Chapter 7 Abstraction, Refinement, and Testing

Outline

Abstract diagrams are good for the whiteboard. You don't want to bore your colleagues with your code (we hope): rather, you want to exhibit the main ideas of how it works. Or maybe you just want to say what it does, not how; or maybe, what it requires of other components that can plug into it. And whether it's a single procedure or an entire planet-wide distributed system, you want to fit it all on that board and convey useful things about it.

The same goes for documentation. A document that makes clear the essential vision of your design will help future maintainers understand it quickly: their short-order updates will be more likely to cohere with the rest of the design, so giving your brilliant ideas a longer life. And for users of the component (whether end-users or other programmers), you want to show not how the thing works inside, but just what it can do.

We've seen how to draw and write these abstract descriptions; but how can you be sure they accurately represent the code you've written? The concepts of abstraction and refinement capture the essential relationship between these descriptions. This chapter is about different forms of abstraction and refinement, and the rules for checking that a more detailed design or spec *refines* (or *implements*, or *conforms to*) a more abstract one.

One result of the approach to abstraction and refinement is that testing falls out as a natural by-product. This chapter also discusses how to test different kinds of refinement relations, including the one between the implementation of an operation and its specification.

7.1 *Zooming in and out*

7.1.1 Objectives

Abstraction is the most valuable design tool

A major theme of Catalysis is the ability to look at a design or a model in a variable degree of detail. You can describe your code to your colleagues in documents or in presentations, or just sketching over coffee, without getting into superfluous detail.

Abstraction horizontal and vertical

One of the properties of the abstract views is that they can isolate different concerns: the behavior of a component can be presented from the point of view of a particular user, or another component, leaving out the details seen by another user, or through another interface. You can restrict yourself to the externally-visible behavior, or you can go inside and describe the overall scheme of the design. You can describe just one component, or how the components fit together; or you can describe particular architectural conventions observed by all the components.

Refinement and conformance provide tracability

The more 'zoomed out' abstract views and the more 'zoomed in' detailed views can be related to one another. For example, you can tell whether the code for a component conforms to the interface expected by those it is coupled to. You can tell whether a system, if designed to behave as specified, would really contribute to the business needs. You can link individual requirements to specific features of a design. And you have a much better start on change-management.

Just-enough rigor for verification

The degree of rigor of this traceability is itself variable. In a very critical context, you can do these checks in mathematical detail. In more ordinary circumstances, you just outline the main points of correspondence in the documentation, to guide reviewers and maintainers.

...and testing

Testing of object and component designs can be even more difficult than more traditional systems, exacerbated by polymorphism, inheritance, and arbitrary overriding of behaviors. The essential idea of testing is to verify that an implementation meets its specification — the very same goal as that of refinement; except that testing tackles the problem by monitoring behaviors under a (hopefully) systematically derived set of test cases. This chapter also outlines such a systematic approach, based on refinement.

7.1.2 Terms

We will use these terms:

Abstraction of	(1) A description that leaves out some detail. A specification or high-level design.
	(2) The act of creating an abstraction.
Refinement of	(1) A more detailed description that conforms to another.
	(2) The act of creating a refinement.
Conforms to	One behavioral description conforms to another iff any object that behaves as described by one is also behaving as described by the other.

Implementation Program code that conforms to some abstraction: requiring no further refinement.

7.1.3 Abstract specifications and abstract designs

Catalysis is a about being able to make precise statements at different chosen levels of detail about either the behavior expected of objects (large and small), or the designs for object collaborations; and about being able to relate the different levels of detail together.

Describe one object's behavior, or a group collaboration

7.1.3.1 Behavior

We describe an object's behavior using types. A type has two parts: the operation specifications (usually pre / postconditions) defining what it does; and the model, providing the vocabulary of terms for the operation specs.

A type specifies behavior: model + operations

An abstract behavioral spec is a bit like operating instructions for a machine: it tells you what to expect as a user. But if we lift the top off the machine, we see the rather different picture that explains how it works. That is what happens when we look at the class code that meets a type spec.

The external spec and internal design may differ

If the type model and the class look rather different, we need ways of determining whether the class conforms to the type. We can treat the two main parts of a the descriptions separately: giving us "operation abstraction" and "model abstraction".

This gives 2 kinds of behavior abstraction

7.1.3.2 Collaborations

We describe designs for objects using collaborations — collections of actions. An action defines a goal achieved collaboratively between the participant objects, using postconditions whose vocabulary is the models of each of the participants. Typical collaborations include everything from business interactions ("banks trade stocks") through hardware ("fax sender sends document to receiver") to software components ("scrollbar displays file position").

Collaboration describe design of interactions

A very abstract action-spec just states the goal achieved, and the participants it affects. But we can put more detail into the design, by working out in more detail the protocol of more detailed interactions between the participants. Typical refinements might be into collaborations such as: banks make deal, confirm, settle; fax connects, sends page, confirms, repeats; file notifies scrollbar on change of position.

Each action can be refined to collaboration

And we may also discover that each of the participants is made up of distinct parts. The banks' traders make the deal, but their back offices settle.

Each participant can be pried apart

Again, the two aspects of the collaboration can be treated separately, giving "action abstraction" and "object abstraction".

This gives 2 kinds of collaboration abstraction

7.1.3.3 Four basic kinds of abstraction

Most abstractions can be understood as combinations of the four basic varieties we've just identified:

Total of 4 basic abstraction kinds

- Operational — Specifying what an operation achieves in terms of its effect on the object executing it, rather than how it works.
- Model — Presenting the state of an object (or component) as a smaller and simpler set of attributes than the actual variables or fields used in the design; or simpler than a model that presents a more detailed view.
- Action — Presenting a complex protocol of interaction between objects as a single entity, again characterised by what effect it has on the participants.
- Object — treating an entire group of objects (such as a component or subsystem or corporation) as if it were one, and characterizing its behavior with a type.

We'll now take a brief look at each of these abstractions. Later sections will treat them in much more detail.

7.1.4 Operation abstraction

Operation spec captures net external effect

One of the most basic forms of abstraction is the operation specification: the idea that I can tell you some of the things that are achieved by an operation, omitting the detail of how it works; and omitting things it achieves that I don't know or don't care about, such as sequences of algorithmic steps, intermediate states, variables used, etc. The operation specs we usually encounter are pre / postconditions; though they may also include rely / guarantee conditions. (Rely: a condition the designer assumes will remain undisturbed by anyone else while the operation is executing; guarantee: a condition the designer undertakes to maintain true during execution.)

They enable testing of operations

Operation specs can be used as the basis of a test harness: they can be incorporated into the code in such a way that an exception is raised if any of them ever evaluates to false. (Eiffel supports pre and postconditions directly; the standard C++ library includes an assert macro.)

```
float square_root (float y);                    // specification
// pre:       y > 0
// post:      abs(return*return – y) < y/1e6          -- almost equal

float square_root (float y)                     // implementation
{
    assert (y > 0);                             // precondition
    float x= y;
    while (abs(x*x–y)>=y/1e6) { x= (x+y/x)/2; }          // miracle
    assert (abs (x*x – y) < y/1e6);             // post: x*x == y (almost)
    return x;
}
```

Op specs can be checked explicitly e.g. in debug mode

Quality Assurance departments like them, because apart from helping to document the code, they act as a very definite test harness. In a component-based environment, you frequently plug components together that were not originally designed together:

integration testing becomes a much more frequent activity. Every component (which might mean individual objects or huge subsystems) therefore has to come with its own test kit, to monitor its behavior when employed in some new configuration.

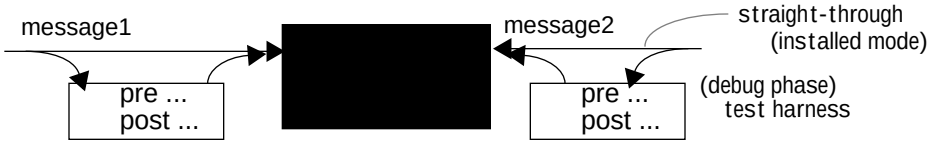


Figure 113: Operation specs: in testing vs. in production

In Catalysis, we characterize behavior with type definitions, attaching postconditions to the operations. In Java, the corresponding construct is the interface: classes that implement an interface must provide the listed operations. Wise interface-writers append comments specifying what clients will expect each operation to do, and classes that claim to implement the interface should conform to those specifications, even though each will do so in its own way. In C++, the pure abstract class plays the role of the Java interface in design.

Op specs influence Java interfaces and C++ abstract classes

Operational refinement, then, just means writing code that conforms to an effects spec; which can be tested by writing the spec in executable form.

Refinement decides how to meet the spec

(Operational is the form of abstraction with the longest history, dating back to Turing. Those that crave mathematical certainty that their code conforms to the spec are referred to [Morgan] or [Hoare].)

7.1.5 Model abstraction

The postconditions of the operations on an object usually need to refer to attributes that help describe the object's state. It's very difficult to describe any but the most primitive types without using them. We have also seen how we use associations as pictorially-presented attributes.

Op specs need attributes to abstract state

And of course, any implementation will also use internal stored variables, operated on by the code of the operations. But they need not be the same as the model attributes used by the postconditions. Model attributes are just hypothetical means of describing the object's state, that help explain its behavior.

These can be different from stored variables

For example, you might use the concept of its length to help describe operations on a queue (of tasks, orders, etc.).

Example: queue

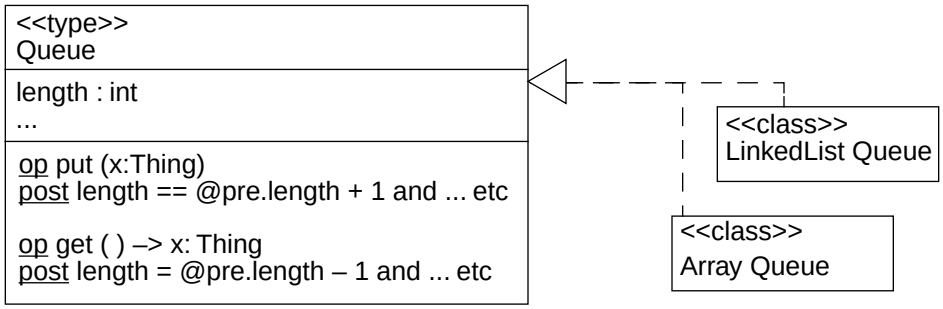


Figure 114: Different implementation of the same type model

Implementation may not directly have 'length'

We can think of several implementations of a Queue. But, not all of them will have a length instance variable or method. Nevertheless, it is undeniable that every Queue does have a length. That is what an attribute is about: it is a piece of information about the object; and not necessarily a feature of any implementation.

...as long as length can be 'retrieved'

An attribute can always be 'retrieved' from an implementation. Suppose I publish the above type on the Web, and invite tenders for implementation. A hopeful programmer sends me an array implementation:

```
class ArrayQueue implements Queue
{   private Object array [ ] ;          // the list of items
    private int insertionIndex;         // where items are put to
    private int extractionIndex;        // where items are got from
    public void put (Object x)
    {   array[insertionIndex]= x;
        insertionIndex= (insertionIndex+1) % array.length;
    } ...
```

Retrieval = Abstraction function

As a quality assurance exercise, I want to check whether the code of put (say) conforms to my spec. But my spec and this code talk in different vocabularies: it has no length. I need to translate from one model to the other — map to the abstraction — before I can begin. So I write back to the programmer and say, "Where's your length?" The answer comes back:

```
private int length ( )
{ return (insertionIndex – extractionIndex) % array.length; }
```

With the 'retrieval', the op spec on attributes can execute against the implementation

In other words, I have been provided with a function that 'retrieves' or abstracts from the implementation's terms to the spec's. (And of course, it is read-only: it would be confusing if it changed anything.) *Now* I can see that the put code does indeed increase the length, as I required. And, if working that out just by looking at the code feels a little too much after a heavy lunch, I can ask that the designer please to include my pre and postconditions as assertions in the code, so that we can test it properly.

Which is great for testing

Again, quality assurance chiefs love this stuff. They will demand that every implementation be supplied complete with a set of retrieval functions: that is, a read-only 'abstraction function' for computing the value of every attribute in the abstract spec. This applies to associations as well, of course, which we have previously observed are just pictorial presentations of attributes.

Another implementation of a Queue is based on a Linked List. There is not necessarily any variable that corresponds directly to the model's length: but you can retrieve it by counting the nodes.

The 'retrieval' can be inefficient — thats OK

It doesn't matter if retrieval functions are very slow and inefficient: they are only required for verification, either by testing or by reasoning. The exercise of writing them often exposes mistakes in an implementation, when the designer realizes that some vital piece of information is missing.

Notice that in Catalysis, we do not expect that attributes / associations will always be publicly supplied for use by clients. There are, of course, many attributes which it is useful for clients to have 'get' and sometimes 'set' access to; but modelers often use attributes to express intermediate information they don't expect to be available directly to clients.

Not all attributes need be publicly accessible

On a larger scale, more complex models can be used to represent the types of whole systems or components, and are usually shown pictorially. In an abstract model, the attributes and their types are chosen to help specify the operations on the component as a whole, and according to good object-oriented analysis practice, are based on a model of the domain. But anyone who has been involved in practical OOD is aware that the design phase introduces all sorts of extra classes, as patterns are applied to help generalize the design or make it more efficient; and to distribute the design, provide persistence, a GUI, and so on. But we can still retrieve the abstract model from any true implementation, in the same way as the simpler models.

Attributes and their types reflect the domain model; implementation may differ

Model refinement, then, means establishing the relationship between the more abstract model used to define postconditions, and the more complex practical implementation. Retrieve functions translate from the refined model attributes to the abstract ones.

Model refinement relates concrete to abstract state

(Model refinement has the second longest history, dating back to VDM and Z in the 1970's. See [Jones86] or [Spivey].)

7.1.6 Action abstraction

7.1.6.1 Messages and actions

At the programming level, 'messages' are the interactions between objects, in most OO programming languages. Some have variants on the basic theme, such as synchronous vs. asynchronous (waiting for the invoked operation to complete, or not). Complex sequences of messages can be difficult to envisage, so we draw sequence diagrams:

'Message' is a good programming construct

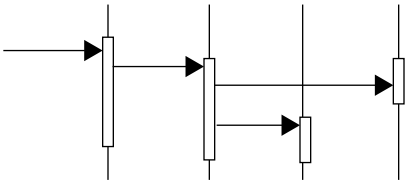


Figure 115: Sequences of messages realize abstract actions

Sequence diagrams have the disadvantages of being bad for encapsulation, particularly when multiple levels of calls are shown on one diagram. They allow you to see in one diagram the response to various messages of several objects at once: and so encourage you to base the design of one object on the internal mechanisms of the others. They also show only one sequence of events, and so make it easy to forget other cases, and that each object may receive the same messages in other configurations. However, they do make it easy to get an initial grip on a design, so we use them with caution. It is often good practice to limit the diagram to just one level of nesting, and to use postconditions to understand what those achieve.

Nested sequence diagrams may encourage dependence on internal mechanisms

The same diagrams can be used to show the interactions between objects in the business world; and interactions between large components.

But abstract actions have many possible message-level protocols

But in any but the most detailed level of design, we usually deal in actions: dialogues with an outcome definable with a postcondition, but made up of messages we do not care about. For example, I might tell you "I bought some coffee": rather than expect you to listen to a long tale about how I approached a vending machine, inserted several coins, pressed one of the selector buttons, and so on. The former statement abstracts the latter detailed sequence, and includes any other means of achieving the same effects. In Catalysis, actions are characterised principally by their effects; to show how an action is achieved by some combination of smaller ones, you document a refinement.

An abstract action may also involve many participants

A single action may involve several participants, since it may abstract several smaller actions which may have different participants. On a Catalysis sequence diagram, the small ellipses mark the participating objects (vertical lines) in each occurrence of an action (horizontal lines). We also mark the states or changes of states of participants:

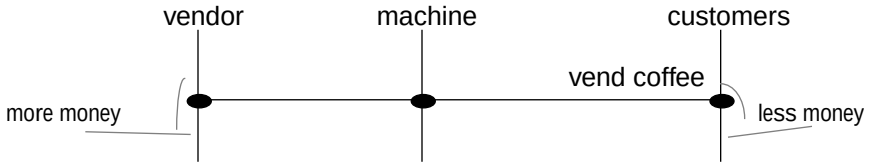


Figure 116: Occurrence of joint action

OOP messages are a special case of actions

OO messages in programming languages are just one subtype of actions, that always have a distinct sender and receiver (Section 5.4.1, "From Joint to Localized Actions," on page 213).[1] We can use action abstractions to represent interactions external to the software; and between users and software; and between objects within the software. Within the software, they are very useful to abstract the standard interactions that happen within certain frameworks and patterns, such as 'observer'.

You can draw and specify an abstract action

Sequence diagrams illustrate sample occurrences of actions; but an action type can be drawn with the types of its participants, and a postcondition written in terms of the participants' attributes (again, whether the types represent software or domain objects):

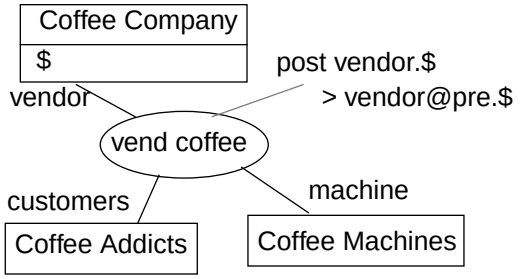


Figure 117: Action type shown on a collaboration

1. CLOS is an exception: its operations are not attached to any particular receiver. Like Java and C++, there may be several operations with the same name and different parameter lists; but the operation to be executed is chosen at run-time on the basis of the classes of all the parameters (not just the receiver, like the other languages). Catalysis works as well for CLOS as it does for Java, etc., because multi-receiver messages are a type of action.

The ability to define the effects even at an abstract level is what makes it worth doing. Abstraction without precision is often just waffle; until some precision is used, it is typically not reliable.

The abstract definition is precise, hence reliable

7.1.6.2 Refining actions

We can ‘zoom into’ or refine the action, to see more detail. What was one action is now seen to be composed of several:

Action refined sequentially

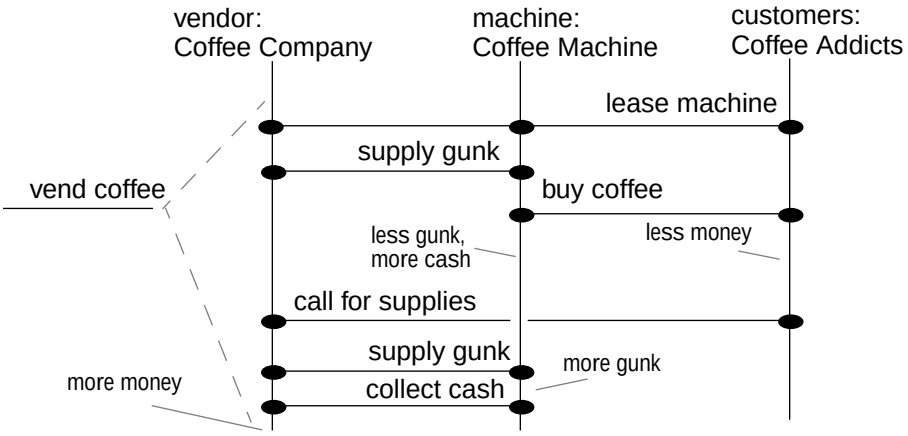


Figure 118: Refinement into further joint actions

Each of these actions can be split again into smaller ones, into as much detail as you like. Some of the actions might be performed by software; some of them might be performed by some mixture of software, hardware, and people; some of them might be the interactions between those things. At any level — deep inside the software or at the overall business level — we can treat them the same way. Catalysis is a ‘fractal’ method: it works the same at any scale.

Recursive and fractal method

Notice that the abstract action has not gone away: we have just filled in more detail. It is still the name we give to the accomplishment of a particular effect by a combination of smaller actions, and that effect is still there.

Abstractions are correct descriptions of their refinements

These illustrations might suggest that an action is always made up of a sequence; but often, the composition is of several concurrent processes that interact in some way. Recall that ‘action’ is the Catalysis blanket word for process, activity, task, function, subroutine, message, operation, etc.

Actions can be refined to concurrent actions too

Abstract and refined actions can be connected by ‘aggregation’

We can summaries any kind of action abstraction on a type diagram, showing which collaboration (set of actions) can be abstracted into a single abstract action:

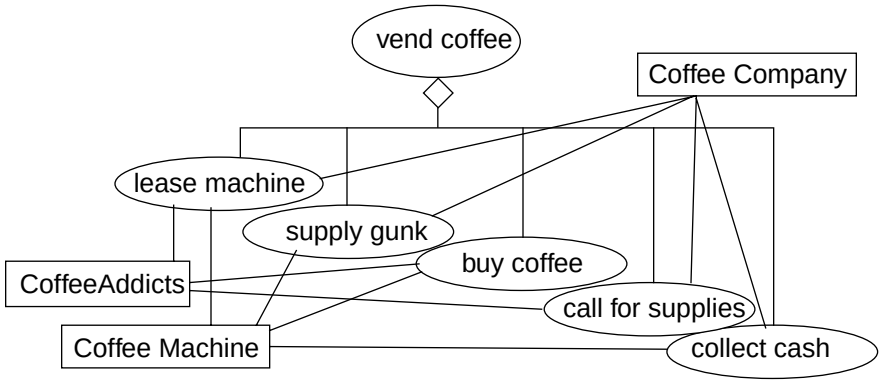


Figure 119: Refined actions shown using "aggregation"

The diamond aggregation (“part of”) symbol is used to indicate the fact that the abstract action is an encompassing term for compositions of the smaller actions. It shows constituent parts of the abstraction, though we have to state separately how they are combined, whether in parallel or some sequence, whether some are optional, or repeated. Also, another diagram may show a different set of detailed actions abstracted to the same abstract action.

7.1.6.3 Summary: action abstraction

Hence, dialog details are deferred

Action abstraction is the technique of treating a dialogue between several participants as a single interaction with a result definable by an effect.

Slightly different from operation abstraction

Action abstraction and operation abstraction are both about treating several ‘smaller’ actions as one. The differences are that:

- In operation abstraction, the initial invocation resulted in a sequence of smaller actions determined by the design; the abstraction still has the same initial invocation, and captures the net desired effect.
- In action abstraction, no invoker need be identified: any participant could initiate it. The exact sequence is determined by the designs of all participants (some of whom could be human); all we can say is what the smaller actions are, and constraints on how they might be combined. Also, the abstract action may never be directly ‘invoked’ at the detailed level.

Action abstraction comes from the idea of transaction, developed in the database world in the 1970s; though of course it is very natural in everyday conversation.

7.1.7 Object abstraction

A group of objects may be treated as one

We can refine or ‘zoom into’ objects, splitting any of the vertical bars on the diagram into several. Thinking about it in more detail, we realize that the vendor Company has different departments that deal with different parts of the business. Also, the customer organization will have both users, and a site manager who liaises with the vendor:

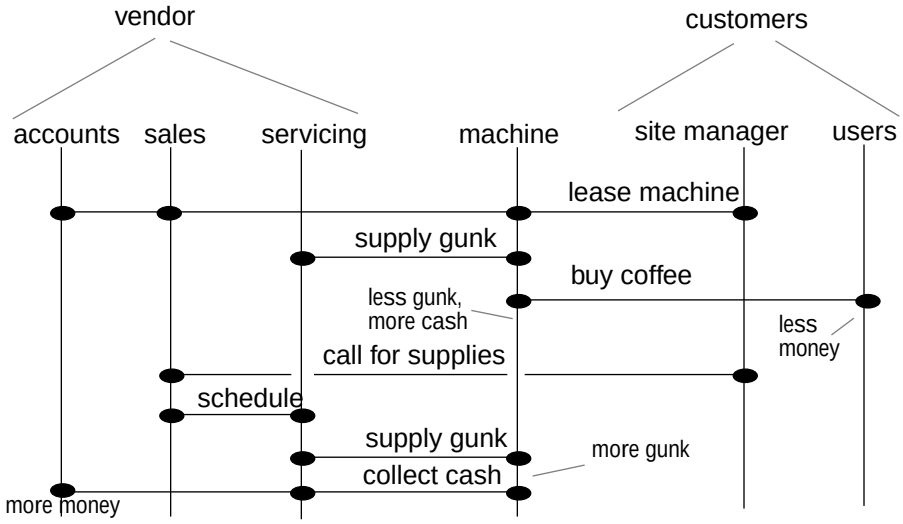


Figure 120: Abstracting objects

The abstract objects are ‘virtually’ present

Again, the more abstract objects have not gone away: the Company is still there; it is what we call a particular configuration of inter-related smaller objects. Once again, each object may be split further into smaller parts: servicing may turn out to be a department full of people with different smaller roles.

There may be many valid object refinements

Just as what characterizes an action is its effect, so what characterizes an object is the set of actions it takes part in — its type. There may be many refinements that will satisfy one abstract object type. (Indeed there may be refinements that successfully satisfy several roles: some small companies have just a single object that plays the roles of accounts, sales, and servicing.)

This applies to software and domain objects

Some of the objects may be software components. For example, zooming in on the accounts department would probably reveal some combination of people and computers, and more detail on the computers would reveal a configuration of software packages, and going into them would reveal lines of Cobol, or, if we’re lucky, objects in an OO language. The same would happen if we peered into the coffee vending machine. And we can continue using the same interaction sequence diagrams (and other tools), right down into the software.

Abstract objects 'aggregate' their refined group

The relationship between an abstract object and its constituents can be shown on a type diagram:

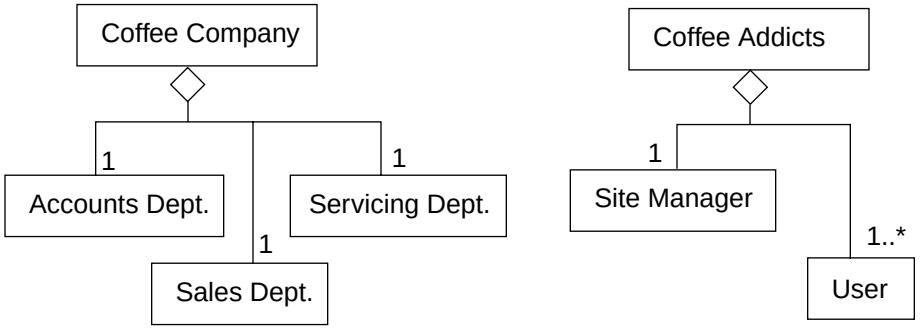


Figure 121: Object refinement shown as "aggregation"

The diamond again indicates the refinement relationship; and once again doesn't by itself give us every detail about what constraints there are between the constituents of one abstraction.

7.1.7.1 Summary: object abstraction

Hence, object details are deferred

So object abstraction means treating a group of objects, whether in software or in a human organization, or a mechanical assembly, as one thing.

The idea is as old as language, and was discussed by Plato, Michael Jackson [Jackson 95, *Individuals*], and others — often beginning with "how many parts of a car do have to change before it's a different car?" or "are you the same person as you were when you were born?" The examples serve to highlight that the identity of anything is, in the end, a model constructed for our convenience, rather than something inherent.

7.1.7.2 Abstracting objects and actions together

Actions and objects often refine together

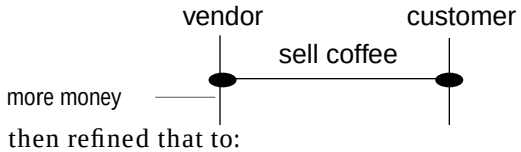
It is usual to zoom in or out in both dimensions at the same time. As soon as you resolve each object into several, you have to introduce interactions specifically with, or between them. Conversely, when you put more detail into an action, you need more objects to represent the intermediate states between the actions.

When we looked inside the vendor Company, we assigned sales to receive supply requests from the customer, but they then schedule a visit by servicing.

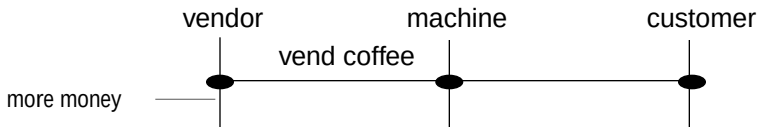
By contrast, we left the collection of cash as one action involving both servicing and accounts. That just means we've deferred until later some decision about how that action splits into smaller steps.

Zooming into an action often involves more objects. As an example, let's go the other way, and abstract the original 'vend coffee' action. If the overall requirement is just to get money out of people by giving them coffee, then there are more ways of doing it

than by installing a machine near them: you could run a café or street stall. So we could have started with:



and then refined that to:



— although the machine is only really required when we go to the more detailed scenario of exactly how we're going to sell it.

7.1.8 Zooming into the software

There are refinement 'zooms' purely at the business level

In Catalysis, the standard pattern for developing a software system or component is to begin with the business context, and represent the business goals in terms of actions. Then we look at how these goals are met by a more refined view, that shows the various roles within the business, and their interactions with each other.

...and down through software interactions

Some of these interacting objects may be computer systems. We can treat them as single objects, and describe their interactions with the world around them. Subsequently, we refine the system into a community of interacting software objects. (A standard OO design is based on a model of the external world, so that we now have two of everything: a real coffee cup, user, and coin, plus their representations inside the software.)

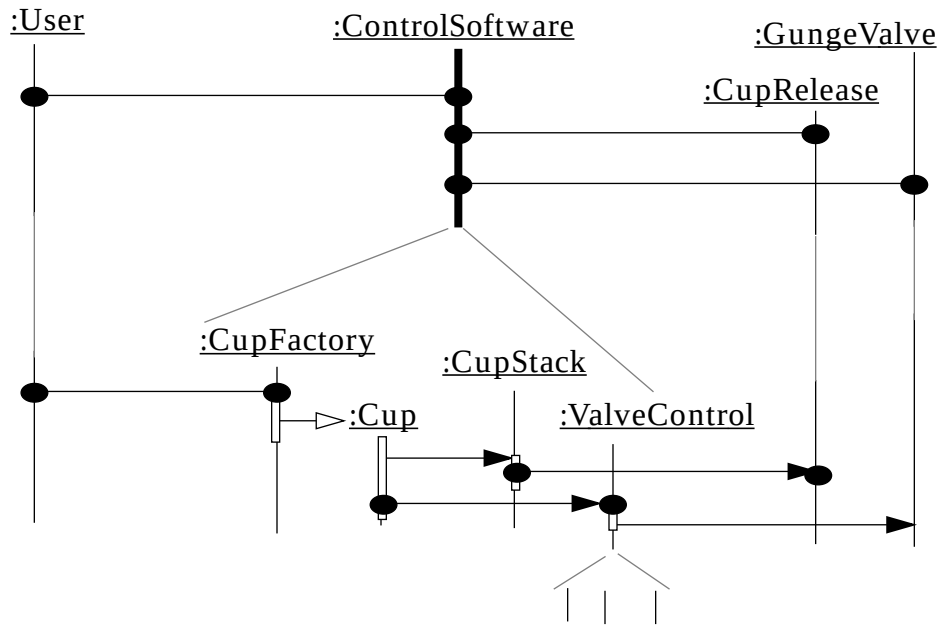


Figure 122: A continuum of refinement: real-world to code

The same abstractions apply in software

When we look inside the software, we can continue to use the more general multiple-participant actions, and objects that actually represent whole subsystems; but ultimately, we get down to the level of individual message-sends between pairs of objects.[1]

Once again, the sequence diagrams help us illustrate particular cases, but we prefer to document the refinement using type diagrams. This gives us software component context diagrams:

Figure 123: Context diagram (collaboration) for one software compor

and high-level design diagrams:

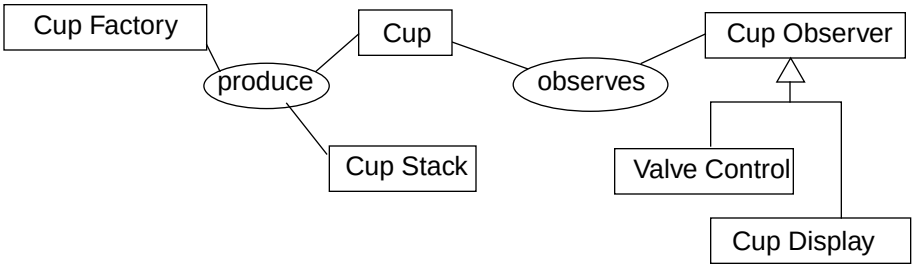


Figure 124: Collaboration between software components

7.1.9 Reified and virtual abstractions

Some abstractions will also appear in a concrete form

When designing objects inside the software, you have a choice about whether to *reify* abstract objects and actions: that is, represent them directly as software objects; or whether just to leave them as ideas in the design that help you understand the various configurations of objects and protocols of messages.

A 'system' object is common, but not all requests pass through it

Representing each entire system and each component as an object is usual, because it gives something to anchor the parts to. A more difficult decision is whether to make it the *façade* for the group of pieces it represents — that is, through which all communication with the outside world should go. Sometimes this works well: it helps ensure everything inside the component is consistent. Sometimes it is not efficient — any more than insisting that every communication with customers should go directly through the president of your company.

1. A significant difference between Catalysis notation and UML version 1.1 (current at the time of writing) is that UML does not have generalized actions in sequence diagrams; nor are the messages in a sequence diagram understood as instances of use-cases. However, at the message-passing level, our sequence diagrams are the same as UML.

7.1.10 Composition and Refinement

Compositional techniques are elsewhere

Other forms of abstraction of abstraction and refinement, more related to the separation and subsequent joining of multiple views, are covered in Chapter 9, *Composing Models and Specs* (p.367).

7.1.11 Summary: kinds of abstraction

We need traceability through different abstractions

An abstraction presents material of interest to particular users, leaving out some detail, but without being inaccurate. Abstractions make it possible to understand complex systems; and to deal with the major issues before getting involved in the detail. In Catalysis, we can treat a multi-party interaction as one thing; or a group of objects as one; and at different layers of detail, we can choose to change the way we model the business and systems. But through all these transformations, we can still trace the relationships between business goals and program code, and therefore understand how changes in one will impact the other.

We've identified four particularly interesting varieties of abstraction:

- Operational — Pre / post (or rely / guarantee) specs.
- Model — Model attributes may be different from actual implementations.
- Action — A complex dialogue presented with a single overall pre / post or rely / guarantee spec.
- Object — A group of objects presented as one object.

7.2 *Using refinement*

Spec and implementation must match

When you write a requirements specification (in any style and language you like), you want it to be a true statement about the product that ends up being delivered. If QA shows that the product falls short of the spec, you fix the code; or, it turns out to be impractical to deliver what was first specified, you can change the specification. But one way or another, you can't (or shouldn't!) call the job complete until the delivered design matches the specified requirements.

Components should have up to date specifications

For valuable components (as opposed to throwaway assemblages of them), we believe in keeping the specification after you've written the code; and in keeping it up to date. Long term, the spec (and high-level design documents) helps to keep the design coherent, because people who do updates have a clearer idea of what the component is about, and how it is supposed to work. Designs without good documents degenerate into fractal warts and patches, and pretty soon end up unmodifiable. Remember that over 70% of the effort on a typical piece of software is done after it was first delivered; and consider whether you want your vision of the design to be long-lived.

Spec and coding usually proceed concurrently

This is not to argue that you should complete the high-level documentation before embarking on coding. There are plenty of times when it's a tactical necessity to do things the other way around. Prototypes, and Rapidly Approaching Deadlines are the usual reasons. The most useful cycle alternates between coding (to obtain feedback from testing and users, and to get things done) and specifying (to get overall insights). Never go beyond getting either cycle 80% more complete than the other. All we need is that, by the appropriate milestone, the specs should correctly describe the code.

7.2.1 Documenting the refinement relation

The relationship between an abstraction and a refinement is the ‘refinement relation’. It is an assertion that one description of a configuration of objects and actions is a more abstract view of another.

Refinement relates two descriptions

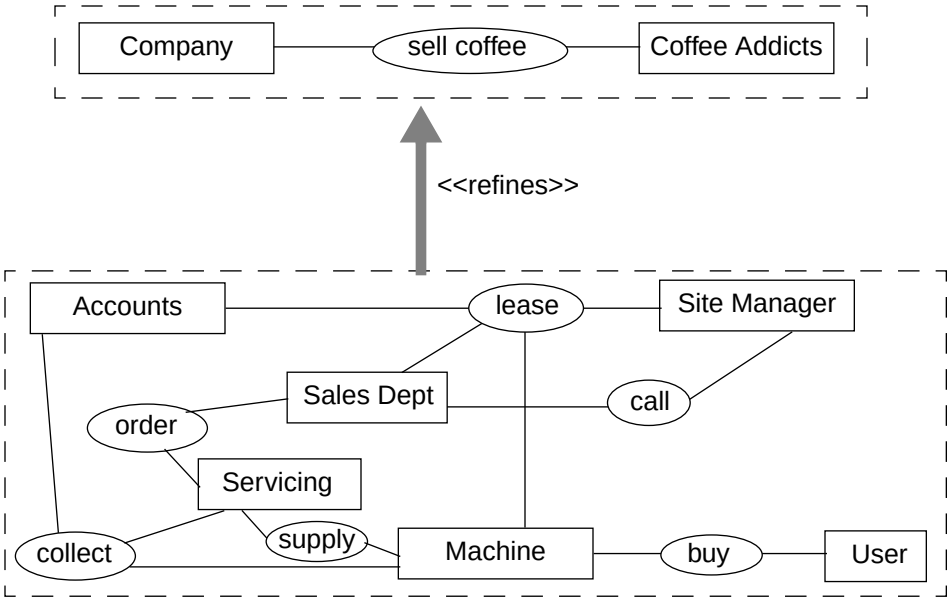


Figure 125: The refinement symbol: collaboration refinement

7.2.1.1 Refinement and subtype

The refinement symbol is a version of the subtype symbol[1]. It states that the more detailed model achieves everything expected from the more abstract one. But where a subtype symbol says to the reader “the subtype is defined *a priori* as an extension of the supertype, having all its properties and more”, the refinement symbol says “the refinement, a self-contained model even without the abstraction, is believed to specify everything defined for the abstraction, and more”.

Notation

7.2.1.2 Refinement and aggregation

There is also a slight difference between the refinement symbol and the diamond “aggregation” symbol. The diamond makes the abstract and the refined types (or actions) part of the same model: in these models, it is explicit that a wheel is a part of some particular car, and that there is some way of knowing which buy action a partic-

The aggregation symbol has a common idiomatic usage

1. UML 1.1 makes mention of refinement, with a default notation that uses a stereotype on the generic ‘dependency’ arrow. In our presentation here, we have chosen to highlight refinement with a distinguished arrow.

ular pay is part of e.g. some portion of the ‘retrieval’ is defined by the aggregation itself. The aggregation symbol is commonly used when, in a design, you have a reified and distinguished ‘head’ object representing the abstraction itself.

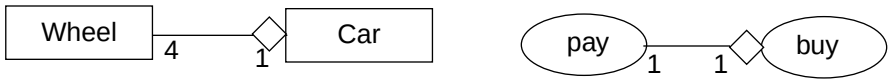


Figure 126: Refinement using "aggregation"

But the refinement symbol is somewhat more general: it says that, even if we didn’t think of it that way when creating one model, the other is nevertheless a more abstract or refined view of it; and we can deal with either separately.

Refinement is central to the design process

In some idealized sense, every design project can be thought of as generating a series of refinements, even if some of them arise as a result of re-factoring or bottom-up abstraction. You begin with a general idea of what is required, refine it to a solid requirements spec, refine it further to a high-level design, and so on to detailed design and code. For example, you might start with very abstract actions between users and system, and refine down to the actual sequences of GUI actions required for each action. If the project is critical enough to be fully documented, each of these refinement-layers is kept separately written-up; and just as importantly, the refinement-relationships themselves are documented and checked.

Documentation includes rationale and justification

Documenting a refinement means writing down:

- the reasons for choosing this realization from the alternatives (so those that follow won’t fall down the same pits you fell into along the way); and
- a justification for believing you’ve done the refinement correctly — that is, that the abstraction really does describe the realization accurately. (Doing this is an invaluable sanity check and helps reviewers and maintainers.)

Refinement documentation is attached to the refinement relation

Since refinement is a many-many relation, the refinement documentation shouldn’t properly attach to either the abstraction or the refinement, but to the refinement relationship between them. (A component you found in a library may be a good realization of your requirements; but it will presumably fit others’ as well.)

Be pragmatic about the documentation

Now of course, if you’re designing a nuclear power station or a jumbo jet, we’d hope you would take all the above writing and verification very seriously, with each layer individually written up and each refinement carefully established. But for most of us, it’s both acceptable and desirable to opt for the somewhat more practical solution of clarifying some essential issues at the requirements level, then working on the design and code while resolving other issues, and only then updating a few class diagrams.

...but keep interoperability needs in mind

Well, maybe. In the new world of component-based development, the successful components will be those that interoperate with many others. Each interface will be designed to couple with a range of other components, and must be specified in a way that admits any component with the right behavior, and excludes others. And a designer aiming to meet a spec should be confident that it works and be able to justify that belief.

We should therefore have some idea of what it means to conform to a specification — what has to be checked, and how you would go about it. In other words, documenting the refinement relation.

7.2.2 Refinement trees

Big refinements are made up of smaller ones. A requirements spec and a completely programmed implementation of it may look quite different; but their relationship can be seen as a combination of several smaller refinement steps.

Big refinements consist of smaller ones

So we can, in theory at least, relate a most abstract specification to the most detailed program code (or business model) through a succession of primitive refinements, each documented by a complete model. If you have a very great deal of time to spare, you might try it for a small design! This is what computers are for, of course, and appropriate tools can do a great deal to help.

In theory this can be traced; tools can help

In practice, we use the ideas more informally; and use design patterns as ready-made refinement schemes.

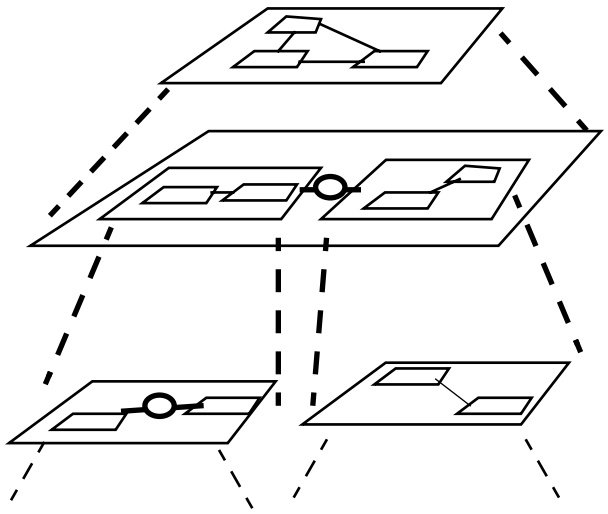


Figure 127: Refinements continue recursively

7.2.3 Traceability and verification

It is important to be able to understand how each business goal relates to each system requirement; and how each requirement relates to each facet of the design, and ultimately each line of the code. Documenting the refinement relationships between these layers makes it easy to trace the impact of changes in the goals.

Refinement enables traceability

Traceability is a much-advertised claim of object-oriented design: because the classes in your program are the same as in your business analysis, so the story goes, you should easily be able to see the effects on your design of any changes in the business. But anyone who has done serious OO design knows that in practice, the designs can get pretty far from this simple ideal. Applying a variety of design patterns to generalize, improve decoupling, and optimize performance, you separate the simple analysis concepts into a plethora of delegations, policies, factories, and plug-in pieces.

It is naive to expect code, design, and business models to be the same

Documenting the refinement relation puts back the traceability, showing how each piece of the analysis relates to the design.

Refinement re-links pieces

Safety-critical projects may prove consistency

In safety critical systems, it is possible to document refinements precisely enough to perform automatic consistency checks on them. However, achieving this level of precision is rarely cost-effective, and we do not deal with that topic in this book.

For the majority of projects, it is sufficient to use pre/postconditions as the basis of test harnesses; and document just enough of a refinement that other developers and maintainers clearly see the design intent. We'll see how to do this later in this chapter.

7.3 *A refinement example*

The sections that follow look in more detail at the four main kinds of refinement we mentioned above. This section introduces an example that runs through those sections.

7.3.1 A specification for a spreadsheet

Let's look first at what can be done with a good abstract model. It's a model of a self-contained program; but it could equally well be a component in a larger system.

Spreadsheet model

Figure 128 shows a model of a spreadsheet, together with a Dictionary interpreting the meaning of the pieces in the model. A spreadsheet is a matrix of named cells, into each of which the user can type either a number or a formula. In this simplified example, a formula can be only the sum of two other chosen cells (themselves either sum or number cells).

...with invariants

The model shows exactly what is expected of a spreadsheet: that it maintains the arithmetic relationships between the cells, no matter how they are altered: in particular, the invariant "Sum::..." says that the value of every Sum-cell is always the addition of its two operands.

...in user's vocabulary

Notice that this is very much a diagram not of the code, but of what the user may expect. The boxes and lines illustrate the vocabulary of terms used to define the requirement.

It does not commit to how things are inside

The box marked Cell, for example, represents the idea that a spreadsheet has a number of addressable units that can be displayed. It doesn't say how they get displayed, and it doesn't say that if you look inside the program code you'll *necessarily* find a class called Cell. If the designer is keen on performance, some other internal structure might be thought more efficient. On the other hand, if the designer is interested in maintainability, using a structure that follows this model would be helpful to whoever will have to do the updates.

Like most, it is an incomplete model

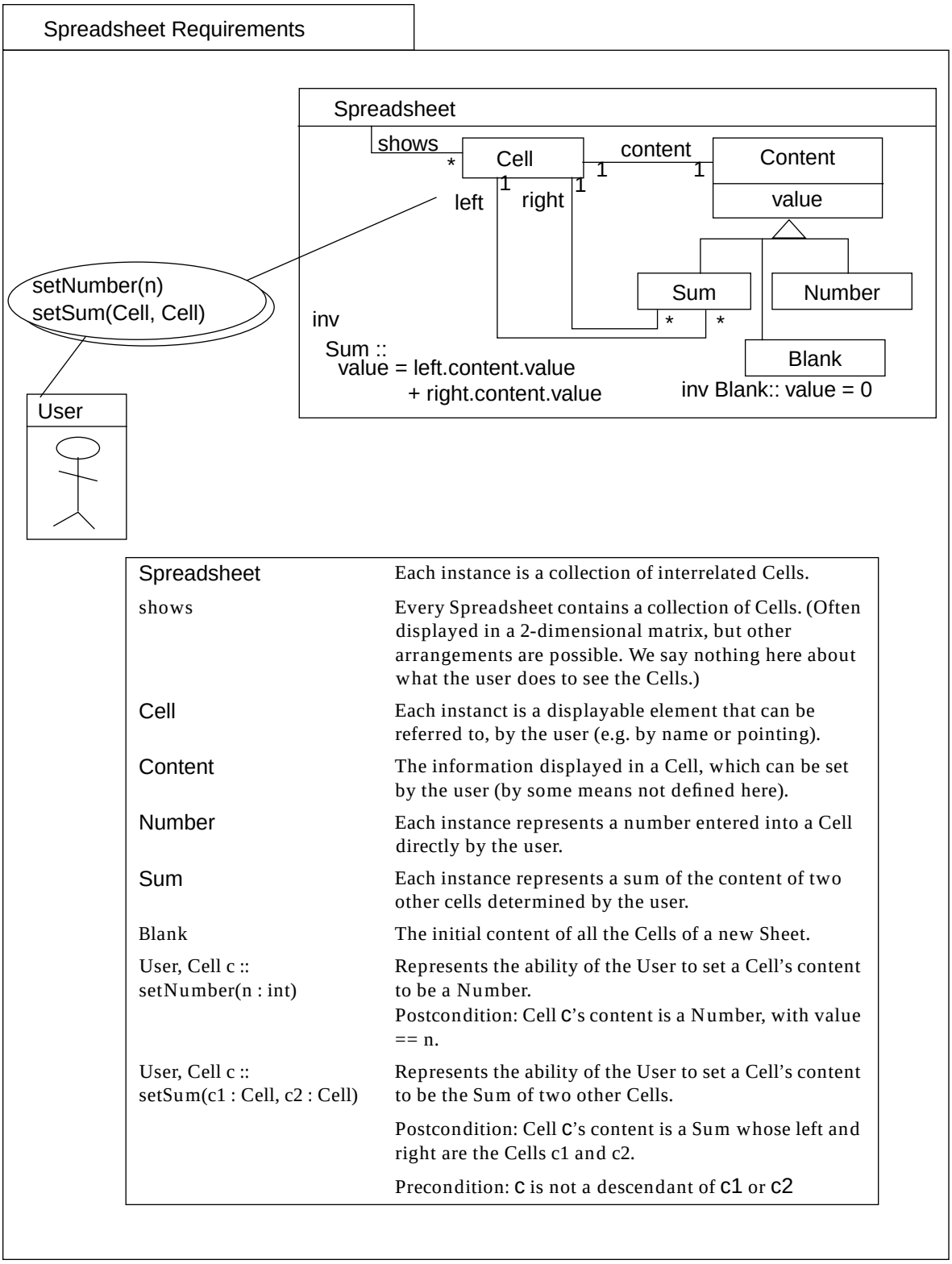
The model doesn't even say everything that you could think of to say about the requirements. For example, are the Cells arranged in any given order on the screen? How does the user refer to a Cell when making a Sum? If all the Cells won't fit on the screen at a time, is there a scrolling mechanism?

It's part of the utility of abstract modeling that you can say or not say as many of these things as you like. And you can be as precise or ambiguous as you like — we could have put the "{Sum::..." invariant as a sentence in English. This facility for abstraction allows us to use modeling notation to focus just on the matters of most interest.

7.3.1.1 Using snapshots to animate the spec

The abstract model is still precise e.g. snapshots

Although (or perhaps because) the model omits a lot of the details you'd see in the code, you can do useful things with it. For example, we can draw 'snapshots' — instance diagrams that illustrate quite clearly what effect each operation has.

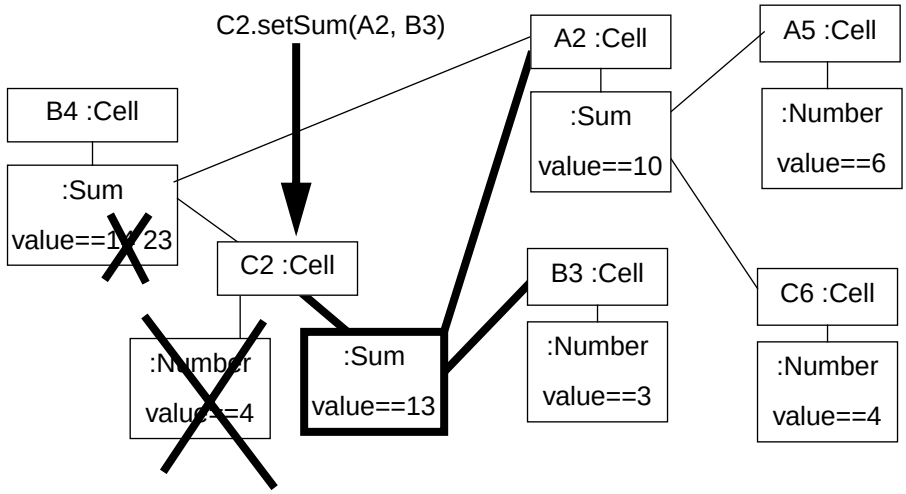


Spreadsheet	Each instance is a collection of interrelated Cells.
shows	Every Spreadsheet contains a collection of Cells. (Often displayed in a 2-dimensional matrix, but other arrangements are possible. We say nothing here about what the user does to see the Cells.)
Cell	Each instanct is a displayable element that can be referred to, by the user (e.g. by name or pointing).
Content	The information displayed in a Cell, which can be set by the user (by some means not defined here).
Number	Each instance represents a number entered into a Cell directly by the user.
Sum	Each instance represents a sum of the content of two other cells determined by the user.
Blank	The initial content of all the Cells of a new Sheet.
User, Cell c :: setNumber(n : int)	Represents the ability of the User to set a Cell's content to be a Number. Postcondition: Cell c's content is a Number, with value == n.
User, Cell c :: setSum(c1 : Cell, c2 : Cell)	Represents the ability of the User to set a Cell's content to be the Sum of two other Cells. Postcondition: Cell c's content is a Sum whose left and right are the Cells c1 and c2. Precondition: c is not a descendant of c1 or c2

Figure 128: Model for a Spreadsheet

Snapshots illustrate specific example situations. Figure 129 shows a snapshots showing the state of our spreadsheet before and after an operation. (The thicker lines and bold type represent the state after the operation.)

Figure 129: Spreadsheet snapshots



Notice that because we are dealing with a requirements model here, we show no messages (function calls) between the objects: those will be decided in the design process. Here we're only concerned with the effects of the operation invoked by the user. This is part of how the layering of decisions works in Catalysis: we start with the effects of operations, and then work out how they are implemented in terms of collaborations between objects.

..easily describe net desired effect

7.3.2 An implementation

Part of the program code

Now let's consider an implementation. The first thing to be said is that the program code is much larger than the model, so we'll only see glimpses of it. Here is some:

```
package SpreadSheetImpl_1;

class SpreadSheet_I             // This class implements SpreadSheet
{
   private Cell_I [ ] cells;       // array of cells of my spreadsheet
   ...      // various code here
}

class Cell_I
{
   private int m_value;            // The current value of the Cell
   private Sum_I sumpart;          // Null if it s a plain number
   public int value ( ) { return m_value; }
   // will need some mechanism to keep m_value in sync with sumpart!
   ... // More code for manipulating Cells goes here
}

class Sum_I                        // Represents Sums
{
   private Cell_I [ ] operands;    // An array of several operands
   int get_value ( )
   {  int sum=0;
      for (int i= 0; i<operands.length; i++)
         sum += operands[i].value( );
   }
... // More code ...
}
```

Figure 130: Implementation code for spreadsheet

(The "_I" suffixes distinguish the 'implementation'.)

It has differences from the model

Now the first thing a reviewer might notice is that the classes I have written don't seem to correspond directly to the classes mentioned in the specification. Their attributes are different, etc. "We are doing object-oriented design," says the reviewer; "Your code ought to mirror the spec. It says so in fifty different textbooks."

...but the internals should not matter

I am quick to my defense. Point (a), Encapsulation: The spreadsheet as a whole, seen through the eyes of a user (including other programs driving it through an API), does exactly what the spec leads them to expect. They won't know the difference. This is encapsulation, "also mentioned in about a hundred and fifty different books," I point out.

... it is performance optimized

Point (b), Engineering: while the spec was written to be easily understood and general to all implementations, my particular implementation is better. It works faster, uses fewer resources, and is better decoupled, than some amateur attempt that slavishly follows the model in the spec. And for many programs, there would be practical issues like persistence not dealt with in the spec.

...and it can do much more

"And fourthly," I add, "my program is a whole lot more than just a spreadsheet: to use it as such is to play *Chopsticks* upon the mighty organ of a grand cathedral; to ask Sean Connery to advertise socks." In other words, the spec from my point of view is partial, expressing the requirements of only one class of user; and so the implementa-

tion classes you see here are chosen to suit a much broader scheme. Nevertheless, I am able to function merely as a spreadsheet when required, and wish to be validated as such against the spec.

7.3.3 The refinement relationship

Bottom line: is it a valid refinement?

Now whether you believe all that about my spreadsheet is not too important here. Certainly there are many cases where performance, decoupling, or a partial spec lead to differences between model and code. So the reviewer is faced with trying to determine whether there is some correspondence between them: whether the code conforms to the spec (or 'refines' it).

No need to wait for testing

Naturally, proper testing will be the final judge. But understanding the conformance relationship allows an earlier check, can clarify the design and the rationale for the differences. Also, and very importantly, it provides traceability: the ability to see how any changes in either the spec or the code impact the other.

As we've seen, we can distinguish a few main primitive kinds of refinement. Combinations of them cover most of the valid cases: you can explain most design decisions in terms of them. In a large development, the usual layers of requirements, high-level design, detailed design, and code, can be seen as successive refinements.

...if we check refinements

Understanding refinement has several advantages:

- It makes clear the difference between a model of requirements and a diagram more directly representing the code (one box per class).
- It makes it possible to justify design decisions more clearly.
- It provides a clear trace from design to implementation.

Design patterns are canned refinements

Many of the well-known design patterns are refinements applied in particular ways. The refinements we're about to discuss are, in some sense, the most primitive design patterns; they are themselves combined in various ways to define the more popular design patterns.

We'll now see examples of the four refinements we looked at in Section 7.1.3.3, "Four basic kinds of abstraction," on page 267.

7.4 *Refinement #1: Model conformance*

We make a drawing of the code

The reviewer begins by getting me to produce a drawing of my code. Figure 131 shows a view focusing on the external user operations and their postconditions. You can see the direct correspondence between the static model and the variables. Indeed, there are tools that will take the code and produce the basis of the diagram.

```
class Cell_I                                  class Sum_I
{ private int value;                          {  Cell_I operands [ ];      // array
   Sum_I sumpart;   // null for a Number      ...
...
```

check how the abstract model is represented

So the reviewer begins with how the information in the model is represented.

"How is the spec's Sum represented?" she asks.
"By my Sum_I" I reply.
"So where are the left and right attributes of a Sum?"
"No problem. All the information mentioned by the requirements is there --- it's just that some of the names have changed. If you want to get the left and right operands of any Sum, look at the first and second items of my operands array. But since the requirements doesn't call for any operations that directly ask for the left or right, I haven't bothered to write them." Nevertheless, anyone using my code would see it behaving as they expect from reading the spec.

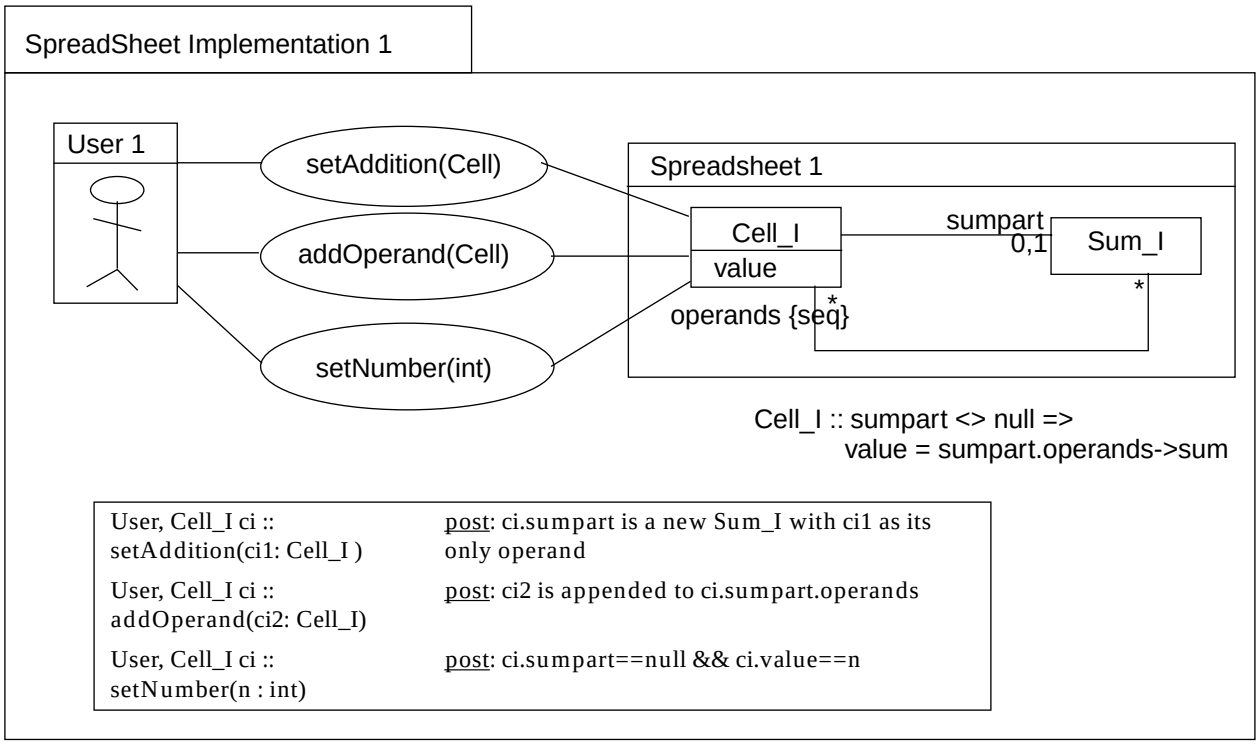


Figure 131: A picture of my code

7.4.1 What a type model means and doesn't mean

The types in a model provide a vocabulary for describing a component's state. The terms can be used to define the effects of actions. However, the model does not provide any information about the internal structure of the component.

But we need to move on to behaviors

A static model (types, attributes, and associations) provides, by itself, no useful information about what behavior to expect. Only the action-specifications based on it provide that. The complete specification is a true one if the statements it makes or implies about the component's behavior (response to actions) are always correct. Features of a static type model not used by action specifications are redundant, having no effect on the specification's meaning.

7.4.2 Documenting model conformance with a Retrieval

First 'retrieve' the model attributes in the code

The general rule is that, for each attribute or association in the abstract models, it should be possible to write a read-only function in the implementation code that abstracts (or 'retrieves') its value. Here are the retrievals for Sum_I I've just mentioned to my reviewer:

```
class Sum_I
{    private Cell_I [ ] operands;     // array
     Cell_I left ( ) {return operands [0]; }
     Cell_I right ( ) {return operands [1];}
     ...
```

It does not matter if the retrieval is complex

These abstractions happen to be particularly easy — the correspondence to the spec model is not too far removed; some are more complex. But it doesn't matter if an abstraction function is hopelessly inefficient: it only needs to demonstrate that the information is in there somewhere. Nor does it matter if there is more information in the code than in the model — I can store more than just two operands, though readers of the official spec won't use more than two.

7.4.2.1 Drawing a model conformance

Drawing a retrieval

Retrievals can be made more clear with a diagram. It can be very helpful to draw both models on a single diagram. You can then visually relate the elements across the refinement using associations, and write invariants that define the abstraction functions for all attributes in the spec[1]. These associations are introduced specifically for this purpose, and are distinguished with a **"//"** marker. The figure below shows that each Cell_I in the implementation corresponds 1-to-1 with a Cell in the spec; ditto for

1. This refinement could even be documented in a separate package from either the spec, or an implementation ()

Sum_I and Sum. So any specification related to a Cell or Sum can be traced to the implementation. All we need is to document how the attributes of Cell or Sum can be computed from the attributes of Cell_I or Sum_I.

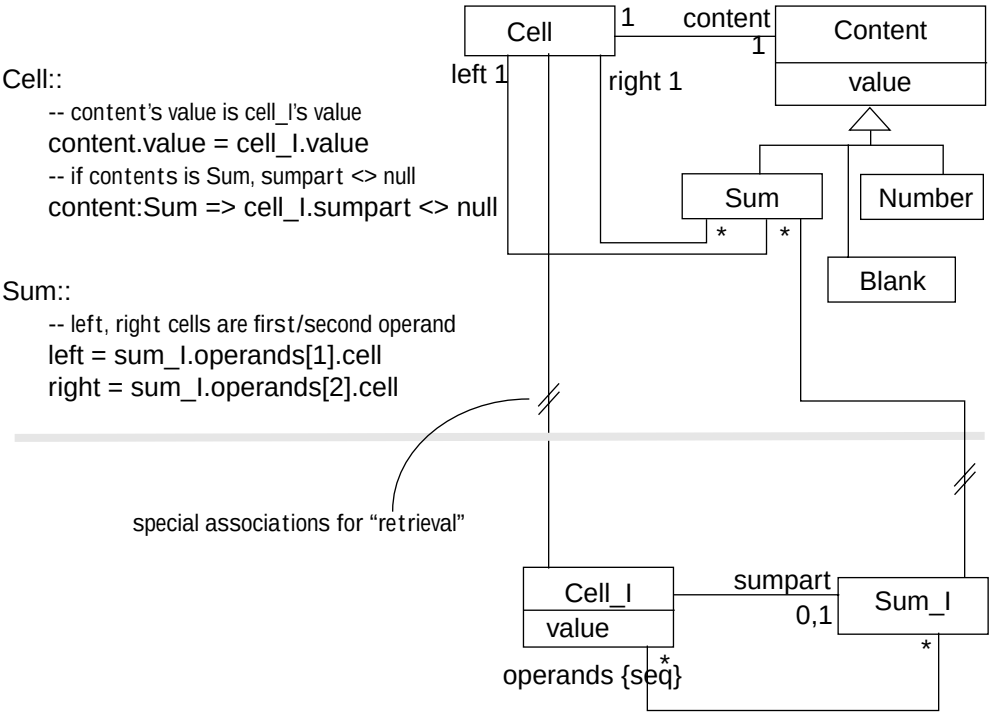


Figure 132: Retrieval diagram

Note that Content has no direct counterpart in the implementation, unless it is a sum; in which case it corresponds to Sum_I. But what about Number and Blank? For such cases, step back a level to Cell, and define the cell's attributes to include those of content e.g. cell.content.value is defined in the figure above. Check

7.4.3 Testing using abstraction functions

Useful for testing

QA departments may insist that such retrieval (abstraction) functions should be written down, or even written into the code. Even though they are not always actually used in the delivered code, they are useful:

- Writing them is a good cross-check, helping to expose inconsistencies that might otherwise have been glossed over.
- They make an unambiguous statement about exactly how the abstract model has been represented in your code.
- For testing purposes, testbeds can be written that execute the postconditions and invariants defined in the requirements models. These talk in terms of the abstract model's attributes, so the abstraction functions will be needed to get their values.

For example, recall we wrote one of the invariants as:

Sum:: value == left.content.value + right.content.value

Now that we've defined what left and right mean as executable functions — and could do the same with content and value — we should be able to insert this expression in our code, and use it as a debugging check before and after every operation.

7.4.4 Model conformance summary

We've seen how a model can be very different from the code, and yet still represent the same information. Now, as we've said before, the Golden Rule of object oriented design is to choose your classes to mirror your specification model. When that is possible, the abstractions are trivial, a 1-1 correspondence. But there are several circumstances where it isn't possible, and model refinement gives us a way of understanding the relationships. Typical cases include:

Where possible, mapping should be direct, except for...

- The model that gives best execution performance is very different from one that explains clearly to clients what the object does.

...performance,

- The implementation adds a lot to the specified functionality. It is possible for one object to satisfy several specifications, especially where it plays roles in separate collaborations (as we will see in later chapters). Each role-specification will have its own model, which will have to be related to the implementation.

partial views,

- You specified a requirement, and then went out and bought a component that comes with its own spec. The first thing you have to work out is how their model corresponds to yours.

and purchased parts

We've also said before, that it is a matter of local policy, how formal you get in your documentation. When you're plugging components together to get an early product delivery, you don't care about all this. But when you're designing a component you hope will be reused many times, it is worth the extra effort. And even if you don't go to the trouble of writing the abstraction functions, it is useful to do a mental check that you believe they could be written if you were challenged to.

7.4.5 Testing by representing the specification model in code

What the Quality Assurance department really wants is to be able to represent a spec in code, so that it can be run as a test-harness. They want to be able to write one set of invariants, postconditions, and so on, that every candidate implementation can be tested against. As far as they are concerned, the spec-writer writes a spec and associated test assertions; and each hopeful designer has to supply two things: the design, plus a set of abstractions that enable the test assertions to execute.

Ideally, we would represent the spec in code: a test harness

This very rigorous view of specification and testing leads to a view in which all models can be cast into program code (which is not so tedious as it was before code-generating tools). The types in a specification turn into abstract classes, of which the designer is expected to supply implementations. Figure 133 shows this done for the invariants; postconditions are omitted.

(In Java, you'd think spec types would be written as interfaces. Unfortunately, if we want to put the invariants and postconditions into the types themselves in real executable form, they need to be classes. This has the uncomfortable effect of disallowing

The coded specs move onto abstract classes

Specification as Code

```
package SpreadSheetSpec;

abstract class SpreadSheet
{
   public abstract Cell [ ] shows ();
         // Cell [ ] --array of Cells
         // abstract --header only, no body

// invariant true if all constituents OK
   public boolean invariant ( )
      { boolean inv= true;
         for (int i= 0;
               i<shows().length; i++)
            inv &&= shows()[i].invariant();
         return inv;
}     }

abstract class Cell
{  abstract Content content ();
//invariant true if content is OK
   boolean invariant ( )
      { return content( ).invariant(); }
}
abstract class Content
{  abstract    int value ( );
// default invariant - depends on subclass:
   boolean invariant ( ) { return true; }
}

abstract class Sum extends Content
{
   abstract    Cell left();
   abstract    Cell right();
// crucial spreadsheet invariant
   boolean invariant ( )
   {  return super.invariant() &&
         value() ==
               left().content().value( )
         +     right().content().value( );
}  }

abstract class Number extends Content
{
   abtsract    void set_value (int v);
      // post: v == value()
}

class Blank extends Content
{
   boolean invariant ( )
   {  return super.invariant()
               && value()==0 ;
   }
   // This is so obvious ... let s just do it
   public int value ( ) { return 0; }
}
```

Implementation (part)

```
package SpreadSheetImpl;
import SpreadSheetSpec;

class SpreadSheet1 extends SpreadSheet
      // This class implements SpreadSheet
{  private Cell_I [ ] cells;
//retrieval:
   public Cell [ ] shows ( )
   {  Cell [ ] r= new Cell [cells.length];
      for (int i= 0; i<r.length; i++)
         r[i]= cells[i];
      return r;
}  }

class Cell_I extends Cell
{  private boolean isBlank= true;
   private int m_value;    // current value
   private Sum_I sumpart;  // null for Number
// retrieval as a Cell:
   public Content content ()
   {  if (isBlank) return new Blank( );
      else if (sumpart==null)
         return new NumberCellAdapter(this);
      else
         return sumpart;
   }
   public int value () {if (sumreturn m_value;}
// services retrieval of Number:
   void set_value (int v) { m_value= v; }
}

class Sum_I extends Sum
{  private Cell_I [ ] operands;
   public int value ( ) { ...add operands...}
// retrievals as a Sum:
   public Cell left ( )
   {  if (operands.length == 0)
         return new Cell_I ( ); //blank
      else return operands[0];
   }
   public Cell right ( )
   {  if (operands.length <= 1)
         return new Cell_I ( ); //blank
      else return operands[1];
} ... }

// This class is used for retrieval
// when debugging -- not required in delivery
class NumberAdapter extends Number
{  private Cell_I myCell;
   NumberAdapter (Cell_I c) // constructor
      { myCell= c; }
   public int value ( )
      { return myCell.value(); }
   public void setValue (int v)
      { myCell.setValue(v); }
}
```

Figure 133: Specification directly translated to code, and implementation with retrievals

one implementation class from playing more than one role. An alternative is to put all the test apparatus in a separate set of classes that interrogates the states of the types. Once again, we find ourselves applauding Eiffel, in which all this is natural and easy.)

Spec and implementation, both in code

If we can write the whole thing, spec and all, in code, we can also show both the abstract model and the more detailed design in one picture: see Figure 134.

But our reviewer has been thinking. “I notice your left and right return Cell_I, so that must be your representation of Cell, right? So what’s a Cell_I’s content?”

“In my terms, that’s its sumpart”

“But that only works if the Cell’s content is a Sum. What if it’s a Number or a Blank?”

Some model attribute types may not correspond to implementation

Well, good question. The spec always models every Cell as having a Content, even if it’s just a plain Number; while I keep the value in the Cell_I itself, and only use an extra object to deal with Sums. The information is all in there, but distributed in a different way. But it does present an obstacle to the executable invariants idea: a term like left().content().value() wouldn’t work where left() is a plain number-cell, since it doesn’t have a content.

We can either just document the retrieval...

What I really would like to say is that expressions in the spec like “content.value” should be translated into the terms of the implementation as a whole: the content.value of a Cell_I is its sumpart.value if it’s a sum, and its own variable m_value otherwise. This is perfectly reasonable if all I want to do is document the abstraction function and persuade my reviewer that my code is OK.

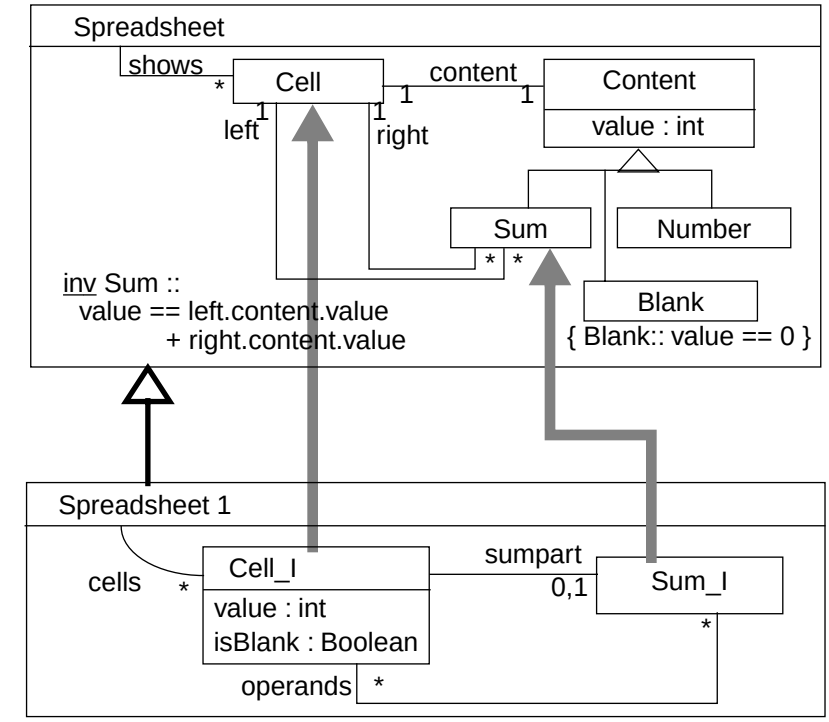


Figure 134: Spec and code, showing main correspondences

..or, to test, add implementation adapters

But for executable test assertions, content() must return something that will then return the appropriate value(). This leads to the invention of the class NumberAdapter (so now we have abstraction classes as well as functions). It can be kept fairly minimal: all it has to do is know our implementation well enough to extract any information required by the test assertions in the spec type Number.

A pattern for executable specs

So the general pattern for executable abstractions is that for every type in the spec, the designer provides a direct subclass. Sometimes these can be the classes of the implementation; in practice they often have to be separately written — mostly because your classes have more interesting things to be subclasses of. Each adapter retrieves from the implementation that part of the component's state that its specification supertype represents. Notice that our adapters are created only when needed.

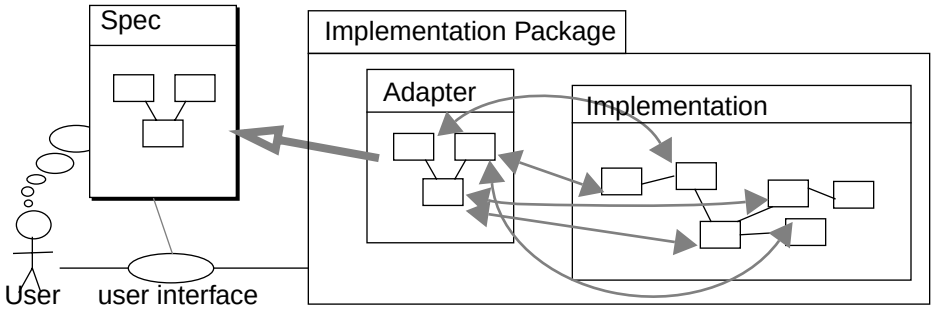


Figure 135: Test by adapting the implementation

Violated encapsulation?

What has happened to our ideal of encapsulation? We started out with the idea that internal workings unseen by the user could be designed any way you like; now we've had to put back all the structure of the model.

Not really; this is for test only

Well, not quite. The adapter classes only have to translate, and are there only for verification. The real implementation still does the hard work. And you'll only need them if your QA department is pretty stringent!

And often useful for other reasons as well

As it turns out in practice, adapters of this kind are frequently needed for each external interface that a component has — whether it is a user interface or to another component. This is because the component's internal state must be translated into the view understood by each external agent. (For human users, the GUI usually encompasses the adapter.)

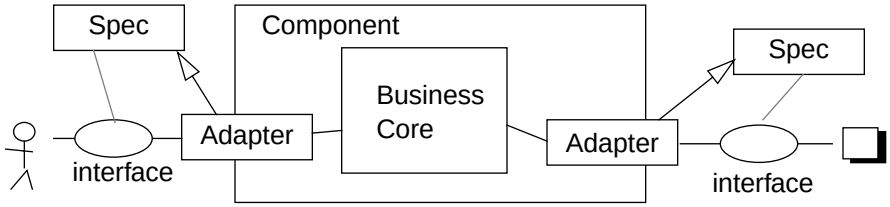


Figure 136: Adaptors for other interfaces

7.4.5.1 Specifications in code — summary

Specifications can be written not just as pictorial models, but also in the form of executable test frameworks. The benefit is a much stronger assurance of conformance, especially where there may be a variety of candidate implementations.

Implementers have to provide executable abstraction functions, to translate from the component's internal vocabulary to that of the specification. Often, this leads to providing an 'adaptor,' a set of classes directly mirroring the types in the spec. However, adaptors can be useful at the interface of a component, in addition to their role in verification.

7.5 *Refinement #2: Action conformance*

A specified function is missing — refined to a collaboration

The next thing my code reviewer notices is that nowhere is there a function setSum(Cell, Cell). I explain that I have decided to refine this action to a finer-grained series of interactions between the User and the Spreadsheet — see Figure 131 on page 290. To set a cell to be the sum of two others as per requirement, the user performs this scenario:

- select the Cell in question by clicking the mouse;
- setAddition: type "=" and click one of the operand Cells;
- addOperand: type "+" and click on the other.

Same overall effect

The requirement is provided for by a combination of features in my implementation. My "=" operation turns the Cell into a single-operand sum, and each "+" operation adds another operand. So although there is no single operation with the signature setSum(Cell, Cell), either at the user interface, or anywhere inside the code, the user is nevertheless able to achieve the specified effect.

It is an important feature of a Catalysis action that it represents a goal attained by some collaboration between the participants, without stating how. The goal can be unambiguously documented with a postcondition or guarantee condition. A conformant implementation is one that provides the means of achieving the goal.

A collaboration changes protocol, and affects the user

Different implementations will require different behavior on the part of every participant, since they will involve different protocols of interaction. A user who knows how to use my spreadsheet-implementation will not necessarily know how to use another design. It is the collaboration that has been refined here, not the participants individually.

7.5.1 Action conformance: realization of business goals

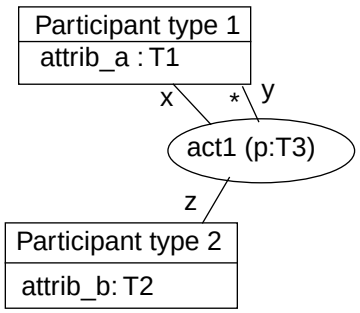
Software specs are (parts of) business goals

In general, the specification-actions are about the business goals of the system. "The user of our drawing-editor must be able to duplicate picture elements; and must be able to copy them from one drawing to another." The actual actions we provide break up these larger goals into decoupled pieces: we invent a clipboard and provide cut and paste operations — with which the user can achieve the stated goals.

Another example: "The customer of our bank must be able to get money at any time of the day or night." So we invent cash machines and cash cards, and provide actions of inserting card into machine, selecting service, etc. — with which the user, with the help of a good user interface, can achieve the stated goals.

7.5.2 What an action means and doesn't mean

7.5.2.1 Defining an action



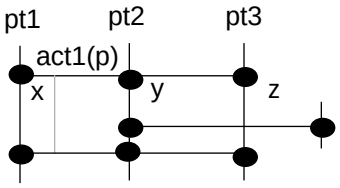
action act1 (p:T3)
post ...x.attrib_a ... z...

The action represents an effect which is achieved by some interaction between the participants. You may draw an action with any number of participants and parameters, each of which has a type.

Actions have participants and parameters

An action ellipse is a variant of the type box, and may have labeled 'associations', its participants; the 'attributes' are its parameters.

You can specify its effect in any desired degree of formality or informality, but should for preference refer to attributes and associations of any of the participants' and parameters' types.



An occurrence of an action may be shown as a horizontal bar on a sequence diagram, linking the instances affected. To show an action extended in time, draw two bars linked with a dotted vertical bar.

..and individual occurrences

An action is declared within the context of a particular model, at a given level of detail.

7.5.2.2 The meaning of an action

Drawing an ellipse on a type diagram signifies that this change of state, documented as the spec of this action, may occur as a result of interactions between instances of these types. The parameters and participants of an action together list those aspects that can differ, from one occurrence to another. Any occurrence of the action may, on refinement, be found to consist of a set of smaller actions, involving parameters and participants not mentioned in the more abstract model.

The ellipse defines a state change caused by some interactions

The postcondition states a fact about changes that can be seen in any occurrence, comparing the participants' states at the beginning and end of the occurrence. Other changes not mentioned by the postcondition may also have happened. Any precondition states the circumstances under which an occurrence can begin.

The pre / post apply to all (refined) occurrences

An occurrence of an action extends in time: other actions may occur concurrently.

Actions are non-atomic

Any guarantee condition states a fact that will be maintained true by the refining collaboration of the participants while the action is in progress. Any rely condition states a fact that must be maintained true by other objects, throughout the period of an occurrence. It sets limits on the possible effects of concurrent actions.

Rely and guarantee define concurrency

The only objects affected are the participants and parameters. Any other objects mentioned in the same model are unaffected.

7.5.3 Checking action conformance

Mapping of action refinements

We need to check that, for every action defined in the Specification, there is some combination of Implementation actions that the user could follow, to achieve the defined goal. Let's take setSum as an example.

These action specs were given (in Figure 128 and Figure 131):

From the Spec:

User, Cell c :: setSum(ci1:Cell , ci2:Cell)	Represents the ability of the User to set a Cell's content to be the Sum of two other Cells. Post: Cell c's content is a Sum whose left and right are the Cells c1 and c2.

From the Implementation:

User, Cell_I ci :: setAddition(ci1:Cell_I)	Post: ci.sumpart is a new Sum_I with ci1 as its only operand
User, Cell_I ci :: addOperand(ci2:Cell_I)	Post: c2 is appended to ci.sumpart.operands Pre: ci.sumpart != null — this is already a Sum

Abstract and concrete snapshots...

A little thought suggests that a setAddition followed by an addOperand should achieve the effect of a setSum. A comparison of snapshots in the two views will help; the arrows show which links are created by each action:

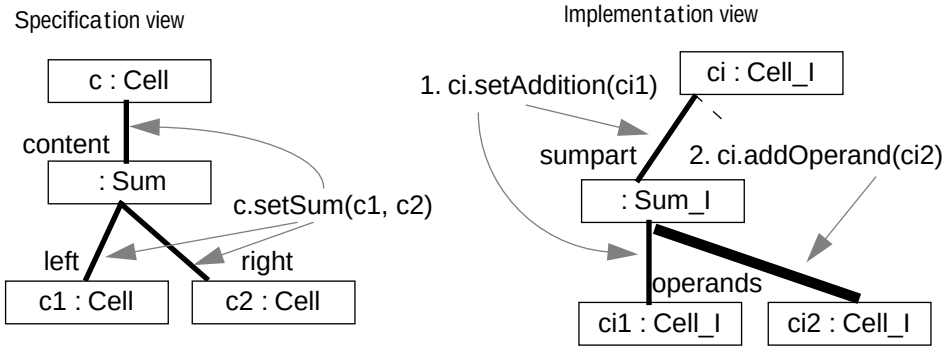


Figure 137: Snapshots in specification and implementation

Now we can see that performing the sequence

<ci.setAddition(ci1), ci.addOperand(ci2)>

should achieve an effect corresponding to the spec's c.setSum(c1, c2). To be absolutely sure, we can use the abstraction functions we worked out earlier.

with retrieval functions, show the mapping

setSum's postcondition talks about the left and right of Cell c. In our implementation, we claim that Cell_Is represent Cells, and so in our example snapshot, ci*n* represents c*n*.; we also decided that content.left in the spec is represented by sumpart.operands[0]. So does this sequence of steps achieve that "Cell c's content is a Sum whose

left is Cell c1”? Yes, because the first step makes ci.sumpart.operands[0] ==ci1. And does it achieve that c.content.right == c2? Yes, because the second step achieves ci.sumpart.operands[1] == ci2.

7.5.4 Documenting action conformance

Refinement outlines the mapping

We can document the refinement as in Figure 138. The diamond “assembly” symbol just says that there is some way in which the specification effect can be achieved by some combination of the implementation actions. But exactly what sort of combination — concurrent, sequential, some sort of loop — needs to be said separately.

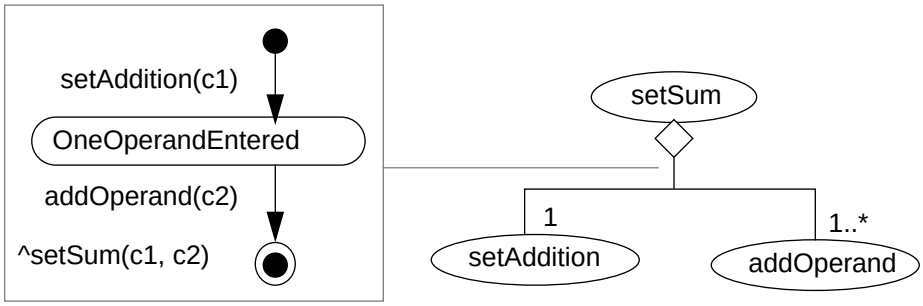


Figure 138: Action conformance

Statechart can show temporal combination

In the case of a sequence, this can be done with a statechart. Each implementation action is a transition on the diagram; each state represents how much of the overall job has been achieved so far.

Exception paths can be added separately

This one is very simple: the initial setAddition takes us into an intermediate state, OneOperandEntered, from which an addOperand will complete the overall setSum action. This achievement is denoted by the caret mark “^”. It is not specified here, what happens if addOperand occurs when we’re no in OneOpnd, or what happens if setAddition is performed again when we’re in OneOpnd. Those exceptions can be shown separately.

Can add guards etc. scoped within statechart

Names are chosen in the statechart (here, c1 and c2), to indicate how the arguments of the actions are related. You can use the names in guards and postconditions written in the chart, to show other constraints and results in more detail.

As usual, the diagrams are not intended to be a substitute for good explanation in your natural language: they are supposed to complement it and take out the ambiguities.

7.5.5 Testing action conformance

Testing is trickier

Testing action conformance at run time isn’t so easy as just inserting a few lines of code to monitor the postconditions. The problem is that there is nowhere that my user explicitly says to my implementation, “Now I want to do a setSum”. It’s the same with our other examples: a user doesn’t say to the drawing program “now I’m going to move a shape from one drawing to another” — they just use the select, cut and paste operations in such a way as to achieve that.

But can still be automated

Therefore we have two options: to write a test harness that performs the requisite sequences and checks the postconditions by comparing states before and after; or to perform the equivalent checks manually, following a written test procedure. This is a matter of policy in your project — each approach will be appropriate in different circumstances. Either way, the documented action conformance should be used to guide the creation of the tests; and the retrievals (whether just documented, or actually coded as we saw earlier) provide the mapping between the different levels of model.

7.5.6 Action conformance and layered design

7.5.6.1 Action refinement brings model refinement

Action and model refinement go together

Action refinement is nearly always associated with model refinement. The more detailed model needs more information, in order to represent the intermediate states.

Refined model tracks intermediate states

Action refinement is about taking some large interaction with many parameters, and breaking it down into several steps with fewer and simpler parameters. For example, "get_cash(ATM, person, account, $)" breaks down to several steps like insert_card(ATM, card) and enter_amount($) each of which identifies just a few parameters at a time. After the first step, the ATM system needs to remember whose card has been inserted, so that when the later $ step happens, it knows which account to debit. The association of "Account x currently using ATM y" is not needed at the more abstract level. Ultimately, the process can be taken right down to individual keystrokes and mouse clicks.

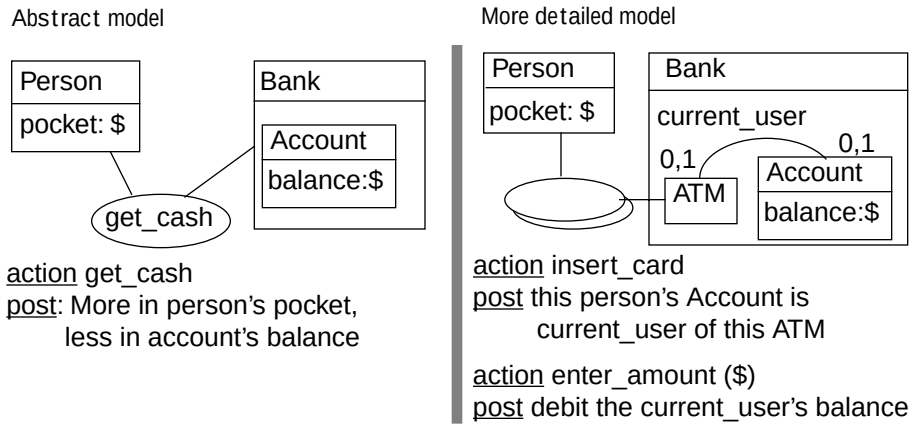


Figure 139: Action refinement means intermediate attribute

For example, selecting a cell

In our spreadsheet, we glossed over something of this nature. We originally said that the user first selects a cell, and then performs a setAddition operation, to identify the first operand. In other words, select(Cell_I) sets some current_focus attribute of the spreadsheet as a whole; and setAddition's postcondition should properly have been written in terms of current_focus.

action setAddition (ci1: Cell_I)
post: current_focus.sumpart is a new Sum_I with ci1 as its only operand

And of course, we can take the action refinement even further. How is a setAddition performed? By typing '=' and clicking in a Cell. For that we need even more model: to record that after the '=', we're now in the 'identify Cell for adding' state; and to map mouse coordinates to Cells — yes, finally, we've got down to something that uses the graphical layout of the spreadsheet.

...down to mouse co-ordinates

7.5.6.2 Reifying actions

'Reification' means making real — we use it to mean making an object out of some concept. In a model, we can feel free to represent any concept we like as an object; in a design, reification is the decision to write a class to represent that concept.

To have a concrete representation of a concept

It is often useful to represent an action as an object, particularly an abstract action with refinements. Its purpose is to guide the refined actions through the steps that lead to the achievement of the abstract action's postcondition; and to hold the extra information that we have seen is always required to represent how far the interaction has progressed.

Abstract action is often reified in software...

So in the bank example, we might create an ATM_transaction object as soon as a user inserts a card. This would keep all the information about which account has been selected, what the current screen display should be, what the menu options are, and so on.

Secondary purposes to reified action objects include: forming a record of the action after it has completed, for audit trail purposes; and keeping the information necessary for an 'undo' operation. (See [Gamma 94], *Command*..) This transaction object is also the place to put all the functionality about exceptional outcomes, rollbacks, and so on.

We can see reified actions in real-world business transactions. An 'order' is the business concept representing the progression of a 'buy' action through various sub-actions such as asking for the goods, paying, delivery. An 'account' is the reification of the ongoing action of entrusting your money to your bank.

...and in the real world

Reified actions are often called 'control objects' or 'transaction objects'. Whenever we draw an action ellipse on a type diagram, we are really drawing a type of object — which might or might not be reified in an implementation.

These are called 'control' objects

And it's easy to see the direct correspondence: the participants of the action are drawn as, and are, associations of the reified action; the action's parameters, variable values we don't bother to depict as links on the diagram, are the object's attributes:

It is a direct correspondence

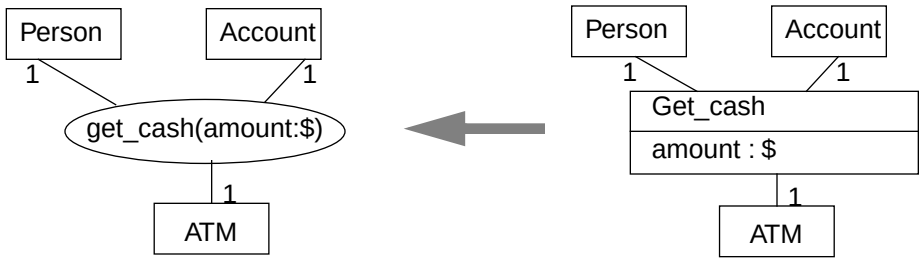


Figure 140: Reification of "get_cash" action

As we put in detail about the constituent actions, the extra information is added to this object. The state diagram we drew, depicting the correspondence between the abstract and detailed actions, becomes the state chart of this object.

7.5.6.3 Action refinement and design layers

Reified actions show up at all levels

The relationships between abstract actions and their more detailed constituents covers a scale from individual keystrokes or electrical signals, to large-scale operations. The objects that reify these actions range from GUI controllers to whole application programs: for example, the operations of editing words and pictures in a drawing program can be seen as refinements of the overall action of creating or updating a document; the in-memory version of the document, and mechanisms such as the cursor and scrollbars, are part of the extra information attached to the object at the more detailed level. At an even bigger scale, workflow and teamwork systems help control the progress of a project through many individual tasks, with documents as intermediate artifacts.

They also often separate architectural layers

Within a software design, the different levels of refinement are generally associated with different, decoupled, layers in the software. We can see this, for example, in the conventional separation into business and GUI layers. We can also see it in communications protocols, from the individual bits up through to the secure movement of files and web pages.

7.5.7 Action conformance within software

Actions defer software protocols, not just UI

Action refinement is not just about user interactions. The same principle can be used when describing dialogues between objects in the software. This enables collaborations to be described in terms of the effects achieved by the collaborations, before going into the precise nature of the dialogue.

e.g. change-propagation on Cells

For example, one of the invariants of the spreadsheet implementation is that the value of a Cell_I with a sumpart shall always be the sum of sumpart.operands. To achieve this, my design makes each Sum register as an observer of its operand Cells. When any Cell's value changes, it will notify all its observers; Sums in turn notify their parent Cells. The effect is to propagate any change in a value.

We can be precise whilst deferring protocol detail

We can document that requirement, whilst deferring the details of how the change is propagated. Does a Cell send the new value in the notify message? Or the difference between old and new values? Or does it send the notification without a value, and let the observer come back and ask for it? In the midst of creating the grand scheme of our design, we don't care: such details can be worked out later, we want to get on to

other important issues first. The art of abstraction is about not getting bogged down in detail! All we need do at this stage is to record the relationship and the effect it achieves:

update occurs whenever any operation changes the Cell's value
Cell c, Sum s :: c.value <> c@pre.value && c in s.operands => update ( )

post: The difference in value is propagated to the Sum.
c.value – c@pre.value == s.~sumpart.value – s@pre.~sumpart.value

Figure 141: Deferring update protocol

7.5.8 Action conformance — Summary

Actions are used to describe the way in which objects collaborate — whether they are people, hardware, or software. Actions are primarily described by their effects on the participants; and secondarily as a series of steps, or another refinement to smaller actions.

Action abstraction allows us to describe a complex business or software interaction as a single entity. Action refinement can be traced all the way from business goals down to the fine detail of keystrokes and bits in wires.

A useful technique in documenting action conformance is to draw a statechart of the progress of the abstract action through smaller steps.

7.6 *Refinement#3: Object conformance*

Object conformance means recognizing that a particular single object is an abstraction representing several constituent parts; and that the state and responsibilities attributed to the abstract object are in fact distributed between the constituents.

Often goes with action refinement

Object conformance often accompanies action conformance, when it turns out that the detailed dialogue is not conducted between the participants as wholes, but between their parts. You can get cash from a Bank, but more specifically from one of its cashiers or ATMs. In fact, looking at the actions in detail, when you receive cash from an ATM, it actually comes out of its money-dispenser.

In Catalysis, we separate writing the requirements of a system from designing the internal messages that deal with each action. In requirements specs, we treat the system or component we're designing as a single object. The closer inspection of a system to treat it as a set of interacting objects is but one example of object refinement.

Process is entirely fractal

The process can be fractal: like all of our techniques, it applies as well to a complete system as to a small object inside the software. A system design can start by refining into major components, and define the high-level actions between them. Subsequently, those actions can be refined to provide more interactive detail; and each component can be itself be refined into constituent objects; and so on, until the actions are individual messages, and the objects are things you can write directly in your favorite programming language.

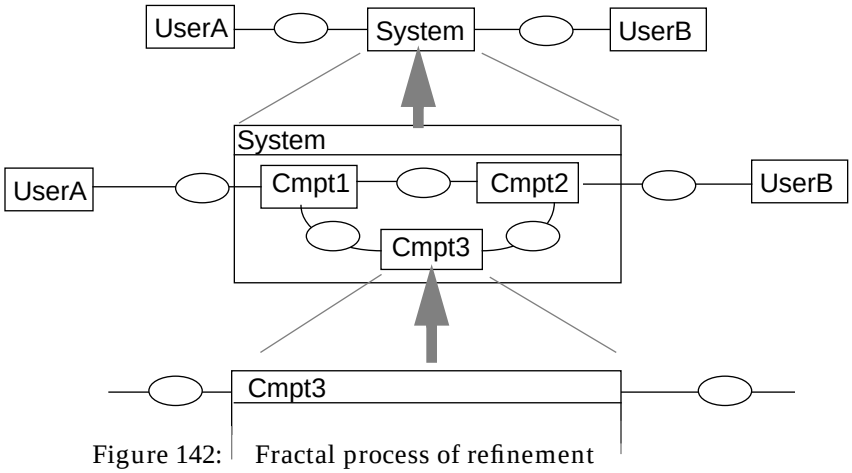


Figure 142: Fractal process of refinement

'Zoom' in or out

Looking in the other direction, the system we're interested in designing is part of some larger machine or organization or software system. By 'zooming out', we can understand better how our design will help fulfill the overall goals of that larger object.

This section will use the term 'component' to refer to the object we are interested in refining — but the same principles apply on every scale.

7.6.1 What an object does and doesn't mean

Object and type apply at all levels

We can use an object to describe any concept, including the active components of some system — whether people, hardware, or software; or groups or parts thereof. Likewise, we can use a type to describe the behavior of such objects, and actions to describe its participation in interactions with other objects.

An object is an abstraction of several others

A type always describes the behavior of some linked-together collection of smaller constituents. A closer look will reveal the parts, and that different parts deal with different behavior.

Type and collaboration describe each level

To specify the abstract object's behavior, a type model is used, that tells us nothing about the abstract object's internal structure. The internal structure can be described as a set of linked objects participating in actions; the actions are both between themselves, and the actions in the outside world. A more detailed picture that reveals the internal structure must account for how the external actions visible in the abstract view are dealt with by the internal structure.

7.6.2 The process of object refinement

The kinds of refinement we've already looked at can be used in a variety of ways; but they fit together particularly well as part of an object refinement. So let's review those techniques as steps forming part of object refinement.

(We've looked at this process in other parts of this book. The point here is to see the steps in terms of the formal refinements defined in this chapter.)

7.6.2.1 Start with the specification

Specify all high-level actions

The model we begin with specifies the behavior of our component. Most objects are involved in more than one action: our spreadsheet has addOperand, setNumber, etc. Some are involved in actions with several other objects: while the spreadsheet has one user, the bank's ATM has customers, operators, and the bank's host machine to deal with. Each of the actions may be specified at a fairly high level, with details to be worked out later. For each action, we have a specification, in terms of more-or-less rigorously defined pre / postconditions (and possibly rely / guarantee conditions).

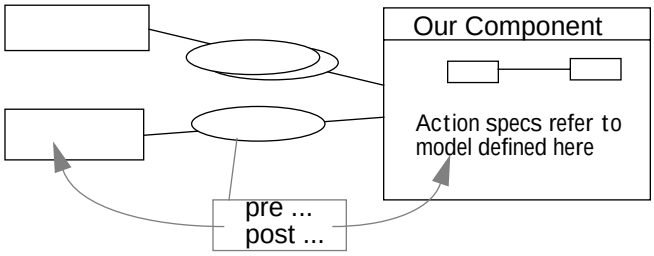


Figure 143: Joint action specified in terms of multiple participants

Next, need to work out 'inside' of the object

Given this context, what we need to do is work out what objects there are inside our abstract object, and how they collaborate to achieve the effects specified for each of the abstract object's actions.

Notice that we might either be refining just one of these objects — perhaps the others are users, whose behavior is the province of the GUI designer and the manual writer. Or it might be that we will be refining several of them, if they are components in a larger system.

7.6.2.2 Decide design object model

Action affect our object, among others

The actions are specified in terms of their effects on the participants, of which our component is one. For that purpose, each participant has a model: for simple objects, it may just be a few attributes; for complex ones, it will be a picture based on the business model. Our spreadsheet had Cells and their various Contents.

Pick the design objects

Focusing on the system we're interesting in refining (the spreadsheet rather than the user), we may decide, as we did in this example, to use a different set of objects in the actual design. We have already seen how to use abstraction functions to relate the design model back to the specification.

7.6.2.3 Refine actions and mask specs

Refine the specified actions

We refined the actions (so that setSum became <setAddition, addOperand>), and specified the more detailed versions in the terms of the design model (Cell_I rather than Cell etc.).

Focus on entirely 'one-sided' action specs

At this stage, we 'mask': remove anything from the action specs that talks about the other participants in the actions. For example, we don't care about the state of the ATM-user's pocket, so long as the ATM dishes out the dosh. In the spreadsheet example, we had never said much about the user's state anyway. In an action involving two or more software components, the team working on each component would make their own version of the action spec, that just defined its effect on their component.

...by ignoring some conjuncts

In some cases, masking is easy: the postcondition says "oneParticipant.someEffect AND theOtherParticipant.someOtherEffect". We just throw away the clauses that don't apply to us — the other components' teams will worry about them.

...or designing finer actions with the outside

Sometimes the postcondition is written in such a way that the effect on our component depends on something elsewhere. In the ATM, for example: "IF the account in the user's bank is in credit, THEN the user gets the money." This means that, in refining the action, we must include an action with the user's bank that transfers that information here. (This also applies to preconditions, which are equivalent to an if-clause like this.)

Masking should also make the actions directional, with a definite initiator and receiver: more like messages. However, they may still encapsulate a dialogue refined later. This enables us to summaries the component spec in the form of a type diagram, with model and action specifications:

We get down to fully directed actions

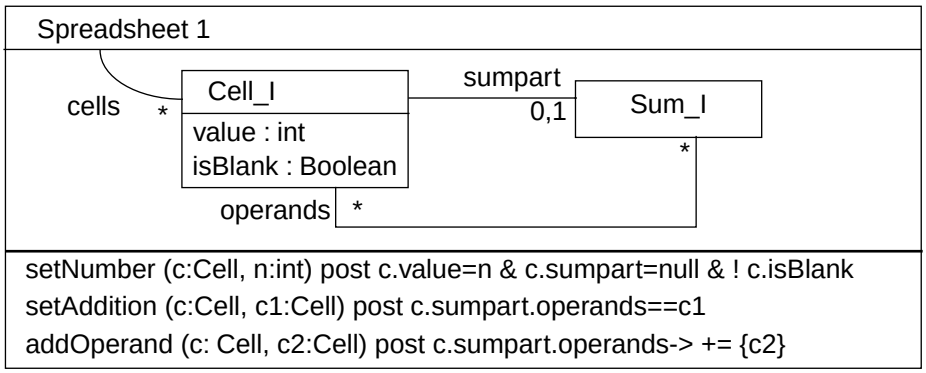


Figure 144: Localize and specify actions

We refine the actions down from the highest level business abstractions, but not right down to individual keystrokes or electrical pulses. We refine actions to the point where they make sense as individual actions on the objects we have chosen for our design — if you like, to the same conceptual layer as our design. “Make a spreadsheet” is too abstract for our design; “click mouse at (x,y)” is too detailed; but “setNumber(n)” makes sense as a message to one of our Cells.

But stop at the level of meaningful business ops

Such judgements are all relative. Working in the GUI layer, mouse clicks and keystrokes are what we deal with, and the job is to parse them into actions that can be dealt with by the spreadsheet core. Working in a workflow system, the creation of a spreadsheet might be just one action, so that from its point of view, the spreadsheet program is all a detail of the user interface that builds a series of cell-setting operations into one spreadsheet-creation. A good guideline is: “what would constitute a useful unit of work for the client?” i.e. something they would cheerfully pay for.

7.6.2.4 Localize each action to a constituent object

We now make each action the prime responsibility of one of the design objects internal to our component. Usually, a good choice is the proxy object that represents the external participant of the action — which might be a user or a piece of hardware, or a software object in a different component. If there is no such proxy — as in our spreadsheet, which has not representation of its user — then one of the action’s parameters is the next choice. We’ll choose the target Cell c. (In any case, a proxy often just sends the message on to one of the parameters. But there is good pattern, that all messages to and from an external object should go through their local proxy; which thereby keeps abreast of what its external counterpart is up to.)

Pick the primary internal ‘recipient’ object

With the assignment of a component action to one of its constituents must, of course, go the specification of what is supposed to achieve.

7.6.2.5 Operation refinement for each component-level action

Delegate internally

Now we work out how each component-level action is dealt with by the object it is assigned to — in other words, design the sequence, or write the operation code so as to meet the spec. Typically this will mean sending messages to other objects.

Internal actions before messages can help defer

We could put it more generally, and say that each object may initiate actions with one or more other objects: because you may know what you need to achieve, and who should be involved, but not care (yet) how, or who should be primarily responsible for each bit.

The receiving objects must themselves be designed, and so they then pass messages to other objects. The result is that many of the component's internal objects now have their own specified actions.

We repeat the procedure for each of the component-level actions. The component's responsibilities are now distributed between its constituents: this is the essence of object refinement.

7.6.2.6 Visibilities

Choose directional links

One last step is to decide who can see whom i.e. object attributes: we append arrow-heads to the links in our model. The simple criteria here is: any object needs some state corresponding to a link to every other objects that it must 'remember' across its operations. Once again, these links do not have to correspond to stored pointers in program code; they can themselves be abstractions, subject to further refinement.

In some cases, each object of a linked pair needs to know about the other. This in itself can add some final operations, relating to the management of the link. Important patterns are:

Two-way link

When two objects refer to each other, it is important that they do not get out of sync: pointing at different objects, or pointing at a deleted object.

Therefore, the only messages that immediately set up or taking down the link should come from the object at the other end. When an object wishes to construct a two-way link with another, it should set its own pointer, and then register with the other.

7.6.3 Documenting object conformance

Interaction diagrams

Interaction diagrams (also called 'object interaction graphs' or OIGs) are snapshots with messages added. They are useful for illustrating particular cases. One or more collaboration can be drawn for each component-level action.

For example, our spreadsheet Cell will respond to an addOperand by sending the message on to its Sum, which will set up an observer action with the target Cell:

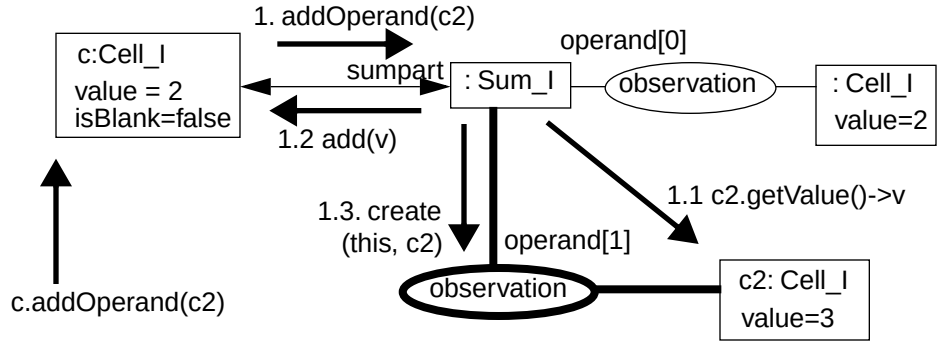
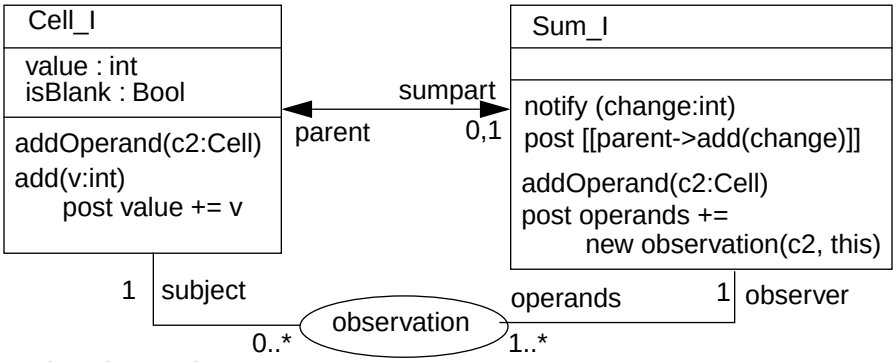


Figure 145: Internal interactions in spreadsheet

The diagram shows the new links and objects in bold. A new observation relationship is created; it is an ongoing action, characterised by its guarantee rather than its post-condition. It is treated just like an object in these diagrams. We have decided that, at this stage, we don't care how it works, although we will specify what it achieves.

Update the design model of types and classes

As is often the case when working out an interaction, we find we have brought new elements into the design. These should now be incorporated into the class diagram and specified. At the same time, we can begin to add localized actions to the classes:




```
action observation
guarantee subject.value <> subject@pre. value =>
          [[observer->notify(subject.value – subject@pre.value)]]
        -- whenever the value of the subject changes, the difference will be
           sent to the observer, using the notify message
```

Figure 146: Update class diagram with localized actions and specs

(Recall that the [[action]] notation is used within a postcondition to specify that another action has been performed as part of this one.)

We have specified the observation action; it may become an object in its own right, or (more likely) will turn out just to be a contractual relationship between its participants.

Notice that while the interaction diagram is useful for illustrating specific cases, the class diagram (with associated dictionary, specifications and code) is the canonical description of the design. As more collaborations are drawn, more detail can be added to the class diagram. Ultimately, we must also resolve 'observation' to specific messages and attributes.

A sequence diagram is an alternative presentation of the same information as is in a collaboration diagram, and is better at showing the sequence of events; but the collaboration diagram also shows the links and objects. As an example, here is the sequence showing how the propagation of changes works:

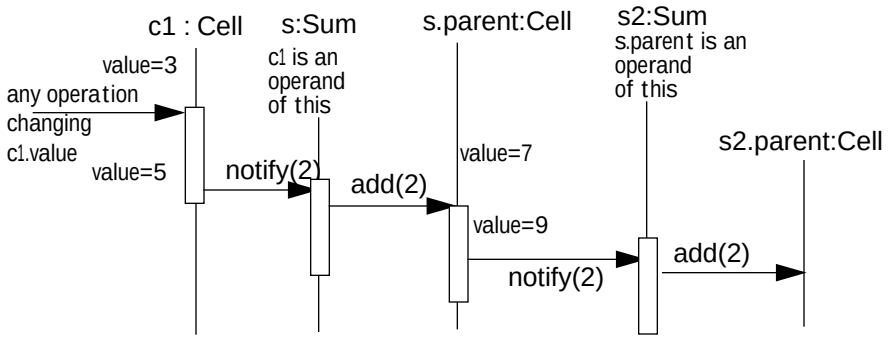


Figure 147: Sequence diagram alternative

7.6.4 Object access: conformance, façades, and adapters

How do the external objects get access to the appropriate constituent objects within our component?

7.6.4.1 Direct access

We can directly expose the internal objects

If the external objects and our component are all in the same body of software, there is a straightforward answer: they all have references to those of our constituents they want to communicate with. When we do an object refinement, the main task is to work out how the externals will initially connect to the appropriate internal. Our abstract component object is just a grouping of smaller objects, and we allow the boundary to be crossed arbitrarily.

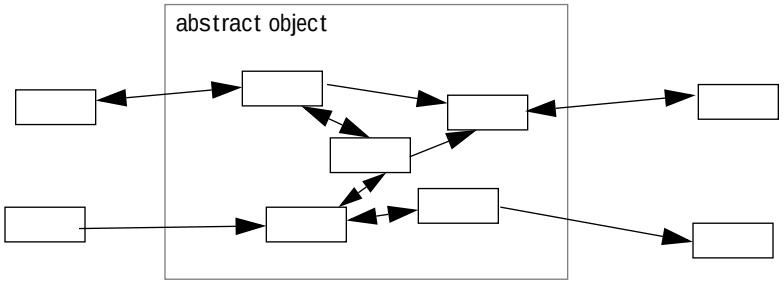


Figure 148: Direct external access to internal objects

There can be a variety of drawbacks to the direct access scheme, though each of them applies only in certain circumstances.

7.6.4.2 Façade

A façade is a single object through which all communication to a component flows. It simplifies several aspects of object refinement.

Or build a 'facade'

- The external objects do not have to be designed to know which constituent to connect to. In the Direct Access scheme, refining our object also means designing the externals to work with our organization.
  An business analogy is where a company provides a single point of contact for each customer or supplier. It saves them from having to understand the company's internal organization.
  A corollary is that, if the design of the external objects is already finalized, a façade is necessary.
- Invariants applying to the component as a whole can be supervised by the façade, since it is aware of every action that could affect it. Conversely, the external objects have no direct access to any objects inside.

The supervision of invariants is a common motivation. Consider for example a SortedList object, where the members of the list are NumberContainers; the value of a NumberContainer can be changed by sending it an appropriate message. The documented invariant of SortedList is that its members are always arranged in ascending order. But if external objects have direct access to the list's members, they can sneakily disorder the list by altering the values, without the anchor SortedList-instance knowing.

It makes maintaining invariants simpler

Making the façade the only way of accessing the list elements is one way to keep control; the other is to implement a more complex Observation scheme, whereby the elements notify the façade when they are changed.

7.6.4.3 Façade components

The big drawback of a single façade object is poor decoupling: as soon as you change or add to any of the types in the component, you need to extend the façade to cope with the new messages.

But the facade becomes coupled to the insides

However, we can make the façade itself an abstract object consisting of a collection of 'peers' — objects that each handle communication with a given class of internal object. Most GUIs are constructed this way: corresponding to each spreadsheet Cell, for example, will be a CellDisplay object, that deals with position on the screen and appearance; and also translates mouse operations back to Cell operations.

Instead, can have many smaller interface objects

In a more general scheme, different categories of external object may have different 'ports' through which to gain access — each of them a different façade.

7.6.4.4 Adapters

It's easy enough to draw links and messages crossing the boundary of a component; but what if the objects are in different host machines, or written different languages; or what if the external objects are machines or people? How do the messages cross the boundary, and what constitutes an association across the boundary?

How to refer across software boundaries?

Build an adapter

These objections are met by building an appropriate adapter — a layer of design that translates object references and messages to and from bits on wires (such as CORBA); or to pixels and from keystrokes and mouse clicks (the GUI). An adapter is a façade with strong translation capabilities.

Map object IDs to strings, screen positions, voice-prints

References across boundaries are typically dealt with by mapping strings to object identities: your bank card's number is the association between that real card object, and the software account object in your bank's computer; the spreadsheet cells can be identified by their row-column tags. The other common method is by mapping screen position to object identity, as when you point at its appearance on the screen.

The adaptor decouples from this mapping

It is because of the Adapter(s) that we can always begin the object refinement by assuming that the appropriate internal object will be involved in the external actions: we just leave it up to the Adapter design.

In the case of the spreadsheet, a considerable GUI will be required to display the spreadsheet; and to translate mouse operations into the specified incoming messages, and direct them to the appropriate Cells.

7.6.4.5 Boundary decisions in object refinements

Object access is an important design point

Object refinements, whether on a small or large scale, always involve some decision about how the boundary is managed — whether direct access is OK, or whether a façade or the more general adapter layer are required.

Our original picture of an object refinement as a simple group of objects has now become a group of objects plus a layer of adapters:

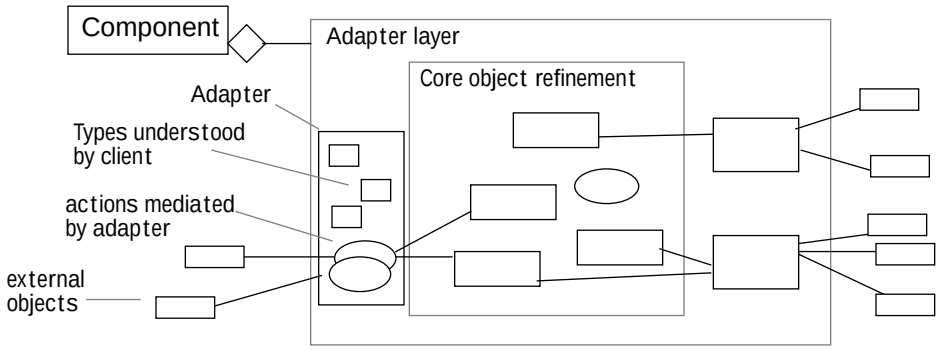


Figure 149: Object refinement with adapter objects

7.6.5 Object conformance — Summary

Object abstraction gives us the ability to treat a whole collection of objects — people, hardware, software, or a mixture — as a single thing with a definable behavior. This can be applied on the large scale of complete computer systems and businesses; or the small scale such as software sets and lists.

The key to documenting conformance is to show how the responsibilities of the abstract object are distributed to its constituents.

7.7 *Refinement #4: Operation conformance*

Operation refinement means writing a program that fulfills a given pre / post (and / or rely / guarantee) specification. This is where we finally get down to program code. In one sense, this is the bit we need to say least about, since the point of this book is not to teach programming: you know how to put together loops and branches and sequences, and can trivially learn any (far more verbose) graphical representations of the same. However, there are some special considerations, mostly about decoupling — since that is what OO programming is all about.

From operation spec to program code

7.7.1 What operation abstraction does and doesn't mean

Operation is a subtype of Action: operations are actions consisting of an invocation from a sender to a receiver, followed by some activity on the part of the receiver, and possibly ending with a return signal to the sender; they are a one-sided view of the sender-receiver interaction.

Operations are one-sided actions

If an operation is invoked in a situation when the precondition is false; or if, during the execution of the operation, its rely condition becomes false: then the specification does not state what the outcome should be.

Some cases can always be left unspecified

If an operation is invoked in a situation when the precondition is true and the rely condition is true throughout, then by the end of the operation, the postcondition will have been achieved; and during its execution, the guarantee condition will have been maintained.

Others must have guaranteed outcomes

Unstated pre- and rely conditions are equivalent to 'true' — no obligation on the sender. Unstated post- or guarantee conditions are equivalent to 'false' — no obligation on the implementer.

The operation should not alter variables which could be left untouched whilst achieving the post and guarantee conditions.

Any invariant in the model should be considered to be ANDed to both the pre and postconditions (but not the rely and guarantee conditions).

7.7.2 Strong decoupling

In conventional programs, you split a big program up into subroutines so that common routines can be invoked individually; and so that the program is easier to understand. In OO programming, the process goes even further: for every statement, you think: Which object should this be attached to? Which object has the information and the other operations most strongly relevant to this? Send off a message to the object that should. Actually, that's not for every statement: it's for every subexpression.

OO design imposes *more* decisions: *who?*

The purpose is to ensure the program is well decoupled. If the preceding stages of design have gone well, many of these questions should be sorted out. However, you definitely do have more decisions to make in an OO development than in procedural.

Done well, this provides decoupling

7.7.2.1 Parameters and Internal variables

Declare all variables as types, not classes

The choice of local variable types — temporaries, inputs, and even return values — for an operation is crucial to good decoupling. As soon as you declare a variable or parameter that belongs to a particular class, you have made your part of the program dependent on that class. That means that any changes there may impact here. Instead, it is usually better to declare all variables as *types*; the only place in a program where you absolutely have to refer to a concrete class is to instantiate a new object.

There is a variety of powerful patterns to help reduce dependencies.

Role decoupling

A variable or parameter contains an object or a reference to one. Frequently, the component in which the declaration occurs will use only some of the facilities provided by that object.
Therefore, define a type (= interface / abstract class + specification) that characterizes only the features that you will use. Declare the variable to be of this type. Declare the object's class to be an implementer of this type.

This will minimize the dependency of your component on the other object. Related patterns are *Adapter* and *Bridge* [Gamma 94].

(Role decoupling is particular important for two-way links.)

Use factories for instantiation

Factory

An object has to be created as a member of a definite class. This gives rise to a dependency between your class and that.
Therefore, devolve to a separate class, the decision about which class the new object should belong to. By doing this, you encapsulate the class dependency behind a type-based creation method. [Gamma 94]

7.7.3 Opspec conformance

How to check that implementation meets spec?

Whether we have written them informally or in more precise form, we should have some specification of what is expected of each operation. We should attempt to check that the program code of each operation conforms to what is expected of it.

Test, or show correctness? Both are useful

The most reliable method is to turn them into test harnesses as we discussed earlier. But there is also the Deep Thought approach: as a general principle, it is possible to inspect both specification and code, and check that one meets the other by a judicious mixture of careful reasoning and guesswork. Although Deep Thought is not an economical method of verification outside safety critical circles, being aware of the basic principles can help anyone root out obvious mistakes before getting to the testing stage.

...depending on code reused vs. written

There are two ways of fulfilling a spec. The hard way is to write the entire implementation yourself. The easy way is to find something that comes close to already does the job — and possibly bend the requirements to suit what you've found. The latter is usually more economical, if you're fairly confident of its provenance and integrity (See Section 11.7, "Heterogenous components," on page 472, for example).

7.7.3.1 Comparing two operation specs

For reused code with specs

Suppose we find some code in a library, and suspect it does the job we require. Miraculously, it comes with a spec and model of some sort, which we want to compare to our requirements spec.

...the two specs should be compared thusly

The rules for comparing two pre / postcondition specs are:

- Any invariants in each model should be ANDed with both pre and postconditions written in that model.
- If the requirement has several pre / postcondition pairs — which may come from different supertypes — they should be ANDed in pairs: (pre1=>post1) AND (pre2=>post2) to give an overall requirement postcondition; if the pre / post conditions are not from supertypes, and are subject to 'joining' (Section 9.4.4, "Joining action specifications," on page 373), compose them accordingly.
- The implementation's vocabulary should be translated to the specification's by using retrieve functions as in Section 7.4 (Refinement #1: Model conformance).
- The precondition of the requirement should imply the precondition of the implementation. For example, if the requirement says "this operation should work whenever Cell c contains a Number", then an implementation that works "whenever c is non-Blank" is fine — because it is always true that: "a Cell contains a Number => it is not Blank".
- The postcondition of the implementation should imply the postcondition of the specification. (Notice the other way around from preconditions — a feature called "contravariance"). So if the implementation claims "this operation adds 3 to the Cell's value" while the requirement is more vague: "this operation should increase the Cell's value" — then we are OK, because it is always true that: "3 added to value => value increased".

7.7.3.2 Comparing operation specs with code

But reused code usually has no specs

More likely, you will have to compare your specification with code (whether you've written it yourself, or someone else has). Once again, the most effective strategy is to write test harnesses, as we discussed before.[1]

Turn on pre / postcondition checking in debug more

In debug mode, preconditions should be written as tests performed on entry to an operation; postconditions are tested on the way out. The only complication is that, since postconditions can contain "@pre", you have to save copies of those items while checking the precondition. (Or you may be lucky enough to be using a language with this facility built-in.) The main caveat is to ensure that the pre and postconditions don't themselves change anything.

Deep Thought 'lite' can help

If you insist on the Deep Thought approach, many (rather academic) books have been written on how to document this kind of refinement. [E.g. Jones, Morgan.] To summarise their works:

1. For more on the Deep Thought approach, see Morgan 88 or Jones 86. For more on executing pre and postconditions, see Meyer 88.

Write assertion for intermediate states

- For sequences of statements separated by ";", it is useful to write assertions within the sequence — conditional expressions that should always be true. This helps see how the requirements are built up with each statement, and can be a useful debug tool if the expressions are actually executed. The assert macro provided with C++ can be parallelled in other languages.
- Where some operation is called within a sequence, its precondition should be satisfied by the assertion immediately before it; and the assertion after it should be implied by its postcondition.

Branches

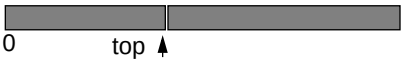
- If-statements ensure preconditions for their branches. The postcondition of the whole thing an OR of the branches':

```
if (x>0) z= square_root(x)          // pre of square_root is 'x>=0'
    else z=square_root(–x);
assert z*z == x  or  z*z == –x
```

Loops

- In a loop, it is useful to find the loop invariant: an assertion that is true every time round the loop. It works as both pre and postcondition of the body of the loop. Together with the condition that ends the loop, it ensures the required result of the loop. For example, in this routine:

```
post sort queue into order of size
{   int top= queue.length;
    while (top > 0)
    invariant everything from top to end of queue is sorted,
    //   and everything before top is smaller than everything beyond top
    {   post move biggest of items from 0 to top along to top
        { ... to be written ... }
        top = top –1;
    }
    // top == 0 and everything from top to end of queue is sorted
}
```



the invariant separates the queue into two parts, unsorted and sorted. The 'top' pointer moves gradually downwards, until the whole queue is in the sorted part.

If you were called to review a refinement like this, you should check that (1) the claimed invariant is bound to be satisfied on first entry to the loop (which it is here, because there's nothing beyond top); (2) if the invariant is true before any iteration of the body, then it is bound to be after; (3) that the invariant and the exit condition together guarantee the effect claimed for the whole thing (true in this case because when top gets to 0, the whole queue must be sorted).

Notice how a postcondition has been written to stand in for a chunk of code not done yet; we are simply using techniques we know to defer details, except now in code. (How would you design it? What would be the loop invariant?)

7.7.3.3 Operation conformance and action conformance — Differences

Both operation and action conformance show how a single action with an overall goal is released by a composition of smaller actions, and some of the techniques for establishing the relationship are the same. The differences are:

- An operation begins with an invocation — a function call or message-send. This prescribes the limits on the sequence of events that should happen. So we know from the beginning of the sequence, which operation we're dealing with.

  Operations start with a specific invocation

  An action is identified only in retrospect. If someone selects some goods in a shop, it will often be the first step to a sale; but they might decide not to buy, or they might walk off without paying; so it turns out not to have been a sale, but a theft. An action is a name for an effect. We can provide a protocol of smaller actions whereby it can be achieved, but the same actions may be composed in some other way to achieve something different.

  An action may start in many different ways; each start may end in different actions

- When we design a procedure that refines an operation spec, we can focus on the receiving object, and devolve subtasks out to other objects we send messages to. If we change the design, we are changing this one object.

  Operations are one-sided

  When we design a protocol of actions that satisfies an action spec, we focus not on any one participant, but on the interactions between them. If we change the refinement, we are affecting the specs of all of the participants.

  Actions involve multiple parties

- Refining an operation is like designing a computer monitor: you have the specs of the signals that come along the wires, and their required manifestations on the screen. Your work is to define the insides of the box; you may involve others by specifying parts you will use inside. Your design talks about the specifications of the constituent parts, and how they are wired together.

  Operation refinement is encapsulated

  Refining an action is like devising an interface standard for the signals on a video connection. You have to involve all of the designers who might make a box to go on either end of this connection — or at least, all those you know at the moment. You have to agree a specification for what is achieved, and then refine it to a set of signals that, in some parallel or sequential combination, achieve the overall required effect.

  Action refinement affects all participants

  You cannot talk in terms of anything inside the boxes, because they will all be different; you can only make models that abstract the various boxes, and talk in terms of the signals' effects on them.

- To test an operation refinement, you push a variety of signals in, from different initial states, and see if the right responses come out. There are various ways to use a precise specification to generate test cases that reasonably cover the state space of input parameters and initial state.

  Test an op by response

  To test an action refinement, you have to see whether it permits a variety of different combinations of participants to collaborate together, and achieve the desired effect. Interoperability is the general goal. The goal here is to explore each sequence of refined actions that should realize the abstract one — a much larger, sequential state space. Scenarios defined during modeling form a useful basis.

  Test an action by traces and net effect

7.7.4 Operation conformance

Operation abstraction makes it possible to state what is required of an object, without going into the detail of how.

7.8 *Statecharts and refinement*

Transitions can take time

In the Catalysis interpretation, the states in a statechart are boolean conditions; and the transitions are actions. As such, actions may take time; there is a gap between one state going false, and the next state lighting up (rather like the floor indicator in many elevators).[1]

States are boolean functions of attributes

Most statecharts focus on a given type, and the states are defined in terms of its attributes. For example, we might define the statechart of a Cell, as containing Blank, a Number, or a Sum.

We would have to wait for the user to enter two operands before the transition to becoming a Sum was completed. In between times, the implementation is in no state that the abstract spec understands. The addition of the intermediate state is a valid change specifically in a refinement:

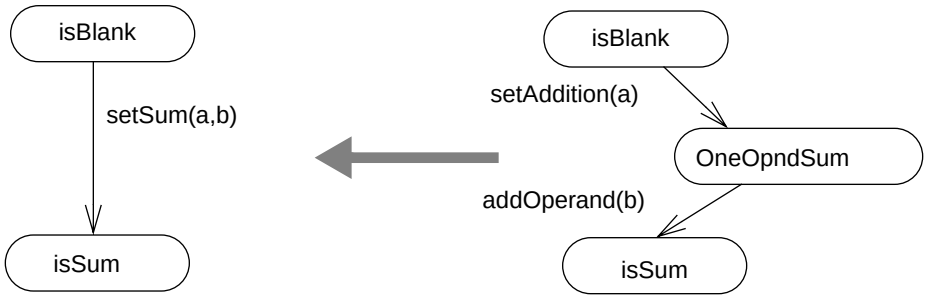


Figure 150: Intermediate states due to refinement

Each state in a statechart should be documented with a condition stating when it is true. In this case, isBlank corresponds directly to the attribute of that name; isSum == (content : Sum) ("the content attribute is of type Sum").

States are retrieved just like any attribute

Since states are just boolean attributes, retrieving a state is no different than retrieving any model. The abstract states can still be seen in the refinement, though you have to use the abstraction functions to translate from sumpart to content. The extra state has to be given its own definition.

1. We prefer our statecharts like this, as they are easier to reconcile with the actuality of the software compared with 'instantaneous event' models; and they tie in better with the actions. In some notations, they would have an event representing the start of an action, and another for the end of it, and a state in between representing the transitional period. But at a given level of abstraction, we do not know enough to characterize the intermediate state, because that is only defined in the more detailed layers of the model.

7.8.1 States definitions are not restricted

States for cell selection

Considering the GUI for a moment, it may be useful to make a small statechart about whether a Cell is selected or not.

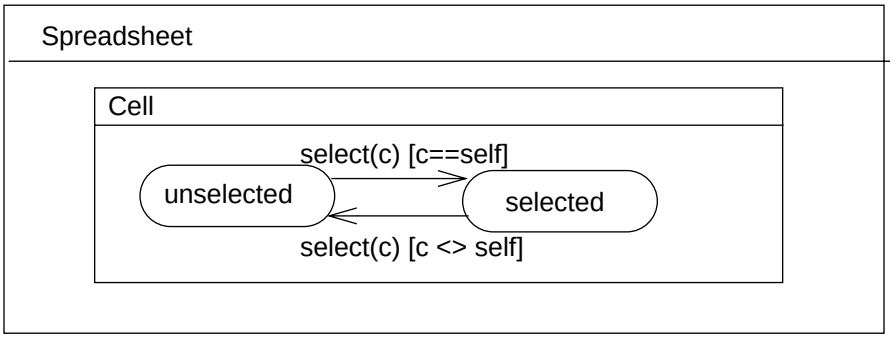


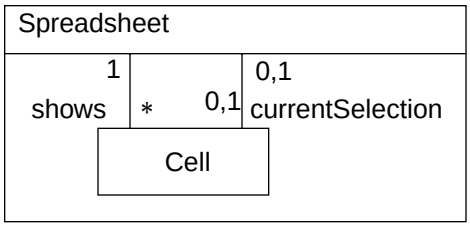
Figure 151: Cell selection statechard

This defines (part of) the spreadsheet state

The Cell type forms part of the Spreadsheet's model; the actions are those of the spreadsheet as a whole (Section 4.9.3, "State Charts of Specification Types," on page 177). The states diagram applies simultaneously to every Cell in the model. Any Cell that is the subject of a select operation gets into the selected state; all others go into unselected, as defined by the guard.

Add attributes to define the state

To accommodate the new bit of information, we could add an optional link to the model. It is then easy to write a definition for the selected state:



Cell:: selected == (~currentSelection ≠null)

Figure 152: Link for indicating cell selection

But in the code, a 'cell' has no 'selected' state

Now, when we come to the implementation, the pointer to the currently selected Cell comes from somewhere in the GUI, and since the pointer is one way, the Cell itself does not know that it is selected. Is the state diagram still valid if the Cell has no information about its state of selection?

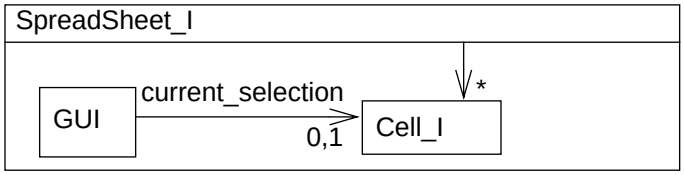


Figure 153: Implementation: does a cell still have a "selected" state?

That's OK; just interpret carefully

Yes, it is. For one thing, the state diagram is of a Cell, not a Cell_I; and the abstraction function to Spreadsheet from Spreadsheet_I should yield all the information about each Cell. And in any case, it would still be valid to draw that diagram for a Cell_I: we just have to define the state with a little more imagination — something like:

```
Cell :: selected = (there is a Spreadsheet_I si, for which
                        si.gui.current_selection == self)
```

The 'there is' means in practice that you have to go searching around the object space until you come to a Spreadsheet, and try it for that property; but we did say that specification functions don't have to execute efficiently to be meaningful.

7.8.2 Other statechart refinement rules

If your project uses statecharts extensively, you may wish to look at [Cook & Daniels], where a complete set of rules for refining statecharts is provided.

7.9 *Abstraction, refinement, and testing*

The various kinds of abstraction make it possible to discuss the essentials of a specification or design, clear of the details; and to define systematic approaches to verifying that an implementation does what it was supposed to do.

Catalysis provides a coherent set of abstraction techniques, and also provides the rationale to relate more detailed accounts back to the abstractions.

The verification techniques can be applied in varying degrees of rigor, from casual inspection to mathematical proof. In between, there is the more cost-effective option of basing systematically defined test code on the specifications.

7.10 *Process Patterns for Refinement*

Here are a set of high-level process guidelines for applying refinement techniques.

Pattern-119, *The OO Golden Rule (Seamlessness or Continuity)* (p.325): how to achieve one of the most important benefits of objects, a seamless path from problem domain to code.

Pattern-120, *The Golden Rule vs. Optimization* (p.327): when stringent performance requirements keep you from maintaining straightforward continuity to code.

Pattern-121, *Refinement is a relation, not a sequence* (p.328): do not make the common mistake of thinking that refinement means top-down development; it is a fundamental relation between different descriptions, regardless of which one was built first.

Pattern-122, *Recursive refinement* (p.330): the ideas of refinement apply at all levels, from describing organizations and business processes, to program code. This means it can form the single consistent basis for traceability.

Pattern-119 The OO Golden Rule (Seamlessness or Continuity)

Summary

Build a system that mirrors the real world; and keep it that way.

Intent

An object oriented design is one in which the structure of the designed system mirrors, to the extent possible, the structure of the world in which it works. Many of the advertised benefits of object technology come from this. It is to this end that languages supporting OOD provide mechanisms that simulate the 'real' world's dynamic interaction between state-storing entities; and this is the reason that object technology originates in simulation techniques and languages (such as Simula). Objects have a strong relationship to the AI subculture's 'frames', which are units of an agent's understanding of the world around it.

Considerations

Truth and Reality. Obviously you have to begin by making a model of the real world. But what does 'real' mean?

- One person's view of it will be different from another's. There are numerous views of reality. Many of them may be self-consistent, but cast in terms different from each other. It is important that the model of each (class of) user must be clearly reflected in the system
- There are numerous ways to use the notation to model the same set of ideas.
- While constructing a model, you will ask and (hopefully) get resolved many questions that were never resolved until now. This is a good thing, but you have to be aware that you are actually constructing reality, not just passively discovering it; and again, the different interested parties will have different views on what the answers ought to be.

Compromise with practicality. The design that mirrors the users' concepts most closely will not always be the most efficient. Compromises must be made, and there is an architectural decision about how far to do so. Fortunately, this can be taken to different degrees in different parts of the design — see Pattern-120, *The Golden Rule vs. Optimization* (p.327).

Strategy

Build and integrate users' business models. See Pattern-15, *Make a business model* (p.553)

Cast system requirements in terms of business model. See Pattern-33, *Construct a system behavior spec* (p.594)

Choose classes based on business model. From system model. Deviations forced by performance and other constraints should be local and clearly documented as refinements to maintain traceability.

Maintain development layers (business model to code) in step. This clearly conflicts with the usual short-term imperative of getting changes done last week; but the changes are normally localized, and experience shows clear long term benefits. Furthermore, since the documents as you go up the tree are more abstract, you come to a point where there is no change: performance improvements, for example, will usually change the code but not the requirements or the model in which they are expressed.

Build many projects on same model. But don't take this as a reason for perfecting a model before building your first system — see Pattern-15, *Make a business model* (p.553).

Benefits

Many of the advertised benefits of object technology come from what is variously called 'continuity', 'seamlessness', or 'the OO Golden Rule':

- The resulting system relates well to the end users. It's easy to learn because it deals in the terms they are familiar with. The relationships are as they expect.
- Changes are easy to make because users express their requirements in terms that are easy to trace through to the model.
- The same business model can be used for many projects within the same business; and of course, much of the code can also be generalized and re-used.

Pattern-120 The Golden Rule vs. Optimization

Summary

Consciously trade-off localized performance optimization against seamless design.

Intent

Ideally, the structure of the design is based on the structure of the world in which it works (to an appropriate extent). Performance constraints will sometimes dictate against such continuity, and elements from architectural design down may need to differ from a 'real-world' model. How to balance the goals of maintainability and tuned performance?

Considerations

Should the program code be object oriented? If it is, how do we choose the classes?

Mirroring the world in code exactly is very good for simulations: every time you change your picture of the world, you can easily find which bits of the code to change. This is how OOP originated with the language 'Simula'; and also what makes it great for writing any kind of program whose requirements change regularly — that is, just about all of them. It's been estimated that 70–80% of total spend in a program's lifecycle is in the maintenance phase, after first delivery.

On the other hand, we gain that flexibility at the cost of making all those decisions about which object should take responsibility for each small part of the job; and at the cost to run-time performance of all those objects passing control to one another. This can make a big difference in, for example, communications or process control software, where throughput is vital.

Flexibility comes from decoupling — making components independent of each other. Optimization for performance generally means that pieces of code that were ideally decoupled become dependent on each other's details. So the more you optimize, the more you mix concerns that were independent before — so that in the extreme you end up with a traditional monolithic program.

Strategy

1. Make it an upfront architectural decision how much you're going to optimize for performance against code flexibility. If your clients shout for little functional enhancements every day (typical for in-house financial trading software), optimize the underlying communications stuff, but leave the business model pristine. But if your software will be embedded in a million car engines for ten years, optimize for performance.
2. 80% of the execution is done by 20% of the code. Design your system in a straightforward object-mirroring way, then abstract and re-refine the pieces that are going to be most critical to performance. (Or analyses a prototype and worry about the execution hotspots.) See π 10. "Refine the system type spec" (p.32)
3. Buy faster hardware and more memory. It's a lot cheaper than programmers.

Benefit

An OO program is one that is refined from an OO design. A rigorously applied refinement carries a 'retrieval' that relates it to the abstraction, even when the design has been optimized from a real-world abstraction.

Pattern-121 Refinement is a relation, not a sequence

Summary Use refinement for any combination of top-down, bottom-up, inside-out, or assembly-based development; it does not imply sequential top-down development

Intent Realistic deliverables for each development cycle, depending upon what development process is most suited to the project.

We have a clear picture of the ideal refinement relations from business model to code. How is this related to the actual series of cycles of the development process?

Considerations Clearly there are some dependencies between prior phases and consequent ones. Even the most unregenerate hacker does not begin to code without at least some vague idea of an objective (well; not many of them, anyway!)

But it is obstructive to be too concerned about completing all the final touches on any phase before going forward to the next. This is a well known demotivator, paralyzing the creative processes — and is often an excuse for people who don't know how to proceed.

But again, if too much is undertaken without clear documentation of the aims of each phase (as determined by the outcome of the preceding ones), all the usual misunderstandings and divergences will arise between team members, and between original target and final landing.

In any case, it's an illusion that the design is determined by the requirements. It's often the way other around. When they discovered the laser, it was not so they could make CDs. Few of us knew we needed musical birthday cards before they were provided for us. We build and use what we have the technology for.

Strategy The typical deliverables of a development cycle (Pattern-39, *Interpreting Models for "Clients"* (p.606)) are *not* individual completed documents from the linear lifecycle. More usually, they might be "first draft requirements; GUI mockup; client feedback; second draft requirements; critical core code version 1; requirements modified to what we find we can achieve; ...."

Get it 80% right. To avoid "analysis paralysis" — the tendency to want to get a perfect business model, requirements, or high-level design before moving on — deliberately start the next phase when you know its precursor is still imperfect. As early as you like, but no later than when you estimate there's still about 20% finishing off to do. (Personal wisdom about the exact figures varies of course.) The results from the next phase will in any case feed back to the first.

Bottom up and top down. The proposition that the delivered system should be documented in such a way that the spec is consistent with the design (and other layers of documentation) in no way constrains you to begin with the requirements and end with the code. Called upon to write an article, there's no need to begin at the beginning — you edit the whole in any order you like, as long as it makes sense when you've finished.

In the same way, consider the various documents of a development to be elements in a structure that you are editing. The goal of the project is to fill in all the slots in the structure and make them all consistent when done. Feel free to begin at the bottom; and don't worry if things get inconsistent en route, provided they come back in line at some planned milestone.

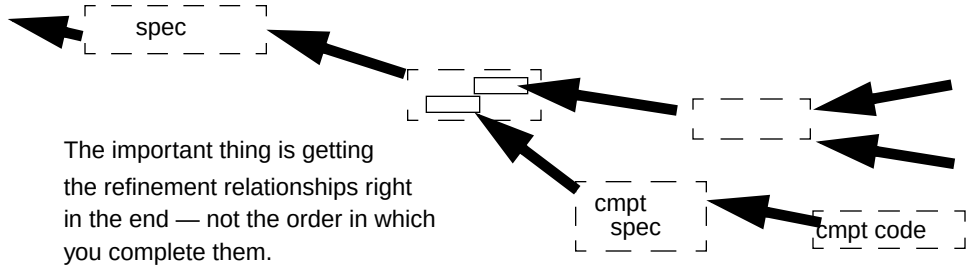


Figure 154: Refinement is a relation, not a sequence

Coding can even help with "analysis". People know what they *don't* want better than they know what they *do* want. (Ask any parent!) When you put a finished system in front of a customer, they'll soon tell you what changes they need. And the system will alter their mode of work, so their requirements will change anyway. To circumvent some of this, deliver early: a slide show or a prototype or a vertical slice — whatever will stimulate the imagination. OO is great for this incremental design. But it must be a clear part of the plan:

- What information will be extracted and fed back to analysis from an early delivery? (And how will the information be obtained? And is the planned delivery adequate to expose that information?)
- What will happen to the code? Is it a throwaway? If not, all proper documentation and reviews must be applied. Prototypes are too often excused proper QA — and then incorporated into the real thing!
- What will happen to the other design documents? The design material usually represents more valuable work than the code itself — so you can indeed build a prototype as a throwaway, even in some other language. This assumes that, as a reader of this book, you're documenting the design in some form other than just the code.

Pattern-122 Recursive refinement

Summary Establish a traceable relationship, based on refinement, between most abstract spec and detailed implementation (program code).

Intent The abstract spec should bears a systematic relationship to the code, expressed as a series of refinements. Many of these are decompositions, which break the problem into smaller pieces.

We wish to document the outcome of design.

Strategy

- Use frameworks to build specifications.
- Model separate views.
- Compose views or frameworks into one spec.
- Refine spec using one of the standard refinements. Some refinements are 1-1; some are decompositions. Decompositions split the spec into several separate specs, each of which can be dealt with as a separate goal.
- Each decomposition is documented with a refinement that states how the constituents work together to fulfill the abstract spec.
- Each decomposition divides the job into a set of constituents, each with a specification that can be fulfilled separately. The same principles can be applied to its design, recursively.
- 'Basic design' short-cuts some of the recursive refinement process by going straight for a set of decisions that accept the specification types as proto-classes. A judicious combination of basic design, subsequent optimization, and recursive refinement is practical.

Result A design traceable through the refinements.