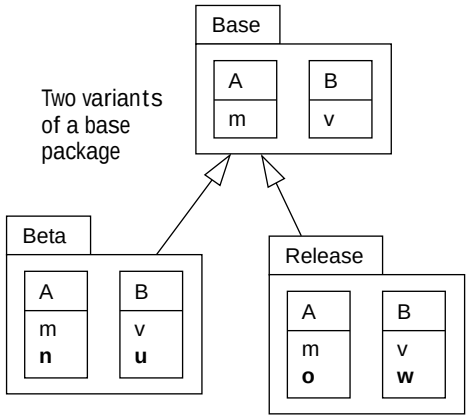


Figure 7.25 Package-level beta and release versions.

"Release 4.5 of the SunshineOS configuration of NetManic, with all changes made to fix bugs Gripe72 and Crib34 that have been approved by QA"

Or it might look like this:

"All interfaces that are believed to be affected by change request RedoItAll81"

A package is also a unit of *build*. Traditionally you perform a build by running various external programs (compilers, linkers, test scripts, documentation tools, and so on) over the contents of the package—to create various derived elements such as executable code and test results—and, recursively, over the contents of imported packages. In Catalysis, each package is associated with a special function called its build function, which, given a BuildEnvironment, creates a resulting BuildObject.

Package::build (BuildEnvironment): BuildObject

The BuildEnvironment enscapsulates external development environment specifics about doing the builds. This technique keeps system construction modularized and separates the logical function that creates various derived artifacts (such as compiled code) from any specifics of the environment in which the build is taking place. The underlying machinery for evaluating a build function may cache information, perhaps using traditional versioning, time-stamping, or facilities such as make.

Packages can be used to deal effectively with variations and versioning across members of a product family. You build base packages containing either models or framework implementations. Each member of that product family then imports that package. Because a Catalysis package can "say more" about any imported element—including types, actions, and refinements—each product can define a variant suited to its own needs. Configurations and versions are defined by the import structure (see Figure 7.26).

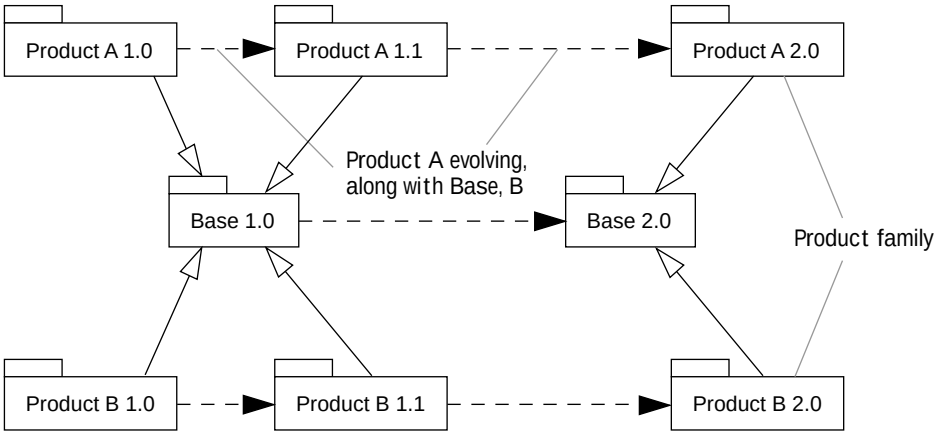


Figure 7.26 Packages with product families and evolving versions.

7.9 *Programming Language Packages*

Programming languages have a variety of forms of packaging.

Java packages can contain any combination of interfaces (types) and implementations (classes). Augmentation and redefinition of classes or interfaces across packages are not allowed. A Java package or group of packages can be used to implement the requirements expressed by a model package. A sensible use of Java packages to clearly layer the architectural design would be that from Section 7.4.1, Role-based Decoupling of Classes.

A C++ include file is a primitive form of package. Header files or groups of them can implement requirements expressed by a Catalysis model package. ANSI C++ has the namespace construct, which provides somewhat better packaging facilities.

In Smalltalk/Envy, an executing system is made up of a series of packages (or "applications") loaded in a specific order, starting with the kernel, which defines all the primitives. Each package can extend existing classes and redefine specific methods.

Smalltalk (in its marketed version) has no types, so types are at best a matter of documentation. A programmer can associate model packages with the code packages or can use conventions to group sets of messages to define named types.

The classes in a code package can be documented as implementing the requirements specified in a model package. If certain principles are followed, this can be preserved even though other code packages may be overlaid. The main points are as follows.

- Each redefinition of a method should satisfy all the specifications, from different model packages, that refer to that method.
- The packages loaded onto a particular target system may be different from those where any one package was designed. Nevertheless, packages that were designed indepen-

dently should not be allowed to interfere with one another: A package may not redefine any method code that was not redefined as part of its own design.

7.10 *Summary*

Packages are the containers for all development work, from models to documentation to code and test (see Figure 7.27). Packages define units of versioning, configuration management, and reuse.

A package declares names and asserts facts about those names. The declarations and facts may be in pictorial form. A package has an associated dictionary or narrative (or

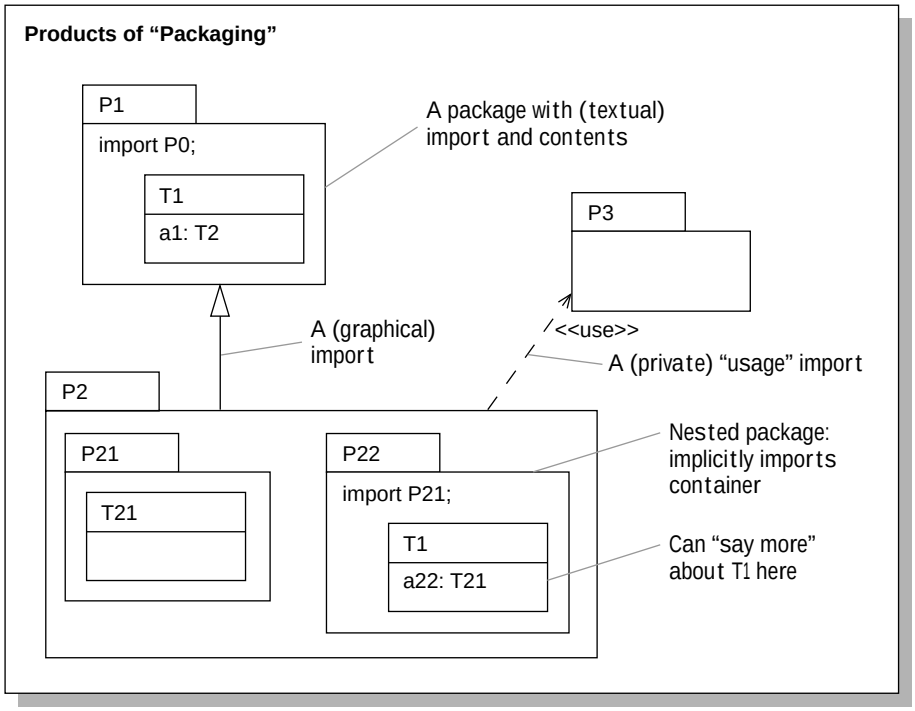


Figure 7.27 Packages.

both), which explains the formal assertions in more readable language. The more formal presentations are used to disambiguate the text.

Packages import other packages either to extend them or to make private use of their contents. The import structure is a fundamental part of planning a project.

The purpose of packages is to make explicit the dependencies between different areas of the development work. A variety of patterns can be applied to help separate concerns, including vertical separation of different views, horizontal separation of different architectural and business levels, and code separations of interfaces from implementations.