Apache Isis Application Library

Developing domain driven applications using Apache Isis

Dan Haywood
Robert Matthews

Apache Isis Application Library: Developing domain driven applications using Apache Isis

by Dan Haywood and Robert Matthews

1.0.0-SNAPSHOT

Table of Contents

List of Tables

Preface

Apache Isis is designed to allow programmers rapidly develop domain-driven applications following the Naked Objects [http://en.wikipedia.org/wiki/Naked_Objects] pattern. It is made up of a core framework plus a number of alternate implementations, and supports various viewers and runtimes/object stores. Apache Isis is hosted at the Apache Foundation [http://incubator.apache.org/isis], and is licensed under Apache Software License v2 [http://www.apache.org/licenses/LICENSE-2.0.html].

This guide is written for programmers looking to understand the programming conventions, annotations and supporting utilities within the *Apache Isis* application library (or *applib*), in order that the framework can correctly pick up and render the business rules and logic encoded within their domain objects.

Chapter 1. Introduction

What this guide contains, who it is for, and how the applib relates to other parts of the framework.

What's in this Guide

This guide describes the set of conventions for writing domain objects, that are together known as the *Apache Isis* Programming Model. It's important reading for developers who are looking to write domain-driven applications either for prototyping or deployment on *Apache Isis*.

More correctly, this guide actually documents the *default* programming model, targetted for the Java programming language, and implemented by the `org.apache.isis.progmodels:dflt` module. *Isis* is in fact capable of supporting different programming models, either for other JVM-based languages; for example the Groovy language is supported in the `org.apache.isis.progmodels:groovy` module. Those other programming models have their own documentation.

This Guide vs the Core Documentation

This guide assumes that you have at least the outline of a running *Apache Isis* application, for example having run the quickstart archetype. Details of how to run the quickstart archetype can be found on the *Isis* website [http://incubator.apache.org/isis].

This guide has two main objectives:

- to describe the general approach for developing Isis applications solely by writing domain objects
- to describe the specifics of writing domain objects according to the conventions of the *default* programming model.

The core documentation, on the other hand, explains the wider landscape of what makes up an *Isis* application, dealing with such matters as running the app with different components, eg viewers, security, programming models, profile stores and runtimes/object stores. Included within this is an explanation of how you can build your *own* programming model (typically as a subset of the default programming model described here, but possibly with additional custom elements).

Where Next?

This guide is in several parts. The first part is intended to explain, from a developers' perspective, what exactly *Apache Isis* is. The second part has chapters that describe the specifics in the style of recipes/ cookbooks, providing a "how-to" guide on developing domain objects. The third part provides details on other supporting features relevant to writing domain applications. The final part is a series of appendices that provide details of the programming model in reference form. You should find this more useful if you've used *Isis* for a while and just need to check on a specific detail.

Part I. Understanding Apache Isis

The chapters in this part of the guide are intended to help you understand *Apache Isis* from a developers' perspective:

- the principles and patterns that underpin *Isis* (Chapter 2, *Apache Isis and Naked Objects*)
- a development process to follow (Chapter 3, *A Development Process*)

Chapter 2. Apache Isis and Naked Objects

Understanding Apache Isis and the naked objects pattern by way of a series of questions and answers.

The headline feature that distinguishes *Apache Isis* from other frameworks is its support for the naked objects pattern - the ability to generate a user interface (at runtime) directly from the domain model. However, there's much more to *Apache Isis* than this.

This chapter introduces the *Apache Isis* and the naked objects pattern by way of a series of questions and answers. It is adapted from an interview originally published on InfoQ [http://www.infoq.com/articles/haywood-ddd-no].

Apache Isis implements the naked objects pattern... but what is that, exactly?

Naked objects is an architectural pattern where the idea is to automatically expose a domain model objects directly within a object-oriented user interface... not just their state (properties and collections) but also their behaviour (what we call actions). You can think of it as analogous to an ORM such as Hibernate [http://hibernate.org]; but whereas an ORM reflects the domain model into the persistence layer, naked objects reflects the domain model into the presentation layer.

Naked Objects (capitals) was a Java framework that implemented the naked objects (lower case) pattern. Since then, we've taken the original framework, along with a number of sister projects developed by the community, into the Apache Incubator. So, what was originally "Naked Objects" is now Apache Isis.

What type of applications are best suited to Isis?

The naked objects pattern requires that your primary consideration is in building an object-oriented domain model, so it's most suitable for enterprise applications that have complex business rules where there's a desire to represented them within such a domain model. The idea - at least during the initial stages - is to build up a domain model quickly, and to get feedback from the domain experts by exploiting the framework's ability to expose that directly domain model within the user interface. A picture tells a thousand words.

Apache Isis "sweet spot" are those applications that are used internally within the organization, by experienced users who are comfortable with the entities within the domain model, and just need an application that imposes as few constraints as possible on how it is used. These are sometimes called *sovereign applications*. If you're a developer then your IDE is probably the sovereign application you use the most, and how often did you use any of its wizards? And if you use telephone banking then you'll also know how frustrating a sovereign application can be that is too invasive in dictating the workflow... that poor person on the other end asks making apologies while she goes exits from one screen and then goes into another to check some detail on a bank account. The generic OO UIs generated by naked objects impose no such restrictions, and so are ideal for this sort of application.

And are there applications where the OO UIs generated by Isis are less suitable?

The opposite of a sovereign application (above) is sometimes called a *transient application*. These are ones used only occasionally or by inexperienced users, typically outside the organization (ie your customers), who don't know or care to know the domain model, and just want to be led through the system to accomplish some well-defined goal. A good example here is a check-in kiosk at an airport... you just want to get on the plane, and be given the opportunity to choose your seat. But you likely won't care which plane, or even plane type, nor who the pilot is; these are unimportant details.

For this sort of application a generic OO UI that exposes lots of the domain is clearly not appropriate. Instead, we want an application that exposes view models rather than entities, and where the UI can be customized. The view model object is responsible for managing the workflow for the user story, exposing just the subset of the domain that is relevant, and hiding the rest of the domain. We find that these view models can be layered on top of the entities once those entities are understood.

Using view model objects is necessary but probably not sufficient for transient applications; we also usually need to customize the UI. *Apache Isis* has two viewers that allow the UI to be customized, one that provides a set of taglibs, and one that provides a set of Apache Wicket [http://wicket.apache.org] components.

Putting both of these techniques together (custom view models and views) means that Apache Isis is suitable for transient applications as well as sovereign applications. However, the former will necessarily take more work as opposed to the latter.

What is the typical development process life cycle for applications built with Isis, and how does it compare to traditional application development process?

The main difference you're likely to encounter is the emphasis on developing the domain model at the same time as identifying and prioritizing user stories, what those practicing domain driven design call a "ubiquitous language" for the team.

So, rather than let the developers in the team "discover" the domain model as part of story implementation, we'd expect that the domain experts/business analysts have already identified some of the main domain concepts through prototyping or spikes, and these can be used to help communicate the stories to the developers during the planning game. Not every business analyst is going to feel comfortable working with an IDE, but it works well to pair with a developer. Because an Apache Isis application can be runs just from domain classes, it's possible to develop the app very very quickly. We've found a good technique is to run workshops with the business analyst facilitating the meeting with the domain experts, and the developer acting as "code monkey" to rapidly convert the ideas into an app. Obviously the code isn't production quality, but it allows the team as a whole to go very quickly and experiment with different representations of the model.

Another alternative is to go a little more slowly, and have the analyst and developer pairing to write production code. The analyst still focuses on identifying and naming the domain concepts and their relationships, while the developer's role (as well as learning about the domain) is to ensure that there's enough rigour and tests around the code that's been written while this is being done. Of course, both

approaches to pairing can be used, sometimes spiking ideas, sometimes going straight to writing tested code.

Of course, while the naked objects pattern means that no UI code needs to be written in order to elicit feedback, there do still need to be objects in the app for the domain expert to view. Apache Isis has a pluggable runtime/persistence layer, and for prototyping and most of development we recommend using the in-memory object store. To populate this object store for each run we use fixtures, which we usually arrange into a composite pattern to setup data needed for the particular scenario we might be working on. Using Isis in this way means that we spend our time alternating between defining domain classes and writing fixture data. This can be a great benefit because it helps the domain experts - who after all aren't necessarily technical - distinguish between what are classes and what are instances of classes. For example, if I'm writing a library system and we identify there are loanable books and reference books, we need to know whether "loanable book" and "reference book" are different subclasses of a book superclass, or merely different instances of a book class with, say, a loanable property set to true or false. Seeing a running application makes it much easier for the domain expert to make the call.

In terms of writing tests, there are several options. First, because the domain objects you write are basically pojos that follow some naming conventions, they are very easy to test using standard TDD tools such as JUnit [http://junit.org] and JMock [http://jmock.org]. And if you wanted to, you could test the UI generated by Isis using Selenium [http://seleniumhq.org/] and similar; indeed we'd recommend that if you've customized the UI.

In addition, though, Isis provides a couple of additional BDD/testing framework integrations. One of these is an integration with FitNesse [http://fitnesse.org], and a newer one we've worked on is an integration with Concordion [http://concordion.org]. In both cases the integrations provide the glue code so you can exercise your domain objects directly. For example, your spec might say "the place order action cannot be invoked if the product is out of stock". All the developer needs to do is to wire the spec to these pre-canned integrations, specifying that the `placeOrder()` action should be called on a `Product`.

Isis also provides a JUnit runner that works by bootstrapping the Isis runtime for each test, and proxying the domain objects with a bit of cglib so that they are interacted with "as if" through the UI. For example, let's go back to that `placeOrder()` action. To prevent this from being called for a `Product` that's out of stock, the developer would write a corresponding `validatePlaceOrder()` method. Isis automatically calls this validate method prior to the `placeOrder()` method, and uses the return value ("that product is out of stock") as the reason why the action can't be invoked. In the UI, this business rule is represented by greying out the OK button of the action. In the JUnit runner, this rule is represented by having the cglib proxy throw a validation exception if the test calls `placeOrder()` with an out of stock `Product`.

In terms of deployment, there's several steps involved, probably the most significant being to select the runtime and object store. As noted above, Isis has a pluggable persistence layer; you can use the in-memory object store for developing, but will want to switch to a "real" object store for deployment. The amount of work required here depends on the object store selected; in the case of JPA/Hibernate support, for example, it amounts to annotating your domain entities with annotations, and writing implementations of some of the repositories to make the relevant SQL calls.

What are the limitations of Apache Isis framework?

It's certainly true that naked objects pattern implemented by *Apache Isis* is opinionated, so if you don't agree with all of its opinions then you're going to find it limiting in one way or another.

The first of these is that everything that the user wants to interact with has to be an object of some sort. As already explained, for sovereign applications, these are likely to be the persisted domain entities, but

for a transient application, the object may be a view model to be support a particular workflow, and which may or may not be persisted. Such an approach probably wouldn't be considered much of a limitation, but it's worth contrasting with architectures (eg Spring Web Flow or Struts) where put the responsibility for tracking workflow lives not in an object but instead within some sort of declarative XML markup (or even just in the interplay between controllers and views).

A slightly more subtle consequence of the above is that integration of different technologies happens through the domain objects, rather than in front of them through application-layer or UI layer mashups. For example, supposing that we wanted to have an SMS sent out to confirm a checkout. If you were writing the UI and application service layer yourself, you might choose to have the application layer make the call to the SMS service, judging it to be a coordination responsibility. With *Isis*, though, you don't get the chance to write any application layer code, and so this would be done by having the domain object call out to the SMS service. The SMS service would be defined by an interface, and the implementation would be injected into the domain object by the framework. For some reason not everyone is necessarily comfortable about injecting domain services into entities; but it works well for us and we're sticking with it.

Another area where *Isis* is going to feel different - and perhaps limiting to some - is that it pushes some responsibilities onto the domain objects that otherwise might sit in other layers. For example, above we discussed that a `placeOrder()` action on a `Customer` can have a supporting `validatePlaceOrder()` method. It's also possible to have `disablePlaceOrder()` method, which if returning non-null will cause the action to be greyed out the action in the UI. For example, a blacklisted `Customer` might not allow any orders to be placed, and would indicate this through the disable method. Some might consider this as a misplaced presentation concern. However, the `Customer` isn't greying out the UI itself; the disable method is merely a convention by which the presentation layer can interrogate the entity.

Something else we hear sometimes about naked objects-style systems is that they are really only suitable for CRUD style applications. It has to be said that it's a criticism that does irk; because although its true that naked objects frameworks automatically expose object state, they also expose object's behaviour. That is, every public method that is not a property or a collection is taken to be an action and will rendered by the UI as a button or a link. Indeed, we sometimes like to talk of behaviourally complete domain objects; it's the very antithesis of the anaemic domain model anti-pattern.

All the above notwithstanding, probably the biggest impediment to going out and using Isis right now is that it's a small community and as such the codebase is still relatively new. But if that is a turn-off, note that you can still use Apache Isis to prototype your domain model, because they are just pojos after all. Indeed, embedding Isis' metamodel for a deployment hosted on some other framework is the objective of the *embedded runtime* component of Isis

How extensive is the application security support provided by Isis?

Apache Isis exposes a pluggable authentication API and an authorization API, with default implementations of both.

The job of the authentication API is to authenticate the user credentials and return the user Id and a set of roles for that user. This information is then used in one of two ways. The authorization API uses it to implement declarative class-based security, that is, associating access to features based on the roles of the user. In addition, though, a domain object instance can also access the user credentials, and so can implement imperative instance-based checks if required.

For example, we might say that only a user with HR role can award pay rises to an `Employee`. This is a class-based check and would be implemented through the authorization API. But, we might also say that a

user is allowed to view (though not modify) their own salary of their `Employee` object, but no-one else's. This would be an instance-based check, and would be implemented in the object itself.

In terms of how naked objects restricts access, this is done either by making an object member (property, collection or action) invisible, or, if it is visible, then making it unusable (greyed out).

How does an Isis applications manage the custom logic in terms of business rules, workflow and other business logic that developers have to manually implement outside the generated code?

"Business rules" is one of those amorphous terms that means different things to different people. A consequence is that the team can struggle to nail down exactly where such rules should reside, so that these rules can start to leach out of the domain layer and the application layer, or even worse, the presentation layer. As noted already, one of the principles of naked objects is that domain objects are behaviourally complete. What that means is that business rules and "other business logic" are the responsibility of domain objects. For workflow, again as noted earlier, we suggest that you introduce a view model object and have that manage the user's interaction.

However, saying that business rules are an object's responsibility doesn't necessarily mean that the implementation has to be in Java. We already said that domain objects can delegate work out to domain services that are injected into them. So if you want to put your business rules into a rules engine, that's fine. Wrap the interface to the rules engine within a domain service, and have the domain object call out to the domain service in order to fulfil its responsibility.

In practical terms, you're going to see business rules both in the small-scale and the large. Small-scale business rules tend to be implemented in terms of the supporting methods (disable, validate and so on) that encode the preconditions for interactions. These ensure that the object or the providing arguments is are valid and will veto the interaction otherwise (blacklisted customers, out-of-stock products). The larger-scale business rules are represented as actions on an object that can perform arbitrary business logic, such as modifying its own state, or related objects, and/or delegating out to domain services.

How does Isis support SOA-based applications, eg where an enterprise service component is consumed by several different applications and other clients?

We can actually answer this both in terms of an Isis application consuming SOA[1] services, and in it *providing* services.

For consuming SOA services, the answer is easy enough: wrap the SOA service in a domain service interface, and then register the implementation with the framework so that the service is injected into each

[1] SOA=Service Oriented Architecture

domain object. That interaction can be either synchronous or asynchronous; there's nothing to prevent the domain object publishing an event to be picked up by some other component out there on your enterprise service bus.

As to providing SOA services, Isis' JSON viewer exposes the domain objects as a set of RESTful resources. We map objects and object members to the standard HTTP verbs, eg so that a GET on an object returns an object representation, a PUT on an object's property will modify the state, or a POST on an object's actions will invoke the action. As its name suggests, the JSON viewer produces a JSON representation, making it easy to consume (eg in HTML5 or RIA applications).

Most enterprise applications need to reuse the domain and service layer classes between several different applications. How does Apache Isis address this?

One way to think of reuse in terms of strategic domain-driven design, which couches the discussion in terms of identifying bounded contexts and understanding the relationship between these contexts - context maps.

You can think of a bounded context as both the user population that understands a particular ubiquitous language as well as the actual system that they use. For example, the term "Customer" is going to mean one thing to the marketing department, but a potentially different thing to those in shipping. Those two user populations therefore have incompatible languages, and so any integration of systems that support both sets of users will need to explicitly map their meanings of those terms.

Simplifying somewhat, an Isis (naked objects) application can be thought of in three parts: the domain model objects themselves, the viewer that provides a channel to interact with the domain model, and the domain services that are used by the domain model objects. The domain model objects encode a ubiquitous language for a particular bounded context, so it only makes sense to reuse them by users who have the same understanding of that language. We can if we want use view model objects to provide projections of the domain entities to different subsets of that user population, but all must agree with what a Customer is, even if they use different parts of said Customer.

Viewers are channels by which the domain model is accessed. *Apache Isis* has viewers that expose the domain model as either a desktop app or as a webapp, and also has a RESTful viewer (producing JSON) which allow the view model objects/domain objects to be surfaced as RESTful resources. In DDD terms the RESTful viewers are examples of an "open host service" context mapping from some client bounded context up to our model's context.

With respect to domain services, these are not reusable per se, rather they allow the domain objects to be the client of some other service within the enterprise. That said, one common type of domain service is to provide an interface to an enterprise service bus, in which case the domain objects can publish events onto that bus. In DDD terms this would be an example of a "published language" context mapping; the downstream systems consuming the events of the publishing domain objects.

Naked objects also allows a somewhat finer-grained level of reuse; indeed its support for "don't repeat yourself" is one of the reasons that some people are attracted to the pattern. For example, if we declare a property or an action parameter to be a date, then the viewers will automatically provide a date chooser widget for that property or parameter unless indicated otherwise. Or, if a property's type is an enum, then we'll get a drop-down box. It's the conventions of the naked objects programming model - establishing a protocol of interaction between the presentation layer and the domain model - that enable this sort of reuse.

Some of Isis' more recent viewers also allow mashups within the UI. For example, the viewer that we've implemented using Apache Wicket uses the chain of responsibility pattern to build the UI. If we have an entity can provide its `Location` (eg, it implements `Locatable`), then a GoogleMapsWidget will render the entity in a map. Or, if an entity has a date, then a CalendarWidget might render the entity on a calendar. This stuff is extensible so you're free to write your own components and reuse (or indeed invent your own) domain semantics as you see fit.

What is the future road map of the project?

Our proposal to join the incubator is online, and that captured the thoughts we had at that point in time [1] [http://wiki.apache.org/incubator/IsisProposal]. A general theme is in the ongoing development of the various viewers, such as the Wicket viewer, improving JSON support within the RESTful viewers, and ongoing development of the taglib-based viewer, Scimpi. Contributors who have expressed interest in developing similar viewers using JSF/Facelets, Tapestry, on Android and in JavaFX/Visage.

Isis has a pluggable architecture with about five major APIs (and numerous more fine-grained APIs) that can be developed. Probably the most significant of these is the runtime/persistence API. One piece of work is to develop an ORM-based object store from Hibernate (which is not compatible with Apache's) to a compatible JPA implementation, We've also had a suggestion [2 [https://issues.apache.org/jira/browse/ISIS-14]] to re-implement using JDO 3.0/DataNucleus, which would allow us to cover a lot of persistence technologies in one fell swoop

Another objective during incubation is to make Isis more easily bootstrappable and embeddable, and for that JSR-299 (Contexts and Dependency Injection) looks promising. For example, it should be easy to take just the *Isis* metamodel component and use it within your own applications, eg to drive the rendering of domain objects within custom UI code.

One other area we might see work is in support for other JVM languages. Isis already supports Groovy as well as Java, and the way that Isis builds up its metamodel means that supporting other languages (Scala, Fantom, Go) ought to be pretty straightforward.

Ultimately, though, Apache Isis' roadmap depends on where its community wants to take it; so any of the above might change as we progress through to graduation.

Chapter 3. A Development Process

This chapter describes the general approach to follow that we recommend for developing domain-driven applications on Isis.

There's quite a lot to *Apache Isis*, with lots of optional components, and we do recognize that it may be difficult to know how to get going. This chapter outlines a development process which you can use as the basis for developing your own way of working.

Run the Quickstart Archetype

We tend to organize *Isis* applications to follow a standard structure, and you can use the *quickstart* Maven archetype (see the *Isis* website [http://incubator.apache.org/isis]) to set this up for you. This sets up a simple application that can be run out-of-the-box using the default runtime and in-memory object store, and supporting a number of the webapp viewers. The WAR file can be run either standalone from the command line or deployed to a servlet container such as Tomcat.

Programming Model

Once you've got the archetype application running, you're ready to start developing your own domain objects. But no matter what application you are developing, you'll need to understand the *Isis programming model*.

The programming model is the set of annotations and programming conventions that Isis recognizes. You'll find full details of the programming model in this guide; see Part II, “Writing Domain Objects”.

In addition, the applib contains a small number of utilities which can be useful when writing your application: one such is the ability to create XML snapshots of your domain object graphs, see Chapter 14, *XML Snapshots*.

Fixtures and Prototyping

We suggest that the fastest way to develop your application is to start prototyping using the in-memory object store (set up automatically by the quickstart archetype). The nice thing about working this way is that there is no database schema to slow you down; you can make changes and then rapidly try them out.

On the other hand, the in-memory object store doesn't persist objects between runs, so you'll soon tire of continually recreating test objects to try out your changes. You should therefore use fixtures: blocks of code that are used to setup objects in the in-memory object store prior to the app running.

You can find more information about using fixtures in Chapter 13, *Fixtures and SwitchUser*. It's also worth knowing about fixtures because they are used when writing tests for your domain application (see the section called “Agile Testing”).

Viewers

Apache Isis allows the same domain object model to be consumed by multiple presentation layers. You'll find that the quickstart archetype sets up a webapp module that bundles together two of the viewers currently available, namely:

- The HTML viewer (in `viewers/html`) provides a basic webapp.

  Other than tweaking CSS, the views that it provides of objects cannot be customized.

- The JSON viewer (in `viewers/restfulobjects`) which exposes your domain objects through a RESTful [http://en.wikipedia.org/wiki/Representational_State_Transfer] interface.

  The intention of this viewer is to allow programmatic access to your domain objects from other (perhaps non-Java) clients.

- The BDD viewer (in `viewers/bdd`) provides an integration with the Concordion [http://concordion.org] BDD framework; see the section called “Story (BDD) Testing”.

- The JUnit viewer (in `viewers/junit`) uses wrappers and a custom Isis JUnit test class runner (`IsisTestRunner`); see the section called “Unit Testing”.

Agile Testing

Apache Isis is very much aligned to agile development practices, and provides two complementary mechanisms for you to test-drive the development of your application. A story test (BDD ensures that the right system is built. A unit test ensures that the system is built right.

Story (BDD) Testing

Many agile practitioners use story tests as a means to capture the acceptance (or completion) criteria for high-level user stories. These story tests are typically captured in a non-programmatic form so that is understandable to domain experts as well as programmers.

Apache Isis provides an out-of-the-box integration with Concordion [http://concordion.org] (where the story test is captured as HTML).[1]There is full coverage of how to use this integration in the BDD viewer's (`viewers/bdd`) documentation.

Unit Testing

Unlike story tests, unit tests are normally written in a programming language, typically in a framework such as JUnit [http://junit.org].

When writing unit tests, you have a choice. Since all the business logic in Isis applications is encapsulated in domain object pojos, you can just write unit tests using nothing more than JUnit and perhaps also a mocking library such as JMock [http://jmock.org].

A slightly more sophisticated approach is to use the JUnit integrations and supporting classes of the Wrapper programming model . The idea of these utilities is to wrap your domain objects in proxies that apply the same rules as an *Apache Isis* viewer. For example, if you try to change a property or invoke an action that is disabled, then the proxy will throw an exception. You write your test to pass if the exception is thrown, and failed otherwise (eg using `@Test(expected=DisabledException.class)`).

Full documentation for unit testing support is in the JUnit viewer's guide (`viewers/junit`).

A Domain Library

The idea behind the `domain` module is to provide some off-the-shelf code for you to use and adapt in your own applications. This code is fully tested (comes with tests), and is intended to be well-designed. Using code from the library should give you a kick-start in writing your own domain applications.

[1]An integration with FitNesse [http://fitnesse.org] (where the story test is captured within a wiki) also exists. Due to licensing this is not part of *Isis* proper.

Note

The library is currently very modest, but we hope it might build up in time.

Of course, there's a limit to the complexity of the code that's included in the library, because every domain is different. But having a full set of tests should allow you to safely refactor the code to your own requirements.

There's also no guarantee that the library will contain code for your specific requirement. But if you do write some domain logic that you think might be reusable by others, why not consider donating it back to Isis when you've done so?

The domain library breaks out into three: services, entities and values.

Domain Services

Apache Isis applications use *domain service*s (a domain-driven design pattern) to allow objects to interact with other domains (or *bounded context*s, to use the correct term). These domain services are automatically injected into each domain object as it is instantiated.

Domain services split into two: those that interact with technical domains (such as sending email, or rendering PDFs), and those that interact with business domains (such as general ledger, CRMs or legacy systems).

Obviously domain services that bridge to business domain services are always likely to be specific to each individual application. So the services in the `domain/services` module focus on providing off-the-shelf implementations for some of the more common *technical* domain services.

Domain Values

As you probably would expect, *Apache Isis* treats Java primitives, Java wrapper classes, `java.lang.String`, `java.math.BigDecimal` and `java.math.BigInteger` as immutable values, It allows `java.util.Date` and its subclasses (java.sql.Date, java.sql.Time and `java.sql.Timestamp`) to be treated as values also. In addition, it offers a small number of its own value types, such as `Money`.

In addition, it is possible to extend Isis support for value types using its `@Value` annotation. We hope in time to provide out-of-the-box integrations for popular 3rd party libraries such as JodaTime.

Domain Entities

In the future we hope that Isis will provide some well-defined and ready-tested domain entities for use in your systems. For example, a set of entities to model customers (`Customer`/`Name`/`Address`) might be useful in many systems.

Deploying an Isis Application

There are two main decisions to make in deploying an *Isis* application.

Persistence (the Default Runtime / Object Store)

In the same way that viewers provide pluggability for the "front-end", *Apache Isis* also offers pluggability on the back-end, i.e. persistence.

In fact, there are two decisions to be made here: which runtime to use, and (if using the default runtime), which object store to use. The default runtime is the one originally developed within the *Naked Objects Framework*, and therefore has been inherited by *Apache Isis*.[2]The default runtime is also the one configured by the quickstart archetype.

If using the default runtime, you have the option of choosing the objectstore. As already discussed, the quickstart archetype configures the in-memory objectstore by default, which is great for rapid prototyping. However, at some point you will need to integrate with a "real" object store that persists object to some sort of serialized persistence.

Of the object stores to provided by the default runtime, some are easy to configure, some more complex; some are only suitable for single-user apps, others for multi-users. Each of the object stores has its own documentation, so you can select the correct object store to choose:[3]

- the XML object store (in `objectstores/xml`) is designed for single-user systems, and persists to its own internal (proprietary) XML format;
- the SQL object store (in `objectstores/sql`) is a multi-user object store that persists to an RDBMS (direct over JDBC);
- the NoSQL object store (in `objectstores/nosql`) is a multi-user object store that persists to NoSQL databases in JSON form.

You'll also find coverage of the object store API itself in the default runtime's documentation.

Alternatively, the default runtime can be configured to support client/server remoting. This configuration only makes sense for the DnD viewer, whereby it is configured as the client and a separate instance of Isis runs as the server. Again, full guidance is available in the default runtime's documentation.

Authentication and Authorization

Apache Isis provides an APIs for both authentication and authorization.

The core implementation of these APIs are simple basic noop-based authentication and authorization mechanisms. For deployment into production, you'll need to configure with another alternative implementation.

One option is the file-based implementation (in `security/file`), that stores the security information in flat files. This is simple, but unlikely to be robust enough for enterprise use.

Another alternative is to use the LDAP implementation (in `security/ldap`), which can integrate with an LDAP infrastructure if you have one. If you have a different security infrastructure, then you might consider to write your own implementation (the API is not complex).

[2]At the time of writing the default runtime is the only real runtime available; however we expect to be developing lighter-weight replacements for the default runtime in the future.

[3]In addition to those listed here, there have also been objectstores for JPA (using Hibernate) and BerkeleyDB. These are currently not part of Apache Isis distribution because of incompatible licenses. Support for an ORM will in the future be provided either by porting the JPA object store to the default runtime, or, we may support it via a JDO-based new runtime.

Part II. Writing Domain Objects

The conventions of the programming model are best described as 'intentional' - they convey an intention as to how domain objects, their properties and behaviours, are to be made available to users. The specific way in which those intentions are interpreted or implemented will depend upon the framework, and/or the particular components or options selected within that framework.

To pick a single example, marking up a domain class with the annotation `@Bounded` is an indication that the class is intended to have only a small number of instances and that the set does not change very often - such as the class `Country`. This is an indication to a viewer, for example, that the whole set of instances might be offered to the user in a convenient form such as a drop-down list. The programming convention has *not* been defined as `@DropDownList` because any equivalent mechanism will suffice: a viewer might not support drop-down-lists but instead might provide a capability to select from an `@Bounded` class by typing the initial letters of the desired instance.

This part of the guide is a set of chapters that provides how-to's for writing domain objects, by which we mean domain entities, value types, services and repositories/factories.

Chapter 4. Domain Entities

How-to's relating to the domain objects, specifically domain entities.

This chapter contains how-to's for programming conventions that apply to the domain objects, or more specifically domain entities. Domain values conventions are covered in Chapter 10, *Value Types*.

How to not inherit from framework superclasses

It isn't mandatory for domain entities to inherit from any framework superclass; they can be plain old java objects (pojos) if required. However, they do at a minimum need to have a `org.apache.isis.applib.DomainObjectContainer` injected into them (an interface), from which other framework services can be accessed.

If you don't have a requirement to inherit from any other superclass, then it usually makes sense to inherit from mention `org.apache.isis.applib.AbstractDomainObject`, which already supports the `DomainObjectContainer` and has a number of convenience helper methods.

There is further coverage of `DomainObjectContainer` in the section called “How to insert behaviour into the object life cycle” and also in Chapter 9, *Domain Services, Repositories and Factories*.

How to specify a title for an object

A title is used to identify an object to the user in the user interface. For example, a `Customer`'s title might be the organization's customer reference, or perhaps (more informally) their first and last names.

By default, the framework will use the object's `toString()` method as the title. Most titles tend to be made up of the same set of elements: for example a Customer's name might be the concatenation of their customer first name and their ;ast name. For these the `@Title` annotation can be used:

```
public class Customer {
  @Title
  public String getFirstName() { ... }
  @Title
  public String getLastName() { ... }
  ...
}
```

For more control, the order of the title components can be specified using a sequence number (specified in Dewey decimal format):

```
public class Customer {
  @Title("1.0")
  public String getFirstName() { ... }
  @Title("1.1")
  public String getLastName() { ... }
  ...
}
```

For more control the title can be declared imperately using the `title()` method (returning a `String`). This leaves the programmer needs to make use of the `toString()` method for other purposes, such as

for debugging. For example, to return the title for a customer which is their last name and then first initial of their first name, we could use:

```
public class Customer {
  public String title() {
    return getLastName() + ", " + getFirstName().substring(0,1);
  }
  ...
}
```

The applib contains a class, `org.apache.isis.applib.util.TitleBuffer`, which you can use to help create title strings if you so wish. See Appendix E, *Utility Classes* for more details.

How to specify the icon for an object's class

By default, the framework will look for an image in the `images` directory (either from the classpath or from the filesystem) that has the same name as the object class. Multiple file extensions are searched for, including `.png`, `.gif` and `.jpg` (in order of preference). For example, fan object of type `Customer` it will look for `Customer.png`, `Customer.gif`, `Customer.jpg` etc.

If the framework finds no such file, then it will work up the inheritance hierarchy to see if there is an icon matching the name of any of the super-classes, and use that instead. If no matching icon is found then the framework will look for an image called `default.png`, `default.gif` or `default.jpg` in the images directory, and if this has not been specified, then the framework will use its own default image for an icon.

We strongly recommend that you adopt 'pascal case' as the convention for icon file names: if you have a class called `OrderLine`, then call the icon `OrderLine.png`. Actually, the framework will also recognise `orderline.png`, but some operating systems and deployment environments are case sensitive, so it is good practice to adopt an unambiguous convention.

Alternatively, you can use the `iconName`() method instead:

```
public String iconName() {
    return "Person";
}
```

This makes it easy for more than one class to use the same icon, without having to duplicate the image file.

How to specify the icon for an individual object's state

As discussed in the section called “How to specify the icon for an object's class”, the `iconName()` method may be used to specify an object. The value returned from this method need not be static, and so it can be used to represent the state of an individual object.

For example, an instance of `Product` could use a photograph of the product as an icon, using:

```
public class Product {
    public String iconName() {
        return "Product-" + getPhotograph();
    }
    ...
```

```
}
```

Alternatively, an `Order` might vary the icon according to the status of the object:

```
public class Order {
    public String iconName() {
        return "Order-" + getStatus();
    }
    ...
}
```

How to specify a name and/or description for an object

By default, the name (or type) of an object, as displayed to the user will be the class name. However, if an `@Named` annotation is included, then this will override the default name. This might be used to include punctuation or other characters that may not be used within a class name, or when - for whatever reason - the name of the class includes technical artifacts (for example project-defined prefixes or suffices). It is also useful if the required name cannot be used because it is a keyword in the language.

By default the framework will create a plural version of the object name by adding an 's' to singular name, or a 'ies' to names ending 'y'. For irregular nouns or other special case, the `@Plural` annotation may be used to specify the plural form of the name explicitly.t

The programmer may optionally also provide a `@DescribedAs` annotations, containing a brief description of the object's purpose, from a user perspective. The framework will make this available to the user in a form appropriate to the user interface style - for example as a tooltip.

For example:

```
@Named("Customer")
@Plural("Customers")
@DescribedAs("Individuals or organizations that have either "+
             "purchased from us in the past or "+
             "are likely to in the future")
public class CustomerImpl implements ICustomer {
    ...
}
```

Note

There is an entirely separate mechanism for dealing with Internationalisation, details of which can be found in the core documentation.

How to specify that an object should not be persisted

Non-persisted objects are intended to be used as view models; they aggregate some state with respect to a certain process. This may be read-only (eg a projection of certain informaiton) or read-write (eg a wizard-like process object). Either way, the viewer is expected to interpret this by not providing any sort of automatic "save" menu item if such an object is returned to the GUI.

Non-persisted objects that are read-only are typically also marked as immutable (see the section called “How to specify that an object should never be modified by the user”).

To indicate that an object cannot be persisted, use the `@NotPersistable` annotation.

How to specify that an object should never be modified by the user

Some objects have state which should not be modifiable; for example those representing reference data. The viewer is expected to interpret this by which suppressing any sort of "edit" button.

To indicate that an object cannot be modified, use the `@Immutable` annotation.

For example:

```
@Immutable
public class ChasingLetter implements PaymentReclaimStrategy {
    ...
}
```

See also the section called “How to control when an object can be modified”.

How to control when an object can be modified

Some objects have state which should not be modifiable only under certain conditions; for example an invoice can not be modified after it has been paid. The viewer is expected to interpret this by suppressing any sort of "edit" button.

To indicate that an object cannot be modified, use the `String disabled(Type type)` method.

For example:

```
public class FeeInvoice implements Invoice {
   public String disabled(Type type){
    ...
   }
}
```

The Type is from `org.apache.isis.applib.Identifier`:

```
    /**
     * What type of feature this identifies.
     */
    public static enum Type {
        CLASS, PROPERTY_OR_COLLECTION, ACTION
    }
```

and provides fine grain control over disabling actions and properties.

The return String is null if the the object (action or property) is not disabled, or the reason why it is disabled, similar to the section called “How to prevent a property from being modified”.

See also the section called “How to control when an object can be modified”.

How to control when an object is visible

To when an object is visible, provide a hidden() method:

```
public class TrackingAction implements Tracking {
   public boolean hidden(){
    ...
   }
}
```

If the function returns true, all properties and methods will be hidden from the user, similar to the section called “How to hide a property from the user”.

How to specify that a class of objects has a limited number of instances

Sometimes an entity may only have a relatively small number of instances, for example the types of credit cards accepted (Visa, Mastercard, Amex). Viewers will typically expected to render the complete set of instances as a drop down list whenever the object type is used (ie as a property or action parameter).

To indicate that a class has a limited number of instances, use the `@Bounded` annotation. Note that there is an implied expectation is that the list of objects is small, and relatively cheap to obtain from the object store.

An alternative way to specify a selection of objects is using the `choicesXxx()` supporting methods.

For example:

```
@Bounded
public class PaymentMethod {
    ...
}
```

Alternatively, you might want to use a (Java) `enum`, because these are implicitly bounded.

How to create or delete objects within your code

When you create any domain object within your application code, it is important that the framework is made aware of the existence of this new object - in order that it may be persisted to the object store, and in order that any services that the new object needs are injected into it.

Just specifying `new Customer()`, for example, will create a `Customer` object, but that object will *not* be known to the framework. However, since we do not want to tie our domain objects to a particular framework, we use the idea of a 'container' to mediate, specified by the `org.apache.isis.applib.DomainObjectContainer` interface. See Appendix C, *`DomainObjectContainer` interface* for the full list of methods provided by `DomainObjectContainer`.

This interface defines the following methods for managing domain objects:

- `<T> T newTransientInstance(final Class<T> ofClass)`

Returns a new instance of the specified class, that is transient (unsaved). The object may subsequently be saved either by the user invoking the Save action (that will automatically be rendered on the object view) or programmatically by calling `persist(Object transientObject)`

- `<T> T newPersistentInstance(final Class<T> ofClass)`

  Creates a new object already persisted.

- `boolean isPersistent()`

  Checks whether an object has already been persisted. This is often useful in controlling visibility or availability of properties or actions.

- `void persist(Object transientObject)`

  Persists a transient object (created using `newTransientInstance(...)`, see above).

- `void persistIfNotAlready(Object domainObject)`

  It is an error to persist an object if it is already persistent; this method will persist only if the object is not already persistent (otherwise it will do nothing).

- `void remove(Object persistentObject)`

  Removes (deletes) from the object store, making the reference transient.

- `void removeIfNotAlready(Object domainObject)`

  It is an error to remove an object if it is not persistent; this method will remove only if the object is known to be persistent (otherwise it will do nothing).

A domain object specifies that it needs to have a reference to the `DomainObjectContainer` injected into by including the following code:

```
private DomainObjectContainer container;
protected DomainObjectContainer getContainer() {
    return container;
}
public final void setContainer(final DomainObjectContainer container) {
    this.container = container;
}
```

Creating or deleting objects is then done by invoking those methods on the container. For example the following code would then create a new `Customer` object within another method:

```
Customer newCust = getContainer().newTransientInstance(Customer.class);
newCust.setName("Charlie");
getContainer().persist(newCust);
```

If you are able to make your domain object inherit from `org.apache.isis.applib.AbstractDomainObject` then you have direct access to those methods, so the code would become:

```
Customer newCust = newTransientInstance(Customer.class);
newCust.setName("Charlie");
persist(newCust);
```

As an alternative to putting the creation logic within your domain objects, you could alternatively delegate to an injected factory (see Chapter 9, *Domain Services, Repositories and Factories*). Ultimately factories just delegate back to `DomainObjectContainer` in the same way, so from a technical standpoint there is little difference. However it is generally worth introducing a factory because it provides a place to centralize any business logic. It also affords the opportunity to introduce a domain term (eg `ProductCatalog` or `StudentRegister`), thereby reinforcing the "ubiquitous language".

These methods are actually provided by the `org.apache.isis.applib.AbstractContainedObject` and so are also available on `org.apache.isis.applib.AbstractService` (and, hence, on `org.apache.isis.applib.AbstractFactoryAndRepository`) for creating objects within a service.

Warning

It is possible to create a transient object within another transient object. When the framework persists any transient object, by default it will automatically persist any other transient object referenced by that object. However, if any of these transient objects are to be exposed to the user (while in their transient state), then you need to write your code very carefully - anticipating the fact that the user could elect to save any of the transient objects at any point - which could cause the graph of related objects to be persisted in an invalid state.

The recommended approach is, if possible, to mark these supplementary classes as not persistable by the user (see the section called “@NotPersistable”), or not to permit the user to create a new transient object that is a child of an existing transient object, but, rather, to require the user to save the parent object first.

How to insert behaviour into the object life cycle

Apache Isis is responsible for managing the object lifecycle, persisting, updating or removing objects from the persistent object store as required. For many applications the domain objects are unaware of this. If required, though, an object can provide callback methods (all optional) so that the framework can notify it of its persistence state.

For example, the `persisted`() method is called after an object has been persisted. This could be used to setup a reverse association that should only be created once the new object has been persisted.

The full list of callbacks is shown below.

Table 4.1. Object lifecycle methods

Method	When called by framework
`created()`	following the logical creation of the object (that is, after `newTransientInstance()` has been called)
`loading()`	when a persistent object is about to be loaded into memory
`loaded()`	once the persistent object has just been loaded into memory
`persisting()` or `saving()`	just before a transient object is first persisted.
`persisted()` or `saved()`	just after a transient object is first persisted.

Method	When called by framework
`updating()`	after any property on a persistent object has been changed and just before this change is persisted
`updated()`	after a changed property on a persistent object has been persisted
`removing()` or `deleting()`	when a persistent object is just about to be deleted from the persistent object store.
`removed()` or `deleted()`	when a persistent object has just been deleted from the persistent object store.

How to perform lazy loading (generally done automatically)

The `DomainObjectContainer` provides the `resolve()` method in order to lazily resolve the value of a property or a collection. In earlier versions of the framework it was necessary to call this method prior to accessing or mutating any property or collection. This is no longer required because *Apache Isis* uses bytecode enhancement to automatically call this method.

While it is possible to disable this bytecode enhancement using `isis.properties` file, this is not generally recommended. If it is disabled then the `resolve()` method may need to be called manually.

How to perform dirty object tracking (generally done automatically)

The `DomainObjectContainer` provides the `objectChanged()` method in order to mark an object's state as having changed, and therefore requiring an update to the persistent object store. In earlier versions of the framework it was necessary to call this method after mutating any the property or collection. This is no longer required because *Apache Isis* uses bytecode enhancement to automatically call this method.

While it is possible to disable this bytecode enhancement using `isis.properties` file, this is not generally recommended. If it is disabled then the `objectChanged()` method may need to be called manually.

How to ensure object is in valid state

A `validate()` method may be added to provided validation at object level, prior to making an object persistent.

The syntax is:

```
public String validate()
```

A non-`null` value is the taken to be the reason why the object cannot be saved.

This is particularly useful for validating fields in relation to each other.

For example:

```
public class Booking {
```

```
    private Date fromDate;
    public Date getFromDate() {...}
    public void setFromDate(Date d) {...}

    private Date toDate;
    public Date getToDate() {...}
    public void setToDate(Date d) {...}

    public String validate() {
        if (fromDate.getTicks() > toDate.getTicks()) {
            return "From Date cannot be after To Date";
        }
        return null;
    }
    ...
}
```

This will prevent the user from saving a transient `Booking` where the From Date falls after the To Date. Note that in this example, the two date properties could also have their own individual validate methods - for example in order to test that each date was after today.

Warning

At the time of writing, the `validate()` method is called only when the object is first saved, not when it is subsequently updated. For validation of subsequent updates, the workaround is necessary to build the validation logic into the individual property validation methods (though these could delegate to a common `validate()` method).

See ISIS-18 [https://issues.apache.org/jira/browse/ISIS-18] for the status of this issue.

Chapter 5. Domain Entity Properties

How-to's relating to an domain entity's properties.

The following conventions are concerned with specifying the properties of a domain entity, and the means by which a user can interact with those properties.

How to add a property to an object

Properties are specified using the JavaBean conventions, recognizing a standard accessor/mutator pair (`get` and `set`).

The syntax is:

```
public PropertyType getPropertyName()

public void setPropertyName(PropertyType param)
```

where `PropertyType` is a primitive, a value object or an entity object.

Properties may either be for a value type or may reference another entity. Values include Java primitives, and JDK classes with value semantics (eg `java.lang.Strings` and `java.util.Dates`; see Chapter 10, *Value Types* for the full list). It is also possible to write your own value types (see Chapter 10, *Value Types*). A property referencing another domain object is sometimes called an association.

For example, the following example contains both a value (`String`) property and a reference (`Organisation`) property:

```
public class Customer {

    private String firstName;
    public String getFirstName() {
        return firstName;
    }
    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }


    private Organisation organisation;
    public Organisation getOrganisation() {
        return organisation;
    }
    public void setOrganisation(Organisation organisation) {
        this.organisation = organisation;
    }

    ...
}
```

How to specify a name and/or description for a property

Specifying the name for a property

By default the framework will use the property name itself to label the property on the user interface. If you wish to override this, use the `@Named` annotation on the property.

For example:

```
public class Customer() {
    @Named("Given Name")
    public String getFirstName() { ... }
    ...
}
```

Specifying a description for a property

An additional description can be provided on a property using the `@DescribedAs` annotation. The framework will take responsibility to make this description available to the user, for example in the form of a tooltip.

For example:

```
public class Customer() {
    @DescribedAs("The customer's given name")
    public String getFirstName() { ... }
    ...
}
```

How to specify the order in which properties are displayed

The `@MemberOrder` annotation provides a hint to the viewer as to the order in which the properties (and also collections, see the section called “How to specify the order in which collections are displayed”) should appear in the GUI.

For example:

```
public class Customer() {
    @MemberOrder("1")
    public String getFirstName() { ... }
    ...

    @MemberOrder("2")
    public String getLastName() { ... }
    ...
}
```

How to make a property optional (when saving an object)

By default, when a new transient (unsaved) object is presented to the user, values must be specified for all properties before the object may be saved.

To specify that a particular property is optional, use the `@Optional` annotation.

How to specify the size of `String` properties

Use:

- the `@MaxLength` to specify the maximum number of characters that may be stored within a `String` property.
- the `@TypicalLength` to specify the typical number of characters that are likely to be stored within a `String` property. Viewers are expected to use this as a hint as to the size of the field to render for the property.
- the `@MultiLine` annotation as a hint to indicate that the property should be displayed over multiple lines (eg as a text area rather than a text field).

For example:

```
public class Ticket {
    @TypicalLength(50)
    @MaxLength(255)
    public String getDescription() { ... }
    ...

    @MaxLength(2048)
    @MultiLine
    public String getNotes() { ... }
    ...
}
```

How to make a derived property

Lazily derived

Omitting the mutator (`setXxx()`) method for a property indicates both that the field is derived, and is not be persisted. This is the standard approach to derive a property from other information available to the object. It also happens to be compatible with Java Persistence Api (JPA) semantics.

For example:

```
public class Employee {
    public Department getDepartment() { ... }
    ...

    // this is the derived property
```

```
    public Employee getManager() {
        if (getDepartment() == null) { return null; }
        return getDepartment().getManager();
    }
    ...
}
```

Although there is no setter, such properties can nevertheless be modified if a modifyXxx supporting method is provided:

```
public class Employee {
    public Department getDepartment() { ... }
    ...

    // this is the derived property
    public Employee getManager() { ... }

    // this makes the derived property modifiable
    public void modifyManager(Employee manager) {
        if (getDepartment() == null) { return; }
        getDepartment().modifyManager(manager);
    }

    ...
}
```

Note how the implementation of such a modifyXxx() method typically modifies the original source of the information (the `Department` object).

Eagerly derived

An alternative to omitting the mutator is to mark a property is to keep the mutator, but to annotate the property with `@NotPersisted`.

Because the data is non-persisted, some other mechanism must be used to initialize the property. Typically this will be a lifecycle method (see the section called “How to insert behaviour into the object life cycle”), which eagerly derive the value of the property.

For example:

```
public class Employee {
    public void loaded() {
        if (getDepartment() != null) {
            setManager(getDepartment().getManager());
        }
    }

    public Department getDepartment() { ... }
    ...

    @NotPersisted
    public Employee getManager() { ... }
    private void setManager(Employee manager) { ... }
    ...
}
```

Eagerly derived properties can be made modifiable either changing the setter to have public visibility, or introducing a modifyXxx() method.

How to hide a property from the user

The mechanism for hiding a property is broadly the same as for hiding a collection (see the section called “How to hide a collection”) or an action (see the section called “How to hide an action”).

For control over the entire object, see the section called “How to control when an object is visible”.

Hiding a property always

To prevent a user from viewing a property at all, use the `@Hidden` annotation. A common use case is to hide an internal Id, eg perhaps as required by the object store.

For example:

```
public class OrderLine {
    private Integer id;
    @Hidden
    public Integer getId() { ... }
    public void setId(Integer id) { ... }
    ...
}
```

Hiding a property based on the persistence state of the object

As a refinement of the above, a property may be optionally hidden using the `@Hidden` annotation based on the persistence state of the object:

- to hide the property when the object is transient, use `@Hidden(When.UNTIL_PERSISTED)`
- to hide the property when the object is persistent, use `@Hidden(When.ONCE_PERSISTED)`

Hiding a property under certain conditions

A `hideXxx()` method can be used to indicate that a particular object's property should be hidden under certain conditions, typically relating to the state of that instance.

The syntax is:

```
public boolean hidePropertyName()
```

Returning a value of `true` indicates that the property should be hidden.

For example:

```
public class Order {
    public String getShippingInstructions() { ... }
    public void setShippingInstructions(String shippingInstructions) { ... }
    public boolean hideShippingInstructions() {
        return hasShipped();
    }
```

```
    ...
}
```

Hiding a property for specific users or roles

It is possible to hide properties for certain users/roles by calling the `DomainObjectContainer#getUser()` method. See the section called “Hiding, disabling or validating for specific users or roles”for further discussion.

How to prevent a property from being modified

Preventing the user from modifying a property value is known as 'disabling' the property. Note that this doesn't prevent the property from being modified programmatically.

The mechanism for disabling a property is broadly the same as for disabling a collection (see the section called “How to prevent a collection from being modified”) or a collection (see the section called “How to prevent a collection from being modified”).

For control over the entire object, see the section called “How to control when an object can be modified”.

Disabling a property permanently

To prevent a user from being able to modify the property at all, use the `@Disabled` annotation.

For example:

```
public class OrderLine {
    private int quantity;
    @Disabled
    public String getQuantity() { ... }
    public void setQuantity(int quantity) { ... }
    ...
}
```

Note that a setter is still required; this is used by the framework to recreate the object when pulled back from the persistent object store.

Disabling a property based on the persistence state of the object

As a refinement of the above, a property may be optionally disabled using the `@Disabled` annotation based on the persistence state of the object:

- to disable the property when the object is transient, use `@Disabled(When.UNTIL_PERSISTED)`
- to disable the property when the object is persistent, use `@Disabled(When.ONCE_PERSISTED)`

Disabling a property under certain conditions

A supporting `disableXxx()` method can be used to disable a particular instance's member under certain conditions

The syntax is:

```
public String disablePropertyName()
```

A non-`null` return value indicates the reason why the property cannot be modified. The framework is responsible for providing this feedback to the user.

For example:

```
public class OrderLine {
    public String getQuantity() { ... }
    public void setQuantity(int quantity) { ... }
    public String disableQuantity() {
        if (isSubmitted()) {
            return "Cannot alter any quantity after Order has been submitted";
        }
        return null;
    }
}
```

If there are multiple reasons to disable a property, take a look at the `org.apache.isis.applib.util.ReasonBuffer` helper.

Disabling a property for certain users/roles

It is possible to disable properties for certain users/roles by calling the `DomainObjectContainer#getUser()` method. See the section called “Hiding, disabling or validating for specific users or roles”for further discussion.

How to validate user input for a property

Declarative validation

For properties that accept a text input string, such as `String` and `Date`, there are convenient mechanisms to validate and/or normalise the values typed in:

- For `Date` and number values the `@Mask` annotation may be used.
- For `String` properties the `@RegEx` annotation may be used.

More complex validation can also be performed imperatively (below).

Imperative validation

A supporting `validateXxx()` method is used to check that a new value for a property is valid.

If the proffered value is deemed to be invalid then the property will not be changed. A non-null return `String` indicates the reason why the member cannot be modified/action be invoked; the framework is responsible for providing this feedback to the user.

The syntax is:

```
public String validatePropertyName(PropertyType param)
```

where `PropertyType` is the same type as that of the property itself.

For example:

```
public class Exam {
    public int getMark() { ... }
    public void setMark(int mark) { ... }
    public validateMark(int mark) {
        return !withinRange(mark)? "Mark must be in range 0 to 30":null;
    }
    private boolean withinRange(int mark) { return mark >= 0 && mark <= 30; }
}
```

How to set up the initial value of a property programmatically

After an object has been created (see the section called “How to create or delete objects within your code”), there are several different ways to setup the initial values for an object's properties.

By each property's default values

Firstly, the default value for a property can be supplied using a supporting `defaultXxx()` method. The syntax for specifying a default value is:

```
public PropertyType defaultPropertyName()
```

where `PropertyType` is the same type as that of the property itself.

```
public class Order {
    public Address getShippingAddress() { ... }
    public void setShippingAddress() { ... }
    public Address defaultShippingAddress() {
        return getCustomer().normalAddress();
    }
    ...
}
```

By the `created()` lifecycle method

Alternatively, the domain object may choose to initialize its property values in the `created()` lifecycle method (see the section called “How to insert behaviour into the object life cycle”). This is called after any `defaultXxx()` methods are called.

Programmatically, by the creator

Third, and perhaps most obviously, the creator of the object could initialize the properties of the object immediately after calling `newTransientInstance(...)`. This would be appropriate if the creator had reason to override any values setup in the `defaultXxx()` or `created()` methods discussed above.

How to specify a set of choices for a property

The simplest way to provide the user with a set of choices for a property (possibly rendered as a drop-down list, for example) is to ensure that the type used by the property is marked up as `@Bounded` (see the section called “How to specify that a class of objects has a limited number of instances”).

However, this is not always appropriate. For example you might wish to provide the user with the choice of all the `Address`es known for that `Customer`, with the most recently-used address as the default.

The syntax for specifying a list of choices is either:

```
public Collection<PropertyType> choicesPropertyName()
```

or alternatively

```
public PropertyType[] choicesPropertyName()
```

where `PropertyType` is the same type as that of the property itself.

For example:

```
public class Order {
    public Address getShippingAddress() { ... }
    public void setShippingAddress() { ... }
    public List<Address> choicesShippingAddress() {
        return getCustomer().allActiveAddresses();
    }
    ...
}
```

How to trigger other behaviour when a property is changed

If you want to invoke functionality whenever a property is changed by the user, then you should create a supporting `modifyXxx()` method and include the functionality within that. The syntax is:

```
public void modifyPropertyName(PropertyType param)
```

Why not just put this functionality in the setter? Well, the setter is used by the object store to recreate the state of an already persisted object. Putting additional behaviour in the setter would cause it to be triggered incorrectly.

For example:

```
public class Order() {
    public Integer getAmount() { ... }
    public void setAmount(Integer amount) { ... }
    public void modifyAmount(Integer amount) {
        setAmount(amount);
        addToTotal(amount);
    }
    ...
}
```

Here the `modifyAmount()` method also calls the `addToTotal()` call as well as the `setAmount()` method. We don't want this `addToCall()` method to be called when pulling the object back from the object store, so we put it into the modify, not the setter.

You may optionally also specify a `clearXxx()` which works the same way as modify `modify Xxx()` but is called when the property is cleared by the user (i.e. the current value replaced by nothing). The syntax is:

```
public void clearPropertyName()
```

To extend the above example:

```
public class Order() {
    public Integer getAmount() { ... }
    public void setAmount(Integer amount) { ... }
    public void modifyAmount(Integer amount) { ... }
    public void clearAmount() {
        removeFromTotal(this.amount);
        setAmount(null);
    }
    ...
}
```

How to setup a bidirectional relationship

The `modifyXxx()` and `clearXxx()` methods (see the section called “How to trigger other behaviour when a property is changed”) can be used to setup bidirectional relationships. This is typically done with 1:m relationships, eg between `Order` and `OrderLine`, or `Department` and `Employee`.

For further discussion and a complete example, see the section called “How to maintain bidirectional relationships”.

Chapter 6. Domain Entity Collections

How-to's relating to a domain entity's collections.

A collection is a list of references to several entity that have a common type. The following conventions are concerned with specifying the collections of an object, and the means by which a user can interact with those collections.

How to add a collection to an object

A collection is recognized via an accessor/mutator method pair (`get` and set) for any type of collection provided by the programming language.

The syntax is either:

```
public Collection<EntityType> getCollectionName()
private void setCollectionName(Collection<EntityType> param)
```

or:

```
public List<EntityType> getCollectionName()
private void setCollectionName(List<EntityType> param)
```

or:

```
public Set<EntityType> getCollectionName()
private void setCollectionName(Set<EntityType> param)
```

A mutator is required, but it need only have `private` visibility.

Note:

Note

`Map`s cannot be used for collections.

It is recommended that the collections be specified using generics (for example: `List<Customer>`). That way the framework will be able to display the collection based on that type definition. If a raw type is used then the framework will attempt to infer the type from the addToXxx() / removeFromXxx() supporting methods, if specified (see the section called “How to trigger other behaviour when an object is added or removed”). If the framework is unable to determine the type of the collection, it will mean that some viewers will represent the collection is a less sophisticated manner (eg a simple list of titles rather than a table).

For example:

```
public class Employee { ... }

public class Department {
    private List<Employee> employees = new ArrayList<Employee>();
    public List <Employee> getEmployees() {
        return employees;
    }
    private void setEmployees(List<Employee> employees) {
```

```
        this.employees = employees;
    }
    ...
}
```

How to specify a name and/or description for a collection

Specifying the name for a collection

By default the framework will use the collection name itself to label the collection on the user interface. If you wish to override this, use the `@Named` annotation on the collection.

For example:

```
public class Customer {
    @Named("Placed Orders")
    public List<Order> getOrders() { ... }
    ...
}
```

Specifying a description for a collection

An additional description can be provided on a collection using the `@DescribedAs` annotation. The framework will take responsibility to make this description available to the user, for example in the form of a tooltip.

For example:

```
public class Customer {
    @DescribedAs("Those orders that have been placed (and possibly shipped) " +
                 "by this customer given name by which this customer is known")
    public List<Order> getOrders() { ... }
    ...
}
```

How to specify the order in which collections are displayed

The `@MemberOrder` annotation provides a hint to the viewer as to the order in which the collections (and also properties, see the section called “How to specify the order in which properties are displayed”) should appear in the GUI.

For example:

```
public class Customer {
    @MemberOrder("3")
    public Collection<Order> getRecentOrders() { ... }
    ...

    @MemberOrder("4")
```

```
    public Collection<Order> getOrders() { ... }
    ...
}
```

How to make a derived collection

Collections can be derived by omitting the mutator (the same way as properties, see the section called “How to make a derived property”).

For example:

```
public class Department {
    // Standard collection
    private List<Employee> employees = new ArrayList<Employee>();
    public List<Employee> getEmployees() { ... }
    private void setEmployees(List<Employee>) { ... }

    // Derived collection
    public List<Employee> getTerminatedEmployees() {
        List<Employee> terminatedEmployees = new ArrayList<Employee>();
        for(Employee e: employees) {
            if (e.isTerminated()) {
                addTo(terminatedEmployees, e);
            }
        }
        return terminatedEmployees;
    }
    ...
}
```

Derived collections are not persisted, though may be modified if there is an addToXxx() or removeFromXxx() supporting method. As for derived properties, the implementations of these mutators change the underlying data. For example:

```
public class Department {
    ...

    public void addToTerminatedEmployees(Employee employee) {
        employee.setTerminated(true);
    }
    public void removeFromTerminatedEmployees(Employee employee) {
        employee.setTerminated(false);
    }
}
```

How to hide a collection

The mechanism for hiding a collection is broadly the same as for hiding a property (see the section called “How to hide a property from the user”) or an action (see the section called “How to hide an action”).

Hiding a collection permanently

To prevent a user from viewing a collection at all, use the `@Hidden` annotation.

For example:

```
public class Order {
    private List<Order> cancelledOrders = new ArrayList<Order>();
    @Hidden
    public List<Order> getCancelledOrders() { ... }
    private void setCancelledOrders(List<Order> cancelledOrders) { ... }
    ...
}
```

Hiding a collection based on the persistence state of the object

As a refinement of the above, a collection may be optionally hidden using the `@Hidden` annotation based on the persistence state of the object:

- to hide the collection when the object is transient, use `@Hidden(When.UNTIL_PERSISTED)`
- to hide the collection when the object is persistent, use `@Hidden(When.ONCE_PERSISTED)`

Hiding a collection under certain conditions

A `hideXxx()` method can be used to indicate that a particular object's collection should be hidden under certain conditions, typically relating to the state of that instance.

The syntax is:

```
public boolean hideCollectionName()
```

Returning a value of `true` indicates that the collection should be hidden.

For example:

```
public class Order {
    @Hidden
    public List<Order> getRushOrders() { ... }
    ...
}
```

Hiding a collection for specific users or roles

It is possible to hide collections for certain users/roles by calling the `DomainObjectContainer#getUser()` method. See the section called “Hiding, disabling or validating for specific users or roles” for further discussion.

How to prevent a collection from being modified

Preventing the user from adding to or removing from a collection is known as 'disabling' the collection.

The mechanism for disabling a collection is broadly the same as for disabling a property (see the section called “How to prevent a property from being modified”) or a action (see the section called “How to prevent an action from being invoked”).

Disabling a collection permanently

Some, though not all, viewers allow the user to directly manipulate the contents of a collection. For example, the DnD viewer will allow new objects to be "dropped" into a collection, and existing objects removed from a collection.

Although it is possible to associate behaviour with such actions (see the section called “How to trigger other behaviour when an object is added or removed”), it may be preferred to only allow modification through actions. Or, the application may be deployed using a viewer that doesn't fully support direct manipulation of collections.

In either case, annotate the collection using the `@Disabled` annotation.

For example:

```
public class Order {
    private List<Order> cancelledOrders = new ArrayList<Order>();
    @Disabled
    public List<Order> getCancelledOrders() { ... }
    private void setCancelledOrders(List<Order> cancelledOrders) { ... }
    ...
}
```

Disabling a collection based on the persistence state of the object

As a refinement of the above, a collection may be optionally disabled using the `@Disabled` annotation based on the persistence state of the object:

- to disable the collection when the object is transient, use `@Disabled(When.UNTIL_PERSISTED)`
- to disable the collection when the object is persistent, use `@Disabled(When.ONCE_PERSISTED)`

Disabling a collection under certain conditions

A `disableXxx()` method can be used to disable a particular instance's collection under certain conditions:

The syntax is:

```
public String disableCollectionName()
```

For example:

```
public class Department {
    public List<Employee> getEmployees() { ... }
    private void setEmployees(List<Employee> employees) { ... }
    public void disableEmployees() {
        return isClosed()? "This department is closed" : null;
    }
    ...
}
```

Disabling a collection for specific users or roles

It is possible to disable collections for certain users/roles by calling the `DoymainObjectContainer#getUser()` method. See the section called “Hiding, disabling or validating for specific users or roles” for further discussion.

How to validate an object being added or removed

A `validateAddToXxx()` method can be used to check that an object is valid to be added to a collection. Conversely, the `validateRemoveFromXxx()` method can be used to check that it is valid to remove an object from a collection is valid.

The syntax is:

```
public String validateAddToCollectionName(EntityType param)
```

and

```
public String validateRemoveFromCollectionName(EntityType param)
```

A non-`null` return `String` indicates the reason why the object cannot be added/removed, and the viewing mechanism will display this to the user.

For example:

```
public class Department {
    public List<Employee> getEmployees() { ... }
    private void setEmployees(List<Employee> employees) { ... }
    public String validateAddToEmployee(Employee employee) {
        return employee.isRetired()?
            "Cannot add retired employees to department"
            :null;
    ...
}
```

How to trigger other behaviour when an object is added or removed

A collection may have a corresponding `addToXxx()` and/or `removeFromXxx()` method. If present, and direct manipulation of the contents of the connection has not been disabled (see the section called “How to prevent a collection from being modified”), then they will be called (instead of adding/removing an object directly to the collection returned by the accessor).

The reason for this behaviour is to allow other behaviour to be triggered when the contents of the collection is altered. That is, it is directly equivalent to the supporting `modifyXxx()` and `clearXxx()` methods for properties (see the section called “How to trigger other behaviour when a property is changed”).

The syntax is:

```
public void addTo<CollectionName>(EntityType param)
```

and

```
public void removeFromCollectionName(EntityType param)
```

where `EntityType` is the same type as the generic collection type.

For example:

```
public class Employee { ... }

public class Department {
    private List<Employee> employees = new ArrayList<Employee>();
    public List <Employee> getEmployees() {
        return employees;
    }
    private void setEmployees(List<Employee> employees) {
        this.employees = employees;
    }
    public void addToEmployees(Employee employee) {
        numMaleEmployees += countOneMale(employee);
        numFemaleEmployees += countOneFemale(employee);
        employees.add(employee);
    }
    public void removeFromEmployees(Employee employee) {
        numMaleEmployees -= countOneMale(employee);
        numFemaleEmployees -= countOneFemale(employee);
        employees.remove(employee);
    }
    private int countOneMale(Employee employee) { return employee.isMale()?1:0; }
    private int countOneFemale(Employee employee) { return employee.isFemale()?1:0

    ...
}
```

How to maintain bidirectional relationships

The recommended way of maintaining a bidirectional relationship is to use the 'mutual registration pattern', a write-up of which can be found at http://www.two-sdg.demon.co.uk/curbralan/papers/MutualRegistration.pdf. The general idea is that one side of the relationship is responsible for maintaining the associations, while the other side simply delegates.

To implement this in *Isis* for a 1:m relationship, use the `addToXxx()` / `removeFromXxx()` and `modifyXxx()` / `clearXxx()` methods.

For example:

```
public class Department {
    private List<Employee> employees = new ArrayList<Employee>();
    public List<Employee> getEmployees() { ... }
    private void setEmployees(List<Employee> employees) { ... }
    public void addToEmployees(Employee e) {
        if(e == null || employees.contains(e)) return;
        e.setDepartment(this);
        employees.add(e);
```

```
    }
    public void removeFromEmployees(Employee e) {
        if(e == null || !employees.contains(e)) return;
        e.setDepartment(null);
        employees.remove(e);
    }
    ...
}
```

and

```
public class Employee {
    private Department department;
    public Department getDepartment() { ... }
    private void setDepartment(Department department) { ... }
    public void modifyDepartment(Department d) {
        if(d==null || department==d) return;
        if(department != null) {
            department.removeFromEmployees(this);
        }
        d.addToEmployees(this);
    }
    public void clearDepartment() {
        if(department==null) return;
        department.removeFromEmployees(this);
    }
    ...
}
```

Chapter 7. Domain Entity Actions

How-to's relating to a domain entity's actions.

An 'action' is a method that we expect the user to be able to invoke on a domain entity via the user interface, though it may also be invoked programmatically within the object model. The following conventions are used to determine when and how methods are made available to the user as actions, and the means by which a user can interact with those actions.

How to add an action to an object

By default, any `public` instance method that you add to a class will be treated as a user action, unless it represents a property, collection, or another reserved method defined in this guide.

The syntax is:

```
public void actionName([ValueOrEntityType param] ...)
```

or

```
public ReturnType actionName([ValueOrEntityType param] ...)
```

When a method returns a reference the viewer will attempt to display that object. If the return value is `null` then nothing is displayed.

We refer to all methods that are intended to be invoked by users as 'action methods'.

If you have a method that you don't want to be made available as a user-action you can either:

- make it non-`public` (eg `protected` or `private`)
- annotate it with `@Ignore`
- annotate it with `@Hidden` (discussed further in the section called “How to hide an action”)

Note also that `static` methods are ignored: such functionality should reside in a service, such as a repository or factory (see Chapter 9, *Domain Services, Repositories and Factories*).

How to specify names and/or description for an action

Specifying the name for an action

By default the framework will use the action name itself to label the menu item on the user interface. If you wish to override this, use the `@Named` annotation on the action.

For example:

```
public class Customer {
    @Named("Place Order")
```

```
    public void createOrder() { ... }
    ...
}
```

Specifying a description for a collection

An additional description can be provided on an action using the `@DescribedAs` annotation. The framework will take responsibility to make this description available to the user, for example in the form of a tooltip.

For example:

```
public class Customer {
    @DescribedAs("Places an order, causing a shipping note "+
                 "to be generated and invoice to be dispatched")
    public void createOrder() { ... }
    ...
}
```

How to specify the order in which actions appear on the menu

The `@MemberOrder` annotation provides a hint to the viewer as to the order in which the actions should be displayed, eg in a menu.

For example:

```
public class Customer {
    @MemberOrder("3")
    public void placeOrder(Product p) { ... }
    ...

    @MemberOrder("4")
    public void blackList() { ... }
    ...
}
```

How to hide an action

The mechanism for hiding an action is broadly the same as for hiding a property (see the section called “How to hide a property from the user”) or a collection (see the section called “How to hide a collection”).

Hiding an action permanently

To prevent a user from viewing an action at all, use the `@Hidden` annotation. This is generally used where a `public` method on an object is not intended to be a user action

For example:

```
public class Order {
    @Hidden
```

```
    public void markAsCancelled() { ... }
    ...
}
```

Hiding an action based on the persistence state of the object

As a refinement of the above, an action may be optionally hidden using the `@Hidden` annotation based on the persistence state of the object:

- to hide the action when the object is transient, use `@Hidden(When.UNTIL_PERSISTED)`
- to hide the action when the object is persistent, use `@Hidden(When.ONCE_PERSISTED)`

Hiding an action under certain conditions

A `hideXxx()` method can be used to indicate that a particular object's action should be hidden under certain conditions, typically relating to the state of that instance.

The syntax is:

```
public boolean hideActionName([ValueOrEntityType param] ...)
```

where the parameter types should match the action itself (allowing for overloading)

or

```
public boolean hideActionName()
```

which applies to (all overloaded versions of) the action.

in both cases, returning a value of `true` indicates that the action should be hidden.

For example:

```
public class Order {
    public void applyDiscount(int percentage) { ... }
    public boolean hideApplyDiscount() {
        return isWholesaleOrder();
    }
}
```

Hiding an action for specific users or roles

It is possible to hide actions for certain users/roles by calling the `DomainObjectContainer#getUser()` method. See the section called “Hiding, disabling or validating for specific users or roles” for further discussion.

How to prevent an action from being invoked

Preventing the user from invoking an action is known as 'disabling' the action.

The mechanism for disabling an action is broadly the same as for disabling a property (see the section called “How to prevent a property from being modified”) or a collection (see the section called “How to prevent a collection from being modified”).

Disabling an action permanently

It is possible to prevent an action from ever being invoked using the `@Disabled` annotation, exactly equivalent to the use of the annotation for properties and collections. However, it's not a particularly meaningful usecase: why display an action that can never be invoked? The only reason we can think of is as a placeholder during prototyping - to indicate to the user that an action is planned, but has not yet been implemented.

Disabling an action based on the persistence state of the object

Whereas annotating an action simply as `@Disabled` probably does not make sense (see above), it does make sense to optionally disable an action using the `@Disabled` annotation based on the persistence state of the object:

- to disable the action when the object is transient, use `@Disabled(When.UNTIL_PERSISTED)`
- to disable the action when the object is persistent, use `@Disabled(When.ONCE_PERSISTED)`

Based on the state of the object

There may be circumstances in which we do not want the user to be able to initiate the action at all - for example because that action is not appropriate to the current state of the object on which the action resides. Such rules are enforced by means of a `disableXxx()` supporting method.

The syntax is:

```
public String disableActionName([ValueOrEntityType param]...)
```

A non-`null` return `String` indicates the reason why the action may not be invoked. The framework takes responsibility to provide this feedback to the user.

For example:

```
public class Customer {
    public Order placeOrder(Product p, int quantity) { ... }
    public String disablePlaceOrder(Product p, int quantity) {
        return isBlackListed()?
            "Blacklisted customers cannot place orders"
            :null;
    }
    ...
}
```

Disabling an action for certain users or roles

It is possible to disable actions for certain users/roles by calling the `DomainObjectContainer#getUser()` method. See the section called “Hiding, disabling or validating for specific users or roles” for further discussion.

How to specify names and/or descriptions for an action parameter

Unlike with properties, the framework cannot pick up the names of parameters that you use within the domain code. By default parameters will be labelled only with the type of the object required (e.g. 'String:' or 'Customer:)

If you want a parameter to have a different name (such as 'First Name', 'Last Name') then that parameter should be marked up with an `@Named` annotation - very often taking the same form as the parameter name used in the code. Alternatively though, you could create a user-defined value type, using `@Value` (see Chapter 10, *Value Types*).

Similarly, any parameter may be given a short user-description using the `@DescribedAs` annotation. The framework takes responsibility to make this available to the user.

For example:

```
public class Customer {
    public Order placeOrder(
            Product p,
            @Named("Quantity")
            @DescribedAs("The number of units of the specified product in this ord
            int quantity) {
        ...
    }
    ...
}
```

How to specify the size of `String` action parameters

As for properties (see the section called “How to specify the size of `String` properties”), use:

- the `@MaxLength` to specify the maximum number of characters that may be stored within a `String` parameter.
- the `@TypicalLength` to specify the typical number of characters that are likely to be stored within a `String` parameter. Viewers are expected to use this as a hint as to the size of the field to render for the parameter.
- the `@MultiLine` annotation as a hint to indicate that the parameter should be displayed over multiple lines (eg as a text area rather than a text field).

For example:

```
public class TicketRaiser {

    public void raiseTicket(
            @TypicalLength(50) @MaxLength(255) @Named("Description")
            String getDescription,
            @MaxLength(2048) @MultiLine @Named("Notes")
```

```
            String notes) {
        ...
    }
    ...
}
```

How to make an action parameter optional

By default, the framework assumes that when an action method is to be invoked, all the parameters are mandatory. You may over-ride this behaviour by marking up one or more of the paramaters with the `@Optional` annotation.

How to validate an action parameter argument

Declarative validation

For parameters that accept a text input string, such as `String` and `Date`, there are convenient mechanisms to validate and/or normalise the values typed in:

- For `Date` and number values the `@Mask` annotation may be used.
- For `String` parameters the `@RegEx` annotation may be used.

More complex validation can also be performed imperatively (below).

Imperative validation

A `validateXxx()` method is used to check that the set of arguments used by an action method is valid. If the arguments are invalid then the framework will not invoke the action.

The syntax is:

```
public String validate<ActionName>([ValueOrEntityType param]...)
```

A non-`null` return `String` indicates the reason why the member cannot be modified/action be invoked, and the viewing mechanism will display this feedback to the user.

For example:

```
public class Customer {
    public Order placeOrder(Product p, int quantity) { ... }
    public String validatePlaceOrder(Product p, int quantity) {
        if (p.isOutOfStock()) { return "Product is out of stock"; }
        if (quantity <= 0) { return "Quantity must be a positive value"; }
        return null;
    }
    ...
}
```

For complex validation, you may wish to use the `org.apache.isis.applib.util.ReasonBuffer` helper class.

How to specify default values for an action parameter

When an action is about to be invoked, then default values can be provided for any or all of its parameters.

There are two different ways to specify parameters; either per parameter, or for all parameters.

Per-parameter syntax (preferred)

The per-parameter form is simpler and generally to be preferred.

The syntax is:

```
public ParameterType defaultNActionName()
```

where N indicates the 0-based parameter number. For example:

```
public class Customer {
    public Order placeOrder(
             Product product,
             @Named("Quantity")
             int quantity) {
        ...
    }
    public Product default0PlaceOrder() {
        return productMostRecentlyOrderedBy(this.getCustomer());
    }
    public int default1PlaceOrder() {
        return 1;
    }
}
```

All parameters syntax

The syntax for specifying all the parameter default values in one go is:

```
public Object[] defaultActionName([ValueOrEntityType param]...)
```

returning an array which must have one element per parameter in the action method signature of corresponding default values.

For example:

```
public class Customer {
    public Order placeOrder(
            Product product,
            @Named("Quantity")
            int quantity) {
        ...
    }
    public Object[] defaultPlaceOrder(
            Product product,
            int quantity) {
```

```
        return new Object[] {
            productMostRecentlyOrderedBy(this.getCustomer()),
            1
        };
    }
    ...
}
```

How to specify a set of choices for an action parameter

The programmer may provide a set of choices for the value of any or all of the parameters of an action. These will be made available to the user - for example as a drop-down list.

If the type of the parameter is annotated with `@Bounded`, then it is not necessary to specify the choices for that parameter, as the user will automatically be offered the full set to choose from.

If this isn't the case, then - as for defaults - there are two different ways to specify parameters; either per parameter, or for all parameters.

Per parameter syntax (preferred)

The per-parameter form is simpler and generally preferred.

The syntax is:

```
public List<ParameterType> choicesNActionName()
```

where N indicates the 0-based parameter number.

For example:

```
public class Order {
    public Order placeOrder(
            Product product,
            @Named("Quantity")
            int quantity) {
        ...
    }
    public List<Product> choices0PlaceOrder() {
        return lastFiveProductsOrderedBy(this.getCustomer());
    }
    public List<Integer> choices1PlaceOrder() {
        return Arrays.asList(1,2,3,4,5);
    }
    ....
}
```

All parameters syntax

The alternative mechanism is to supply all the parameter choices in one go:

```
public Object[] choices<ActionName>([<parameter type> param]...)
```

returning an array, which must have one element per parameter in the method signature. Each element of the array should itself either be an array or a list, containing the set of values representing the choices for that parameter, or `null` if there are no choices to be specified for that parameter.

For example:

```
public class Order {
    public Order placeOrder(
            Product product,
            @Named("Quantity")
            int quantity) {
        ...
    }
    public Object[] choicesPlaceOrder(
            Product product,
            int quantity) {
        return new Object[] {
            lastFiveProductsOrderedBy(this.getCustomer()).toArray(),
            Arrays.asList(1,2,3,4,5)
        };
    }
    ...
}
```

Chapter 8. Further Business Rule How-Tos

Further validation how-to's that apply across all class members

This chapter has some additional recipes/how-tos relating to implementing business rules. They apply across all class members.

@MustSatisfy Specification

The `@MustSatisfy` annotation is an alternative to using imperative validation, allowing validation rules to be captured in an (implementation of a) `org.apache.isis.applib.spec.Specification`.

For example:

```
public class DomainObjectWithMustSatisfyAnnotations extends AbstractDomainObject {

    private String lastName;
    @MustSatisfy(SpecificationRequiresFirstLetterToBeUpperCase.class)
    public String getLastName() {
        resolve(lastName);
        return lastName;
    }
    public void setLastName(String lastName) {
        this.lastName = lastName;
        objectChanged();
    }

    public void changeLastName(
            @MustSatisfy(SpecificationRequiresFirstLetterToBeUpperCase.class)
            String lastName
            ) {
        setLastName(lastName);
    }

}
```

To help you write your own `Specification`s, there are some adapter classes in org.apache.isis.applib.specs:

- `AbstractSpecification`, which implements `Specification` and takes inspiration from the Hamcrest [http://code.google.com/p/hamcrest/] library's `TypeSafeMatcher` class
- `SpecificationAnd`, which allows a set of `Specification`s to be grouped together and require that *all* of them are satisfied
- `SpecificationOr`, which allows a set of `Specification`s to be grouped together and require that *at least one* of them is satisfied
- `SpecificationNot`, which requires that the provided `Specification` is *not* satisfied

Hiding, disabling or validating for specific users or roles

Generally it is not good practice to embed knowledge of roles and/or users into the domain classes; instead, this should be the responsibility of the framework or platform and should be specified and administered externally to the domain model. However, in rare circumstances it might be necessary or pragmatic to implement access control within the domain model.

The current user can be obtained from `DomainObjectContainer`, using its `getUser()` method. Alternatively, if the domain object inherits from `AbstractDomainObject`, then `getUser()` is also inherited. In either case the method returns an object of type `org.apache.isis.security.UserMemento`, which holds both username and the set of roles for that user. The full details of the security classes can be found in Appendix D, *Security Classes*.

The mechanism to apply a business rule is just to return an appropriate value from a supporting `hideXxx()`, `disableXxx()` or `validateXxx()` method.

For example, the following requires that the MODIFY_SALARY role is assigned to the current user in order to update a salary property beyond a certain value:

```
public class Employee extends AbstractDomainObject {
    public BigDecimal getSalary() { ... }
    public void setSalary(BigDecimal salary) { ... }
    public String validateSalary() {
        return salary.doubleValue() >= 30000 &&
              !getUser().hasRole("MODIFY_SALARY")?
              "Need MODIFY_SALARY role to increase salary above 30000": null;
    }
}
```

How to pass a messages and errors back to the user

Sometimes, within an action it is necessary or desirable to pass a message to the user, for example to inform them of the results of their action ('5 payments have been issued') or that the action was not successful ('No Customer found with name John Smith').

`DomainObjectContainer` defines three methods for this purpose:

- `informUser(String message)`

  Inform the user of some event. The user should not be expected to acknowledge the message; typically the viewer will display the message for a period of time in a non-modal notification window.

- `warnUser(String message)`

  Warn the user of some event. Because this is more serious, the viewer should require the user to acknowledge the message.

- `raiseError(String message)`

Indicate that a serious application error has occurred. The viewer should again require the user to acknowledge the message, and quite possibly indicate further steps that the user should perform (eg notify the help desk).

The precise mechanics of how each of these messages is rendered visible to the user is determined by the viewer being used.

Chapter 9. Domain Services, Repositories and Factories

How-to's relating to writing services, repositories and factories.

This chapter contains how-to's for programming conventions that writing domain services (by which we also mean repositories and factories); ie everything that isn't a domain object or a value type.

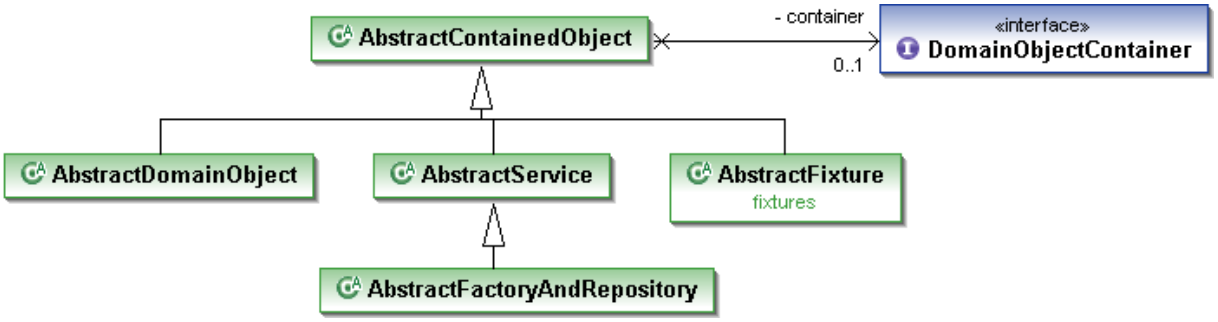
Domain services are instantiated once and once only by the framework, and are used to centralize any domain logic that does not logically belong in a domain entity or value. *Isis* will automatically inject services into every domain entity that requests them, and into each other.

How to not inherit from framework classes

Like entities, it isn't mandatory for domain services to inherit from any framework superclass; they can be plain-old pojos if required. However, again, like entities, they do at a minimum need to have a `org.apache.isis.applib.DomainObjectContainer` injected into them (an interface), from which other framework services can be accessed.

If you don't have a requirement to inherit from any other superclass, then it usually makes sense to inherit from one of the abstract classes in the applib, either `org.apache.isis.applib.AbstractService` or `org.apache.isis.applib.AbstractRepositoryAndFactory`. These already supports the `DomainObjectContainer` and have a number of convenience helper methods.

The UML class diagram below shows the relationship between these types and the `DomainObjectContainer`.



What this means is that *Apache Isis* treats factories and repositories as just another type of domain service.

How to register domain services, repositories and factories

All domain services (which includes repositories and factories) should be registered in the isis.properties configuration file, under the `isis.services.prefix` (a common package name) and `isis.services` key (a comma-separated list).

For example:

```
isis.services.prefix = org.apache.isis.support.prototype.objstore.dflt
```

```
isis.services = employee.EmployeeRepositoryDefault, claim.ClaimRepositoryDefault
```

This will instantiate a single instance of each of the two services listed.

How to inject services into entities

All that is required to inject a service into an entity (or indeed into another service) is to provide an appropriately typed setter. The name of the method does not matter, only that it is prefixed "set", is public, and has a single parameter of the correct type.

For example:

```
public class Customer {
    private OrderRepository orderRepository;
    public void setOrderRepository(OrderRepository orderRepository) {
        this.orderRepository = orderRepository;
    }
    ...
}
```

Note that we consider `DomainObjectContainer` to be a service too; hence it can be injected in exactly the same manner.

How to write a typical domain service

Services consist of a set of logically grouped actions, and as such follow the same conventions as for entities (see Chapter 7, *Domain Entity Actions*). However, a service cannot have (persisted) properties, nor can it have (persisted) collections.

For convenience you can inherit from `AbstractService` or one of its subclasses (see the section called “How to not inherit from framework classes”), but this is not mandatory.

The `getId()` method

Optionally, a service may provide a `getId()` method:

```
public String getId()
```

This method returns a logical identifier for a service, independent of its implementation. Currently it used only by perspectives, providing a label by which to record the services that are available for a current user's profile. See Chapter 12, *Profiles* for more about profiles and perspectives.

(Suppressing) contributed actions

Any n-parameter action provided by a service will automatically be contributed to the list of actions for each of its (entity) parameters. From the viewpoint of the entity the action is called a contributed action.

For example, given a service:

```
public interface Library {
    public Loan borrow(Loanable l, Borrower b);
}
```

and the entities:

```
public class Book implements Loanable { ... }y
```

and

```
public class LibraryMember implements Borrower { ... }
```

then the `borrow(...)` action will be contributed to both `Book` and to `LibraryMember`.

This is an important capability because it helps to decouple the concrete classes from the services.

If necessary, though, this behaviour can be suppressed by annotating the service action with `@org.apache.isis.applib.annotations.NotContributed`.

For example:

```
public interface Library {
    @NotContributed
    public Loan borrow(Loanable l, Borrower b);
}
```

If annotated at the interface level, then the annotation will be inherited by every concrete class. Alternatively the annotation can be applied at the implementation class level and only apply to that particular implementation.

Note that an action annotated as being `@NotContributed` will still appear in the service menu for the service. If an action should neither be contributed nor appear in service menu items, then simply annotate it as `@Hidden`.

(Suppressing) service menu items

By default every action of a service (by which we also mean repositories and factories) will be rendered in the viewer, eg as a menu item for that service menu. This behaviour can be suppressed by annotating the action using `@org.apache.isis.applib.annotations.NotInServiceMenu`.

For example:

```
public interface Library {
    @NotInServiceMenu
    public Loan borrow(Loanable l, Borrower b);
}
```

Note that an action annotated as being `@NotInServiceMenu` will still be contributed. If an action should neither be contributed nor appear in service menu items, then simply annotate it as `@Hidden`.

(Suppressing) service menus

If none of the service menu items should appear, then the service itself should be annotated as @Hidden.

For example:

```
@Hidden
public interface EmailService {
    public void sendEmail(String to, String from, String subject, String body);
    public void forwardEmail(String to, String from, String subject, String body);
}
```

How to use a generic repository

To speed up initial prototype the framework allows so-called generic repositories to be defined, one per entity. Such a repository will, for its specified type, provide methods to:

- Create a new transient instance
- Create a new persisted instance
- Find all persisted instances
- Find instances with a specified title

To register such a service prefix the class name with the prefix `repository#`.

For example:

```
isis.services = repository#dom.Booking
```

Over time, you should expect most if not all of these generic repositories will be replaced with regular repository types (see the section called “How to write a custom repository”).

How to write a custom repository

Repositories are defined as interfaces within the domain, and their implementation will vary by object store. During prototyping and for much of development, you will probably find it easiest to use an in-memory object store or perhaps the XML object store, with only a small number of instances. The `DomainObjectContainer` provides a set of methods that make it easy to pull back all instances from the object store which can then be filtered as required. Later on, you can replace the implementation depending upon the specifics of the object store that you'll be using for production.

If you inherit from the `org.apache.isis.applib.AbstractFactoryAndRepository` adapter class then this will automatically have the `DomainObjectContainer` injected, and provides convenience methods that delegate to the container. Using this is not mandatory, however.

The methods provided by the `DomainObjectContainer` to support repositories are:

- `allInstances(Class<T> ofType)`

  Returns all instances of the specified type. Note that this includes all instances of any subtypes.

- `allMatches(...)`

  Returns all instances matching the provided arguments.

- `firstMatch(...)`

  Returns the first instance matching the provided arguments.

- `uniqueMatch(...)`

  Returns the one-and-only instance matching the provided arguments (else is an exception).

The last three methods, `*Match(...)` are all overloaded in order to return a subset of object instances. Some of these are "naive"; all instances are returned from the object store, and the filtering is performed

within the repository. Others are designed to pass the query predicate back to the object store so that only the matching rows are returned.

Each of these options are discussed in more detail below.

Finding by Title

The first version of finding instances is to specify the required title for the matching objects:

- `allMatches(Class<T> ofType, String title)`
- `firstMatch(Class<T> ofType, String title)`
- `uniqueMatch(Class<T> ofType, String title)`

Although easy and intuitive, this isn't generally recommended for production use because (a) the matching is performed within the repository rather than the object store, and (b) the title string can often change as business requirements are refined.

That said, it is possible to eliminate the first disadvantage by using the `Query` API, discussed below; this provides an implementation that is equivalent to find by title.

Finding by Pattern

The next technique of finding instances is to specify pattern object to match against (sometimes called "query-by-example", or QBE):

- `allMatches(Class<T> ofType, Object pattern)`
- `firstMatch(Class<T> ofType, Object pattern)`
- `uniqueMatch(Class<T> ofType, Object pattern)`

Any non-null value of the pattern object is used as the predicate.

Although more robust that searching by title, this technique is also not likely to be valid for production code because the matching is still performed within the repository rather than within the object store.

That said, it is possible to eliminate the first disadvantage by using the `Query` API, discussed below; this provides an implementation that is equivalent to find by pattern.

Note

If the pattern object is created using `newTransientInstance(...)`, then any default values for properties will automatically be set (see the section called “How to set up the initial value of a property programmatically”). If this isn't required, they will need to be manually cleared.

Finding using the `Filter` API

The third overloaded version of the matching methods to find instances all take an `org.apache.isis.applib.Filter<T>` instance:

- `allMatches(Class<T> ofType, Filter<T> filter)`
- `firstMatch(Class<T> ofType, Filter<T> filter)`

- `uniqueMatch(Class<T> ofType, Filter<T> filter)`

The `Filter<T>` interface is very straightforward:

```
public interface Filter<T> {
    public boolean accept(T obj);
}
```

Every object of the given type (and subclasses) is passed into the `Filter` instance; only those `accept()`'ed are returned from the `*Match()` method.

Although flexible, with this technique the matching is also performed within the repository rather than the object store, and so is also likely not to be suitable for production use where there are many instances of the given type.

Finding using the `Query` API

The finaly overloaded version of the matching methods take an instance of `org.apache.isis.applib.query.Query<T>` interface:

- `allMatches(Query<T> query)`
- `firstMatch(Query<T> query)`
- `uniqueMatch(Query<T> query)`

Unlike all the other matching mechanisms, the point of the `Query` interface is for it to be passed back to the object store and evaluated there.

The applib provides several implementations that implement the `Query<T>` interface. Probably the most important of these is `QueryDefault<T>`, which provides a set of factory methods for constructing `Query` instances that represent a named query with a map of parameter/argument pairs.

For example:

```
public class CustomerRepositoryImpl implements CustomerRepository {
    public List<Customer> findCustomers(
            @Named("Last Name") String lastName,
            @Named("Postcode")  String postCode
        ) {
        QueryDefault<Customer> query =
            QueryDefault.create(
                Customer.class,
                "findCustomers",
                "lastName", lastName,
                "postCode", postCode);

        return getContainer().allMatches(query);
    }
    ...
}
```

Above it was noted that the other overloaded versions of the matching API have the disadvantage that the matching is performed within the repository. As an alternative to using "find by title" or "find by pattern", you may wish to use QueryFindByTitle and QueryFind

- `QueryFindByTitle<T>`, which corresponds to the `allMatches(...)` for searching by title
- `QueryFindByPattern<T>`, which corresponds to the `allMatches(...)` for searching by pattern

There is also a `QueryFindAllInstances<T>`, which corresponds to the `allInstances()` method.

The interpretation of a `Query` instance ultimately depends on the object store. All object stores will support `QueryFindAllInstances`, and most will provide a mechanism to support `QueryDefault`. Check the object store documentation to determine whether they support other `Query` implementations (ie, `QueryFindByTitle` and `QueryFindByPattern`).

Factories

Like repositories, factories are defined by interface in the domain, decoupling the domain objects from their actual implementation. Unlike repositories, there is no particular need to change the implementation when moving from one object store to another, because in all cases the factory can simply delegate to its injected `DomainObjectContainer`.

The methods for `DomainObjectContainer` that are relevant for a factory are:

- `<T> T newTransientInstance(final Class<T> ofClass)`
- `<T> T newPersistentInstance(final Class<T> ofClass)`
- `persist(Object)`

These are discussed in more detail in the section called “How to create or delete objects within your code”. See also Appendix C, *`DomainObjectContainer` interface* for full coverage of the methods available in `DomainObjectContainer`.

Chapter 10. Value Types

Built-in value types, writing your own value types, and supporting third-party value types.

The state of any given entity is characterized by properties (Chapter 5, *Domain Entity Properties*) and collections (Chapter 6, *Domain Entity Collections*). A collections is a one-to-many reference to another entities, while a property is either a one-to-one reference to another entity, or it is a value.

But what's a value? Well, it's an atomic piece of state. A string is a value, so is a number, so is a date. Values should be designed to be immutable (though some system value types, such as `java.util.Date`, famously are not).

Isis supports all the standard JDK value types, and defines a number of its own (eg `Percentage` and `Color`). It also allows you to define your own value types, such as `LastName`, or `Celsius`, or `ComplexNumber`.

Finally, it's also possible to make Isis integrate with third-party value types, such as JodaTime [http://joda-time.sourceforge.net/].

Note

Isis' support for a particular value type does not necessarily imply that there is a custom widget for that type in a particular viewer. Rather, it means that the state of the object can be serialized, is expected to have equal-by-content semantics, and is expected to be immutable. It may also be parseable from a string.

Built-in Value Types

The following are the value types supported by *Isis* out-of-the-box.

JDK Types

The following JDK types are supported by *Isis*.

Primitive Types

All the primitive types may be used as values: `byte`, `short`, `int`, `long`, `float`, `double`, `char`, and `boolean`.

Wrapper Types

The wrapper types for each of the primitives can also be used as value types: `java.lang.Byte`, `java.lang.Short`, `java.lang.Integer`, `java.lang.Long`, `java.lang.Float`, `java.lang.Double`, `java.lang.Character`, `java.lang.Boolean`.

Java Classes

The following java classes have value semantics and may be used as value types:

- `java.lang.String`
- `java.math.BigInteger` and `java.math.BigDecimal`

- `java.util.Date` (date and time), `java.sql.Date` (date only), and `java.sql.Time` (time only)
- `java.sql.Timestamp`
- `java.awt.Image`

Isis AppLib

Isis itself also provides a number of its own value types. These are all in the `org.apache.applib.value` package:

- `Color`
- `Date` (date only), `DateTime` (date and time) and `Time` (time only)
- `TimeStamp`
- `Image`
- `Money`
- `Password`
- `Percentage`

Value formats

Isis provides default formats for the inbuilt value types, according to type. These can be modified using `isis.properties`.

These formats cut across the above categories; for example the byte format relates to both `byte` (primitive) and `java.lang.Byte` (wrapper). In all cases this setting can be overriden for a specific field using the @Mask annotation (see the section called “@Mask”).

Byte format

The format for all bytes can be set, replacing the default format derived from the system, using the following property to specify a mask:

```
isis.value.format.byte=####
```

The mask is used to set up a `java.text.DecimalFormat` formatting object so details of the mask format can be found in the Java documentation.

Date Format

The format for all dates can be set, replacing the default format derived from the system, using the following property to specify one of *long*, *medium*, *short*, *isolong*, *isoshort* or a mask:

```
isis.value.format.date=dd/MM/yy
```

When a mask is specified it is used to set up a `java.text.SimpleDateFormat` formatting object so details of the mask format can be found in the Java documentation.

Date/time Format

The format for all date/time values can be set, replacing the default format derived from the system, using the following property to specify one of *long*, *medium*, *short*, *isolong*, *isoshort* or a mask:

```
isis.value.format.datetime=dd/MM/yy
```

When a mask is specified it is used to set up a `java.text.SimpleDateFormat` formatting object so details of the mask format can be found in the Java documentation.

Decimal format

The format for `BigDecimal` values can be set, replacing the default format derived from the system, using the following property to specify a mask:

```
isis.value.format.decimal=####
```

The mask is used to set up a `java.text.DecimalFormat` formatting object so details of the mask format can be found in the Java documentation.

Double format

The format for all double values can be set, replacing the default format derived from the system, using the following property to specify a mask:

```
isis.value.format.double=####
```

The mask is used to set up a `java.text.DecimalFormat` formatting object so details of the mask format can be found in the Java documentation.

Float format

The format for all float values can be set, replacing the default format derived from the system, using the following property to specify a mask:

```
isis.value.format.float=####
```

The mask is used to set up a `java.text.DecimalFormat` formatting object so details of the mask format can be found in the Java documentation.

Integer format

The format for all integers (including `BigInteger`) can be set, replacing the default format derived from the system, using the following property to specify a mask:

```
isis.value.format.int=####
```

The mask is used to set up a `java.text.DecimalFormat` formatting object so details of the mask format can be found in the Java documentation.

Long format

The format for all long values can be set, replacing the default format derived from the system, using the following property to specify a mask:

```
isis.value.format.long=####
```

The mask is used to set up a `java.text.DecimalFormat` formatting object so details of the mask format can be found in the Java documentation.

Short format

The format for all short values can be set, replacing the default format derived from the system, using the following property to specify a mask:

```
isis.value.format.short=####
```

The mask is used to set up a `java.text.DecimalFormat` formatting object so details of the mask format can be found in the Java documentation.

Time Format

The format for all time values can be set, replacing the default format derived from the system, using the following property to specify one of *long*, *medium*, *short*, *isolong*, *isoshort* or a mask:

```
isis.value.format.time=ddMMyyyy hhmm
```

When a mask is specified it is used to set up a `java.text.SimpleDateFormat` formatting object so details of the mask format can be found in the Java documentation.

Timestamp Format

The format for time stamp values can be set, replacing the default format derived from the system, using the following property to specify one of *long*, *medium*, *short*, *isolong*, *isoshort* or a mask:

```
isis.value.format.timestamp=hh:mm
```

When a mask is specified it is used to set up a `java.text.SimpleDateFormat` formatting object so details of the mask format can be found in the Java documentation.

Custom Value Types

In addition to the built-in value types it is also possible to define user-defined value types. This is typically done using the `@Value` annotation.

The `@Value` annotation is used to provide an implementation of the `org.apache.isis.applib.adapters.ValueSemanticsProvider` interface. In turn this provides objects that allow the framework to interact with the value, specifically:

- the `EncoderDecoder` is used to convert the value into and back out of serializable form

  This is used by some object stores (eg the XML Object Store), for remoting and also by the XML Snapshot capability (see Chapter 14, *XML Snapshots*);

- the `Parser` is used to convert `String`s into the value type

  This is used as a fallback by viewers that do not have any specific widgets to support the particular value type, and make do with a simple text field instead.

  An obvious example is to parse a date. But it could be used to parse "TRUE" and "FALSE" into a boolean (as opposed to using a checkbox).

- the `DefaultsProvider` is used to provide a meaningful default for the value

  Not every value type will have a default, but some do (eg false for a boolean, 0 for a number). This is used as the default value for non-`@Optional` properties and parameters.

Each of these interfaces also reside in `org.apache.isis.applib.adapters`.

For more details, explore the built-in types within the applib, for example `org.apache.isis.applib.value.Money`.

```
@Value(semanticsProviderName =
         "org.apache.isis.core.progmodel.facets.value.MoneyValueSemanticsProvider"
public class Money extends Magnitude {
    ...
}
```

where `MoneyValueSemanticsProvider` is the implementation of `ValueSemanticsProvider` described above.

Note

Using value types generally removes the need for using `@MustSatisfy` annotation (see the section called "@MustSatisfy Specification"); the rules can instead move down into a `validate()` method on the value type itself.

Third-party Value Types

Third party value types, such as those in JodaTime [http://joda-time.sourceforge.net/], can also supported, again through the use of a `ValueSemanticsProvider`. However, since the source code cannot be altered, the provider must be supplied using a key value in `isis.properties` configuration file.

For example, the following would register a semantics provider for `org.jodatime.time.DateTime`:

```
isis.core.progmodel.value.org.jodatime.time.DateTime.semanticsProviderName=\
    com.mycompany.values.CalendarIntervalValueSemanticsProvider
```

Note

At the time of writing *Apache Isis* does not currently ship with any `ValueSemanticsProvider`s, though doing so is on the roadmap.

Part III. Supporting Features

Table of Contents

Chapter 11. Clock

The default Clock and alternative implementations.

Many if not all enterprise applications deal with dates and times in one way or another. For example, if an `Order` is placed, then the `Customer` may have 30 days to pay the Invoice, otherwise a penalty may be levied. However, this can complicate automated testing: "today+30" will be a different date every time the test is run.

A common solution is to require that domain objects do not go directly to the system for the current date (ie don't simply instantiate a new `java.util.Date` in order to get the current time); instead they should call some sort of facade.

The *Apache Isis* framework provides such a facade through the `org.apache.isis.applib.clock.Clock` class. The defaults for all values refer back to the `Clock`, and - because the `Clock` is a singleton - it is easy for any application code to obtain the current time also.

For example:

```
public class Customer {
     public Order placeOrder(Product p) {
         Date now = Clock.getTimeAsDate();
         ...
     }
     ...
}
```

Lazily Instantiated

The first call to `Clock.getTime()` will lazily instantiate the singleton, with the default implementation being one that simply delegates to the system's internal clock. To use a different `Clock` implementation, eg one that delegates to an NNTP server, all that is required is to instantiate it any time prior to bootstrapping *Isis* itself.

One notable implementation that notably takes advantage of this is `FixtureClock`, used for testing. See Chapter 13, *Fixtures and SwitchUser* for more information.

Possibly Replaceable

Clock implementations can indicate whether they are replaceable as the singleton, or not.

Most (all?) production implementations (eg the default system clock) are *not* replaceable; once instantiated, any attempt to instantiate another subclass will be rejected with an exception.

However implementations to work with tests (such as `FixtureClock`, already mentioned) are more likely to be replaceable, so that they can be setup multiple times as required.

Chapter 12. Profiles

Support for user profiles

As well as allowing domain entities to be persisted into object stores, *Apache Isis* also allows user *profiles* to be persisted into a profile store.

Profiles and Perspectives

Every user can have one profile associated with them, which consists of:

- a set of options or preferences
- a set of perspectives

A perspective is something akin to a layout or desktop, representing a configuration of relevant objects as might be displayed on the home page of the viewer. In particular, perspectives allow the set of services available to a user (eg as icons in the DnD viewer) to be customized for that user. Since these services represent the "start points" for the user to interact with the domain model, they in a sense define an application on a per-user basis.

The elements that make up a perspective are:

- a set of services (identified by the string returned from each service's `getId()` method, see Chapter 9, *Domain Services, Repositories and Factories*)

  These are the services (eg icons) that can be accessed from that perspective

- a set of objects

  These might represent bookmarks to objects saved from a previous session.

The applib contains types to define these two concepts; specifically `Profile` and `Perspective`. These are actually interfaces, the implementation is provided by framework itself.

Runtime and Viewer Support

Support for profiles will vary across runtime implementations and across viewers. The *default runtime* implementation *does* support profiles, however, and works such that if a user logs in and no perspective exists for that user one will automatically be created. This will either be a copy of the 'template' perspective, or, if no such template exists, then simply a perspective containing all the known services.

Moreover, the runtime must be configured to use some sort of persistence mechanism for profiles such that they are persisted between runs. The default runtime does support this (though the default is the in-memory profilestore that does not persist any information between sessions).

Even if the configured runtime does support profiles, not every viewer necessarily uses the concept. The most notable viewer that *does* support the idea is the *dnd viewer*.

Chapter 13. Fixtures and SwitchUser

Using fixtures to setup the system, generally for testing purposes.

Fixtures are used to setup the framework into a known state. This is predominantly done for testing, and in particular when running with the in-memory object store. However, fixtures can also be used to specify the user that has "logged on", and to set the clock.

Whether a fixture has any effect will depend on several factors:

- the first is the `DeploymentType`; specifying the logged on user will only be honoured if running in exploration mode (or more precisely, where `DeploymentType#isExploring()` returns true);
- the second is the object store; most persistent object stores will either ignore fixtures, or only allow them to be installed once or if run with an (object store-specific) flag set

How to register fixtures

All domain services (which includes repositories and factories) should be registered in the *isis.properties* configuration file, under the `isis.fixtures.prefix` and `isis.fixtures` keys.

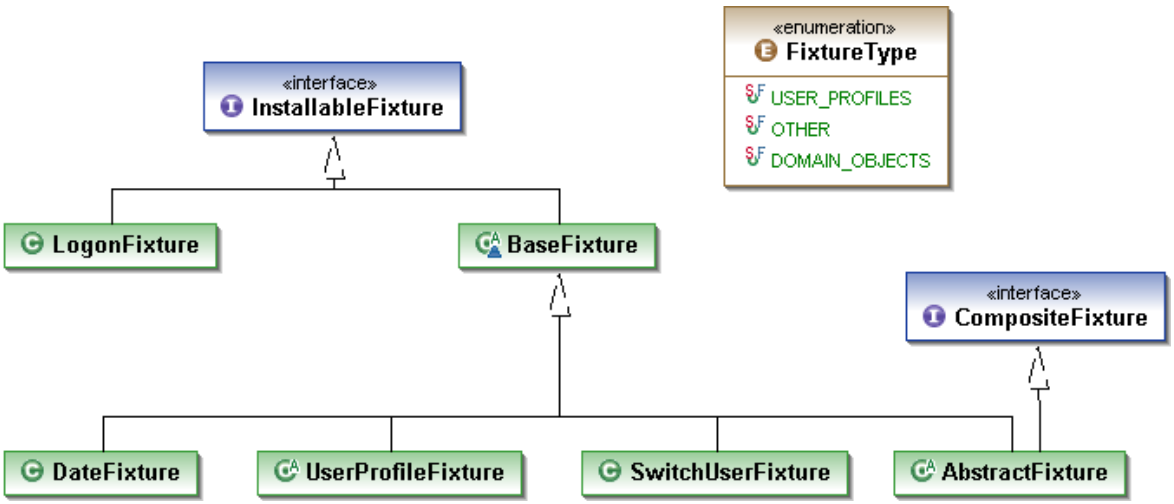
For example:

```
isis.fixtures.prefix= org.apache.isis.support.prototype.fixture
isis.fixtures= ClaimsFixture,LogOnAsCliveFixture
```

It is also possible to specify fixtures from the command line, using the --fixture flag.

How to write custom fixtures

The applib defines several interfaces and classes for writing fixtures. The following UML diagram shows their relationships:



`InstallableFixture` and `FixtureType`

The `InstallableFixture` interface defines the general contract between a fixture and the framework: the `FixtureType`, and an `install()` method.

The `FixtureType` is used to characterize whether the fixture:

- if `FixtureType.DOMAIN_OBJECTS` (the default), then the fixture will be installed only so long as the configured object store does not indicate that fixtures have already been installed
- if `FixtureType.USER_PROFILES`, then the fixture will be installed only so long as the configured profile store does not indicate that fixtures have already been installed
- if `FixtureType.OTHER`, then the fixture will always be installed. This usually refers to setting the clock or user.

`BaseFixture` and `AbstractFixture`

The `BaseFixture` class is not API, but is responsible for instantiating `FixtureClock`. This is an implementation of `Clock` singleton, which (as described in Chapter 11, *Clock*), is used by the rest of the *Isis* framework delegates to in order to obtain the current time. The `FixtureClock`'s purpose is to allow the current date/time to be set by other fixtures.

The `AbstractFixture` class (inheriting from `BaseFixture`) is an general-purpose adapter for writing fixtures. It implements `CompositeFixture` interface, meaning that hierarchies of fixtures can be created following the composite design pattern. It also provides a number of convenience methods:

- the `setDate(...)` and `setTime(...)` method allow the date/time to be set.

  This is done using by obtaining the `FixtureClock` (from `BaseFixture`). There are also some convenience methods to move the date/time either earlier or later.

- the `switchUser(...)` method allow the logged-on user to be changed

  This changes the user logged-on while the fixtures are being installed. This is to facilitate tests that have complex setup requirements, eg verifying workflow between different user roles.

An alternative to using `setDate(...)`/`setTime(...)` is to use `DateFixture`, and an alternative to using `switchUser(...)` is to use `SwitchUserFixture`. Which you use is largely a matter of personal preference.

`DateFixture`

The `DateFixture` provides an alternative to `AbstractFixture`, just allowing the current date/time to be set.

The main difference is one of style; `DateFixture` can be used in a declarative way, whereas `AbstractFixture` is more imperative. For example:

```
public class DateIs13Jan2007Fixture extends DateFixture {
    public DateIs13Jan2007Fixture() {
        super(2007,1,13);
    }
}
```

A fixture in this style could then be used within a composite fixture hierarchy.

`SwitchUserFixture`

The `SwitchUserFixture` provides an alternative to `AbstractFixture`, just allowing the current user to be switched.

The main difference is one of style; `SwitchUserFixture` can be used in a declarative way, whereas `AbstractFixture` is more imperative. For example:

```
public class SwitchToFrankSupervisorFixture extends SwitchUserFixture {
    public SwitchToFrankSupervisorFixture() {
        super("frank", "user", "supervisor");
    }
}
```

A fixture in this style could then be used within a composite fixture hierarchy.

Warning

Do be aware of when you call switchUser() in a `Fixture` as it causes the current transaction to be ended and a new one started (for the new user). If you share a reference between the two you will get an exception.

For example:

```
Account account = accounts.createAccount(
    "ACME", "Peter Planner", "pplanner@acme.com");
Participant peterPlanner = account.getAdmin();

switchUser("pplanner@acme.com", new String[0]);

Work work = plan1.createWork();
BcpPlan plan = (BcpPlan) work.getTarget();
plan.getOwner().modifyLeader(peterPlanner);
```

This will fail because `peterPlanner` reference is no longer valid after the switch user.

The solution is to retrieve the object again so it is part of the second transaction. In this example we can change to code to this:

```
accounts.createAccount("ACME", "Peter Planner", "pplanner@acme.com");

switchUser("pplanner@acme.com", new String[0]);

Account account = uniqueMatch(Account.class, "0 ACME");
Participant peterPlanner = account.getAdmin();

Work work = plan1.createWork();
BcpPlan plan = (BcpPlan) work.getTarget();
plan.getOwner().modifyLeader(peterPlanner);
```

`LogonFixture`

Unlike the very similar `SwitchUserFixture`, this fixture does not affect the currently logged on user while the fixtures are being installed. Instead, it is used to specify the user to logon as once all the fixtures have been installed.

If more than one `LogonFixture` is specified, the last one encountered is used.

UserProfileFixture

The `UserProfileFixture` is used to populate the configured profile store[1] with user profiles (the concept of which is described in Chapter 12, *Profiles*).

To create a user profile for a specific user, inherit from `UserProfileFixture`, use the inherited `newUserProfile()` to create a profile and use the inherited `saveForUser(...)` method to save that profile.

It is also possible to create a 'template' perspective using the `saveAsDefault(...)` method. This is used as the basis for any new perspectives that are automatically created, eg for users who are logging in and for whom there is no perspective in existence.

For example:

```
public class PerspectivesFixture extends UserProfileFixture {
  @Override
  protected void installProfiles() {
    Profile profile = newUserProfile();
    Perspective perspective = profile.newPerspective("ECS");
    perspective.addToServices(LocationFactory.class);
    perspective.addToServices(CustomerRepository.class);
    perspective.addToServices(PaymentMethodFactory.class);

    saveAsDefault(profile);
  }
}
```

With that set up, when a new user now logs in they will see three service icons on the screen for locations, customers and payment methods. In the *dnd viewer* the user will be able to add and remove services from their perspective. See the DnD Viewer documentation for further details.

Any fixtures, if specified, are only ever loaded once; the fixture installer checks with `UserProfileSevice.isInitialized()` to see if the fixture is already installed. This allows fixtures to be used for seeding a persisting profile store, and then be ignored thereafter.

[1] A profile store is a persistence mechanism provided by the configured runtime that allows user profiles to be stored between runs. It is similar to, but independent of, an object store (which stores the domain objects themselves).

Chapter 14. XML Snapshots

Generating XML snapshots from domain objects.

The *Apache Isis* framework provides the capability to generate XML snapshots (and if required corresponding XSD schemas) based on graphs of domain objects. This is done using the `org.apache.isis.core.runtime.snapshot.XmlSnapshot` class.

Generating an XML Snapshot

The `XmlSnapshot` can be created either directly or using a builder.

Basic Usage

The standard usage is to instantiate directly.

```
XmlSnapshot snapshot = new XmlSnapshot(customer);
Element customerAsXml = snapshot.getXmlElement();
```

This will return the `Customer`'s fields, titles of simple references, number of items in collections.

In order to use the `XmlSnapshot`, the domain object must implement `org.apache.isis.applib.snapshot.Snapshottable`. This is just a marker interface.

Including other Elements

It's also possible to instruct the `XmlSnapshot` to "walk" the object graph and include other information within the generated XML.

For example:

```
XmlSnapshot snapshot = new XmlSnapshot(customer);
snapshot.include("placeOfBirth");   // (1)
snapshot.include("orders/product"); // (2)
Element customerAsXml = snapshot.getXmlElement();
```

In (1), we indicate that we want to also navigate to another domain object represented by simple reference "placeOfBirth"; in (2), we indicate that we want to navigate the "orders" collection (presumably of `Order`s) and for each referenced `Order`, to navigate in turn its "product" reference (presumably to a `Product` class).

Note that `XmlSnapshot` is mutable, in that calls to its `getXmlElement()` may return different XML structures based on whether additional paths have been `include()`'d, or whether the state of the domain objects themselves have changed.

Using the Fluent API

An `XmlSnapshot` can also be constructed using an alternative "fluent" API:

```
XmlSnapshot snapshot =
     XmlSnapshot.create(customer)
                .includePath("placeOfBirth")
```

```
                .include("orders/product")
                .build();
Element customerAsXml = snapshot.getXmlElement();
```

The `SnapshottableWithInclusions` interface

As already mentioned, in order to be snapshotted a domain object must implement the `Snapshot` interface. This is just a marker interface, so implementing it is trivial.

Alternatively, the domain object can choose to implement the sub-interface, `SnapshottableWithInclusions`. This moves the responsibility for determining what is included within the snapshot from the caller to the snapshottable object itself:

```
public interface SnapshottableWithInclusions extends Snapshottable {
    List<String> snapshotInclusions();
}
```

If necessary, both approaches can be combined.

Generating an XSD schema

As well as obtaining the XML snapshot, it is also possible to obtain an XSD schema that the XML snapshot conforms to.

```
XmlSnapshot snapshot = ...;
Element customerAsXml = snapshot.getXmlElement();
Element customerXsd = snapshot.getXsdElement();
```

This can be useful for some tools. Note that for the XSD to be correct, the object being snapshotted must have non-null values for the paths that are `include()`'d. If this isn't done then the XSD will not be correct reflect for another snapshotted object that does have non-null values.

Hints and Tips

As an alternative to using include(), you might consider building a non-persisted domain object (a "view model") which can reference only the relevant information required for the snapshot.

For example, if only the 5 most recent `Order`s for a `Customer` were required, a `CustomerAndRecentOrders` view model could hold a collection of just those 5 `Order`s.

Typically such view models would implement `SnapshottableWithInclusions`.

Part IV. Reference Appendices

Table of Contents

Appendix A. Recognized Methods and Prefixes

The following table lists all of the methods or method prefixes that are recognized by *Apache Isis*' default programming model:

Table A.1. Recognized Method Prefixes

Prefix	object	property	collection	action	action param	See also
addTo			Y			removeFrom
choices		Y			Y	
clear		Y				modify
created	Y					
default		Y			Y	
disable		Y	Y	Y		
get		Y	Y			set
getId	Y					(services only)
hide		Y	Y	Y		
iconName	Y					
loaded	Y					
loading	Y					
modify		Y				clear
persisted	Y					
persisting	Y					
removing	Y					
removed	Y					
removeFrom			Y			addTo
set		Y	Y			get
toString	Y					
title	Y					
updating	Y					
updated	Y					
validate	Y	Y	Y		Y	

There are also a number of deprecated methods:

Table A.2. Deprecated Method Prefixes

Prefix	object	property	collection	action	action param	See also
deleted	Y					

Prefix	object	property	collection	action	action param	See also
deleting	Y					
saved	Y					
saving	Y					

In order to be recognized, all methods must be `public`. Any methods that do not match are deemed to be action methods that the user can invoke from the user interface.

Appendix B. Recognized Annotations

All the annotations recognized in the *Apache Isis* default programming model.

This chapter defines the set of annotations that are recognised by the *Apache Isis* default programming model.

@ActionOrder

Note

The recommended mechanism for specifying the order in which actions are listed to the user is `@MemberOrder` (see the section called “@MemberOrder”)

`@ActionOrder` provides a mechanism to specify the order in which actions appear in the user interface, in which the order is specified in one place in the class.

For example:

```
@ActionOrder("PlaceNewOrder, CheckCredit")
public class Customer {

    public Order placeNewOrder() {}

    public CreditRating checkCredit() {}

...
}
```

The action names are not case sensitive.

Compared to `@MemberOrder`, there is (currently) one additional advantage in that you can easily specify groupings (which may be rendered by the viewer as sub-menus). This information may be used by the viewing mechanism to render actions into sub-menus.

For example:

```
@ActionOrder("(Account Management: PlaceOrder, CheckCredit), (Personal Details: Ch
public class Customer {
    public CreditRating checkCredit() { ... }
    public void changeOfAddress() { ... }
    public Order placeNewOrder() { ... }
    public void addEmail(String emailAddress) { ... }
    ...
}
```

However, `@ActionOrder` is more 'brittle' to change: if you change the name of an existing action you will need to ensure that the corresponding name within the `@ActionOrder` annotation is also changed.

@ActionSemantics

This annotation, which applies only to actions, describes whether the invocation is safe (as no side-effects), is idempotent (may have side-effects but always has the same postconditions), or is neither safe nor idempotent. If the annotation is missing then the framework assumes non-idempotent.

For example:

```
public class Customer {
    @ActionSemantics(Of.SAFE)
    public CreditRating checkCredit() { ... }

    @ActionSemantics(Of.IDEMPOTENT)
    public void changeOfAddress(Address address) { ... }

    @ActionSemantics(Of.NON_IDEMPOTENT)
    public Order placeNewOrder() { ... }
    ...
}
```

The annotation was introduced for the restfulobjects viewer in order that action invocations could be made available using either HTTP GET, PUT or POST (respectively). It is now also used in core runtime's in-built concurrency checking; the invocation of a safe action does not perform a concurrency check, whereas non-safe actions do perform a concurrency check.

@AutoComplete

This annotation is to support an auto-complete capability for reference properties and action parameters, the idea being that the user enters a few characters to locate a reference, and these are shown - for example - in a drop-down list box.

The annotation is specified on the type, and specifies an action on a repository; this action should take a single string and should return a list of the type.

For example:

```
@AutoComplete(repository=Customers.class, action="autoComplete")
public class Customer extends AbstractDomainObject {
   ....
}
```

where:

```
public interface Customers {

    @Hidden
    List<Customer> autoComplete(String search);
    ...
}
```

This annotation was first implemented in the Wicket viewer.

@Aggregated

This annotation indicates that the object is aggregated, or wholly owned, by a root object. This information is of use by some object stores implementations (to store the aggregated objects "inline"). At the time of writing none of the viewers exploit this information, though this may change in the future. (For example, the DnD viewer could be enhanced to prevent aggregated objects from being "dragged out" from their root object).

All value types and all collections are automatically aggregated.

Warning

Outside of value types and collections, the `@Aggregated` semantics are not completely well-defined and so its use is currently discouraged.

@Bounded

For immutable objects where there is a bounded set of instances, the `@Bounded` annotation can be used.

For example:

```
@Bounded
public class County {
    ...
}
```

The number of instances is expected to be small enough that all instance can be held in memory. The viewer will use this information to render all the instances of this class in a drop-down list or equivalent.

Note

Although this is not enforced, `@Bounded` is intended for use on `final` classes. Its behaviour when used on interfaces, or classes with sub-classes is not specified

@Debug

The `@Debug` annotation marks an action method as available in debug mode only, and so will not normally be displayed by the user interface.

For example:

```
public class Customer {
    public Order placeNewOrder() { ... }
    @Debug
    public List<Order> listRecentOrders() { ... }
    ...
}
```

The exact means to access `@Debug` actions depends on the viewer.

@Defaulted

The concept of "defaulted" means being able to provide a default value for the type by way of the `org.apache.isis.applib.adapters.DefaultsProvider` interface. Generally this only applies to value types, where the `@Value` annotation (see the section called “@Value”) implies encodability through the `ValueSemanticsProvider` interface (see Chapter 10, *Value Types*).

For these reasons the `@Defaulted` annotation is generally never applied directly, but can be thought of as a placeholder for future enhancements whereby non-value types might also have a default value provided for them.

@DescribedAs

The `@DescribedAs` annotation is used to provide a short description of something that features on the user interface. How this description is used will depend upon the viewing mechanism - but it may be thought of as being like a 'tool tip'.

Descriptions may be provided for objects, members (properties, collections and actions), and for individual parameters within an action method. `@DescribedAs` therefore works in a very similar manner to `@Named` (see the section called “@Named”).

Providing a description for an object

To provide a description for an object, use the `@DescribedAs` annotation immediately before the declaration of that object class.

For example:

```
@DescribedAs("A customer who may have originally become known to us via " +
             "the marketing system or who may have contacted us directly.")
public class ProspectiveSale {
   ...
}
```

Providing a description for an object member

Any member (property, collection or action) may provide a description. To specify this description, use the `@DescribedAs` annotation immediately before the declaration of that member.

For example:

```
public class Customer {
    @DescribedAs("The name that the customer has indicated that they wish to be "
                 "addressed as (e.g. Johnny rather than Jonathan)")
    public String getFirstName() { ... }
}
```

Providing a description for an action parameter

To provide a description for an individual action parameter, use the `@DescribedAs` annotation in-line i.e. immediately before the parameter declaration.

For example:

```
public class Customer {
    public Order placeOrder(
                      Product product,
                      @Named("Quantity")
                      @DescribedAs("The quantity of the product being ordered")
                      int quantity) {
        Order order = createTransientInstance(Order.class);
        order.modifyCustomer(this);
        order.modifyProduct(product);
        order.setQuantity(quantity);
        return order;
```

```
    }
    ...
}
```

@Disabled

The `@Disabled` annotation means that the member cannot be used in any instance of the class. When applied to the property it means that the user may not modify the value of that property (though it may still be modified programmatically). When applied to an action method, it means that the user cannot invoke that method.

For example:

```
public class Customer {
    @Disabled
    public void assessCreditWorthiness() { ... }

    @Disabled
    public int getInitialCreditRating(){ ... }
    public void setInitialCreditRating(int initialCreditRating) { ... }
}
```

Note that if an action is marked as `@Disabled`, it will be shown on the user interface but cannot ever be invoked. The only possible reason we can think to do this is during prototyping, to indicate an action that is still to be developed. If a method is intended for programmatic use, but not intended ever to be invoked directly by a user, then it should be marked as `@Hidden` instead.

This annotation can also take two parameter indicating where (in the UI) it is to be disabled, and when (in the object's lifecycle) it is to be disabled. For example:

```
public class Customer {
    @Disabled(when=When.UNTIL_PERSISTED)
    public void assessCreditWorthiness() { ... }
}
```

would disable the action until the object has been saved. And:

```
public class Customer {
    @Disabled(where=Where.PARENTED_TABLES)
    public void getFirstName() { ... }
}
```

would disable the property in parented tables but not in regular object forms (though note: this would only be used by viewers that provide in-table editing capability).

The acceptable values for the `where` parameter are:

- `Where.ANYWHERE`

  The member should be disabled everywhere.

- `Where.OBJECT_FORMS`

  The member should be disabled when displayed within an object form. For most viewers, this applies to property and collection members, not actions.

- `Where.PARENTED_TABLES`

The member should be disabled when displayed as a column of a table within a parent object's collection. For most (all?) viewers, this will have meaning only if applied to a property member.

- `Where.STANDALONE_TABLES`

  The member should be disabled when displayed as a column of a table showing a standalone list of objects, for example as returned by a repository query. For most (all?) viewers, this will have meaning only if applied to a property member.

- `Where.ALL_TABLES`

  The member should be disabled when displayed as a column of a table, either an object's * collection or a standalone list. This combines PARENTED_TABLE} and STANDALONE_TABLE

- `Where.NOWHERE`

  Has no meaning for the `@Disabled` annotation (though is used by the `@Hidden` annotation, see the section called "@Hidden").

The acceptable values for the `when` parameter are:

- `When.ALWAYS`

  The member should be disabled at all times.

- `When.NEVER`

  The member should never be disabled (unless disabled through some other mechanism, for example an imperative `disableXxx()` supporting method)..

- `When.ONCE_PERSISTED`

  The member should be enabled for transient objects, but disabled for persisted objects.

- `When.UNTIL_PERSISTED`

  The member should be disabled for transient objects, but enabled for persisted objects.

By default the annotated property or action is always disabled (ie defaults to `Where.ANYWHERE`, `When.ALWAYS`).

@Encodable

Encodability means the ability to convert an object to-and-from a string, by way of the `org.apache.isis.applib.adapters.EncoderDecoder` interface. Generally this only applies to value types, where the `@Value` annotation (see the section called "@Value") implies encodability through the `ValueSemanticsProvider` interface (see Chapter 10, *Value Types*).

For these reasons the `@Encodable` annotation is generally never applied directly, but can be thought of as a placeholder for future enhancements whereby non-value types might also be directly encoded.

@EqualByContent

Equal-by-content is a characteristic of value types, and is implied by any class annotated with the the `@Value` annotation (see the section called "@Value" and also Chapter 10, *Value Types*).

The `@EqualByContent` annotation exists so that this semantic could, if required, by applied directly to an object without it necessarily also being annotated as a value.

That said, this annotation is only really for completeness. Moreover, the semantic captured by this annotation is currently used by the framework (not in any object store nor viewer).

@Executed

The `@Executed` annotation is relevant only for client/server deployments (eg with the DnD viewer).

In a client/server deployment, all methods for persistent objects are executed on the server-side and for transient objects are executed on the client-side.

This annotation can be used to override that default.

Forcing a method to be executed on the client

The `@Executed(Where.LOCALLY)` annotation marks an action method so that it executes on the client, rather than being forwarded to the server for execution. This is useful for methods that invoke a service that must be run client-side, for example spawning off a separate process (such as a web browser or Acrobat Reader).

Forcing a method to be executed on the server

The `@Executed(Where.REMOTELY)` annotation marks an action method so that it executes on the server, even though it would normally be executed on the client (as methods for transient objects are). This is useful for methods that although based on transient objects need access to persistent objects.

@Exploration

The `@Exploration` annotation marks an action method as available in exploration mode only, and therefore not intended for use in the production system. An exploration action may or may not also be a debug action (annotated with `@Debug`, see the section called “@Debug”).

For example:

```
public class Customer {
    public Order placeNewOrder() { ... }
    @Exploration
    public List<Order> listRecentOrders() { ... }
    ...
}
```

See also the `@Prototype` annotation, the section called “@Prototype”

@Facets

The `@Facets` annotation specifies `FacetFactory` implementations and so can be used to run install arbitrary `Facet`s for a type. Generally this is not needed, but can be useful for overriding a custom programming model where a `FacetFactory` is not typically included.

See the core documentation for more on writing `FacetFactory`s.

@FieldOrder

Note

The recommended mechanism for specifying the order in which fields are listed to the user is `@MemberOrder` (see the section called “@MemberOrder”)

`@FieldOrder` provides a mechanism to specify the order in which fields appear in the user interface, in which the order is specified in one place in the class.

For example:

```
@FieldOrder("Name, Address, DateOfBirth, RecentOrders")
public class Customer {
    public Date getDateOfBirth() {...}
    public List<Order> getRecentOrders() {...}
    public String getAddress() {...}
    public String getName() {...}
    ...
}
```

The field names are not case sensitive.

However, `@FieldOrder` is more 'brittle' to change: if you change the name of an existing property you will need to ensure that the corresponding name within the `@FieldOrder` annotation is also changed.

@Hidden

The `@Hidden` annotation indicates that the member (property, collection or action) to which it is applied should never be visible to the user. It can also be applied to service types (it has no effect if applied to entities or values).

For example:

```
public class Customer {
    @Hidden
    public int getInternalId() { ... }

    @Hidden
    public void updateStatus() { ... }
    ...
}
```

Or, applied to a service:

```
@Hidden
public class EmailService {
    public void sendEmail(...) { ... }
    ...
}
```

This annotation can also take a parameters indicating where and when it is to be hidden. For example:

```
public class Customer {
    @Hidden(when=When.ONCE_PERSISTED)
```

```
    public int getInternalId() { ... }
    ...
}
```

would show the `Id` until the object has been saved, and then would hide it. And:

```
public class Customer {
    @Hidden(where=Where.ALL_TABLES)
    public int getDateOfBirth() { ... }
    ...
}
```

would suppress the `dateOfBirth` property of a Customer from all tables.

The acceptable values for the `where` parameter are:

- `Where.ANYWHERE`

  The member should be hidden everywhere.

- `Where.OBJECT_FORMS`

  The member should be hidden when displayed within an object form. For most viewers, this applies to property and collection members, not actions.

- `Where.PARENTED_TABLES`

  The member should be hidden when displayed as a column of a table within a parent object's collection. For most (all?) viewers, this will have meaning only if applied to a property member.

- `Where.STANDALONE_TABLES`

  The member should be hidden when displayed as a column of a table showing a standalone list of objects, for example as returned by a repository query. For most (all?) viewers, this will have meaning only if applied to a property member.

- `Where.ALL_TABLES`

  The member should be /hidden when displayed as a column of a table, either an object's * collection or a standalone list. This combines `PARENTED_TABLES` and `STANDALONE_TABLES`.

- `Where.NOWHERE`

  Acts as an override if a member would normally be hidden as a result of some other convention. For example, if a property is annotated with `@Title` (see the section called “@Title”), then normally this should be hidden from all tables. Additionally annotating with `@Hidden(where=Where.NOWHERE)` overrides this.

The acceptable values for the `when` parameter are:

- `When.ALWAYS`

  The member should be hidden at all times.

- `When.NEVER`

  The member should never be hidden (unless disabled through some other mechanism, for example an imperative `disableXxx()` supporting method)..

- `When.ONCE_PERSISTED`

  The member should be visible for transient objects, but hidden for persisted objects.

- `When.UNTIL_PERSISTED`

  The member should be hidden for transient objects, but visible for persisted objects.

By default the annotated property or action is always hidden (ie defaults to `Where.ANYWHERE`, `When.ALWAYS`).

@Idempotent (deprecated)

Equivalent to using `@ActionSemantics(Of.IDEMPOTENT)` on an action.

@Ignore (deprecated)

Equivalent to using `@Programmatic`. The `@Programmatic` annotation was introduced because `@Ignore` can easily clash with `@org.junit.Ignore`.

@Immutable

The `@Immutable` annotation may be applied to a class, and indicates to the framework that the state of such objects may not be changed. The viewers will prevent any change through the user interface, and moreover the object stores will reject any changes to the objects that might have occurred programmatically.

For example:

```
@Immutable
public class Country {
    ...
}
```

This annotation can also take a single parameter indicating when it is to become immutable. For example:

```
@Immutable(When.ONCE_PERSISTED)
public class Email {
    ...
}
```

This would allow the user to create an email object and set it up, and then prevent any changes once it has been saved.

The acceptable values for the parameter are:

- `When.ALWAYS`
- `When.NEVER`
- `When.ONCE_PERSISTED`
- `When.UNTIL_PERSISTED`

By default the annotated property or action is always immutable (ie defaults to `When.ALWAYS`).

@Mask

The `@Mask` annotation may be applied to any property, or to any parameter within an action method, that allows the user to type in text as input. It can also annotate a string-based value type, and thereby apply to all properties or parameters of that type.

The mask serves to validate, and potentially to normalise, the format of the input. The characters that can be used are based on Swing's `javax.swing.text.MaskFormatter`, and also Java's `java.util.SimpleDateFormat`.

For example, on a property:

```
public class Email {
    @Mask("(NNN)NNN-NNNN")
    public String getTelephoneNumber() {...}
    public void setTelephoneNumber(String telNo) {...}
    ...
}
```

Or, on an action parameter:

```
public void ContactRepository {
    public void newContact(
            @Named("Contact Name") String contactName
           ,@Mask("(NNN)NNN-NNNN")
            @Named("Telephone Number") String telNo) {
        ...
    }
    ...
}
```

Or, on a value type:

```
@Value(...)
@MaxLength(30)
public class CustomerFirstName {
    ...
}
```

See also `@RegEx` annotation, the section called “@RegEx”.

@MaxLength

The `@MaxLength` annotation indicates the maximum number of characters that the user may enter into a `String` property, or a `String` parameter in an action, or for a string-based value type. It is ignored if applied to any other type.

For example, on a property:

```
public class Customer {
    @MaxLength(30)
    public String getFirstName() { ... }
```

```
    public void setFirstName(String firstName) { ... }
    ...
}
```

Or, on an action parameter:

```
public class CustomerRepository {
    public Customer newCustomer(
        @MaxLength(30)
        @Named("First Name") String firstName
       ,@MaxLength(30)
        @Named("Last Name") String lastName) {
    ...
}
```

Or, for a value type:

```
@Value(...)
@MaxLength(30)
public class CustomerFirstName {
    ...
}
```

If the model is being persisted to a relational database then `@MaxLength` should be specified for all `String` properties and action parameters.

@MemberGroups

`@MemberGroups` is designed to work in conjunction with `@MemberOrder` (see the section called "@MemberOrder"), and specifies the order in which groups of members should be rendered.

For example:

```
@MemberGroups({"General", "Dates", "Other"})
public Class Customer {
    @MemberOrder(name="General", sequence="1.1")
    public String getFirstName() {...}
    public void setFirstName(value as String) {...}

    @MemberOrder(name="General", sequence="1.2")
    public String getLastName() {...}
    public void setLastName(value as String) {...}

    @MemberOrder(name="Other", sequence="1")
    public String getAddress() {...}
    public void setAddress(value as String) {...}

    @MemberOrder(name="Dates", sequence="1")
    public Date getDateOfBirth() {...}
    public void setDateOfBirth(value as Date) {...}
    ...
}
```

If the `@MemberOrder`'s name is not specified, then its group is assumed to be "`General`".

@MemberOrder

`@MemberOrder` is the recommended mechanism for specifying the order in which fields and/or actions are presented to the user. (`@ActionOrder` and `@FieldOrder` provide alternative, deprecated mechanisms).

`@MemberOrder` is specified at the individual member level, relative to other members, as a string. The simplest convention is to use numbers - 1, 2, 3 - though it is a better idea to leave gaps in the numbers - 10, 20, 30 perhaps - such that a new member may be added without having to edit existing numbers. A useful alternative is to adopt the 'dewey-decimal' notation - 1, 1.1, 1.2, 2, 3, 5.1.1, 5.2.2, 5.2, 5.3 - which allows for an indefinite amount of future insertion, and allows subclasses to insert their class members as required.

For example:

```
public Class Customer {
    @MemberOrder(sequence="2.1")
    public String getAddress() {...}
    public void setAddress(value as String) {...}

    @MemberOrder(sequence="1.1")
    public String getFirstName() {...}
    public void setFirstName(value as String) {...}

    @MemberOrder(sequence="1.2")
    public String getLastName() {...}
    public void setLastName(value as String) {...}

    @MemberOrder(sequence="3")
    public Date getDateOfBirth() {...}
    public void setDateOfBirth(value as Date) {...}
    ...
}
```

If a member does not have a specified order then it will be placed after those that are specified. Two members may have the same sequence specifier, but in such a case the relative ordering of those members will be indeterminate.

Note

Certain styles of user interface may lay out an object's properties and its collections separately, in which case the relative member order of properties and collections will be evaluated separately. In general, though, consider the layout of all properties and collections together, and of all actions together.

As a refinement to this, some viewers support the notion of grouping members together. In this case the `name` attribute can be specified.

For example, the following can be used to group properties:

```
public Class Customer {
    @MemberOrder(name="General", sequence="1.1")
    public String getFirstName() {...}
    public void setFirstName(value as String) {...}

    @MemberOrder(name="General", sequence="1.2")
```

```
    public String getLastName() {...}
    public void setLastName(value as String) {...}

    @MemberOrder(name="Other", sequence="1")
    public String getAddress() {...}
    public void setAddress(value as String) {...}

    @MemberOrder(name="Dates", sequence="1")
    public Date getDateOfBirth() {...}
    public void setDateOfBirth(value as Date) {...}
    ...
}
```

In this case the `sequence` is ordered with respect to the `name`. If using group names in this way, typically the MemberGroups annotation (see the section called “@MemberGroups”) should also be specified, allowing the order of the groups themselves to be sorted.

Grouping can also be performed on actions. Some viewers (for example the Wicket viewer) support the idea that the group name can be the name of a collection. In this case, the action is rendered "near to" the collection (for example, to support add or remove of elements).

For example:

```
public Class Customer {

    public List<CommunicationChannel> getCommunicationChannels() { ... }

    ...
    @MemberOrder(name="communicationChannels", sequence="1.1")
    public void addCommunicationChannel(...) { ... }

    @MemberOrder(name="communicationChannels", sequence="1.2")
    public void removeCommunicationChannel(...) { ... }

    ...
}
```

@MultiLine

The `@MultiLine` annotation provides information about the carriage returns in a `String` property or action parameter, or for a string-based value type. It also implies a hint to the viewer that the widget to be used should be over multiple lines (eg a text area rather than a text field), with appropriate wrapping and/or scrollbars.

More formally, the annotation indicates that:

- the `String` property or parameter may contain carriage returns, and
- (optionally) the typical number of such carriage returns (meaning the number of rows in the text area), and
- (optionally) that the text should be wrapped (the default is that text is not wrapped).

Warning

Currently the `preventWrapping` functionality is not fully implemented.

The syntax is:

```
@MultiLine([numberOfLines=<typicalNumberOfCRs>]
[,preventWrapping=<false|true>])
```

For example:

```
public class BugReport {
    @MultiLine(numberOfLines=10)
    public String getStepsToReproduce() { ... }
    public void setStepsToReproduce(String stepsToReproduce) { ... }
    ...
}
```

Here the `stepsToReproduce` may be displayed in a text area of 10 rows, with no wrapping. A horizontal scrollbar may appear if the number of characters on any given row exceeds the width.

Another example:

```
public class Email {
    @MultiLine(numberOfLines=20, preventWrapping=false)
    public String getBody() { ... }
    public void setBody(String body) { ... }
    ...
}
```

Here the body should be displayed in a text area of 20 rows, with wrapping.

If the annotation is combined with the `@TypicalLength`, then the expected width of the text area in the user interface will be determined by the value of the typical length divided by the number of specified lines. For example:

```
public class Email {
    @MultiLine(numberOfLines=20, preventWrapping=false)
    @TypicalLength(800)
    public String getBody() { ... }
    public void setBody(String body) { ... }
    ...
}
```

Here the body will (likely be) displayed in a text area of 20 rows, with 40 columns.

@MustSatisfy

The `@MustSatisfy` annotation allows validation to be applied to properties and parameters using an (implementation of a) `org.apache.isis.applib.spec.Specification` object.

For example, on a property:

```
public class Customer {
    @MustSatisfy(StartWithCapitalLetterSpecification.class)
    public String getFirstName() { ... }
    ...
}
```

Or, on an action parameter:

```
public class CustomerRepository {
    public Customer newCustomer(
            @MustSatisfy(StartWithCapitalLetterSpecification.class)
            @Named("First Name") firstName
           ,@MustSatisfy(StartWithCapitalLetterSpecification.class)
            @Named("Last Name") lastName) {
        ...
    }
    ...
}
```

The `Specification` is consulted during validation, being passed the proposed value.

An alternative to using `@MustSatisfy` is to define a custom value type, see Chapter 10, *Value Types*.

@Named

The `@Named` annotation is used when you want to specify the way something is named on the user interface i.e. when you do not want to use the name generated automatically by the system. It can be applied to objects, members (properties, collections, and actions) and to parameters within an action method.

Warning

Generally speaking it is better to rename the property, collection or action. The only common case where `@Named` is common is to rename parameters for built-in value types. Even here though a custom value type can be defined using `@Value` so that the value type is used as the parameter name. `@Named` may also be used if the name needs punctuation or other symbols in the name presented to the user.

Specifying the name of an object

By default the name of an object is derived, reflectively from the class name. To specify a different name for an object, use the `@Named` annotation in front of the class declaration.

For example:

```
@Named("Customer")
public class CustomerImpl implements Customer{
   ...
}
```

See also the `@Plural` annotation, the section called “@Plural”.

Specifying the name of a class member

By default, the name of a class member (a property, collection or action) presented to the user is derived, reflectively, from the name of the member defined in the program code. To specify a different name use the `@Named` annotation immediately before the member declaration.

For example:

```
public class Customer {
    @Named("Given Name")
    public String getFirstName() { ... }
```

```
    @Named("Family Name")
    public String getSurname() { ... }

    public CreditRating getCreditRating() { ... }
}
```

Note that the framework provides a separate and more powerful mechanism for internationalisation.

Specifying the name for an action parameter

The most common usage of `@Named` is be to specify names for the parameters of an action. This is because the parameter name declared in the code for the action method cannot be picked up reflectively (by default, the user interface will use the type of the parameter as the name; for a String or a Boolean, this is almost certainly not what is required).

To specify the name of a parameter, the `@Named` annotation is applied 'in-line' (i.e. preceding the individual parameter declaration.

For example:

```
public class Customer {
    public Order placeOrder(
            Product product
           ,@Named("Quantity")
            int quantity) {
        Order order = newTransientInstance(Order.class);
        order.modifyCustomer(this);
        order.modifyProduct(product);
        order.setQuantity(quantity);
        return order;
    }
    ...
}
```

An alternative is to use a value type, as described in Chapter 10, *Value Types*.

@NotContributed

The `@NotContributed` annotation applies only to action methods, and specifically to the actions of services. If present, it indicates that the action should not be contributed to any of its domain object arguments.

For example:

```
public class OrderServices {
    @NotContributed
    public void cancel(Order order);
    ...
}
```

If the action should neither be contributed nor appear in the service's service menu (see the section called “@NotInServiceMenu”), then you could instead simply mark it as `@Hidden` (see the section called “@Hidden”).

@NotInServiceMenu

The `@NotInServiceMenu` annotation applies only to action methods, and specifically to the actions of services. If present, it indicates that the action should not appear in the service menu for the service. It may, however, still be contributed to any of its domain object arguments.

For example:

```
public class OrderServices {
    @NotInServiceMenu
    public void cancel(Order order);
    ...
}
```

If the action should neither be contributed (see the section called “@NotContributed”) nor appear in the service's service menu, then you could instead simply mark it as `@Hidden` (see the section called “@Hidden”).

@NotPersistable

The `@NotPersistable` annotation indicates that transient instances of this class may be created but may not be persisted. The framework will not provide the user with an option to 'save' the object, and attempting to persist such an object programmatically would be an error.

For example:

```
@NotPersistable
public class InputForm {
    // members and actions here
}
```

This annotation can also take a single parameter indicating whether it is only the user that cannot persist the object, for example the following code would prevent the user from saving the object (via the viewer) but still allow the program to persist the object.

For example:

```
@NotPersistable(By.USER)
public class InputForm {
    ...
}
```

The acceptable values for the parameter are:

- `By.USER`
- `By.USER_OR_PROGRAM`

By default the annotated object is effectively transient (ie default to `By.USER_OR_PROGRAM`).

@NotPersisted

The `@NotPersisted` annotation indicates that the property is not to be persisted.

Note

In many cases the same thing can be achieved simply by providing the property with a 'getter' but no 'setter'.

For example:

```
public class Order {

    @NotPersisted
    public Order getPreviousOrder() {...}
    public void setPreviousOrder(Order previousOrder) {...}

    ...
}
```

@ObjectType

The `@ObjectType` annotation is used to provide a unique abbreviation for the object's class name. This is used internally to generate a string representation of an objects identity (the Oid).

For example:

```
@ObjectType("ORD")
public class Order {

    ...
}
```

If no `@ObjectType` annotation is present, then the framework uses the fully-qualified class name.

If an `@ObjectType` is not unique, then the framework will fail to boot.

@Optional

By default, the system assumes that all properties of an object are required, and therefore will not let the user save a new object unless a value has been specified for each property. Similarly, by default, the system assumes that all parameters in an action are required and will not let the user execute that action unless values have been specified for each parameter.

To indicate that either a property, or an action parameter, is optional, use the `@Optional` annotation.

Note

The `@Optional`annotation has no meaning for a primitive property (or parameter) such as `int` - because primitives will always return a default value (e.g. zero). If optionality is required, then use the corresponding wrapper class (e.g. java.lang.Integer).

Making a property optional

Annotate the getter to indicate that a property is `@Optional`. For example:

```
public class Order {
```

```
    public Product getProduct() { ... }

    public java.util.Date getShipDate() { ... }
    public void setShipDate(Date java.util.shipDate) { ... }

    @Optional
    public String getComments() { ... }
    public void setComments(String comments) { ... }
}
```

Here the `product` and `shipDate` properties are both required, but the `comments` property is optional.

Making an action parameter optional

To indicate that an action may be invoked without having to specify a value for a particular parameter, annotate with `@Optional`. For example:

```
public class Customer {
    public Order placeOrder(
            Product product
           ,@Named("Quantity") int quantity
           ,@Optional @Named("Special Instructions") String instr) {
        ...
    }
    ...
}
```

@Paged

This annotation is used to indicate that parented and/or standalone collections should be paginated.

When annotated on a collection, `@Paged` indicates the page size of a parented collection. When annotated on a type, `@Paged` indicates the page size of a standalone collection.

For example:

```
@Paged(30)
public class Order {

    @Paged(15)
    public List<LineItem> getDetails() {...}
}
```

This indicates a page size of 15 for parented collections, and a page size of 30 for standalone collections.

When omitting a parameter value or omitting the annotation completely, the configured defaults in `isis.properties` wil be used.

For example:

```
isis.viewers.paged.standalone=20
isis.viewers.paged.parented=5
```

This indicates a page size of 5 for parented collections and a page size of 20 for standalone collections.

@Parseable

Parseability means being able to parse a string representation into an object by way of the `org.apache.isis.applib.adapters.Parser` interface. Generally this only applies to value types, where the `@Value` annotation (see the section called “@Value”) implies encodability through the `ValueSemanticsProvider` interface (see Chapter 10, *Value Types*).

For these reasons the `@Parser` annotation is generally never applied directly, but can be thought of as a placeholder for future enhancements whereby non-value types might also have be able to be parsed. For example, a `Customer` could in theory be parsed from a string representing that `Customer`'s title (perhaps in conjunction with some other annotation to determine the repository by which to perform the lookup).

@Plural

When the framework displays a collection of several objects it may use the plural form of the object type in the title. By default the plural name will be created by adding an 's' to the end of the singular name (whether that is the class name or another name specified using `@Named`). Where the single name ends in 'y' then the default plural name will end in 'ies' - for example a collection of `Country` objects will be titled 'Countries'. Where these conventions do not work, the programmer may specify the plural form of the name using `@Plural`. For example:

```
@Plural("Children")
public class Child {
    // members and actions here
}
```

@Programmatic

The `@Programmatic` annotation can be used to cause Apache Isis to complete ignore a class member. This means it won't appear in any viewer, its value will not be persisted, and it won't appear in any XML snapshots (see Chapter 14, *XML Snapshots*).

A common use-case is to ignore implementation-level artifacts. For example:

```
public class Customer implements Comparable<Customer> {
    ...
    @Programmatic
    public int compareTo(Customer c) {
        return getSalary() - c.getSalary();
    }
    ...
}
```

Note that `@Programmatic` does not simply imply `@Hidden` and `@NotPersisted`; it actually means that the class member will not be part of the Isis metamodel.

@Prototype

The `@Prototype` annotation marks an action method as available in prototype mode only, and therefore not intended for use in the production system. A prototype action may or may not also be a debug action (annotated with `@Debug`, see the section called “@Debug”).

For example:

```
public class Customer {
    public Order placeNewOrder() { ... }
    @Prototype
    public List<Order> listRecentOrders() { ... }
    ...
}
```

See also the `@Exploration` annotation, the section called “@Exploration”

@QueryOnly (deprecated)

Equivalent to using `@ActionSemantics(Of.SAFE)` on an action.

@RegEx

The `@RegEx` annotation may be applied to any string property, or to any parameter within an action method. It can also be applied to any string-based value type. It serves both to validate and potentially to normalise the format of the input. `@Regex` is therefore similar in use to `@Mask` (see the section called “@Mask”) but provides more flexibility.

The syntax is:

```
@RegEx(validation = "regEx string", format = "regEx string",
caseSensitive = <true|false>)
```

Only the first parameter is required; the format defaults to "no formatting", and caseSensitive defaults to false.

For example, on a property:

```
public class Customer {
    @RegEx(validation = "(\\w+\\.)*\\w+@(\\w+\\.)+[A-Za-z]+")
    public String getEmail() {}
    ...
}
```

Or, on a parameter:

```
public class Customer {
    public void updateEmail(
            @RegEx(validation = "(\\w+\\.)*\\w+@(\\w+\\.)+[A-Za-z]+")
            @Named("Email") String email) {
        ...
    }
    ...
)
```

Or, on a value type:

```
@Value(...)
@RegEx(validation = "(\\w+\\.)*\\w+@(\\w+\\.)+[A-Za-z]+")
public class EmailAddress {
```

```
    ...
y}
```

@Resolve

The `@Resolve` annotation is a hint for properties and collections to indicate that a value property should be resolved lazily (rather than eagerly, as usual), or that a reference property or collection should be resolved eagerly (rather than lazily, as usual).

Viewers can use this to present the property/collection in an appropriate manner:

- an `Order`'s `lineItems` collection might initially be rendered expanded form so that the user could see a list of line items immediately when the order is rendered.
- a (reference) property of type `Address` might show the details of the referenced `Address` in a box

At the same time, an object store might use this to defer lazy loading of values that represent blobs or clobs.

For example:

```
public class Order {
    @Resolve(Type.EAGERLY)
    public List<LineItem> getDetails() { ... }

    ...
}
```

For properties and collections there is some similarity between this concept and that of eager-loading as supported by some object stores. Indeed, some object stores may choose use their own specific annotations (eg a JDO default fetch group) in order to infer this semantic.

@Title

The `@Title` annotation is used to indicate which property or properties make up the object title. If more than one property is used, the order can be specified (using the same Dewey-decimal notation as used by `@MemberOrder`) and the string to use between the components can also be specified.

For example:

```
public void Customer {
    @Title(sequence="1.0")
    public String lastName() { ... }
    ...
    @Title(sequence="1.5", prepend=", ")
    public String firstName() { ... }
    ...
    @Title(sequence="1.7", append=".")
    public String midInitial() { ... }
    ...
}
```

could be used to create names of the style "Bloggs, Joe K."

It is also possible to annotate reference properties; in this case the title will return the title of the referenced object (rather than, say, its string representation).

An additional convention for `@Title` properties is that they are hidden in tables (in other words, it implies `@Hidden(where=Where.ALL_TABLES)`. For viewers that support this annotation (for example, the Wicket viewer), this convention excludes any properties whose value is already present in the title column. This convention can be overridden using `@Hidden(where=Where.NOWHERE)`.

@TypeOf

The `@TypeOf` annotation is used to specify the type of elements in a collection, when for whatever reason it is not possible to use generics.

Note

Given that Apache Isis only supports Java 1.6 and later, it's not that obvious what such a reason might be...

For example:

```
public void AccountService {
    @TypeOf(Customer.class)
    public List errantAccounts() {
        return CustomerDatabase.allNewCustomers();
    }
    ...
}
```

@TypicalLength

The `@TypicalLength` annotation indicates the typical length of a `String` property or `String` parameter in an action. It can also be specified for string-based value types.

The information is generally used by the viewing mechanism to determine the space that should be given to that property or parameter in the appropriate view. If the typical length is the same as the `@MaxLength` (see the section called “@MaxLength”) then there is no need to specify `@TypicalLength` as well. If the value specified is zero or negative then it will be ignored.

For example, for a property:

```
public class Customer {
    @MaxLength(30)
    @TypicalLength(20)
    public String getFirstName() { ... }
    public void setFirstName(String firstName) { ... }
}
```

Or, for a parameter:

```
public class CustomerRepository {
    public Customer newCustomer(
            @TypicalLength(20)
            @Named("First Name") String firstName
           ,@TypicalLength(20)
            @Named("Last Name") String lastName) {
        ...
    }
```

```
    ...
}
```

Or, for value type:

```
@Value(...)
@TypicalLength(20)
public class FirstName {
    ...
}
```

@Value

The `@Value` annotation indicates that a class should be treated as a value type rather than as a reference (or entity) type. It does this providing an implementation of a `org.apache.isis.applib.adapters.ValueSemanticsProvider`.

For example:

```
@Value(semanticsProviderClass=ComplexNumberValueSemanticsProvider.class)
public class ComplexNumber {
    ...
}
```

The `ValueSemanticsProvider` allows the framework to interact with the value, parsing strings and displaying as text, and encoding/decoding (for serialization). For more information, see Chapter 10, *Value Types*.

@ViewModel

The `@ViewModel` annotation allows the developer to declare that a domain object is intended to be used as a view model. As such, any changes to its structure are guaranteed to be backwardly compatible.

The annotation was originally introduced to support a requirement of the RestfulObjects viewer which directly expose the domain objects as RESTful representations

For example, a domain object that represents a summary of a `Customer` and their most recent `Order`s might be annotated as:

```
@NotPersistable
@ViewModel
public class CustomerAndOrdersViewModel {
   ...
}
```

Appendix C. `DomainObjectContainer` interface

Provides a single point of contact from domain objects into the *Apache Isis* framework.

The `DomainObjectContainer` interface provides a single point of contact from domain objects into the *Isis* framework. It can also be used as a lightweight general purpose repository during prototyping.

Table C.1. DomainObjectContainer methods (1 of 2)

Category	Method	Description
Object creation	`newTransientInstance(Class<T>)`	Creates new non-persisted object
	`newPersistentInstance(Class<T>)`	Creates new object, will be persisted at end of action
	`newInstance(Class<T>, Object)`	Creates object in state persistence state as that provided
Validation	`isValid(Object)`	whether object is valid
	`validate(Object)`	reason why object is invalid (if any)
Generic Repository	`allInstances(Class<T>)`	All persisted instances of specified type
	`allMatches(Class<T>, Filter<T>)`	All persistenced instances of specified type matching filter
	`allMatches(Class<T>, String)`	All persisted instances with the specified string as their title
	`allMatches(Class<T>, Object)`	All persisted instances matching object (query-by-example)
	`allMatches(Query<T>)`	All instances satisfying the provided query
	`firstMatch(...)`	As for `allMatches(...)`, but returning first instance
	`uniqueMatch(...)`	As for `firstMatch(...)`, but requiring there to be only one match

Table C.2. DomainObjectContainer methods (2 of 2)

Category	Method	Description
Object persistence	`isPersistent(Object)`	whether object is persistent
	`persist(Object)`	persist the transient object
	`persistIfNotAlready(Object)`	persist the object (provided is not already persisted)
	`remove(Object)`	remove the persisted object
	`removeIfNotAlready(Object)`	remove the object (provided is not already transient)
Presentation	`titleOf(Object)`	Returns the title of the object.
Messages and warnings	`informUser(String)`	Inform the user

Category	Method	Description
	`warnUser(String)`	Warn the user about a situation, requiring acknowledgement.
	`raiseError(String)`	Notify user of a serious application error, typically requiring further action on behalf of the user
Security	`getUser()`	The currently-logged on user
Properties	`getProperty(String)`	Value of configuration property
	`getPropertyNames()`	All configuration properties available
Lazy loading, dirty object tracking (*)	`resolve(Object)`	Lazy load object (overloaded to optionally load a property of object)
	`objectChanged(Object)`	Mark object as dirty
Object store control (**)	`flush()`	Flush all pending changes to object store
	`commit()`	Commit all pending changes to object store

Note

(*) generally not necessary to call - performed by bytecode proxies

Note

(**) the use of these methods is discouraged - they are typically used only for tests

Appendix D. Security Classes

A simple set of classes to represent the currently logged on user and their roles.

When the user logs onto an Isis application, the framework constructs a representation of their user and roles using classes from the applib. This allows the application to inspect and act upon those details if required.

The user details are captured in the `org.apache.isis.applib.security.UserMemento` class ("memento" because it is a snapshot of their credentials at the point of logging on). The `UserMemento` class defines the following properties:

- `name` (a `String`)
- collection of roles (as `RoleMemento`)

The org.apache.isis.applib.security.RoleMemento class in turn defines two properties:

- `name` (a `String`)
- `description` (a `String`)

To obtain the current user, the application can call `DomainObjectContainer#getUser()`. For more on the `DomainObjectContainer`, see Appendix C, *`DomainObjectContainer` interface*.

Appendix E. Utility Classes

Simple utility classes for domain objects.

The org.apache.isis.applib.util package has a number of simple utility classes designed to simplify the coding of some common tasks.

Title creation

The `TitleBuffer` utility class is intended to make it easy to construct title strings (returned from the `title()` method). For example, it has overloaded versions of methods called `append()` and `concat()`.

Reason text creation (for disable and validate methods)

There are two different classes provided to help build reasons returned by `disableXxX()` and `validateXxx()` methods:

- the `org.apache.isis.applib.util.ReasonBuffer` helper class
- the `org.apache.isis.applib.util.Reasons` helper class

For example:

```
public class Customer {
    ...
    public String validatePlaceOrder(Product p, int quantity) {
        return Reasons.coalesce(
            whetherCustomerBlacklisted(this),
            whetherProductOutOfStock(p)
        );
    }
}
```

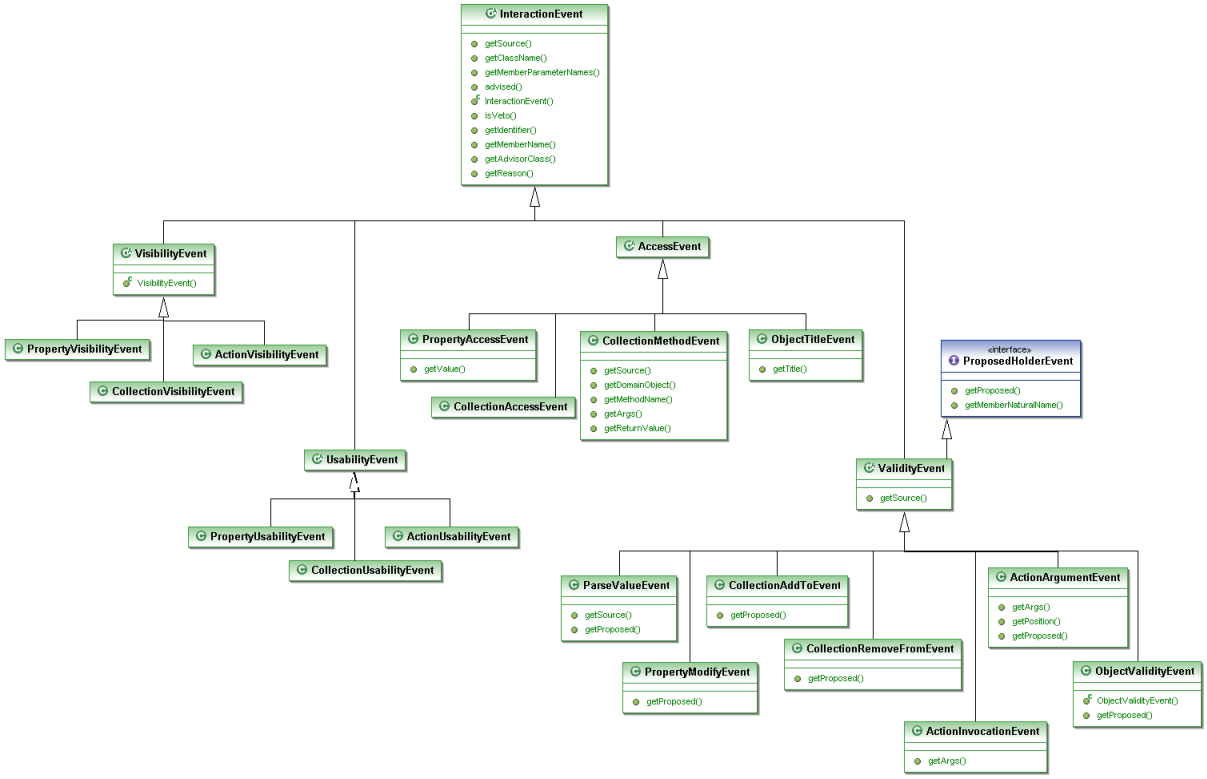
Which you use (if any) is up to you.

Appendix F. Events

The `InteractionEvent` hierarchy.

Although not supported by the default programming model, the applib nevertheless defines an event hierarchy that characterizes all of the different types of interactions that can occur. This is used by the wrapper programming model, and is exploited by the JUnit viewer.

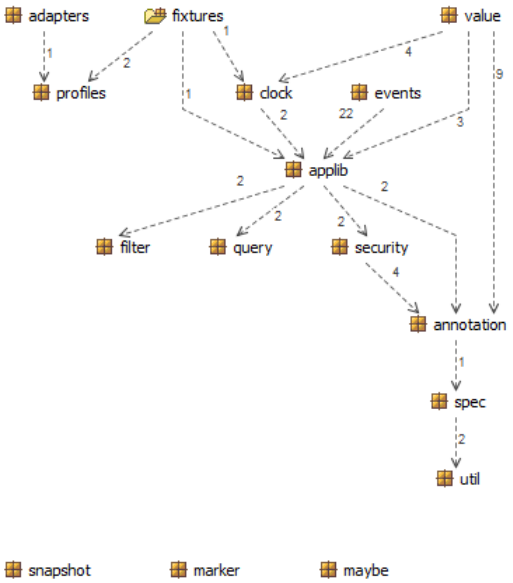
The following UML class diagram shows the hierarchy of events:



Appendix G. Package Dependencies

The dependencies between the packages.

The following diagram shows that the relationship between the different packages that make up the applib (note that there are no cyclic dependencies between the packages):



The following diagram shows the same packages, but from a layered, architecture perspective: