Specifications

by *Eric Evans* <evans@acm.org> and *Martin Fowler* <fowler@acm.org>

Introduction

One of the nice things about working in Silicon Valley is that almost any time of the year you can walk around outside while discussing some design point. Due to serious follicle impairment Martin needs to wear a hat when doing this in the sunshine. Eric was talking to him about the problems of matching shipping services to the requests customers make and the similar process as the shipper makes requests to contractors. There was some similarity here that suggested an abstraction. Eric had been using a model he called "specification" to represent the requirements for a route and to ensure that cargoes were stored in proper containers. Martin had come across similar problems before. In a trading system people were responsible for looking at the risk implied by some subset of the bank's contracts. Healthcare observations could be made on a population to indicate typical ranges of some phenomenon type (such as the breathing flow for a heavy smoker in his 60's). He suggested some extensions to the model. From that point it was clear that there were patterns in our experiences, and we could pool our ideas. As we explored the patterns further, they helped clarify the abstractions for Eric's project and gave Martin ideas on how some of the analysis patterns in his book [Fowler] could be improved.

A valuable approach to these problems is to separate the statement of what kind of objects can be selected from the object that does the selection. A cargo has a separate storage specification to describe what kind of container can contain it. The specification object has a clear and limited responsibility, which can be separated and decoupled from the domain object that uses it.

In this paper we examine the specification idea and some of its ramifications in a series of patterns. The central idea of *Specification* is to separate the statement of how to match a candidate, from the candidate object that it is matched against. As well as its usefulness in selection, it is also valuable for validation and for building to order.

This paper is not going to go into details about how a specification is implemented. We see this as an analysis pattern, a way of capturing how people think about a domain, and a design pattern, a useful mechanism for accomplishing some system tasks. We do have some sample code, though, as is rather too glib to talk about objects that have all these capabilities without at least outlining how this could be done. Also, there are consequences to different implementations. We will look into three implementation strategies you can apply to specifications. A *Hard Coded Specification* implements the specification as a code fragment, essentially treating the specification as a *Strategy* [Gang of Four]. This allows a great deal of flexibility, but requires programming for every new specification. A *Parameterized Specification* allows new specifications to be built without programming, indeed at run time, but you are limited as to what kind of specifications you can build by what the programmers have set up. Although programmers can be generous in providing parameters to customize, eventually they can make the parameterized specification too complex to use and difficult to maintain. A *Composite Specification* uses an interpreter [Gang of Four] to cut a very agreeable middle path. The programmers provide basic elements of the specification and ways to combine them, later users can then assemble specifications with a great deal flexibility.

Once you have used specification, a powerful pattern that builds on it is *Subsumption*. The usual use of specification tests the specification against a candidate object to see if that object satisfies all the requirements expressed in the specification. *Subsumption* allows you to compare specifications to see if satisfying one implies satisfaction of the other. It is also sometimes possible to use *subsumption* to implement satisfaction. If a candidate object can produce a specification that characterizes it, the testing with a specification then becomes a comparison of similar specifications — removing the

coupling of specification mechanism entirely from the domain. *Subsumption* works particularly well with *Composite Specifications*.

Sometimes it is useful to think about cases where you have a *Partially Satisfied Specification*. You may not find any matches for the complete specification. Or you may have completed part of the specification and need to consider alternatives for the part that remains.

Problem	Solution	Pattern	
You need to select a subset of objects based on some criteria, and to refresh the selection at various times You need to check that only suitable objects are used for a certain role You need to describe what an object might do, without explaining the details of how the object does it, but in such a way that a candidate might be built to fulfill the requirement	Create a specification that is able to tell if a candidate object matches some criteria. The specification has a method isSatisfiedBy (anObject) : Boolean that returns "true" if all criteria are met by anObject.	Specification	
How you implement a specification?	Code the selection criteria into the isSatisfied method as a block of code Create attributes in the specification for values that commonly vary. Code the isSatisfiedBy method to combine these parameters to make the test. Create a particular form of interpreter for the specification. Create leaf elements for the various kinds of tests. Create composite nodes for the and, or, and not operators (see Combining Specifications, below).	Hard Coded Specification Parameterized Specification Composite Specification	

How do I compare two specifications to see if one is a special case of the other, or is substitutable for another?	Create an operation called isGeneralizationOf(Specification): Boolean that will answer whether the receiver is in every way equal or more general than the argument.	Subsumption	
You need to figure out what still must be done to satisfy your requirements. You need to explain to the user why the specification was not satisfied.	Add a method "remainderUnsatisfiedBy" that returns a Specification that expresses only the requirements not met by the target object. (Best used together with Composite Specification).	Partially Satisfied Specification	

The Need for a Specification

Imagine a trader in a bank who is responsible for a managing the risk on some set of contracts that the bank deals in. He might consider a portfolio of all contracts that involve the yen. A colleague, concerned about a particular set of risks wants to examine all Deutchmark contracts with US based companies. Each of these traders wants the ability to specify selection criteria for the kind of contracts that they want to monitor in their portfolio, and for the system to remember these selection criteria. They might look at the resulting portfolio at regular intervals, or they may want to keep the portfolio always on the screen and have it automatically recognize any new contracts that fit the conditions that they are concerned about. The portfolio is responsible for **selection** of appropriate contracts.

Imagine a shipping company that moves various kinds of goods around the world. It has a cargo of meat that it wants to put in a container. Not all containers would be suitable for such a cargo. To hold meat a container must be refrigerated (known in the trade as a "reefer") and must be clean enough to safely store food. Whenever a cargo is assigned to a container you need to check that the container is a suitable container for the cargo. The system has the responsibility for **validation** when we assign a cargo to a container.

Imagine a customer booking a shipment. They want to move meat from Des Moines to Beijing entering China at Shanghai. There are various routes that a shipping company could use to do this, but they need to know the constraints on the routes in order to construct an appropriate itinerary. This way they can **build** the route **to order** from the request.

Specification

Problem

Selection: You need to select a subset of objects based on some criteria, and to refresh the selection at various times

Validation: You need to check that only suitable objects are used for a certain purpose

Construction-to-order: You need to describe what an object might do, without explaining the details of how the object does it, but in such a way that a candidate might be built to fulfill the requirement

Solution

Create a specification that is able to tell if a candidate object matches some criteria. The specification has a method isSatisfiedBy (anObject) : Boolean that returns true if all criteria are met by anObject.

Consequences

+ decouple the design of requirements, fulfillment, and validation

+ allows clear and declarative system definitions

Each of these situations is common in business systems. Although they seem different they share a common theme. This theme is the notion of some test that we can apply to candidate objects to see if they fit our needs, this test is a specification of what is needed. The specification is, in essence, a boolean function which applies to the candidate which indicates whether or not it matches. If you use specification in a strongly typed language, you usually set the parameter type of isSatisfiedBy to be the type of candidate object you are looking for.

Uses of Specification

The trader defines a specification for the kind of contracts that he wants to monitor in a portfolio. When the portfolio is created it will look at all contracts and select those that satisfy the specification. Whenever a new contract is created, it may have to check itself against existing portfolios to see if it belongs to some of them. Alternatively, the portfolio's contents might be derived by a query each time it is used. You can think of the portfolio as a subset of contracts that is defined by the specification.

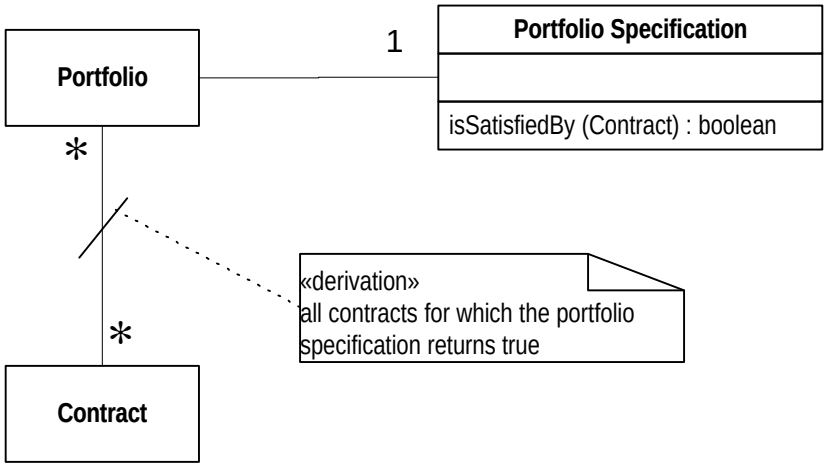

Figure 1. Using specifications to describe the selection criteria for portfolios in contracts (see [Fowler] §9.2). All diagrams use the UML notation [Fowler, UML]

A similar situation appears when you want to make an observation for a group of people — such as typical ranges for respiratory function of heavy smokers over 60. The population has a specification which selects appropriate patients as falling into the population. Observations can then be made of patient, or of the population.

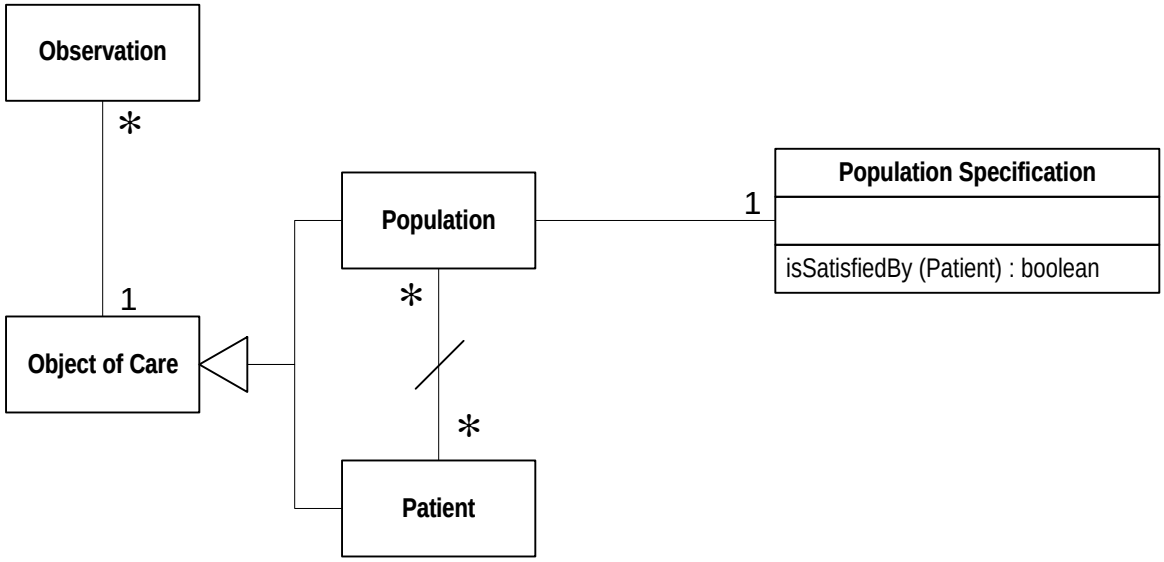

Figure 2. Using specification to define populations in healthcare (see [Fowler] §3.5 and §4.1)

In sales organizations an important part of the business is allocating sales to an appropriate sales representative. This is important to the sales rep as commisions are based on how many sales occur in the territory that the sales rep is responsible for. The territory is a specification which selects deals. Typically they are based on geographic area (often with postal codes), size of the deal, the industry group of the customer, and large customers are individually named. Complications include ensuring that territories are not overlapping. Either the specification should only match one territory, or it should ensure that there is a way of prioritizing or pro-rating between several matched territories.

Temporal specifications can be used to describe recurring calendar events. The elements in these specifications may include the day of the week, a period within a year, specific named days, and fixed length intervals from some seed date. Complications include handling how regular business events can be put off by holidays.

Whenever a shipping company assigns a cargo to a container it checks the cargo's storage specification to ensure that the container is appropriate for the cargo. You can think of this as a constraint on the container for a cargo.

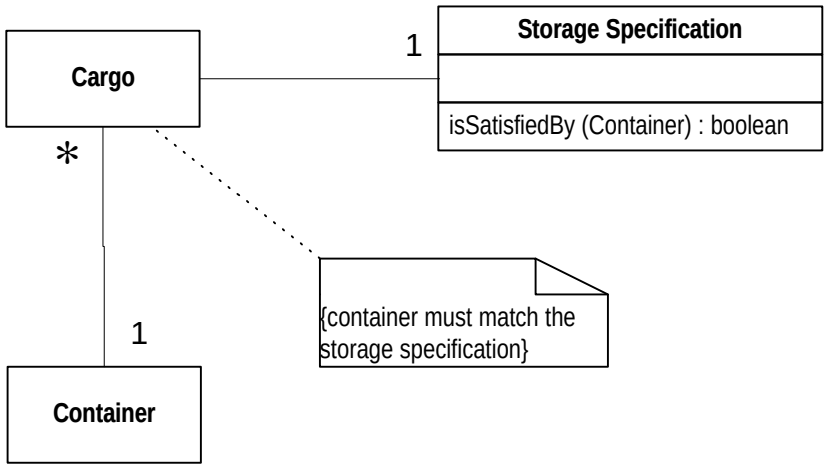

Figure 3. Using specifications to constrain which containers can be used for transporting a cargo.

The customer provides a route specification for the shipment. The shipping company will use the specification as a constraint in coming up with a route.

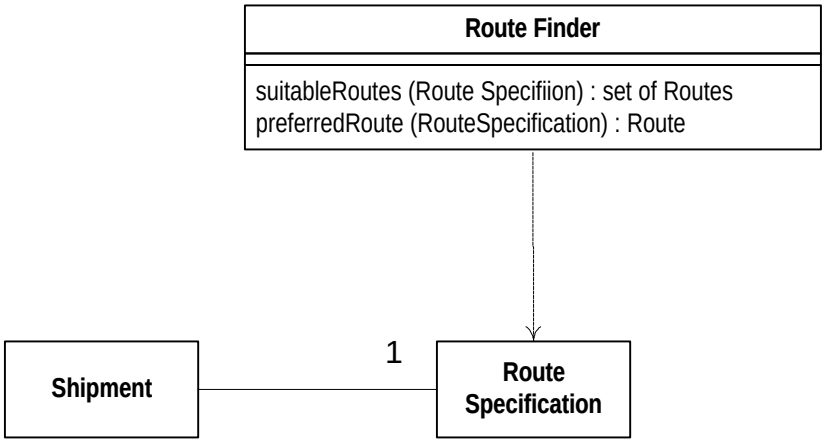

Figure 4. Using a specification as an input to a route selector. This decouples the route selector from the shipment.

In each of these cases the specification object is a separate object to the domain object that uses it. The portfolio could contain an operation to test for contract membership, the cargo could contain an operation for checking that a container is valid, the shipment could provide the necessary information for the route finder.

Treating the specification as a separate object has a number of advantages.

Consequences of Specification

Lets you decouple the design of requirements, fulfillment, and validation

Because the objects involved now have a crisper responsibility, it is easier to decouple them. This allows us to avoid the duplication of effort of implementing similar functionality many times, and makes integration much easier, since different parts of the system use the same constructs to express constraints or requirements, and frequently communicate with each other in these terms.

Allows you to make your systemdefinitions more clear and declarative.

For example, if you have built a mechanism that is supposed to derive an object that fits a spec, even if you trust the mechanism enough not to check, it gives your design a clear definition. RouteSelector finds a Route that satisfies a RouteSpecification. Even if we do not do an isSatisfiedBy: test, we have clearly defined the responsibility of RouteSelector without linking it to a specific process for finding routes.

When Not to Use Specification

There are temptations to over-use Specification. The most common is to use it to build objects with flexible attributes — a sort of poor man's frame system. Remember that the specification has a clear responsibility: to identify an object that satisfies it, and to combine with other specifications in various ways that support that basic responsibility (see below). A route is not the same thing as a route specification. The route should have responsibilities such generating an itinerary or reserving space on the transports being used. The route could tell us that the cargo is expected to arrive at a certain date and time on a certain vessel. The route specification might know that the cargo must arrive on or before a certain date and time, and tell us if a chosen route meets its requirements. Since the attributes of a route specification are so similar to the route, there is a temptation to combine the two behaviors into one object, and a route just becomes a route specification that fully specifies every detail of the journey. This would require the machinery of the specification to handle this quantity of data efficiently, even though route requirements are usually relatively simple, and it would require route behavior to operate off of an internal representation that was far more flexible that route needs to be. Most of all, the clarity of the design is lost.

If you find yourself adding to the protocol other than for initialization, if you find that you don't use isSatisfiedBy or isGeneralizationOf (see below), if you find that your object is representing an actual entity in the domain, rather than placing constraints on some other, possibly hypothetical, entity, you should reconsider the use of this pattern.

Strategies for Implementing Specification

There are various ways you can implement a specification, and now we will look into that in more detail. We won't go into the details of implementation here, as these will vary depending on your language and database. Instead we will just outline the principal approaches you can use. We will illustrate the strategies by discussing how they would handle a storage specification (as in Figure 3). The specification calls for the container to be able to store goods at less than -4°C and meet the sanitary requirements for storage of food. The sample code is in Smalltalk.

Hard Coded Specification

How you implement a specification?

Code the selection criteria into the isSatisfied method as a block of code

Consequences

+ easy

+ expressive

- inflexible

The simplest approach is to just create special purpose objects that specify each target object of interest and implement a single Boolean method.

Define an abstract class (or interface) that declares the isSatisfiedBy operation and provides the type for the parameter.

```
StorageSpecification>>isSatisfiedBy: aContainer
```

Each subclass then implements that operation.

```
MeatStorageSpecification >> isSatisfiedBy: aContainer
   ^ (aContainer canMaintainTemperatureBelow: -4) and: [aContainer
isSanitaryForFood]
```

This is essentially an application of *Strategy* [Gang of Four], where each cargo has a strategy for evaluating a potential container. Each validation rule we need requires a subclass of the strategy.

The problem with this approach is the need to create a subclass for every selection criterion. This works well if we do not need too many selection criteria, and if we don't need to create new ones at runtime. Explicit code is straightforward, and expressive.

A variation on this that works in languages with block closures (such as Smalltalk) is to put the implementation of isSatisfiedBy into a Boolean block object, so that we don't need to use a subclass. The code can even be added at runtime, although you do need someone who knows Smalltalk to write the appropriate block. Also using Blocks in this way can be a pain to debug.

Parameterized Specification

How do you implement a specification?

Create attributes in the specification for values that commonly vary. Code the isSatisfiedBy method to combine these parameters to make the test.

+ some flexibility

- still requires special-purpose classes

The problem with the hard-coded specification is that even small changes require a new subclass. If you have conditions such as a car with a specified top speed, then it is silly to create a different subclass for each possible speed. How do you implement a specification that will give you the

flexibility to express small variations simply? Make the key variables parameters of the specification. (So you can put the speed into the specification as a parameter.)

This implies that you need these parameters in the specification as attributes. When you create the specification, you put the values in the attributes, and the isSatisfiedBy method uses the parameters to make its selection.

In the cargo specification example above, we might have a cargo specification that looks like this.

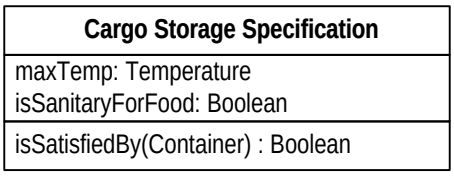

Figure 5. An example of a parameterized specification.

The isSatisfiedBy method would then look like

```
CargoStorageSpecification >> isSatisfiedBy: aContainer
   ^ (aContainer canMaintainTemperatureBelow: maxTemp) and: [
      isSanitaryForFood ifTrue: [ aContainer isSanitaryForFood ] ifFalse:
[true]
   ]
```

Whereas the MeatStorageSpecification was very special purpose, this CargoStorageSpecification could be used for a variety of cargoes.

The parameterized specification makes it easy to create a wide range of specifications as you vary the parameters. Users can usually spell out the exact specification by filling in a dialog box. However you can only vary those parameters that are provided. If you need to add a new parameter, that does require new coding. If you have a lot of parameters, then a single parameterized specification can get very complicated to code and to use.

Composite Specification

How do you implement a generalized specification that will give you the flexibility to express complex requirements without implementing specialized classes for every different type of candidate? The hard coded and parameterized specifications both use a single specification object to handle the specification. The composite specification uses several objects arranged in the composite pattern [Gang of Four] to do the work.

Composite Specification

How do you implement a specification?

Create a particular form of interpreter for the specification. Create leaf elements for the various kinds of tests. Create composite nodes for the and, or, and not operators (see Combining Specifications, below).

Consequences

+ very flexible, without requiring many specialized classes

+ supports logical operations

- must invest in complex framework

The key to this pattern is to provide a class that can combine specifications formed in different ways. A simple example of this is a specification which is true if all its components are true (we will call is a conjunction specification later on).

```
CompositeSpecification isSatisfiedBy: aCandidate
        (self components do: [:each |
        (each isSatisfiedBy: aCandidate) ifFalse: [^false]
        ]
        ^true
```

It need not know anything about how each component specification does its work. In the meat example we might have one parameterized specification to test for temperature.

```
MaximumTemperatureSpecification isSatisfiedBy: aContainer
        ^aContainer canMaintainTemperatureBelow: maxTemp
```

And another for the food sanitation.

```
SanitaryForFoodSpecification isSatisfiedBy: aContainer
        ^aContainer isSanitaryForFood
```

We can then form the overall specification by combining them

```
meatSpecification
   ^CompositeSpecification
      with: MaximumTemperatureSpecification new: -4
      with: SanitaryForFoodSpecification new
```

The class diagram for this structure would look like this

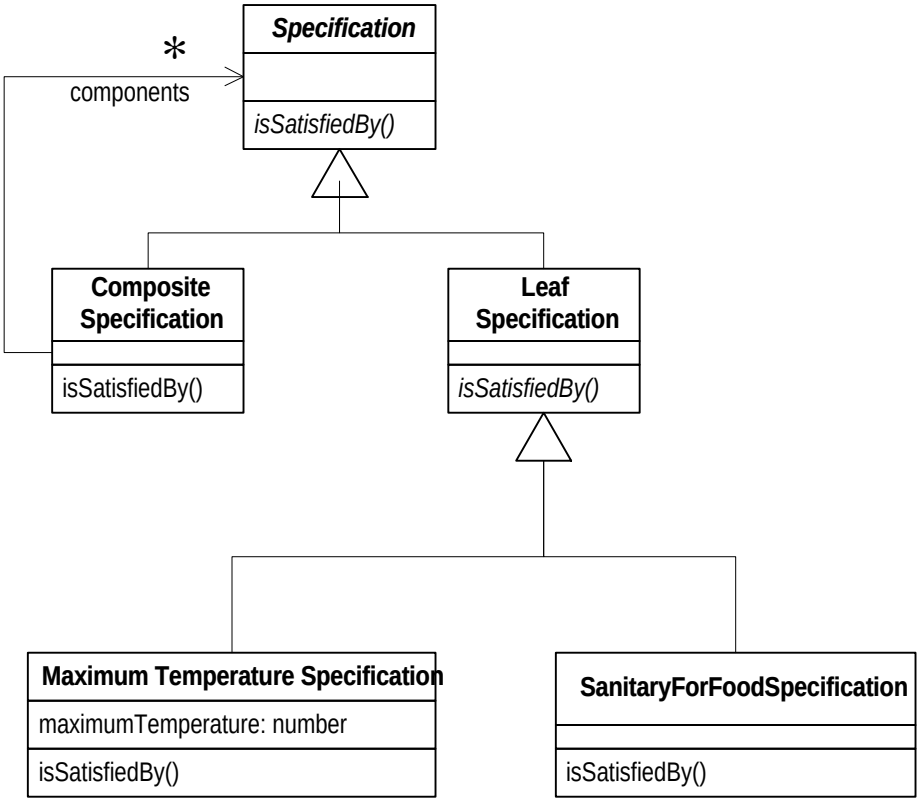

Figure 6. Composite Specification using conjunction.

And the instances would look like this

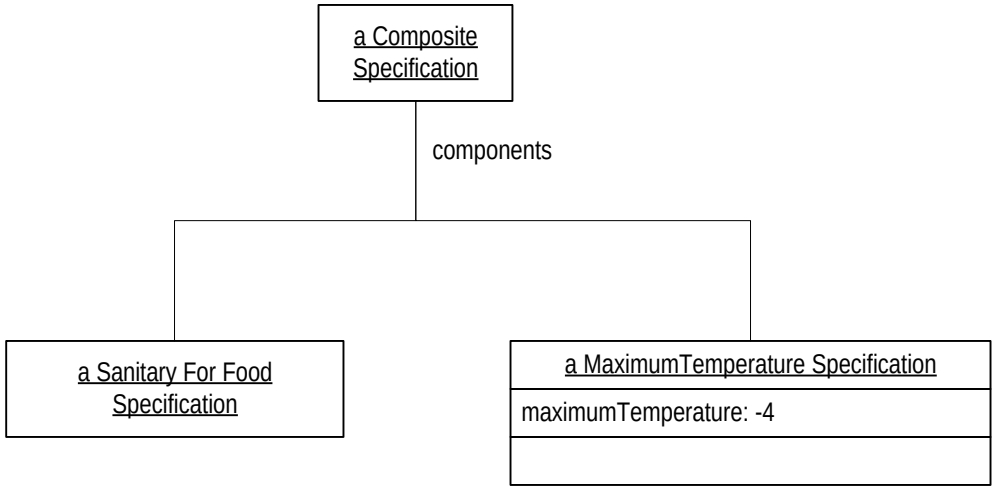

Figure 7. An example of a composite specification.

Define class `CompositeSpecification` as a Specification stated in terms of other Specifications. Special purpose primitive Specification subclasses may be needed for specialized applications. But a few primitive specifications can be built that can serve in different situations. Here we will define just one very versatile one, `ValueBoundSpecification`, to be a named value that expresses a limit on an attribute identified by the name. Lower bounds, upper bounds and

exact values may be defined with separate subclasses, or with case logic. Using these primitives, a range can be expressed by a composite of a lower bound and an upper bound.

Logical Expressions for Composite Specifications

In the example above, the composite specification combines its components by conjunction. All the components must be true for the composite to be true. You can get more flexibility by providing composite specifications for the other logical operators.

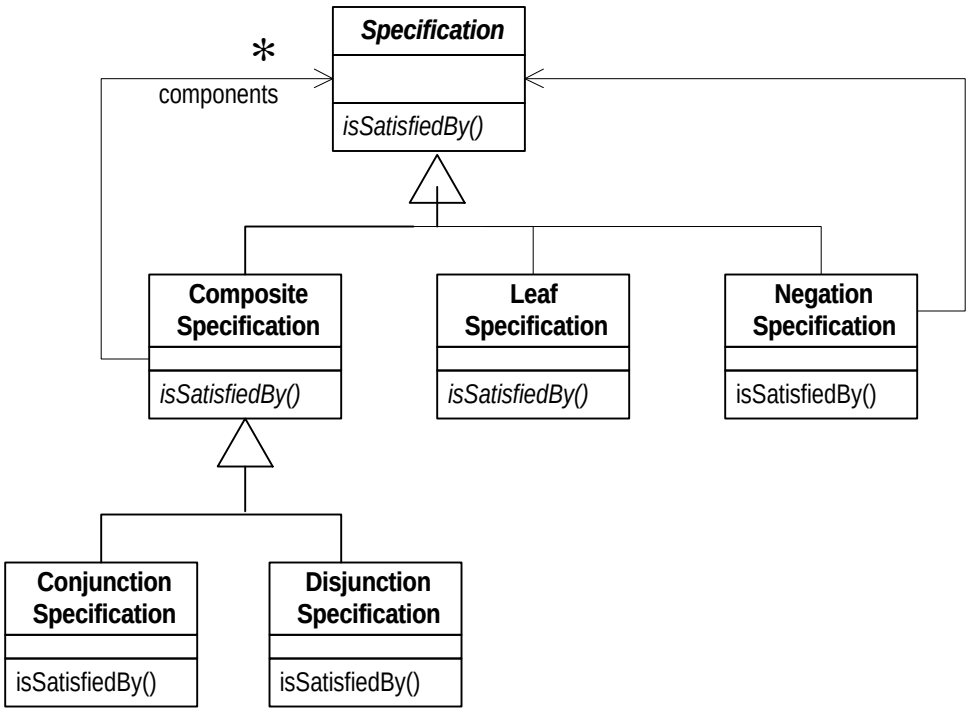

Figure 8. Composite Specification using a full logical expression.

This effectively creates an interpreter [Gang of Four] for the specification. The "language" being interpreted allows us to describe a composite specification. For example in the example above, in which we created a storage specification for a cargo of frozen foods, using the same parameterized specifications as above (which would now be leaf specifications) we could now express the composite like this

```
SanitaryForFoodSpecification new and:
      (MaximumTemperatureSpecification newFor: -4)
```

If we create a few general-purpose leaf specifications, we can do a lot without custom coded specifications. We can use *Variable State* [Beck] (also known as *Property List*) to allow ourselves to declare attributes on the fly.

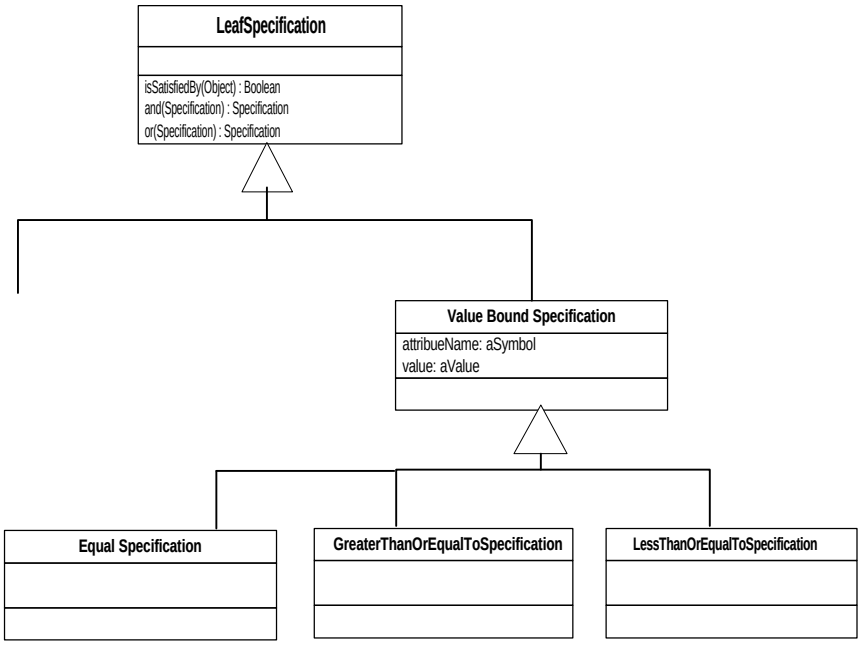

Figure 9. Value bound leaf specification

Placing the creation protocols on the abstract superclass, Specification, (a common practice in Smalltalk), that same specification can now be created as follows

```
(Specification attributeNamed: #isSanitaryForFood equals: true) and:
(Specification attributeNamed: #temperature isLessThanOrEqualTo: -4)
```

To implement these additional classes, the code from CompositeSpecification moves to ConjunctionSpecification. The DisjunctionSpecification implements this

```
DisjunctionSpecification >> isSatisfiedBy: aCandidateObject
   ^ self components contains: [:each | each isSatisfiedBy: aCandidateObject ]
```

Implementation of `and:` and or: can be completely generic, on the Specification superclass, although in practice there are many simplifications of the result that would be desirable. The key, though is to return a new Specification.

```
Specification >> and: aSpecification
   ^ ConjunctionSpecification with: self with: aSpecification
Specification >> or: aSpecification
   ^ DisjunctionSpecification with: self with: aSpecification
```

Comparing the alternatives for implementing specification

Each of these implementation strategies has its strengths and weaknesses.

The hard coded specification is very expressive. It can express any specification that you can represent in the programming language that you are working with. However each specification needs a new

class, and you usually can not create new specifications at runtime. Also you need to know the language to create a specification.

You can easily create new parameterized specifications at run time, and building them is easily done. Most of the time a dialog box can capture the parameters. However you are limited to the parameters supplied, and by the way the objects combine the parameters.

The specification interpreter represents a middle ground between the two. You get much more expressiveness than the parameterized specification, because you can combine the leaf elements in any logical expression you like. You can also set up the expression at run time. It is not quite as easy as the parameterized specification to build, but a simple tool can capture many needs. You are still limited by the leaf elements that are provided however. The specification interpreter is also a lot more effort to build.

You may "upgrade" from one design to another, or use them in combination. For example, you may realize you have hard-coded specifications already and wish to refactor them into parameterized specifications. Later you may decide you need the power of the Composite Specification. Some of your parameterized specifications can serve as leaf nodes. Multi-parameter specifications can be broken down a bit at a time until you reach atomic, one-parameter specifications, which are the simplest and most flexible form for the leaf nodes of a composite specification.

If you have to select from a lot of candidates on a database, then performance may well become an issue. You will need to see how to index the candidates in some way to speed up the matching process. How best to index will depend on your domain and on your database technology.

Subsumption

With Specification in place, there are two supplemental patterns that can provide further benefits: *Subsumption* and *Partially Satisfied Specifications*.

So far we've just looked at how you test a specification against an object. However there is also a purpose to compare specifications to each other, and to combine specifications.

This extension allows two specifications to be compared to one another. Specification B is a special case of Specification A if and only if for any possible candidate object X, where A is satisfied by X, B will always be satisfied by X also. If this is true, it is possible to apply any conclusion reached using B to A, hence B can subsume A. For example, if we have two route requirements:

A: Hong Kong to Chicago

And B: Hong Kong to Chicago via Long Beach

We can find a route that satisfies B, and know that it will satisfy A, without testing A directly. B is a special case of A, and therefore can subsume its place.

Subsumption

How do I compare two specifications to see if one is a special case of the other, or is substitutable for another?

Create an operation called isSpecialCaseOf(Specification): Boolean that will answer whether the receiver is in every way equal or more strict than the argument, and therefore could stand in for the argument. (A complementary isGeneralizationOf operation is convenient, also.)

People frequently want to compare two specifications without ever having a concrete candidate object. For example, in a system that records service contracts and tries to match them to orders, we have two sets of specifications: the specification that describes the type of service offered under the contract; and the specification that describes the type of service being ordered. If a shipping company had a contract with a local trucking company to deliver its containers from Long Beach to any site within California, the specification might be:

Conditions for use of Contract:

Port must be Long Beach

Destination must be located within California

In order to deliver a shipment that is being routed through Long Beach and destined for Newport Beach, the shipment routing system might generate a specification for contract applicability like this:

Service requirements of customer order:

Port must be Long Beach

Destination must be Newport Beach

The question becomes: Is the specification service requirements a special case of the conditions for use of the contract? In this case it is. The contract's specification is more general than the order's specification (is subsumed by it) and therefore this contract can be applied to the order.

Subsumption can also be used in the implementation of satisfaction, when we can characterize the candidate object using a Specification. For example, if a Container is characterized by an attribute called contentsSpecification, which is typed as a StorageSpecification, and a Cargo has an attribute, storageSpecification, which is also a StorageSpecification, then we could implement isSatisfiedBy as follows.

```
Cargo>>storageSpecification isSatisfiedBy: aContainer
^ aContainer contentsSpecification isSpecialCaseOf: self storageSpecification
```

This further decouples the Cargo from the Container.

If we had two shipments, and we wanted to know if they could be packaged together in the same container, we would like to be able to recognize that two refrigerated cargoes with different storage temperatures cannot be packaged together, without any reference to an actual container.

The result of the logical operators can be described in terms of *subsumption*. The result of an "and" operation must subsume both of the terms. In other words, if I derive a specX

```
specX := specA and: specB.
```

Then I know that both of the following will return true

```
specX isSpecialCaseOf: specA
```

```
specX isSpecialCaseOf: specB
```

The result of an "or" operation must be a generalization of both terms.

```
specX := specA or: specB.
```

Therefore, both of the following will return true.

```
specX isGeneralizationOf: specA
specX isGeneralizationOf: specB
```

Implementing Subsumption

Add a method "isSpecialCaseOf(Specification) : Boolean" and a complementary "isGeneralizationOf(Specification) : Boolean" to the Specification object. This can be implemented in either the parameterized specification or the composite. In a parameterized specification, only two members of the same class can be compared, and the isGeneralizationOf method must be written for each specialized Specification. For CompositeSpecification, it can be generalized somewhat. As usual we will illustrate it with Smalltalk. Doing this kind of thing in Smalltalk calls for *Double Dispatch* [Beck], which can look a little odd to non-Smalltalkers.

```
Specification >> isGeneralizationOf: aSpecification
   ^self subclassResponsibility

NamedSpecification >> isGeneralizationOf: aSpecification
   ^ aSpecification isSpecialCaseOfNamedSpecification: self

Specification >> isSpecialCaseOfNamedSpecification: aNamedSpecification
   "In the general case, it does not"
   ^ false

NamedSpecification >> isSpecialCaseOfNamedSpecification: aNamedSpecification
   "Only specifications of the same name can subsume each other.  Double
Dispatch again
within whatever subclasses you have, after comparing name"
   ^ self name = aSpecification  name and: [ self
isSpecialCaseOfSpecificationOfSameName: aSpecification ]
```

as an example, magnitude comparisons would dispatch among themselves:

```
ValueBoundSpecification >> isSpecialCaseOfSpecificationOfSameName:
aNamedSpecification
   ^ aNamedSpecification isGeneralizationOfValueBoundSpecification: self


LessThanOrEqualToSpecification >> isGeneralizationOfValueBoundSpecification:
aValueBoundSpecification
   ^ aValueBoundSpecification isSpecialCaseOfLessThanOrEqualToSpecification:
self

EqualSpecification >> isSpecialCaseOfLessThanOrEqualToSpecification:
aLessThanOrEqualToSpecification
   "Finally, we are comparing two leaf nodes."
   ^ self magnitude <= aLessThanOrEqualToSpecification magnitude

GreaterThanOrEqualToSpecification >>
isSpecialCaseOfLessThanOrEqualToSpecification: aLessThanOrEqualToSpecification
   ^false
```

And so on.

The Composite branch dispatches quite differently:

```
CompositeSpecification >> isGeneralizationOf: aSpecification
   "True if each component is subsumed.  False if any component is not
subsumed."
```

```
   ^ (self components contains: [:each | (each isGeneralizationOf:
aSpecification) not ]) not

CompositeSpecification >> isSpecialCaseOfNamedSpecification:
aNamedSpecification
   "The composite isSpecialCaseOf the other Specification if any of its
components does."
   ^  self components contains: [: each | aNamedSpecification
isGeneralizationOf: each ]
```

Once this dispatching framework is built, new leaf nodes can be added by just making sure that they implement all the messages expected within their branch.

Partially Fulfilled Specifications

The most common need is to know whether a specification is fully satisfied or fully subsumed. In fact, if you are very interested in other matching patterns, it may be an indication that specification is the wrong model for what you are doing. But there are at least two ways in which partial satisfaction or *subsumption* does fit well into the specification model.

One is explanation of results. Using the container example above, it may be necessary not only to say that a container does not meet spec, but to list exactly which part of the spec is not met: Is it not refrigerated? Is it not sanitary? Both? Our example is so simple that a user could easily scan for differences, but in a more realistic case, finding the discrepancy could be much more difficult.

Another situation that calls for *partially satisfied specifications* is when a specification may have to be satisfied by two or more objects. This would typically happen when specifying requirements for a service. If the originally selected service is only partially completed, it may be necessary to determine what remains to be done and then select a service to satisfy the remaining portion.

Partially Satisfied Specification

You need to figure out what still must be done to satisfy your requirements.

You need to satisfy a requirement with more than one object.

You need to explain to the user why the specification was not satisfied.

Add a method "remainderUnsatisfiedBy(CandidateObject):Specification" that returns a Specification that expresses only the requirements not met by the target object. (Best used together with *Composite Specification*).

Here is an example of route selection from the shipping domain.

Original Route Specification:

- Origin: Hong Kong
- Destination: Chicago
- Stop-off: Salt Lake City (to drop off part of the cargo)
- Customs Clearance: Seattle

The original route was selected to satisfy this specification, spelling out a detailed itinerary of ships and trains. According to this itinerary, the shipment was to go by train to Boise, there to be transferred to another train to go to Salt Lake City. But, when the shipment arrives in Seattle, a blizzard has engulfed Boise, and the train yards are temporarily shut down. It is time to select a new route.

The problem is that the portion of the route already traversed cannot be changed. We really want to select a new route that satisfies only those components of the original spec that have not been satisfied by the already traversed route.

We ask the original spec for the remainder unsatisfied by the traversed route:

```
originalSpecification remainderUnsatisfiedBy: traversedRoute      (returns a
new specification)
```

The new specification will look like this:

- Destination: Chicago
- Stop-off: Salt Lake City

There is one more requirement for the new route. It must start at our current location. So we add that:

```
newSpecification and: (Origin: Seattle)
```

Which returns a specification that looks like this:

- Origin: Seattle
- Destination: Chicago
- Stop-off: Salt Lake City

We can now apply the route selection process to finding a way to fulfill these new requirements, and we can check the result using isSatisfiedBy:. In this case, perhaps it will be shipped to Portland and then sent on to Salt Lake. The final leg to Chicago may or may not be the same.

If the route specification is implemented as a composite, this operation is easy to implement. Using the same iteration process used to walk the composite structure in the isGeneralizationOf: or isSatisfiedBy: operations, you test each component:

```
aComponent isSatisfiedBy: anObject
```

and collect all the unsatisfied ones into the new specification.

Summary of the Full Specification Protocol

When all these patterns are used together, Specification as the base, CompositeSpecification to represent more complex expressions, hard coded and parameterized Specifications as leaf nodes, with the supplemental patterns added in, the full protocol would look something like this:

- isSatisfiedBy(Object) : Boolean
- isGeneralizationOf(Specification) : Boolean
- and(Specification) : Specification

- or(Specification) : Specification
- remainderUnsatisfiedBy: (Object) : Specification

In a strongly typed language you would want to ensure that the arguments to isSatisfiedBy and remainderUnsatisfiedBy were restricted to the type of object you are working with.

Final Thoughts

Specifications have proven a valuable technique in current projects, and a technique we wished we had used in previous work. Using a composite specification, in particular, is very capable. It is a bit of effort to set it up, but once the building blocks are in place it can make life much easier. So far much of our implementation experience with this pattern is in a Smalltalk/Gemstone environment, and we will be interested to see how it applies to others. As we run into others who have used it, we are sure we will be able pool ideas and develop this theme further.

Further Reading

Martin describes the notions of portfolio selecting its contracts in [Fowler] §9.2. Essentially this is a use of a parameterized specification.

[Riehle] is a paper which is primarily about you can ask a superclass to create instances of subclasses without knowing about the subclasses. To do this he uses a specification to help the creating object choose an appropriate subclass. It includes good advice for implementing specifications for this kind of problem.

Although we haven't tried it, we would imagine that specifications would work well with the Shopper pattern [Doble] to help specify the item list.

References

[Beck] Beck K, Smalltalk Best Practice Patterns, Prentice-Hall, 1997

[Doble] Doble J, Shopper, in [PloPD2], pp 143–154

[Fowler] Fowler M, Analysis Patterns: Reusable Object Models, Addison-Wesley, 1997

[Fowler, UML] Fowler M, with Scott K, UML Distilled: Applying the Standard Object Modeling Language, Addison-Wesley, 1997

[Gang of Four] Gamma E, Helm R, Johnson R, and Vlissides J, Design Patterns: Elements of Reusable Software, Addison-Wesley, 1995

[PloPD2] Vlissides J, Coplien J, and Kerth N (eds), Pattern Languages of Program Design 2, Addison-Wesley, 1996

[Riehle] Riehle D, Patterns for Encapsulating Class Trees, in [PloPD2] pp 87–104