Classes, Methods and Schema

Table of Contents

Chapter 1. Classes, Methods and Schema

This reference guide lists and describes various elements of the the Apache Isis Programming Model, specifically reserved and prefix methods (such as `title()` and `validate⋯()`) and various utility and supporting classes.

It also describes the XSD schema defined by Apache Isis. One use case is for the JAXB serialization of view models.

1.1. Other Guides

Apache Isis documentation is broken out into a number of user, reference and "supporting procedures" guides.

The user guides available are:

- Fundamentals
- Wicket viewer
- Restful Objects viewer
- DataNucleus object store
- Security
- Testing
- Beyond the Basics

The reference guides are:

- Annotations
- Domain Services
- Configuration Properties
- Classes, Methods and Schema (this guide)
- Apache Isis Maven plugin
- Framework Internal Services

The remaining guides are:

- Developers' Guide (how to set up a development environment for Apache Isis and contribute back to the project)
- Committers' Guide (release procedures and related practices)

Chapter 2. Methods

The Apache Isis metamodel is built up from declaratively (ie, annotations) and imperatively, from "supporting" methods and other reserved methods.

This chapter documents the supporting methods and the reserved methods. It also documents (separately) the reserved methods that act as callback hooks into the persistence lifecycle of domain entities.

2.1. Supporting Method Prefixes

Supporting methods are those that are associated with properties, collections and actions, providing additional imperative business rule checking and behaviour to be performed when the user interacts with those object members.

This association is performed by name matching. Thus, a property called "firstName", derived from a method `getFirstName()` may have supporting methods `hideFirstName()`, `disableFirstName()` and `validateFirstName()`. Supporting methods are, therefore, each characterized by their own particular prefix.

Using name matching to associate supporting methods generally works very well, but of course if an object member's method is renamed, there's always the risk that the developer forgets to rename the supporting method; the supporting methods become "orphaned".

Apache Isis checks for this automatically, and will fail-fast (fail to boot) if any orphaned methods are located. A suitable error message is logged so that the issue can be easily diagnosed.

The table below lists the method prefixes that are recognized as part of Apache Isis' default programming model.

Table 1. Recognized Method Prefixes

Prefix	Object	Property	Collection	Action	Action Param	Description
`addTo···()`			Y			add object to a collection [NOTE] ==== Directly mutable collections are not currently supported by the Wicket viewer. ==== See also `removeFrom···()` `
`autoComplete···()`		Y			Y	Return a list of matching elements for a property or an action parameter. Alternatively, can specify for a class using `@DomainObject #autoCompleteRepository` See also `choices···()`

Prefix	Object	Property	Collection	Action	Action Param	Description
`choices···()`		Y			Y	Provide list of choices for a property or action parameter. See also `autoComplete···()`.
`clear···()`		Y				Clear a property (set it to null). Allows business logic to be placed apart from the setter. See also `modify···()`
`default···()`		Y			Y	Default value for a property or an action parameter.
`disable···()`	Y	Y	Y	Y		Disables (makes read-only) a property, a collection or an action.
`get···()`		Y	Y			Access the value of a property or collection. See also `set···()`.
`hide···()`		Y	Y	Y		Hides a property, a collection or an action.
`modify···()`		Y				Modify a property (set it to a non-null) value. Allows business logic to be placed apart from the setter. See also `clear···()`.
`removeFrom···()`			Y			remove object from a collection. [NOTE] ==== Directly mutable collections are not currently supported by the Wicket viewer. ==== See also `addTo···()`
`set···()`		Y	Y			Sets the value of a property or a collection.
`validate···()`	Y			Y	Y	Check that a proposed value of a property or a set of action parameters or a single action parameter is valid. See also `validateAddTo···()` and `validateRemoveFrom···()` to validate modifications to collections.
`validate AddTo···()`		Y				Check that a proposed object to add to a collection is valid. [NOTE] ==== Directly mutable collections are not currently supported by the Wicket viewer. ==== See also `validateRemoveFrom···()`, and `validate···()` for properties and actions.

Prefix	Object	Property	Collection	Action	Action Param	Description
validateRemoveFrom…()		Y				Check that a proposed object to remove from a collection is valid. [NOTE] ==== Directly mutable collections are not currently supported by the Wicket viewer. ==== See also validateAddTo…(), and validate…() for properties and actions.

2.1.1. addTo…() (deprecated)

The addTo…() supporting method is called whenever an object is added to a collection. Its purpose is to allow additional business logic to be performed.

Directly mutable collections are not currently supported by the Wicket viewer. The suggested workaround is to simply define an action.

For example:

```
public class LibraryMember {
    public SortedSet<Book> getBorrowed() { ... }
    public void setBorrowed(SortedSet<Book> borrowed) { ... }
    public void addToBorrowed(Book book) {
        getBorrowed().add(book);                                                    ①
        reminderService.addReminder(this, book, clock.today().plusDays(21));  ②
    }
    public void removeFromBorrowed(Book book) { ... }
    ...
}
```

① update the collection

② perform some additional business logic

See also removeFrom…()`

2.1.2. autoComplete…()

The autoComplete…() supporting method is called for action parameters and for properties to find objects from a drop-down list box. The use case is when the number of candidate objects is expected to be large, so the user is required to enter some characters to narrow the search down.

If the number of candidate objects is comparatively small, then use choices…() supporting method instead.

The signature of the supporting method depends on whether it is for a parameter or a property.

Parameters

For an action parameter in (0-based) position *N*, and of type `T`, the signature is:

```
public List<T> autoCompleteNXxx(String search) { ... }
```

It is also valid to return `T[]`, a `Set<T>` or a `Collection<T>`.

For example:

```
public class ShoppingCartItem {
    @Property(editing=Editing.DISABLED)
    public Product getProduct() { ... }
    public void setProduct(Product product) { ... }

    @Property(editing=Editing.DISABLED)
    public int getQuantity() { ... }
    public void setQuantity(int quantity) { ... }

    @Action(semantics=SemanticsOf.IDEMPOTENT)
    public ShoppingCartItem updateProduct(
        Product product,
        @ParameterLayout(named="Quantity")
        final int quantity) {
        setProduct(product);
        setQuantity(quantity);
    }
    public Collection<Product> autoComplete0UpdateProduct(  ①
        @MinLength(3) String search                         ②
    ) {
        ...
    }
    ...
}
```

① `product` is the 0th argument of the action.

② the `@MinLength` annotation specifies the minimum number of characters that must be entered before a search is performed for matching objects

Properties

For a property of type `T`, the signature is:

```
public List<T> autoCompleteXxx(String search) { ... }
```

(As for action parameters) it is also valid to return `T[]`, a `Set<T>` or a `Collection<T>`.

For example:

```
public class ShoppingCartItem {
    public Product getProduct() { ... }
    public void setProduct(Product product) { ... }

    public Collection<Product> autoCompleteProduct(
        @MinLength(3) String search                    ①
    ) {
        ...
    }
    ...
}
```

① the `@MinLength` annotation specifies the minimum number of characters that must be entered before a search is performed for matching objects

2.1.3. `choices…()`

The `choices…()` supporting method is called for both action parameters and for properties, to find objects from a drop-down list box. Unlike `autoComplete…()`, the use case is when the number of objects is comparatively small and can be selected from a drop-down without any additional filtering.

The signature of the supporting method depends on whether it is for an action parameter or a property.

Parameters

For an action parameter in (0-based) position *N*, and of type `T`, the signature is:

```
public Collection<T> choicesNXxx() { ... }
```

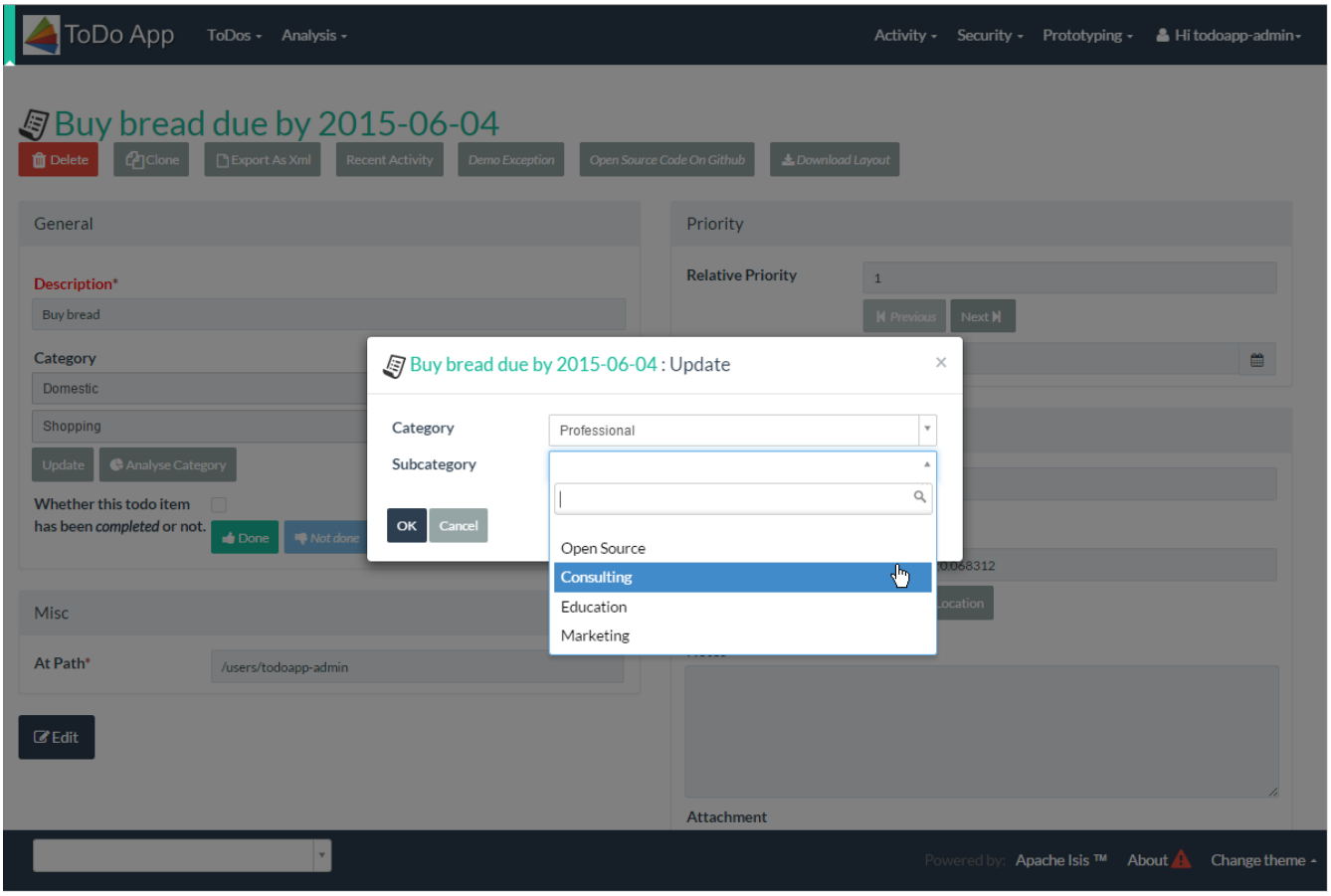
For example:

```
public class ShoppingCartItem {
    @Property(editing=Editing.DISABLED)
    public Product getProduct() { ... }
    public void setProduct(Product product) { ... }

    @Property(editing=Editing.DISABLED)
    public int getQuantity() { ... }
    public void setQuantity(int quantity) { ... }

    @Action(semantics=SemanticsOf.IDEMPOTENT)
    public ShoppingCartItem updateProduct(
        Product product,
        @ParameterLayout(named="Quantity")
        final Integer quantity) {
        setProduct(product);
        setQuantity(quantity);
    }
    public Collection<Integer> choices1UpdateProduct() {
        return Arrays.asList(1,2,3,5,10,25,50,100);
    }
    ...
}
```

Dependent Choices

Action parameters also support the notion of dependent choices, whereby the list of choices is dependent upon the value of some other argument.

An example can be found in the (non-ASF) Isis addons' todoapp, whereby `ToDoItem`s are categorized and then can also be subcategorized:



This functionality is actually implemented as a contributed action, so the code for this is:

```
@DomainService(nature = NatureOfService.VIEW_CONTRIBUTIONS_ONLY)
public class UpdateCategoryContributions ... {
    @ActionLayout(
            describedAs = "Update category and subcategory"
    )
    @Action(semantics = SemanticsOf.IDEMPOTENT)
    public Categorized updateCategory(
            final Categorized item,                                    ①
            final Category category,
            @Parameter(optionality = Optionality.OPTIONAL)
            final Subcategory subcategory) {
        item.setCategory(category);
        item.setSubcategory(subcategory);
        return item;
    }
    public List<Subcategory> choices2UpdateCategory(                   ②
            final Categorized item,                                    ③
            final Category category) {                                 ④
        return Subcategory.listFor(category);
    }
    ...
}
```

① `ToDoItem` implements `Categorized`

② subcategory is the 2-th argument (0-based)

③ the item contributed to

④ the category selected

Dependent choices are not restricted to enums, however. Going back to the shopping cart example shown above, the choices for the `quantity` parameter could be dependent upon the selected `Product`:

```
public class ShoppingCartItem {
    ...
    @Action(semantics=SemanticsOf.IDEMPOTENT)
    public ShoppingCartItem updateProduct(
        Product product,
        @ParameterLayout(named="Quantity")
        final Integer quantity) {
        setProduct(product);
        setQuantity(quantity);
    }
    public Collection<Integer> choices1UpdateProduct(Product product) {
        return productService.quantityChoicesFor(product);                ①
    }
    ...
}
```

① `productService` is a (fictitous) injected service that knows what the quantity choices should be for any given product

Properties

For a property of type `T`, the signature is:

```
public Collection<T> choicesXxx() { ... }
```

For example:

```
public class ShoppingCartItem {
    public Product getProduct() { ... }
    public void setProduct(Product product) { ... }

    public Collection<Product> choicesProduct() {
        ...
    }
```

2.1.4. `clear···()` (deprecated)

The `clear···()` supporting method is called — instead of the setter — whenever an (optional) property is to be set to `null`. Its purpose is to allow additional business logic to be performed.

DataNucleus' smart handling of setters means that this supporting methods are in essence redundant, and so should be considered deprecated.

For example:

```
public class LibraryMember {
    public Title getFavoriteTitle() { ... }
    public void setFavoriteTitle(Title title) { ... }
    public void modifyFavoriteTitle(Title title) { ... }
    public void clearFavoriteTitle() {
        if(getTitle() == null) { return; }
        setFavoriteTitle(null);                          ①
        titleFavoritesService.decrement(title);          ②
    }
    ...
}
```

① update the property

② perform some additional business logic

See also `modify…()`

2.1.5. `default…()`

The `default…()` supporting method is called for action parameters to return the initial argument value. This may be some sensible default (eg today's date, or 0 or 1), or — for an action that is modifying the state of an object — might default to the current value of a corresponding property.

The method is *also* called for properties in the case when an object is newly instantiated using `DomainObjectContainer#newTransientInstance(…)`. This is a much less common use case. If a default is not specified then properties are initialized to a default based on their type (eg 0 or `false`).

The signature of the supporting method depends on whether it is for an action parameter or a property.

Parameters

For an action parameter in (0-based position n), and of type `T`, the signature is:

```
public T defaultNXxx() { ... }
```

For example:

```
public class ShoppingCartItem {
    @Property(editing=Editing.DISABLED)
    public Product getProduct() { ... }
    public void setProduct(Product product) { ... }

    @Property(editing=Editing.DISABLED)
    public int getQuantity() { ... }
    public void setQuantity(int quantity) { ... }

    @Action(semantics=SemanticsOf.IDEMPOTENT)
    public ShoppingCartItem updateProduct(
        Product product,
        @ParameterLayout(named="Quantity")
        final Integer quantity) {
        setProduct(product);
        setQuantity(quantity);
    }
    public Product default0UpdateProduct() {    ①
        return getProduct();
    }
    public int default1UpdateProduct() {        ②
        return getQuantity();
    }
    ...
}
```

① default the 0-th parameter using the current value of the `product` property

② default the 1-th parameter using the current value of the `quantity` property

Defaults are also supported (of course) for contributed actions. For example, here is a contributed action for updating category/subcategory of the (non-ASF) Isis addons' todoapp:

```
@DomainService(nature = NatureOfService.VIEW_CONTRIBUTIONS_ONLY)
public class UpdateCategoryContributions ... {
    @ActionLayout(
            describedAs = "Update category and subcategory"
    )
    @Action(semantics = SemanticsOf.IDEMPOTENT)
    public Categorized updateCategory(
            final Categorized item,                                    ①
            final Category category,
            @Parameter(optionality = Optionality.OPTIONAL)
            final Subcategory subcategory) {
        item.setCategory(category);
        item.setSubcategory(subcategory);
        return item;
    }
    public Category default1UpdateCategory(                            ②
            final Categorized item) {
        return item != null? item.getCategory(): null;
    }
    public Subcategory default2UpdateCategory(                         ③
            final Categorized item) {
        return item != null? item.getSubcategory(): null;
    }
}
```

① `ToDoItem` implements `Categorized`

② defaults the 1-th parameter using the item's `category` property

③ defaults the 2-th parameter using the item's `subcategory` property

Properties

For a property of type `T`, the signature is:

```
public T defaultXxx() { ... }
```

For example:

```
public class ShoppingCartItem {
    public int getQuantity() { ... }
    public void setQuantity(int quantity) { ... }

    public int defaultProduct() {
        return 1;
    }
```

Alternatives

There are, in fact, two other ways to set properties of a newly instantiated object to default values.

The first is to use the `created()` callback, called by the framework when `DomainObjectContainer#newTransientInstance(…)` is called. This method is called after any dependencies have been injected into the service.

The second is more straightforward: simply initialize properties in the constructor. However, this cannot use any injected services as they will not have been initialized.

2.1.6. `disable…()`

The `disable…()` supporting method is called for properties, collections and actions. It allows the modification of the property/collection to be vetoed (ie made read-only) and to prevent the invocation of the action ("grey it out").

Directly mutable collections are not currently supported by the Wicket viewer; they are always implicitly disabled.

Typically modification/invocation is vetoed based on the state of the domain object being interacted with, though it could be any reason at all (eg the current date/time of the interaction, or the state of some other related data such as stock levels, or the identity of the calling user).

The reason for vetoing a modification/invocation is normally returned as a string. However, Apache Isis' i18n support extends this so that reasons can be internationalized.

The signature of the supporting method is simply:

```
public String disableXxx() { ... }
```

where the returned string is the reason the property cannot be edited, or the action invoked.

For i18n, the supporting method returns a `TranslatableString`:

```
public TranslatableString disableXxx() { ... }
```

The returned string is then automatically translated to the locale of the current user.

For example, to disable an action:

```
public class Customer {
    public boolean isBlacklisted() { ... }

    public Order placeOrder(
            final Product product,
            @ParameterLayout(named="Quantity")
            final int quantity) {
        ...
    }
    public String disablePlaceOrder() {
        return isBlacklisted()
                    ? "Blacklisted customers cannot place orders"
                    : null;
    }
    ...
}
```

Or, to disable a property:

```
public class Customer {
    public boolean isBlacklisted() { ... }

    public BigDecimal getCreditLimit() { ... }
    public void setCreditLimit(BigDecimal creditLimit) { ... }
    public String disableCreditLimit() {
        return isBlacklisted()
                    ? "Cannot change credit limit for blacklisted customers"
                    : null;
    }
    ...
}
```

In the case of actions, the framework will also search for supporting method that has the exact same parameter types as the action itself. Enabling `isis.reflector.validator.noParamsOnly` configuration property switches this off, so that the framework will only search for supporting method with no parameters.

Note that enabling this configuration property in effect means that mixins must be used instead of contributed services (because contributed actions are the one case where the value of a parameter to a supporting method may be non-null).

2.1.7. `get···()`

The `get···()` prefix is simply the normal JavaBean getter prefix that denotes properties or collections.

When Apache Isis builds its metamodel, it first searches for the getter methods, characterizing them as either properties or collections based on the return type. It then refines the metamodel based on the presence of annotations and supporting methods.

All remaining `public` methods (that do not use one of the Apache Isis prefixes) are interpreted as actions.

Any methods "left over" that *do* use one of the Apache Isis prefixes, are interpreted to be orphaned. Apache Isis "fails-fast" and will not boot, instead printing an error message to the log so that the issue can be easily diagnosed.

See also `set…()`.

2.1.8. `hide…()`

The `hide…()` supporting method is called for properties, collections and actions. It allows the property/collection to be completely hidden from view.

It's comparatively rare for properties or collections to be imperatively hidden from view, but actions are sometimes hidden or shown visible (as opposed to being just disabled, ie greyed out).

The signature of the supporting method is simply:

```
public boolean hideXxx() { ... }
```

Returning `true` will hide the property, collection or action, returning `false` leaves it visible.

For example, to hide an action:

```
public class Customer {
    public boolean isBlacklisted() { ... }

    public Order placeOrder(
            final Product product,
            @ParameterLayout(named="Quantity")
            final int quantity) {
        ...
    }
    public boolean hidePlaceOrder() {
        return isBlacklisted();
    }
    ...
}
```

Or, to hide a property:

```
public class Customer {
    public boolean isBlacklisted() { ... }

    public BigDecimal getCreditLimit() { ... }
    public void setCreditLimit(BigDecimal creditLimit) { ... }
    public boolean hideCreditLimit() {
        return isBlacklisted();
    }
    ...
}
```

In the case of actions, the framework will also search for supporting method that has the exact same parameter types as the action itself. Enabling `isis.reflector.validator.noParamsOnly` configuration property switches this off, so that the framework will only search for supporting method with no parameters.

Note that enabling this configuration property in effect means that mixins must be used instead of contributed services (because contributed actions are the one case where the value of a parameter to a supporting method may be non-null).

2.1.9. `modify…()` (deprecated)

The `modify…()` supporting method is called — instead of the setter — whenever a property has been set to be set to a new value. Its purpose is to allow additional business logic to be performed.

DataNucleus' smart handling of setters means that this supporting methods are in essence redundant, and so should be considered deprecated.

For example:

```
public class LibraryMember {
    public Title getFavoriteTitle() { ... }
    public void setFavoriteTitle(Title title) { ... }
    public void modifyFavoriteTitle(Title title) {
        if(getTitle() != null) {
            titleFavoritesService.decrement(getTitle());    ①
        }
        setFavoriteTitle(title);                            ②
        titleFavoritesService.decrement(title);             ③
    }
    public void clearFavoriteTitle() { ... }
    ...
}
```

① perform some additional business logic

② update the property

③ perform some additional business logic

See also `clear…()`

2.1.10. `removeFrom…()` (deprecated)

The `removeFrom…()` supporting method is called whenever an object is removed from a collection. Its purpose is to allow additional business logic to be performed.

Directly mutable collections are not currently supported by the Wicket viewer. The suggested workaround is to simply define an action.

For example:

```
public class LibraryMember {
    public SortedSet<Book> getBorrowed() { ... }
    public void setBorrowed(SortedSet<Book> borrowed) { ... }
    public void addToBorrowed(Book book) { ... }
    public void removeFromBorrowed(Book book) {
        getBorrowed().remove(book);                         ①
        reminderService.removeReminder(this, book);         ②
    }
    ...
}
```

① update the collection

② perform some additional business logic

See also `addTo…()`

2.1.11. `set…()`

The `set…()` prefix is simply the normal JavaBean setter prefix that denotes writeable properties or collections.

See also `get…()`.

2.1.12. `validate…()`

The `validate…()` supporting method is called for properties, actions and action parameters. It allows the proposed new value for a property to be rejected, or the proposed argument of an action parameter to be rejected, or to reject a whole set of action arguments for an actio invocation.

The reason for vetoing a modification/invocation is normally returned as a string. However, Apache Isis' i18n support extends this so that reasons can be internationalized if required.

Action Parameter

For an action parameter in (0-based) position *N*, and of type `T`, the signature is:

```
public String validateNXxx(T proposed) { ... }
```

where the returned string is the reason why the argument is rejected (or `null` if not vetoed).

For i18n, the supporting method returns a `TranslatableString`:

```
public TranslatableString validateNXxx(T proposed) { ... }
```

The returned string is then automatically translated to the locale of the current user.

For example:

```
public class Customer {
    public Order placeOrder(
            final Product product,
            @ParameterLayout(named="Quantity")
            final int quantity) {
        ...
    }
    public String validate0PlaceOrder(
            final Product product) {
        return product.isDiscontinued()
                    ? "Product has been discontinued"
                    : null;
    }
    ...
}
```

Action Parameter Set

In addition to validating a single single action argument, it is also possible to validate a complete set of action arguments. The signature is:

```
public String validateXxx(...) { ... }
```

where the returned string is the reason why the argument is rejected (or `null` if not vetoed), and the supporting method takes the same parameter types as the action itself.

For i18n, the supporting method returns a `TranslatableString`:

```
public TranslatableString validateXxx(...) { ... }
```

For example:

```
public class Customer {
    public Order placeOrder(
            final Product product,
            @ParameterLayout(named="Quantity")
            final int quantity) {
        ...
    }
    public String validatePlaceOrder(
            final Product product,
            final int quantity) {
        return quantity > product.getOrderLimit()
                    ? "May not order more than " + product.getOrderLimit() + " items
for this product"
                    : null;
    }
    ...
}
```

Properties

For properties of type `T` the signature of the supporting method is:

```
public String validateXxx(T proposed) { ... }
```

where the returned string is the reason the modification is vetoed (or `null` if not vetoed).

For i18n, the supporting method returns a `TranslatableString`:

```
public TranslatableString validateXxx(T proposed) { ... }
```

For example:

```
public class Customer {
    public BigDecimal getCreditLimit() { ... }
    public void setCreditLimit(BigDecimal creditLimit) { ... }
    public validateCreditLimit(BigDecimal creditLimit) {
        return creditLimit.compareTo(BigDecimal.ZERO) < 0
                    ? "Credit limit cannot be negative"
                    : null;
    }
    ...
}
```

2.1.13. `validateAddTo…()` (deprecated)

The `validateAddTo…()` supporting method is called whenever an object is to be added to a

collection. Its purpose is to validate the proposed object and possibly veto the change.

Directly mutable collections are not currently supported by the Wicket viewer. The suggested workaround is to simply define an action.

The signature of the supporting method for a collection with element type `E` is:

```
public String validateAddToXxx(E element) { ... }
```

where the returned string is the reason the collection modification invocation is vetoed (or `null` if not vetoed). Apache Isis' i18n support extends this so that reasons can be internationalized if required.

For example:

```
public class LibraryMember {
    public SortedSet<Book> getBorrowed() { ... }
    public void setBorrowed(SortedSet<Book> borrowed) { ... }
    public String validateAddToBorrowed(Book book) {
        return book.isReference()? "Reference books cannot be borrowed": null;
    }
    public void validateRemoveFromBorrowed(Book book) { ... }
    ...
}
```

See also `addTo···()` and `validateRemoveFrom···()`

2.1.14. `validateRemoveFrom···()` (deprecated)

The `validateRemoveFrom···()` supporting method is called whenever an object is to be removed from a collection. Its purpose is to validate the proposed object removal and possibly veto the change.

Directly mutable collections are not currently supported by the Wicket viewer. The suggested workaround is to simply define an action.

The signature of the supporting method for a collection with element type `E` is:

```
public String validateRemoveFromXxx(E element) { ... }
```

where the returned string is the reason the collection modification invocation is vetoed (or `null` if not vetoed). Apache Isis' i18n support extends this so that reasons can be internationalized if required.

For example:

```
public class LibraryMember {
    public SortedSet<Book> getBorrowed() { ... }
    public void setBorrowed(SortedSet<Book> borrowed) { ... }
    public String validateAddToBorrowed(Book book) { ... }
    public void validateRemoveFromBorrowed(Book book) {
        return !book.hasBeenReadBy(this)? "You didn't read this book yet": null;
    }
    ...
}
```

See also `removeFrom…()` and `validateAddTo…()`

2.2. Reserved Methods

The table below lists the reserved methods that are recognized as part of Apache Isis' default programming model.

Table 2. Reserved Methods

Method	Description
`cssClass()`	Provides a CSS class for this object instance. In conjunction with `application.css`, can therefore provide custom styling of an object instance wherever it is rendered. See also `title()` and `iconName()`.
`disable(…)`	Disable all or some of an object's properties
`getId()`	Provides an optional unique identifier of a service. If not provided, the service's fully-qualified class name is used.
`hide(…)`	Hide all or some of an object's properties
`iconName()`	Provides the name of the image to render, usually alongside the title, to represent the object. If not provided, then the class name is used to locate an image. See also `title()` and `cssClass()`
`title()`	Provides a title for the object. See also `iconName()` and `cssClass()`
`validate()`	Validate the object's state prior to persisting.

2.2.1. `cssClass()`

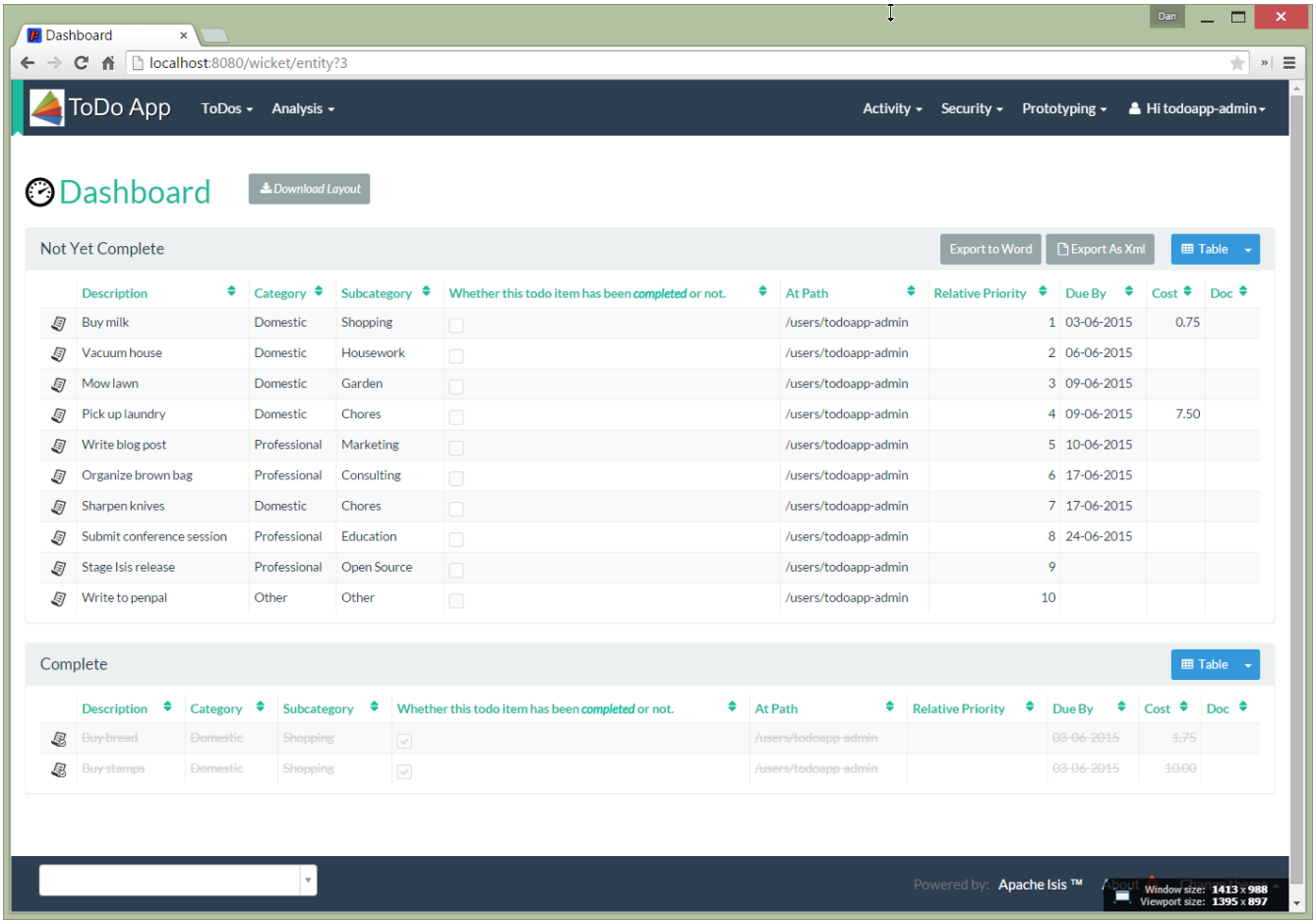
The `cssClass()` returns a CSS class for a particular object instance.

The Wicket viewer wraps the object's representation in a containing `<div>` with the class added. This is done both for rendering the object either in a table or when rendering the object on its own page.

In conjunction with `application.css`, can therefore provide custom styling of an object instance

wherever it is rendered.

For example, the (non-ASF) Isis addons' todoapp uses this technique to add a strikethrough for completed todo items. This is shown on the home page:



The code to accomplish this is straightforward:

```
public class ToDoItem ... {
    public String cssClass() {
        return !isComplete() ? "todo" : "done";
    }
    ...
}
```

In the application.css, the following styles were then added:

```
tr.todo {
}
tr.done {
    text-decoration: line-through;
    color: #d3d3d3;
}
```

See also `title()` and `iconName()`.

2.2.2. disabled()

One use case that Apache Isis supports is that of a domain object with a lifecycle whereby at some stage it should become immutable: all its properties/collections should be disabled, and/or its actions become not invokable.

It would be painful to have to write a separate `disable…()` method for each and every member, so instead Isis allows a single `disable…(…)` method to be implemented that is applied to all members.

The signature of the method is:

```
public String disabled(Identifier.Type identifierType) { ... }
```

where `Identifier.Type` is part of the Isis applib (nested static class of `o.a.i.applib.Identifier`) to distinguish between an interaction with an action, a property or an action.

Note that Apache Isis' i18n support extends this so that the returned reason can also be internationalized.

For example:

```
public String disabled(Identifier.Type identifierType) {
    return !calendarService.isOfficeHours(clock.today()
            ? "Cannot modify objects outside of office hours"
            : null;
}
```

See also the similar methods to `hide()` object members en-masse.

Alternatives

An alternative design — and one that could be easily argued is actually more flexible — is to leverage domain events with vetoing subscribers.

With this approach we define, for a given domain class, a base `PropertyDomainEvent`, `CollectionDomainEvent` and `ActionDomainEvent`. A good pattern is to make these nested static classes. For example:

```
public class ToDoItem ... {
    public static abstract class PropertyDomainEvent<T>
            extends ToDoAppDomainModule.PropertyDomainEvent<ToDoItem, T> {
        ...
    }
    ...
}
```

where in turn:

```
public final class ToDoAppDomainModule {
    private ToDoAppDomainModule(){}
    public abstract static class PropertyDomainEvent<S,T>
            extends org.apache.isis.applib.services.eventbus.PropertyDomainEvent<S,T>
{
        ...
    }
    ...
}
```

Then, each property/collection/action emits either these base domain events or their own subclass:

```
public class ToDoItem ... {
    public static class DescriptionDomainEvent
            extends PropertyDomainEvent<String> {
        ...
    }
    @Property(
        domainEvent = DescriptionDomainEvent.class
    )
    public String getDescription() { ... }
    ...
}
```

A vetoing subscriber can then subscribe to the domain events and veto access, eg:

```
@DomainObject
public class VetoOutOfOfficeHours {
    @Subscribe
    public void on(ToDoItem.PropertyDomainEvent ev) {
        if(!calendarService.isOfficeHours(clock.today()) {
            ev.veto("Cannot modify objects outside of office hours");
        }
    }
    ...
}
```

Obviously there's an awful lot more boilerplate here, but there's also a lot more flexibility.

2.2.3. `getId()`

The `getId()` method applies only to domain services, and is used to provide a unique alias for the domain service's class name.

This value is used internally to generate a string representation of an service identity (the `Oid`). This can appear in several contexts, including:

- as the value of `Bookmark#getObjectType()` and in the `toString()` value of `Bookmark` (see `BookmarkService`)
- in the serialization of `OidDto` in the command and interaction schemas
- in the URLs of the RestfulObjects viewer
- in the URLs of the Wicket viewer (specifically, for bookmarked actions)

Example

For example:

```
@DomainService
public class OrderMenu {
    ...
    public String getId() { return "orders.OrderMenu"; }
}
```

Precedence

The rules of precedence are:

1. `@DomainService#objectType()`
2. `getId()`
3. The fully qualified class name.

This might be obvious, but to make explicit: we recommend that you always specify an object type for your domain services.

Otherwise, if you refactor your code (change class name or move package), then any externally held references to the OID of the service will break. At best this will require a data migration in the database; at worst it could cause external clients accessing data through the Restful Objects viewer to break.

If the object type is not unique across all domain classes then the framework will fail-fast and fail to boot. An error message will be printed in the log to help you determine which classes have duplicate object tyoes.

2.2.4. `hide()`

One use case that Apache Isis supports is that of a domain object with a lifecycle whereby at some stage some number of the object's members should be hidden. For example, for an object that at some stage is logically immutable, we might want to make all its properties/collections unmodifiable and hide all its actions.

While we could write a separate `hide…()` method for each and every action, this could become painful. So instead Isis allows a single `hide…(…)` method to be implemented that is applied to all members.

The signature of the method is:

```
public boolean hide(Identifier.Type identifierType) { ... }
```

where `Identifier.Type` is part of the Isis applib (nested static class of `o.a.i.applib.Identifier`) to distinguish between an interaction with an action, a property or an action.

For example:

```
public boolean hide(Identifier.Type identifierType) {
    return identifierType == Identifier.Type.ACTION && isFrozen();
}
```

See also the similar method to `disable()` object members en-masse.

Alternatives

An alternative design — and one that could be easily argued is actually more flexible — is to leverage domain events with vetoing subscribers.

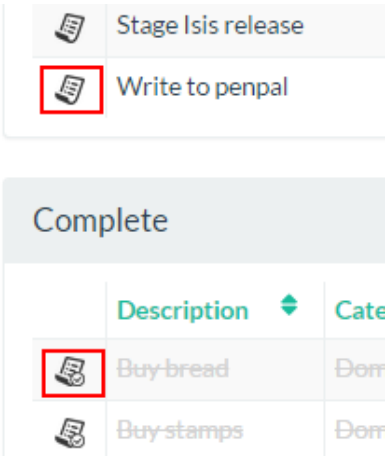
There is further discussion on this approach in here.

2.2.5. `iconName()`

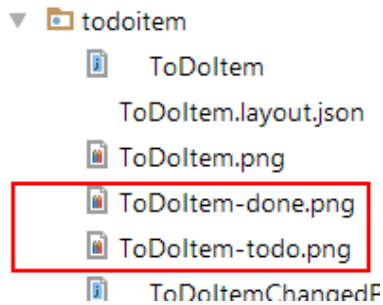
Every object is represented by an icon; this is based on the domain object's simple name. The Wicket viewer searches for the image in the same package as the `.class` file for the domain object or in the `images` package. It will find any matching name and one of the followign suffexes `png`, `gif`, `jpeg`, `jpg`, `svg`. If none is found, then `Default.png` will be used as fallback.

The `iconName()` allows the icon that to be used to change for individual object instances. These are usually quite subtle, for example to reflect the particular status of an object. The value returned by the `iconName()` method is added as a suffix to the base icon name.

For example, the (non-ASF) Isis addons' todoapp uses this technique to add an overlay for todo items that have been completed:



The screenshot below shows the location of these png icon files:



The code to accomplish this is straightforward:

```
public class ToDoItem ... {
    public String iconName() {
        return !isComplete() ? "todo" : "done";
    }
    ...
}
```

See also `title()` and `cssClass()`

2.2.6. `title()`

Every object is represented by a title. This appears both as a main header for the object when viewed as well as being used as a hyperlink within properties and collections. It therefore must contain enough information for the end-user to distinguish the object from any others.

This is most commonly done by including some unique key within the title, for example a customer's SSN, or an order number, and so forth. However note that Apache Isis itself does *not* require the title to be unique; it is merely recommended in most cases.

An object's title can be constructed in various ways, but the most flexible is to use the `title()` method. The signature of this method is usually:

```
public String title() { ... }
```

Note that Apache Isis' i18n support extends this so that titles can also be internationalized.

For example, the (non-ASF) Isis addons' todoapp uses this technique to add an overlay for todo items that have been completed:

```
public String title() {
    final TitleBuffer buf = new TitleBuffer();                          ①
    buf.append(getDescription());
    if (isComplete()) {                                                 ②
        buf.append("- Completed!");
    } else {
        try {
            final LocalDate dueBy = wrapperFactory.wrap(this).getDueBy();   ③
            if (dueBy != null) {
                buf.append(" due by", dueBy);
            }
        } catch(final HiddenException ignored) {                        ④
        }
    }
    return buf.toString();
}
```

① simple tility class to help construct the title string

② imperative conditional logic

③ using the `WrapperFactory` to determine if the `dueBy` field is visible for this user ...

④ ... but ignore if not

As the example above shows, the implementation can be as complex as you like.

In many cases, though, you may be able to use the `@Title` annotation.

See also `iconName()` and `cssClass()`

2.2.7. `validate()`

The `validate()` method is used to specify that invariants pertaining to an object's state are enforced.

(As of 1.8.0) there are known limitations with this functionality. Invariants are enforced when an object is initially created and when it is edited, however invariants are currently *not* enforced if an action is invoked.

The signature of the method is:

```
public String validate() { ... }
```

where the returned string is the reason that the invocation is vetoed.

Note that Apache Isis' i18n support extends this so that the returned reason can also be internationalized.

2.3. Lifecycle Methods

The lifecycle callback methods notify a domain entity about its interaction within the persistence lifecycle. For example, the entity is notified immediately prior to being persisted, or when it is about to be updated.

Note that these callbacks are fired by Apache Isis rather than JDO. In the future we may deprecate them because there are better mechanisms available using listeners/subscribers:

- in Isis 1.9.0 and earlier, you may therefore want to consider using the JDO API directly to set up a lifecycle listener; see here for further discussion.
- alternatively, you can use a subscriber for the lifecycle events fired in Isis.

The lifecycle callback methods supported by Isis are:

Table 3. Lifecycle methods (partial support)

Method	Description
`created()`	called when an object has just been created using `newTransientInstance()`
`loaded()`	called when a (persistent) object has just been loaded from the object store.
`persisted()`	called when object has just been persisted from the object store.
`persisting()`	called when a (not-yet-persistent) object is just about to be persisted from the object store
`removed()`	called when a (persistent) object has just been deleted from the object store
`removing()`	called when a (persistent) object is just about to be deleted from the object store
`updated()`	called when a (persistent) object has just been updated in the object store
`updating()`	called when a (persistent) object is just about to be updated in the object store

Some lifecycle methods have been deprecated:

Table 4. Deprecated lifecycle methods

Method	Notes
`deleted()`	Replaced by `removed()`
`deleting()`	Replaced by `removing()`
`loading()`	callback for when the (persistent) object is just about to be loaded from the object store. [WARNING] ==== This method is never called. ====
`saved()`	Replaced by `persisted()`
`saving()`	Replaced by `persisting()`

2.3.1. `created()`

The `created()` lifecycle callback method is called when an object has just been created using `newTransientInstance()`

Alternatively, consider using a event bus subscriber on the `ObjectCreatedEvent`.

2.3.2. `loaded()`

The `loaded()` lifecycle callback method is called when a (persistent) object has just been loaded from the object store.

Alternatively, consider using a event bus subscriber on the `ObjectLoadedEvent`.

2.3.3. `persisted()`

The `persisted()` lifecycle callback method is called when object has just been persisted from the object store.

See also `persisting()`.

Alternatively, consider using a event bus subscriber on the `ObjectPersistedEvent`.

2.3.4. `persisting()`

The `persisting()` lifecycle callback method is called when a (not-yet-persistent) object is just about to be persisted from the object store

See also `persisted()`.

Alternatively, consider using a event bus subscriber on the `ObjectPersistingEvent`.

2.3.5. `removed()`

The `removed()` lifecycle callback method is called when a (persistent) object has just been deleted from the object store

See also `removing()`.

Alternatively, consider using a event bus subscriber on the `ObjectRemovedEvent`.

2.3.6. `removing()`

The `removing()` lifecycle callback method is called when a (persistent) object is just about to be deleted from the object store

See also `removed()`.

Alternatively, consider using a event bus subscriber on the `ObjectRemovingEvent`.

2.3.7. `updated()`

The `updated()` lifecycle callback method is called when a (persistent) object has just been updated in the object store

See also `updating()`.

Alternatively, consider using a event bus subscriber on the `ObjectUpdatedEvent`.

2.3.8. `updating()`

The `updating()` lifecycle callback method is called when a (persistent) object is just about to be updated in the object store

See also `updated()`.

Alternatively, consider using a event bus subscriber on the `ObjectUpdatingEvent`.

2.3.9. Using the JDO API

As an alternative to relying on Apache Isis to call lifecycle callback methods, you could instead use the JDO lifecycle listener API directly.

We may decide to deprecate the Apache Isis callbacks in the future because they merely duplicate this functionality already available in JDO.

You can gain access to the relevant JDO API using the `IsisJdoSupport` domain service.

For example:

```
@RequestScoped                                                  ①
@DomainService(nature=NatureOfService.DOMAIN)
public class ObjectChangedListenerService
        implements javax.jdo.listener.StoreLifecycleListener {  ②
    @Programmatic
    @PostConstruct
    public void init() {
        getPmFactory().addInstanceLifecycleListener(this);
    }
    @Programmatic
    @PreDestroy
    public void tidyUp() {
        getPmFactory().removeInstanceLifecycleListener(this);
    }
    private PersistenceManager getPersistenceManager() {
        return jdoSupport.getPersistenceManager();              ③
    }
    @Programmatic
    public void preStore (InstanceLifecycleEvent event) { ... }
    @Programmatic
    public void postStore (InstanceLifecycleEvent event) { ... }
    @Inject
    IsisJdoSupport jdoSupport;
}
```

① must be `@RequestScoped` because we register on the `PersistenceManager`, which is different for each request.

② implement whichever callback lifecycle listeners are of interest

③ use the injected `IsisJdoSupport` service to obtain the `PersistenceManager`.

Note that it isn't possible to register on the `PersistenceManagerFactory` because listeners cannot be attached once a persistence session has been created (which it will have been when the service's `@PostConstruct` method is called).

Chapter 3. Classes and Interfaces

This chapter describes the usage of various classes and interfaces that are not otherwise associated with domain services, object layout or configuration.

3.1. `AppManifest` (bootstrapping)

This section describes how to implement the `AppManifest` interface to bootstrap both an Apache Isis web application, and also its integration tests.

(As of `1.15.0`), the framework-provided `AppManifestAbstract` and `AppManifestAbstract.Builder` make it easy to write `AppManifest` that can be used both to bootstrap the application "proper", and to be tweaked for use within integration tests.

3.1.1. API

The `AppManifest` interface allows the constituent parts of an application to be defined programmatically, most specifically the packages that contain domain services and/or persistent entities. Its API is defined as:

```
public interface AppManifest {
    public List<Class<?>> getModules();                          ①
    public List<Class<?>> getAdditionalServices();               ②
    public String getAuthenticationMechanism();                  ③
    public String getAuthorizationMechanism();                   ④
    public List<Class<? extends FixtureScript>> getFixtures();   ⑤
    public Map<String,String> getConfigurationProperties();      ⑥
}
```

① Must return a non-null list of classes, each of which representing the root of one of the modules containing services and possibly entities, which together makes up the running application.

② If non-`null`, overrides the value of `isis.services` configuration property to specify a list of additional classes to be instantiated as domain services (over and above the domain services defined via `getModules()` method.

③ If non-`null`, overrides the value of `isis.authentication` configuration property to specify the authentication mechanism.

④ If non-`null`, overrides the value of `isis.authorization` configuration property to specify the authorization mechanism.

⑤ If non-`null`, overrides the value of `isis.fixtures` configuration property to specify a fixture script to be installed.

⑥ Overrides for any other configuration properties.

The following sections describe each of these methods in a little more detail.

`getModules()`

The most significant method (the only one which must return a non-`null` value) is the `getModules()` method. Each module is identified by a class; the framework simply uses that class' package as the root to search for domain services (annotated with `@DomainService`) and entities (annotated with `@PersistenceCapable`). Generally there is one such module class per Maven module.

A module class for a domain module might for example be defined as:

```
package com.mycompany.myapp.dom;
public final class MyAppDomainModule {
    private MyAppDomainModule() {}
}
```

This tells the framework that the package and subpackages under `com.mycompany.myapp.dom` should be searched for domain services (annotated with `@DomainService`), mixins (`@Mixin`) and entities (`@PersistenceCapabable`).

As is perhaps apparent, the `getModules()` method replaces and overrides both the `isis.services.ServicesInstallerFromAnnotation.packagePrefix` key (usually found in the `isis.properties` file) and also the`isis.persistor.datanucleus.RegisterEntities.packagePrefix` key (usually found in the `persistor_datanucleus.properties` file). The value of the `isis.services-installer` configuration property is also ignored.

For example, the (non-ASF) Isis addons' todoapp defines the following:

```
@Override
public List<Class<?>> getModules() {
    return Arrays.asList(
            ToDoAppDomainModule.class,
            ToDoAppFixtureModule.class,
            ToDoAppAppModule.class,
            org.isisaddons.module.audit.AuditModule.class,
            org.isisaddons.module.command.CommandModule.class,
            org.isisaddons.module.devutils.DevUtilsModule.class,
            org.isisaddons.module.docx.DocxModule.class,
            org.isisaddons.module.publishing.PublishingModule.class,
            org.isisaddons.module.sessionlogger.SessionLoggerModule.class,
            org.isisaddons.module.settings.SettingsModule.class,
            org.isisaddons.wicket.gmap3.cpt.service.Gmap3ServiceModule.class
    );
}
```

As can be seen, the various (non-ASF) Isis Addons modules also each provide a module class that can be easily referenced.

`getAdditionalServices()`

We normally we recommend that services are defined exclusively through `getModules()`, and that

this method should therefore return an empty list. However, there are certain use cases where the a service must be explicitly specified either because the service required does not (for whatever reason) have a `@DomainService` annotation.

For example, the (non-ASF) Isis addons' security module (v1.9.0) allows the policy to evaluate conflicting permissions to be specified by explicitly registering either the `PermissionsEvaluationServiceAllowBeatsVeto` domain service or the `PermissionsEvaluationServiceVetoBeatsAllow` domain service:

```
@Override
public List<Class<?>> getAdditionalServices() {
    return Arrays.asList(
            org.isisaddons.module.security.dom.permission
.PermissionsEvaluationServiceVetoBeatsAllow.class
    );
}
```

If this method returns a non-`null` value, then it overrides the value of `isis.services` configuration property.

`getAuthenticationMechanism()`

If non-`null`, this method specifies the authentication mechanism to use. The valid values are currently `"shiro"` or `"bypass"`. If null is returned then the value of the `isis.authentication` configuration property (in `isis.properties` file) is used instead.

See the security guide for further details on configuring shiro or bypass security.

This property is ignored for integration tests (which always uses the `"bypass"` mechanism).

`getAuthorizationMechanism()`

If non-`null`, this method specifies the authorization mechanism to use. The valid values are currently `"shiro"` or `"bypass"`. If null is returned then the value of the `isis.authorization` configuration property (in `isis.properties` file) is used instead.

See the security guide for further details on configuring shiro or bypass security.

This property is ignored for integration tests (which always uses the `"bypass"` mechanism).

`getFixtures()`

If non-`null`, this method specifies the fixture script(s) to be run on startup. This is particularly useful when developing or demoing while using an in-memory database.

For example:

```
@Override
public List<Class<? extends FixtureScript>> getFixtures() {
    return Lists.newArrayList(todoapp.fixture.demo.DemoFixture.class);
}
```

Note that in order for fixtures to be installed it is also necessary to set the `isis.persistor.datanucleus.install-fixtures` key to `true`. This can most easily be done using the `getConfigurationProperties()` method, discussed below.

`getConfigurationProperties()`

This method allow arbitrary other configuration properties to be overridden. One common use case is in conjunction with the `getFixtures()` method, discussed above:

```
@Override
public Map<String, String> getConfigurationProperties() {
    Map<String, String> props = Maps.newHashMap();
    props.put("isis.persistor.datanucleus.install-fixtures", "true");
    return props;
}
```

3.1.2. Bootstrapping

One of the primary goals of the `AppManifest` is to unify the bootstrapping of both integration tests and the webapp. This requires that the integration tests and webapp can both reference the implementation.

We strongly recommend using a `myapp-app` Maven module to hold the implementation of the `AppManifest`. This Maven module can then also hold dependencies which are common to both integration tests and the webapp, specifically the `org.apache.isis.core:isis-core-runtime` and the `org.apache.isis.core:isis-core-wrapper` modules.

We also strongly recommend that any application-layer domain services and view models (code that references persistent domain entities but that is not referenced back) is moved to this `myapp-app` module. This will allow the architectural layering of the overall application to be enforced by Maven.

What then remains is to update the bootstrapping code itself.

There are several different contexts in which the framework needs to be bootstrapped:

- the first is as a "regular" webapp (using the Wicket viewer). Here the `AppManifest` just needs to be specified as a configuration property, usually done using the `WEB-INF/isis.properties` configuration file:

  ```
  isis.appManifest=domainapp.app.MyAppAppManifest
  ```

- the second is also as a webapp, but from within the context of the IDE.

  Here, it's common to use the `org.apache.isis.WebServer` class to launch your application from the command line. This allows the `AppManifest` to be specified using the `-m` (or `--manifest`) flag:

  ```
  java org.apache.isis.WebServer -m com.mycompany.myapp.MyAppAppManifestWithFixtures
  ```

- the third case is within an integration test.

  The code to boostrap an integration test is shown in the testing guide, but once again an `AppManifest` is required.

In some cases an integration test uses the exact same `AppManifest` as the regular webapp. Sometimes though it is necessary to "tweak" the `AppManifest`:

- it might use additional services, such as services to mock out external dependencies, or to provide fake data
- it might override certain configuration properties, eg to run against an in-memory HSQLDB database.

The next section describes some helper classes that the framework provides to help achieve this.

AppManifestAbstract

(As of `1.15.0`), the `AppManifestAbstract` and its associated builder (`AppManifestAbstract.Builder`) make it easy to bootstrap the application both as a webapp and also as an integration test.

Rather than implement `AppManifest` interface directly, instead your application subclasses from `AppManifestAbstract`. This takes an instance of a `AppManifestAbstract.Builder` in its constructor; the builder is what allows for variation between environments.

Moreover, these classes recognise that configuration properties fall into two broad classes:

- those that are fixed and do not change between environments.

  In other words these describe how the application chooses to configure the framework itself, eg global disable of editing of properties, or enabling of auditing.

- those that change between environments.

  The classic example here is the JDBC URL.

For example, the SimpleApp archetype's `AppManifest` is defined as:

```
public class DomainAppAppManifest extends AppManifestAbstract {

    public static final Builder BUILDER = Builder.forModules(
                    SimpleModuleDomSubmodule.class,                          ①
                    DomainAppApplicationModuleFixtureSubmodule.class,
                    DomainAppApplicationModuleServicesSubmodule.class
            )
            .withConfigurationPropertiesFile(DomainAppAppManifest.class,     ②
                    "isis.properties",
                    "authentication_shiro.properties",
                    "persistor_datanucleus.properties",
                    "viewer_restfulobjects.properties",
                    "viewer_wicket.properties"
            ).withAuthMechanism("shiro");                                    ③

    public DomainAppAppManifest() {
        super(BUILDER);                                                      ④
    }
}
```

① the modules that make up the application; corresponds to `AppManifest#getModules()`

② the (non-changing with environment) set of configuration properties, loaded relative to the manifest itself; corresponds to `AppManifest#getConfigurationProperties()`

③ override of components; correponds to both `AppManifest#getAuthenticationMechanism()` and `AppManifest#getAuthorizationMechanism()`

④ Pass the builder up to the superclass.

If the integration tests requires no tweaking, then the `AppManifest` can be used directly, for example:

```
public abstract class DomainAppIntegTestAbstract extends IntegrationTestAbstract2 {
    @BeforeClass
    public static void initSystem() {
        bootstrapUsing(new DomainAppAppManifest());
    }
}
```

On the other hand, if tweaking is required then exposing the builder as a `public static` field makes this easy to do:

```
public abstract class DomainAppIntegTestAbstract extends IntegrationTestAbstract2 {
    @BeforeClass
    public static void initSystem() {
        bootstrapUsing(DomainAppAppManifest.BUILDER
                               .withAdditionalModules(...)
                               .withAdditionalServices(...)
                               .withConfigurationPropertiesFile("...")
                               .withConfigurationProperty("...","...")
                               .build()
        );
    }
}
```

3.1.3. Subsidiary Goals

There are a number of subsidiary goals of the `AppManifest` class (though not all of these are fully implemented):

- Allow different integration tests to run with different manifests.
  - Normally the running application is shared (on a thread-local) between integration tests. What the framework could perhaps do is to be intelligent enough to keep track of the manifest in use for each integration test and tear down the shared state if the "next" test uses a different manifest
- Provide a programmatic way to contribute elements of `web.xml`.
- Provide a programmatic way to configure Shiro security.
- Anticipate the module changes forthcoming in Java 9.
  - Eventually we see that the `AppManifest` class acting as an "aggregator", with the list of modules will become Java 9 modules each advertising the types that they export.
  - It might even be possible for `AppManifest`s to be switched on and off dynamically (eg if Java9 is compatible with OSGi, being one of the design goals).

3.2. Superclasses

This section catalogues the various convenience (non event) superclasses defined by Apache Isis. These are listed in the table below.

Table 5. Convenience Superclasses

API	Maven Module Impl'n (g: a:)	Implementation	Notes
`o.a.i.applib.AbstractContainedObject`	`o.a.i.core` `isis-core-applib`	(abstract class)	

API	Maven Module Impl'n (g: a:)	Implementation	Notes
o.a.i.applib. AbstractDomainObject	o.a.i.core isis-core-applib	(abstract class)	
o.a.i.applib. AbstractFactoryAndRepository	o.a.i.core isis-core-applib	(abstract class)	
o.a.i.applib. AbstractService	o.a.i.core isis-core-applib	(abstract class)	
o.a.i.applib. AbstractSubscriber	o.a.i.core isis-core-applib	(abstract class)	
o.a.i.applib. AbstractViewModel	o.a.i.core isis-core-applib	(abstract class)	
o.a.i.applib. fixturescript FixtureScript	o.a.i.core isis-core-applib	(abstract class)	
o.a.i.applib. fixturescripts FixtureScripts	o.a.i.core isis-core-applib	(abstract class). FixtureScriptsDefault is a default implementation that is used when the alternative FixtureScriptsSpecificationProvider is provided (and no other implementation of FixtureScripts was found).	depends on: ClassDiscoveryService

3.2.1. AbstractContainedObject

This class is a convenience superclass for domain objects and services, providing general purpose methods for interacting with the framework. These include:

- allMatches(Query) - search for all objects matching the specified Query.

  + Note that this, and other similar methods (eg firstMatch(…), uniqueMatch(…)) will

automatically flush the current transaction.

- `newTransientInstance(Class)` - to create a new instance of an object, with any services injected into it
- `persistIfNotAlready(Object)` - to persist an object

  In fact, the object is queued up to be persisted, and is only actually persisted either when the transaction commits, or when the transaction is flushed (typically when a query is performed).

- `warnUser(String)` - generate a warning to the user
- `getContainer()` - which returns the `DomainObjectContainer`

Each of these methods simply delegates to an equivalent method in `DomainObjectContainer`.

In practice we find that there's little to gain from subclassing; it's easier/less obscure to simply inject `DomainObjectContainer` into a simple pojo class.

3.2.2. `AbstractDomainObject`

This class extends `AbstractContainedObject`, adding in convenience methods for managing the persistence lifecycle of the object instance.

Each of these methods, eg `isPersistent(…)`, delegates to an equivalent method in `DomainObjectContainer`.

3.2.3. `AbstractFactoryAndRepository`

This class extends `AbstractContainedObject`. Its intent was to be a convenience subclass for services acting as either a repository or a factory, however note that all of the methods that it defines are now deprecated.

Instead, indicate that a service is repository using the `@DomainService#repositoryFor()` attribute.

3.2.4. `AbstractService`

This class extends `AbstractContainedObject`, adding in an implementation of `getId()` based upon the classes name.

In practice there is little to gain from subclassing; simply inject `DomainObjectContainer` for broadly equivalent functionality.

3.2.5. `AbstractSubscriber`

This is a convenience superclass for creating subscriber domain services on the `EventBusService`. It uses `@PostConstruct` and `@PreDestroy` callbacks to automatically register/unregister itself with the `EventBusService`.

It's important that subscribers register before any domain services that might emit events on the `EventBusService`. For example, the (non-ASF) Isis addons' security module provides a domain service

that automatically seeds certain domain entities; these will generate lifecycle events and so any subscribers must be registered before such seed services. The easiest way to do this is to use the `@DomainServiceLayout#menuOrder()` attribute.

As a convenience, the `AbstractSubscriber` specifies this attribute.

3.2.6. `AbstractViewModel`

This class extends `AbstractContainedObject`, also implementing the `ViewModel` interface. In and of itself it provides no new behaviour.

As an alternative, consider simply annotating the view model class with {@link org.apache.isis.applib.annotation.ViewModel}.

3.2.7. `FixtureScript`

The `FixtureScript` class is an abstract class defining an API to set up data within the object store, either for integration tests or while demoing/prototyping.

The primary method that subclasses must implement is:

```
protected abstract void execute(final ExecutionContext executionContext);
```

In this method the fixture script can in theory do anything, but in practice it is recommended that it uses injected domain services to set up data. The provided `ExecutionContext` is used to invoke child fixture scripts, and alsocan be used to store references to any created objects (so that the calling test can access these objects/so that they are rendered in the view model).

See the see the user guide's testing chapter for further discussion on the use of fixture scripts, in particular fixture scripts' API and usage.

3.2.8. `FixtureScripts`

This abstract class is intended to allow a domain service that can execute `FixtureScript`s to be easily written.

However, it has now been deprecated; instead we recommend that the `FixtureScriptsSpecificationProvider` service is implemented instead. The framework will then automatically use `FixtureScriptsDefault` as a fallback implementation of this class.

See the see the user guide's testing chapter for further discussion on the use of fixture scripts, in particular fixture scripts' API and usage.

3.3. Domain Event Classes

This section catalogues the various domain event classes defined by Apache Isis.

These events are broadcast on the `EventBusService`. The domain events are broadcast as a result of

being specified in the `@Action#domainEvent()`, `@Property#domainEvent()` or `@Collection#domainEvent()` attributes.

They are listed in the table below.

Table 6. Domain Event Classes

API	Maven Module Impl'n (g: a:)	Implementation	Notes
`o.a.i.applib.AbstractDomainEvent`	`o.a.i.core services.eventbus isis-core-applib`	(abstract class)	Superclass of the other domain events, listed below in this table.
`o.a.i.applib.ActionDomainEvent`	`o.a.i.core services.eventbus isis-core-applib`	(abstract class). `ActionDomainEvent.Default` is the concrete implementation used if no `@Action#domainEvent` attribute is specified	Broadcast whenever there is an interaction (hide/disable/validate/pre-execute/post-execute) with an object's action.
`o.a.i.applib.CollectionDomainEvent`	`o.a.i.core services.eventbus isis-core-applib`	(abstract class). `CollectionDomainEvent.Default` is the concrete implementation used if no `@Collection#domainEvent` attribute is specified.	Broadcast whenever there is an interaction (hide/disable/validate/access) with an object's collection.

API	Maven Module Impl'n (g: a:)	Implementation	Notes
o.a.i.applib. PropertyDomainEvent	o.a.i.core services.eventbus isis-core-applib	(abstract class). PropertyDomainEvent.Default is the concrete implementation used if no @Propert#domainEvent attribute is specified	Broadcast whenever there is an interaction (hide/disable/validate/access) with an object's property.

3.3.1. AbstractDomainEvent

This class is the superclass for all domain events that are raised by the framework when interacting with actions, properties or collections.

Its immediate subclasses are:

- ActionDomainEvent
- PropertyDomainEvent
- CollectionDomainEvent

The main purpose of the class is to define the protocol by which subscribers can influence an interaction (eg hide a collection, disable a property, validate action arguments). It class also provides a simple mechanism to allow adhoc sharing of user data between different phases.

API

The API of the class is:

```
public abstract class AbstractDomainEvent<S> extends java.util.EventObject {

    public Phase getEventPhase();                                          ①
    public S getSource();                                                  ②
    public Identifier getIdentifier();                                     ③

    public void hide();                                                    ④
    public boolean isHidden();                                             ⑤

    public void disable(final String reason);                              ⑥
    public void disable(final TranslatableString reason);
    public String getDisabledReason();                                     ⑦
    public TranslatableString getDisabledReasonTranslatable();
    public boolean isDisabled();

    public void invalidate(final String reason);                           ⑧
    public void invalidate(final TranslatableString reason);
    public String getInvalidityReason();                                   ⑨
    public TranslatableString getInvalidityReasonTranslatable();
    public boolean isInvalid();

    public void veto(final String reason, final Object... args);          ⑩
    public void veto(final TranslatableString translatableReason);

    public Object get(Object key);                                         ⑪
    public void put(Object key, Object value);
}
```

① Whether the framework is checking visibility, enablement, validity or actually executing (invoking action, editing property), as per the `Phase` enum (defined below).

② The domain object raising this event

③ Identifier of the action, property or collection being interacted with.

④ API for subscribers to hide the member

⑤ Used by the framework to determine if the member should be hidden (not rendered)

⑥ API for subscribers to disable the member, specifying the reason why (possibly translated)

⑦ Used by the framework to determine whether the member should be disabled (greyed out) when rendered.

⑧ API for subscribers to invalidate an interaction, eg invalid arguments to an action

⑨ Used by the framework to determine whether the interaction is invalid and should be blocked (eg pressing OK shows message)

⑩ Convenience API for subscribers to veto; will automatically call either `hide()`, `disable(…)` or `invalidate(…)` based on the phase

⑪ Mechanism to allow subscribers to share arbitrary information between phases. One event instance is used for both the hide and disable phases, and a different event instance is shared between validate/pre-execute/post-execute.

The referenced `Phase` enum is in turn:

```
public enum Phase {
    HIDE,
    DISABLE,
    VALIDATE,
    EXECUTING,
    EXECUTED;
    public boolean isValidatingOrLater();    ①
}
```

① The significance being that at this point the proposed values/arguments are known, and so the event can be fully populated.

3.3.2. `ActionDomainEvent`

Subclass of `AbstractDomainEvent` for actions.

The class has a number of responsibilities (in addition to those it inherits):

- capture the target object being interacted with
- capture the arguments for each of the action's parameters
- provide selected metadata about the action parameters from the metamodel (names, types)
- link back to the `CommandContext` service's `Command` object

The class itself is instantiated automatically by the framework whenever interacting with a rendered object's action.

API

The API of the class is:

```
public abstract class ActionDomainEvent<S> extends AbstractDomainEvent<S> {

    public static class Default extends ActionDomainEvent<Object> { ... }   ①
    public static class Noop extends ActionDomainEvent<Object> { ... }      ②
    public static class Doop extends ActionDomainEvent<Object> { ... }      ③

    @Deprecated
    public Command getCommand();                                            ④

    public SemanticsOf getSemantics();

    public List<String> getParameterNames();
    public List<Class<?>> getParameterTypes();

    public Object getMixedIn();                                             ⑤
    public List<Object> getArguments();                                     ⑥
    public Object getReturnValue();                                         ⑦
}
```

① The `Default` nested static class is the default for the `@Action#domainEvent()` annotation attribute. Whether this raises an event or not depends upon the `isis.reflector.facet.actionAnnotation.domainEvent.postForDefault` configuration property.

② The `Noop` class is provided as a convenience to indicate that an event should *not* be posted (irrespective of the configuration property setting).

③ Similarly, the `Doop` class is provided as a convenience to indicate that an event *should* be raised (irrespective of the configuration property setting).

④ Deprecated, use `CommandContext` or (better) `InteractionContext`instead.

⑤ Populated only for mixins; holds the underlying domain object that the mixin contributes to.

⑥ The arguments being used to invoke the action; populated during validate phase and subsequent phases.

⑦ The value returned by the action; populated only in the executed phase.

3.3.3. `CollectionDomainEvent`

Subclass of `AbstractDomainEvent` for collections.

The class has a couple of responsibilities (in addition to those it inherits):

- capture the target object being interacted with
- indicate whether the interaction is to add or remove an object from the collection (or simply to indicate that the collection is being accessed/read)
- capture the object reference being added or removed

The class itself is instantiated automatically by the framework whenever interacting with a rendered object's collection.

API

The API of the class is:

```
public abstract class CollectionDomainEvent<S,T> extends AbstractDomainEvent<S> {

    public static class Default                                     ①
        extends CollectionDomainEvent<Object, Object> { ... }
    public static class Noop                                        ②
        extends CollectionDomainEvent<Object, Object> { ... }
    public static class Doop                                        ③
        extends CollectionDomainEvent<Object, Object> { ... }

    public T getValue();                                            ④
    public Of getOf();                                              ⑤
}
```

① The `Default` nested static class is the default for the `@Collection#domainEvent()` annotation attribute. Whether this raises an event or not depends upon the `isis.reflector.facet.collectionAnnotation.domainEvent.postForDefault` configuration property.

② The `Noop` class is provided as a convenience to indicate that an event should *not* be posted (irrespective of the configuration property setting).

③ Similarly, the `Doop` class is provided as a convenience to indicate that an event *should* be raised (irrespective of the configuration property setting).

④ the object being added or removed

⑤ whether this is to add or to remove

where the `Of` enum indicates in turn how the collection is being interacted with:

```
public static enum Of {
    ACCESS,         ①
    ADD_TO,         ②
    REMOVE_FROM     ③
}
```

① collection is being rendered; set during for hide and disable phases

② collection is being added to; set for validate, executing and executed phases

③ or, collection is being removed from; set for validate, executing and executed phases

3.3.4. `PropertyDomainEvent`

Subclass of `AbstractDomainEvent` for properties.

The class has a couple of responsibilities (in addition to those it inherits):

- capture the target object being interacted with

- capture the old and new values of the property

The class itself is instantiated automatically by the framework whenever interacting with a rendered object's property.

API

The API of the class is:

```
public abstract class PropertyDomainEvent<S,T> extends AbstractDomainEvent<S> {

    public static class Default                                     ①
        extends PropertyDomainEvent<Object, Object> { ... }
    public static class Noop                                        ②
        extends PropertyDomainEvent<Object, Object> { ... }
    public static class Doop                                        ③
        extends PropertyDomainEvent<Object, Object> { ... }

    public T getOldValue();                                         ④
    public T getNewValue();                                         ⑤
}
```

① The `Default` nested static class is the default for the `@Property#domainEvent()` annotation attribute. Whether this raises an event or not depends upon the `isis.reflector.facet.propertyAnnotation.domainEvent.postForDefault` configuration property.

② The `Noop` class is provided as a convenience to indicate that an event should *not* be posted (irrespective of the configuration property setting).

③ Similarly, the `Doop` class is provided as a convenience to indicate that an event *should* be raised (irrespective of the configuration property setting).

④ The pre-modification value of the property; populated at validate and subsequent phases.

⑤ The proposed (post-modification) value of the property; populated at validate and subsequent phases

3.4. UI Event Classes

This section catalogues the various UI event classes defined by Apache Isis.

These events are broadcast on the `EventBusService`. The domain events are broadcast as a result of being specified in the `@DomainObjectLayout#titleUiEvent()`, `@DomainObjectLayout#iconUiEvent()` or `@DomainObjectLayout#cssClassUiEvent()` attributes.

They are listed in the table below.

Table 7. UI Event Classes

API	Maven Module Impl'n (g: a:)	Implementation	Notes

`o.a.i.applib.TitleUiEvent`	`o.a.i.core`
	`services.eventbus`
	`isis-core-applib`
		(abstract class). `TitleUiEvent.Default` is the concrete implementation used if no `@DomainObjectLayout#titleUiEvent` attribute is specified	Broadcast whenever there is a requirement to obtain a title for a domain object. Note that if the domain object defines its own `title()` supporting method, or has `@Title` annotation(s) on its properties, then these will take precedence.

API	Maven Module Impl'n (g: a:)	Implementation	Notes

`o.a.i.applib.IconUiEvent`	`o.a.i.core`
	`services.eventbus`
	`isis-core-applib`
		(abstract class). `IconUiEvent.Default` is the concrete implementation used if no `@DomainObjectLayout#iconUiEvent` attribute is specified	Broadcast whenever there is a requirement to obtain an icon (or rather, the name of an icon) for a domain object. Note that if the domain object defines its own `iconName()` supporting method, or if it has the `@DomainObjectLayout#cssClassFa()` attribute, then these will take precedence.

API	Maven Module Impl'n (g: a:)	Implementation	Notes

`o.a.i.applib.CssClassUiEvent`	`o.a.i.core`
	`services.eventbus`
	`isis-core-applib`
		(abstract class). `CssClassUiEvent.Default` is the concrete implementation used if no `@DomainObjectLayout#cssClassUiEvent` attribute is specified	Broadcast whenever there is a requirement to obtain a CSS class hint for a domain object. Note that if the domain object defines its own `cssClass()` supporting method then this will take precedence.

3.4.1. `TitleUiEvent`

This event class represents a request to obtain the title of a domain object. The class has a number of responsibilities:

- capture the target object being interacted with
- capture the title, if any, as specified to one of the subscribers

The class itself is instantiated automatically by the framework whenever interacting with a rendered object's action.

If the domain object defines its own `title()` supporting method, or has `@Title` annotation(s) on its properties, then these will take precedence.

3.4.2. `IconUiEvent`

This event class represents a request to obtain the icon (or rather, name of icon) of a domain object. The class has a number of responsibilities:

- capture the target object being interacted with
- capture the icon (name), if any, as specified to one of the subscribers

The class itself is instantiated automatically by the framework whenever interacting with a rendered object's action.

If the domain object defines its own `iconName()` supporting method, or if it has the `@DomainObjectLayout#cssClassFa()` attribute, then these will take precedence.

3.4.3. `CssClassUiEvent`

This event class represents a request to obtain the a CSS class hint of a domain object. The class has a number of responsibilities:

- capture the target object being interacted with
- capture the CSS class, if any, as specified to one of the subscribers

The class itself is instantiated automatically by the framework whenever interacting with a rendered object's action.

if the domain object defines its own `cssClass()` supporting method then this will take precedence.

3.5. Lifecycle Events

This section catalogues the various lifecycle event classes defined by Apache Isis. These events are fired automatically when a domain object is loaded, created, updated and so forth.

The lifecycle event classes are listed in the table below:

Table 8. Lifecycle Event Classes

API	Maven Module Impl'n (g: a:)	Implementation	Notes

`o.a.i.applib.AbstractLifecycleEvent`	`o.a.i.core`
	`services.eventbus`
	`isis-core-applib`
		(abstract class)	Superclass of the other lifecycle events, listed below in this table.

`o.a.i.applib.ObjectCreatedEvent`	`o.a.i.core`
	`services.eventbus`
	`isis-core-applib`
		(abstract class). `ObjectCreatedEvent.Default` is the concrete implementation that is used.	Broadcast when an object is first instantiated using the `DomainObjectContainer`'s `#newTransientInstance(…)` method.

API	Maven Module Impl'n (g: a:)	Implementation	Notes
o.a.i.applib. ObjectLoadedEvent	o.a.i.core services.eventbus isis-core-applib	(abstract class). ObjectLoadedEvent.Default is the concrete implementation that is used.	Broadcast when an object is retrieved from the database.
o.a.i.applib. ObjectPersistedEvent	o.a.i.core services.eventbus isis-core-applib	(abstract class). ObjectPersistedEvent.Default is the concrete implementation that is used.	Broadcast when an object is first saved (inserted) into the database using the DomainObjectContainer's #persist(…) method.
o.a.i.applib. ObjectPersistingEvent	o.a.i.core services.eventbus isis-core-applib	(abstract class). ObjectPersistingEvent.Default is the concrete implementation that is used.	Broadcast when an object is about to be saved (inserted) into the database using the DomainObjectContainer's #persist(…) method.

API	Maven Module Impl'n (g: a:)	Implementation	Notes
o.a.i.applib. ObjectRemovingEvent	o.a.i.core services.eventbus isis-core-applib	(abstract class). ObjectRemovingEvent.Default is the concrete implementation that is used.	Broadcast when an object is about to be deleted from the database using the DomainObjectContainer's #remove(…) method.
o.a.i.applib. ObjectUpdatedEvent	o.a.i.core services.eventbus isis-core-applib	(abstract class). ObjectUpdatedEvent.Default is the concrete implementation that is used.	Broadcast when an object has just been updated in the database. This is done either explicitly when the current transaction is flushed using the DomainObjectContainer's #flush(…) method, else is done implicitly when the transaction commits at the end of the user request.

API	Maven Module Impl'n (g: a:)	Implementation	Notes
o.a.i.applib. ObjectUpdatingEvent	o.a.i.core services.eventbus isis-core-applib	(abstract class). ObjectUpdatingEvent.Default is the concrete implementation that is used.	Broadcast when an object is about to be updated in the database. This is done either explicitly when the current transaction is flushed using the DomainObjectContainer's #flush(…) method, else is done implicitly when the transaction commits at the end of the user request.

3.5.1. AbstractLifecycleEvent

This class is the superclass for all lifecycle events that are raised by the framework when loading, saving, updating or deleting objects from the database.

Its immediate subclasses are:

- ObjectCreatedEvent
- ObjectLoadedEvent
- ObjectPersistedEvent
- ObjectPersistingEvent
- ObjectRemovingEvent
- ObjectUpdatedEvent
- ObjectUpdatingEvent

3.5.2. ObjectCreatedEvent

Subclass of `AbstractLifecycleEvent`, broadcast when an object is first instantiated using the `DomainObjectContainer`'s `#newTransientInstance(…)` method.

`ObjectCreatedEvent.Default` is the concrete implementation that is used.

In the future this may be generalized to allow arbitrary subclasses to be broadcast, see ISIS-803.

3.5.3. ObjectLoadedEvent

Subclass of `AbstractLifecycleEvent`, broadcast when an object is retrieved from the database.

`ObjectLoadedEvent.Default` is the concrete implementation that is used.

In the future this may be generalized to allow arbitrary subclasses to be broadcast, see ISIS-803.

3.5.4. ObjectPersistedEvent

Subclass of `AbstractLifecycleEvent`, broadcast when an object is first saved (inserted) into the database using the `DomainObjectContainer`'s `#persist(…)` method.

`ObjectPersistedEvent.Default` is the concrete implementation that is used.

In the future this may be generalized to allow arbitrary subclasses to be broadcast, see ISIS-803.

3.5.5. ObjectPersistingEvent

Subclass of `AbstractLifecycleEvent`, broadcast when an object is about to be saved (inserted) into the database using the `DomainObjectContainer`'s `#persist(…)` method.

`ObjectPersistingEvent.Default` is the concrete implementation that is used.

In the future this may be generalized to allow arbitrary subclasses to be broadcast, see ISIS-803.

3.5.6. ObjectRemovingEvent

Subclass of `AbstractLifecycleEvent`, broadcast when an object is about to be deleted from the database using the `DomainObjectContainer`'s `#remove(…)` method.

`ObjectRemovingEvent.Default` is the concrete implementation that is used.

In the future this may be generalized to allow arbitrary subclasses to be broadcast, see ISIS-803.

3.5.7. `ObjectUpdatedEvent`

Subclass of `AbstractLifecycleEvent`, broadcast when an object has just been updated in the database. This is done either explicitly when the current transaction is flushed using the `DomainObjectContainer`'s `#flush(…)` method, else is done implicitly when the transaction commits at the end of the user request.

`ObjectUpdatedEvent.Default` is the concrete implementation that is used.

In the future this may be generalized to allow arbitrary subclasses to be broadcast, see ISIS-803.

3.5.8. `ObjectUpdatingEvent`

Subclass of `AbstractLifecycleEvent`, broadcast when an object is about to be updated in the database. This is done either explicitly when the current transaction is flushed using the `DomainObjectContainer`'s `#flush(…)` method, else is done implicitly when the transaction commits at the end of the user request.

`ObjectUpdatingEvent.Default` is the concrete implementation that is used.

In the future this may be generalized to allow arbitrary subclasses to be broadcast, see ISIS-803.

3.6. Value Types

Apache Isis can render and persist all of the JDK primitives and wrapper classes, and a number of other JDK (7.x) classes that represent value types.

It also supports some of the Joda-Time datatypes, and a number of value types that are shipped by the framework itself.

In addition to primitives, the JDK Classes supported are:

- the wrapper classes:
  - `java.lang.Boolean`, `java.lang.Character`, `java.lang.Double`, `java.lang.Float`, `java.lang.Long`, `java.lang.Integer`, `java.lang.Short`, `java.lang.Byte`
- `java.lang.String`
- numeric data types:
  - `java.math.BigDecimal`
  - `java.math.BigInteger`
- date types:
  - `java.sql.Date`
  - `java.sql.Time`
  - `java.sql.Timestamp`

- `java.util.Date`

It supports these Joda-Time classes:

- `org.joda.time.DateTime`
- `org.joda.time.LocalDateTime`
- `org.joda.time.LocalDate`

The value types defined by the framework itself (in the applib) are:

- `o.a.i.applib.value.Blob`

  binary large object, eg PDFs or images

- `o.a.i.applib.value.Markup`

  (As of `1.15.1-SNAPSHOT`), intended for use as a read-only property to display arbitrary HTML.

- `o.a.i.applib.value.Clob`

  character large objects, eg XML

- `o.a.i.applib.value.Money`

  A currency and amount

- `o.a.i.applib.value.Password`

  A simple wrapper around a string, but never shown in plain-text.

3.6.1. `Blob`

`Blob` (in the `org.apache.isis.applib.value` package) is a value type defined by the Apache Isis framework to represent a binary large object. Conceptually you can consider it as a set of bytes (a picture, a video etc), though in fact it wraps three pieces of information:

- the set of bytes
- a name
- a mime type.

This is reflected in the class' constructors and properties:

```
public final class Blob ... {
    ...
    public Blob(String name, String primaryType, String subtype, byte[] bytes) { ... }
    public Blob(String name, String mimeTypeBase, byte[] bytes) { ... }
    public Blob(String name, MimeType mimeType, byte[] bytes) { ... }
    ...
    public String getName() { ... }
    public MimeType getMimeType() { ... }
    public byte[] getBytes() { ... }
    ...
}
```

Properties of this type can be mapped to JDO/DataNucleus using:

```
@javax.jdo.annotations.Persistent(defaultFetchGroup="false", columns = {
        @javax.jdo.annotations.Column(name = "someImage_name"),
        @javax.jdo.annotations.Column(name = "someImage_mimetype"),
        @javax.jdo.annotations.Column(name = "someImage_bytes", jdbcType = "BLOB",
sqlType = "LONGVARBINARY")
})
private Blob someImage;
```

For character large objects, use `Clob` value type.

3.6.2. `Clob`

`Clob` (in the `org.apache.isis.applib.value` package) is a value type defined by the Apache Isis framework to represent a character large object. Conceptually you can consider it as a set of characters (an RTF or XML document, for example), though in fact it wraps three pieces of information:

- the set of characters
- a name
- a mime type.

This is reflected in the class' constructors and properties:

```
public final class Clob ... {
    ...
    public Clob(String name, String primaryType, String subType, char[] chars) { ... }
    public Clob(String name, String mimeTypeBase, char[] chars) { ... }
    public Clob(String name, MimeType mimeType, char[] chars) { ... }
    public Clob(String name, String primaryType, String subType, CharSequence chars) {
... }
    public Clob(String name, String mimeTypeBase, CharSequence chars) { ... }
    public Clob(String name, MimeType mimeType, CharSequence chars) { ... }
    ...
    public String getName() { ... }
    public MimeType getMimeType() { ... }
    public CharSequence getChars() { ... }
    ...
}
```

Properties of this type can be mapped to JDO/DataNucleus using:

```
@javax.jdo.annotations.Persistent(defaultFetchGroup="false", columns = {
        @javax.jdo.annotations.Column(name = "someClob_name"),
        @javax.jdo.annotations.Column(name = "someClob_mimetype"),
        @javax.jdo.annotations.Column(name = "someClob_chars", jdbcType = "CLOB",
sqlType = "LONGVARCHAR")
})
private Clob someClob;
```

For binary large objects, use `Blob` value type.

3.6.3. `Markup`

The `Markup` value type (introduced in `1.15.1-SNAPSHOT`) is intended to be used as a read-only property, to render arbitrary markup into the user interface.

For example:

Corresponds to:

```
@javax.jdo.annotations.Persistent
@javax.jdo.annotations.Column(allowsNull = "true", length = 4000)
@lombok.Getter @lombok.Setter
@Property(optionality=Optionality.OPTIONAL, editing = Editing.DISABLED)
private Markup someMarkup;

public BlobClobObject updateSomeMarkup(Markup markup) {
    setSomeMarkup(markup);
    return this;
}
public Markup default0SomeMarkup(String markup) {
    return getSomeMarkup();
}
```

with this corresponding `layout.xml`:

```
<cpt:fieldSet name="Markup" id="markup">
    <cpt:action id="updateSomeMarkup"/>
    <cpt:property id="someMarkup" labelPosition="NONE">
    </cpt:property>
</cpt:fieldSet>
```

If the property is also editable then an text edit box is also displayed - unlikely to be the desired effect.

3.7. Applib Utility Classes

The `org.apache.isis.applib.util` package has a number of simple utility classes designed to simplify the coding of some common tasks.

3.7.1. Enums

FIXME

```
public final class Enums {
    public static String getFriendlyNameOf(Enum<?> anEnum) { ... }
    public static String getFriendlyNameOf(String anEnumName) { ... }
    public static String getEnumNameFromFriendly(String anEnumFriendlyName) { ... }
    public static String enumToHttpHeader(final Enum<?> anEnum) { ... }
    public static String enumNameToHttpHeader(final String name) { ... }
    public static String enumToCamelCase(final Enum<?> anEnum) { ... }
}
```

3.7.2. ObjectContracts

The ObjectContracts test provides a series of methods to make it easy for your domain objects to:

- implement Comparable (eg so can be stored in java.util.SortedSets)
- implement toString()
- implement equals()
- implement hashCode()

For example:

```
public class ToDoItem implements Comparable<ToDoItem> {

    public boolean isComplete() { ... }
    public LocalDate getDueBy() { ... }
    public String getDescription() { ... }
    public String getOwnedBy() { ... }

    public int compareTo(final ToDoItem other) {
        return ObjectContracts.compare(this, other, "complete","dueBy","description");
    }

    public String toString() {
        return ObjectContracts.toString(this, "description","complete","dueBy",
"ownedBy");
    }
}
```

Note that ObjectContracts makes heavy use of Java Reflection. While it's great to get going quickly in prototyping, we recommend you use your IDE to code generate implementations of these methods for production code.

Moreover (and perhaps even more importantly) ObjectContracts implementation can cause DataNucleus to recursively rehydrate a larger number of associated entities (More detail below).

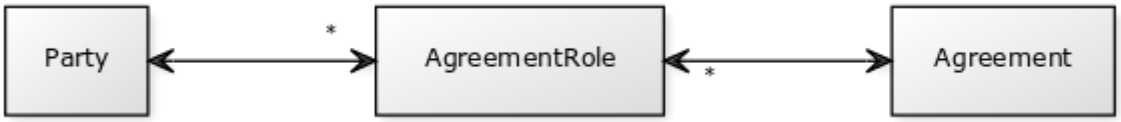
We therefore recommend that you disable persistence-by-reachability by adding:

persistor_datanucleus.properties

```
isis.persistor.datanucleus.impl.datanucleus.persistenceByReachabilityA
tCommit=false
```

The issue in more detail

Consider the entities:



In the course of a transaction, the `Agreement` entity is loaded into memory (not necessarily modified), and then new `AgreementRole`s are associated to it.

All these entities implement `Comparable` using `ObjectContracts`, so that the implementation of `AgreementRole`'s (simplified) is:

```
public class AgreementRole {
    ...
    public int compareTo(AgreementRole other) {
        return ObjectContracts.compareTo(this, other, "agreement","startDate","party"
);
    }
    ...
}
```

while `Agreement`'s is implemented as:

```
    public class Agreement {
        ...
        public int compareTo(Agreement other) {
            return ObjectContracts.compareTo(this, other, "reference");
        }
        ...
    }
```

and `Party`'s is similarly implemented as:

```
public class Party {
    ...
    public int compareTo(Party other) {
        return ObjectContracts.compareTo(this, other, "reference");
    }
    ...
}
```

DataNucleus's persistence-by-reachability algorithm adds the `AgreementRole`s into a `SortedSet`, which causes `AgreementRole#compareTo()` to fire:

- the evaluation of the "agreement" property delegates back to the `Agreement`, whose own `Agreement#compareTo()` uses the scalar `reference` property. As the `Agreement` is already in-memory, this does not trigger any further database queries
- the evaluation of the "startDate" property is just a scalar property of the `AgreementRole`, so will already in-memory

- the evaluation of the "party" property delegates back to the `Party`, whose own `Party#compareTo()` requires the uses the scalar `reference` property. However, since the `Party` is not yet in-memory, using the `reference` property triggers a database query to "rehydrate" the `Party` instance.

In other words, figuring out whether `AgreementRole` is comparable requires the persistence-by-reachability algorithm to run, causing the adjacent associated entity `Party` to also be retrieved.

3.7.3. `Reasons`

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.

3.7.4. `TitleBuffer`

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()`.

3.8. Specification pattern

The interfaces and classes listed in this chapter provide support for the `Specification` pattern, as described in Eric Evans' book *Domain Driven Design*, p224.

Apache Isis will automatically apply such specifications as validation rules on properties (as per `@Property#mustSatisfy()`) and on action parameters (as per `@Parameter#mustSatisfy()`).

3.8.1. `Specification`

The heart of the support for this pattern is the `Specification` interface:

```
public interface Specification {
    public String satisfies(Object obj);  ①
}
```

① if returns `null`, then the constraint is satisfies; otherwise returns the reason why the constraint has not been satisfied.

For example:

```
public class StartWithCapitalLetterSpecification implements Specification {
    public String satisfies(Object proposedObj) {
        String proposed = (String)proposedObj;                ①
        return "".equals(proposed)
            ? "Empty string"
            : !Character.isUpperCase(proposed.charAt(0))
                ? "Does not start with a capital letter"
                : null;
    }
}
public class Customer {
    @Property(mustSatisfy=StartWithCapitalLetterSpecification.class)
    public String getFirstName() { ... }
    ...
}
```

① this ugly cast can be avoided using some of the other classes available; see below.

3.8.2. Specification2

The `Specification2` interface extends the `Specification` API to add support for i18n. This is done by defining an additional method that returns a translatable string:

```
public interface Specification2 extends Specification {
    public TranslatableString satisfiesTranslatable(Object obj);  ①
}
```

① if returns `null`, then the constraint is satisfies; otherwise returns the reason why the constraint has not been satisfied.

Note that if implementing `Specification2` then there is no need to also provide an implementation of the inherited `satisfies(Object)` method; this will never be called by the framework for `Specification2` instances.

3.8.3. Adapter classes

The `AbstractSpecification` and `AbstractSpecification2` adapter classes provide a partial implementation of the respective interfaces, providing type-safety. (Their design is modelled on the `TypesafeMatcher` class within Hamcrest).

For example:

```
public class StartWithCapitalLetterSpecification extends AbstractSpecification<String>
{
    public String satisfiesSafely(String proposed) {
        return "".equals(proposed)
            ? "Empty string"
            : !Character.isUpperCase(proposed.charAt(0))
                ? "Does not start with a capital letter"
                : null;
    }
}
public class Customer {
    @Property(mustSatisfy=StartWithCapitalLetterSpecification.class)
    public String getFirstName() { ... }
    ...
}
```

The `AbstractSpecification2` class is almost identical; its type-safe method is `satisfiesTranslatableSafely(T)` instead.

3.8.4. Combining specifications

There are also adapter classes that can be inherited from to combine specifications:

- `SpecificationAnd` - all provided specifications' constraints must be met
- `SpecificationOr` - at least one provided specifications' constraints must be met
- `SpecificationNot` - its constraints are met if-and-only-if the provided specification's constraint was *not* met.

Note that these adapter classes inherit `Specification` but do not inherit `Specification2`; in other words they do not support i18n.

3.9. i18n support

The `org.apache.isis.applib.services.i18n` package contains a single class to support i18n.

3.9.1. `TranslatableString`

FIXME - see user guide, i18n.

The `TranslatableString` utility class ...

3.10. Contributee

The interfaces listed in this chapter act as contributees; they allow domain services to contribute actions/properties/collections to any domain objects that implement these interfaces.

3.10.1. HasTransactionId

The HasTransactionId interface is a mix-in for any domain objects that reference a transaction id, such as auditing entries or commands, or for Interactions persisted as published events.

Prior to 1.13.0, this identifier was the GUID of the Isis transaction in which the object was created (hence the name). As of 1.13.0, this identifier actually is for the request/interaction in which the object was created, so is actually now mis-named.

The interface is defined is:

```
public interface HasTransactionId {
    public UUID getTransactionId();                                    ①
    public void setTransactionId(final UUID transactionId);
}
```

① unique identifier (a GUID) of this request/interaction.

Modules that either have domain entity that implement and/or services that contribute this interface are:

- (non-ASF) Isis addons' audit module (AuditEntry entity, AuditingServiceContributions service)
- (non-ASF) Isis addons' command module (CommandJdo entity, CommandServiceJdoContributions service)
- (non-ASF) Isis addons' publishing module (PublishedEvent entity, PublishingServiceContributions)
- (non-ASF) Isis addons' publishmq module (PublishedEvent entity)

3.10.2. HasUsername

The HasUsername interface is a mix-in for domain objects to be associated with a username. Other services and modules can then contribute actions/collections to render such additional information relating to the activities of the user.

The interface is defined is:

```
public interface HasUsername {
    public String getUsername();
}
```

Modules that either have domain entity that implement and/or services that contribute this interface are:

- (non-ASF) Isis addons' security module (ApplicationUser entity, HasUsernameContributions service)

- (non-ASF) Isis addons' audit module (`AuditEntry` entity,
- (non-ASF) Isis addons' command module's `CommandJdo` entity, `HasUsernameContributions` service)
- (non-ASF) Isis addons' publishing module (`PublishedEvent` entity)
- (non-ASF) Isis addons' sessionlogger module (`SessionLogEntry` entity, `HasUsernameContributions` service)
- (non-ASF) Isis addons' settings module (`UserSettingJdo` entity)

3.11. Roles

The interfaces listed in this chapter are role interfaces; they define a contract for the framework to interact with those domain objects that implement these interfaces.

3.11.1. `HoldsUpdatedAt`

The `HoldsUpdatedAt` role interface allows the (framework-provided) `TimestampService` to update each object with the current timestamp whenever it is modified in a transaction.

The interface is defined as:

```
public interface HoldsUpdatedAt {
    void setUpdatedAt(java.sql.Timestamp updatedAt);
}
```

The current time is obtained from the `ClockService`.

Entities that implement this interface often also implement `HoldsUpdatedBy` role interface; as a convenience the `Timestampable` interface combines the two roles.

Alternative approaches

An alternative way to maintain a timestamp is to use JDO's `@Version` annotation. With this approach, it is the JDO/DataNucleus that maintains the version, rather than the framework's `TimestampService`.

For example:

```
@javax.jdo.annotations.Version(
        strategy=VersionStrategy.DATE_TIME,
        column="version")
public class Customer {
    ...
    public java.sql.Timestamp getVersionSequence() {
        return (java.sql.Timestamp) JDOHelper.getVersion(this);
    }
}
```

3.11.2. HoldsUpdatedBy

The HoldsUpdatedBy role interface ...

FIXME ... incomplete

```
public interface HoldsUpdatedBy {
    void setUpdatedBy(String updatedBy);
}
```

Entities that implement this interface often also implement HoldsUpdatedAt role interface; as a convenience the Timestampable interface combines the two roles.

3.11.3. Timestampable

The Timestampable role interface is a convenience that combines the HoldsUpdatedAt and HoldsUpdatedBy interfaces. It is defined as:

```
public interface Timestampable
    extends HoldsUpdatedAt, HoldsUpdatedBy {
}
```

The interface no additional methods of its own.

Alternatives

An alternative way to maintain a timestamp is to use JDO's @Version annotation. With this approach, it is the JDO/DataNucleus that maintains the version, rather than the framework's TimestampService. See HoldsUpdatedBy for further details.

3.12. Mixins

This chapter defines a number of role interfaces that definea contract for some framework-defined mixins.

See the fundamentals user guide for a discussion of mixins.

3.12.1. Object

The framework provides a single mixin that contributes to simply java.lang.Object. It provides the ability to download the layout XML for any domain object (in practical terms: entities and view models).

clearHints()

When a domain object is rendered the end-user can select different tabs, and for collections can sort the columns, navigate to second pages, or select different views of collections. If the user

revisits that object, the Wicket viewer will remember these hints and render the domain object in the same state. These rendering hints are also included if the user copies the URL using the anchor link (to right hand of the object's title).

The `Object_clearHints` mixin provides the ability for the end-user to discard these hints so that the object is rendered in its initial state:

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

Appearance in the UI

This mixin actions are all associated with the "Metadata" fieldset. If there is no such field set, then the action will be rendered as a top-level action).

Related Services

This mixin uses the `HintStore` service to store and retrieve UI hints for each rendered object, per user.

`downloadLayoutXml()`

The `Object_downloadLayoutXml` mixin provides an action to download the layout XML for the current domain object. It has the following signature:

```
public Object downloadLayoutXml(
    @ParameterLayout(named = "File name")
    final String fileName,
    final LayoutService.Style style) {          ①
    ...
}
```

① either current, complete, normalized or minimal.

See the documentation on layout XML and also the `LayoutService` for more information on these styles

Appearance in the UI

This mixin actions are all associated with the "Metadata" fieldset.

A number of other mixins also contribute properties and actions to the "Metadata" fieldset.

Related Services

This mixin calls `LayoutService` to obtain the layout XML.

`rebuildMetamodel()`

The `Object_rebuildMetamodel` mixin provides the ability to discard the current internal metamodel data (an instance of `ObjectSpecification`) for the domain class of the rendered object, and recreate from code and other sources (most notably, layout XML data). It has the following signature:

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

Appearance in the UI

This mixin actions are all associated with the "Metadata" fieldset.

A number of other mixins also contribute properties and actions to the "Metadata" fieldset.

Related Services

This mixin calls `MetaModelService` and the `GridService` to invalidate their caches.

3.12.2. `Dto`

The `Dto` role interface is intended to be implemented by JAXB-annotated view models, that is, annotated using `@XmlRootElement`. It enables the ability to download the XML and XSD schema of those objects using two mixins, `Dto_downloadXml` and `Dto_downloadXsd`.

The interface is just a marker interface (with no members), and is defined as:

```
public interface Dto { }
```

The `Dto_downloadXml` mixin defines the following action:

```
@Mixin(method="act")
public class Dto_downloadXml {
    public Dto_downloadXml(final Dto dto) { ... }     ①
    public Object act(final String fileName) { ... }   ②
    ...
}
```

① provided as an action to any class that (trivially) implements the `Dto` interface

② The action's name is derived from the class name.

This will return the XML text wrapped up in a `Clob`.

The `Dto_downloadXsd` mixin is similar:

```
@Mixin(method="act")
public class Dto_downloadXsd {
    public Dto_downloadXsd(final Dto dto) { ... }
①
    public Object act(final String fileName, final IsisSchemes isisSchemas) { ... }
②
}
```

① provided as an action to any class that (trivially) implements the `Dto` interface

② The action's name be derived from the class name.

If the domain object's JAXB annotations reference only a single XSD schema then this will return that XML text as a `Clob` of that XSD. If there are multiple XSD schemas referenced then the action will return a zip of those schemas, wrapped up in a `Blob`. The `IsisSchemas` parameter to the action can be used to optionally ignore the common Apache Isis schemas (useful if there is only one other XSD schema referenced by the DTO).

Related Services

The `Dto_downloadXml` and `Dto_downloadXsd` delegate to the `JaxbService` to actually generate the XML/XSD.

3.12.3. `Persistable`

All domain entities automatically implement the DataNucleus `Persistable` role interface as a result of the enhancer process (the fully qualified class name is `org.datanucleus.enhancement.Persistable`). So as a developer you do not need to write any code to obtain the mixins that contribute to this interface.

`downloadJdoMetadata()`

The `Persistable_downloadJdoMetadata` mixin provides an action which allows the JDO class metadata to be downloaded as XML. It has the following signature:

```
public Clob downloadJdoMetadata(                    ①
    @ParameterLayout(named = ".jdo file name")
    final String fileName) {
    ...
}
```

① returns the XML text wrapped up in a `Clob`.

Appearance in the UI

This mixin action is associated with the "Metadata" fieldset, and will appear as a panel drop-down action.

These mixin properties are all associated with the "Metadata" fieldset. The Object mixin also contribute an action to the "Metadata" fieldset.

Related Services

The mixin delegates to the `IsisJdoSupport` service to obtain a reference to the JDO `PersistenceManagerFactory`.

`datanucleusXxx`

The framework provides a number of mixins that expose the datanucleus Id and version of a persistable domain entity. Several implementations are provided to support different datatypes:

- `Persistable_datanucleusIdLong` will expose the entity's id, assuming that the id is or can be cast to `java.lang.Long`. Otherwise the property will be hidden.
- `Persistable_datanucleusVersionTimestamp` will expose the entity's version, assuming that the version is or can be cast to `java.sql.Timestamp`. Otherwise the property will be hidden.
- `Persistable_datanucleusVersionLong` will expose the entity's version, assuming that the version is or can be cast to `java.lang.Long`. Otherwise the property will be hidden.

Appearance in the UI

These mixin properties are all associated with the "Metadata" fieldset. The Object mixin also contribute an action to the "Metadata" fieldset.

3.13. Layout

The `org.apache.isis.applib.layout` package defines a number of classes that allow the layout of domain objects (entities and view models) to be customized. These classes fall into two main categories:

- grid classes, that define a grid structure of rows, columns, tab groups and tabs, and;
- common component classes, that capture the layout metadata for an object's properties, collections and actions. These are bound (or associated) to the regions of the grid

The framework provides an implementation of the grid classes modelled closely on Bootstrap 3, along with Wicket viewer components capable of rendering that grid system. In principle it is also possible to extend the layout architecture for other grid systems. The component classes, though, are intended to be reusable across all grid systems.

The component classes, meanwhile, are broadly equivalent to the "layout" annotations (`@PropertyLayout`, `@CollectionLayout`, `@ActionLayout` and `@DomainObjectLayout`

All of the classes in this package are JAXB-annotated, meaning that they can be serialized to/from XML (the `component` classes in the `http://isis.apache.org/applib/layout/component` XSD namespace, the bootstrap 3 grid classes in the `http://isis.apache.org/applib/layout/grid/bootstrap3` XSD namespace). This ability to serialize to/from XML is used by the `GridLoaderService`, the default implementation of which reads the grid layout for a domain class from a `.layout.xml` file on the classpath.

3.13.1. Component

The component classes reside in the `org.apache.isis.applib.layout.component` package, and consist of:

- `FieldSet`

  A fieldset (previously also called a property group or member group) of a number of the domain object's properties (along with any associationed actions of those properties).

- layout data classes, which correspond to the similarly named annotations:
  - `PropertyLayoutData`, corresponding to the `@PropertyLayout` annotation;
  - `CollectionLayoutData`, corresponding to the `@CollectionLayout` annotation;
  - `ActionLayoutData`, corresponding to the `@ActionLayout` annotation;
  - `DomainObjectLayoutData`, corresponding to the `@DomainObjectLayout` annotation.

In addition, the component package includes `Grid`, representing the top level container for a custom layout for a domain object. `Grid` itself is merely an interface, but it also defines the visitor pattern to make it easy for validate and normalize the grid layouts. The `GridAbstract` convenience superclass provides a partial implementation of this visitor pattern.

3.13.2. Bootstrap3 Grid

As noted above, the default bootstrap3 grid classes are modelled closely on Bootstrap 3. Bootstrap's grid system divides the page width equally into 12 columns, and so each column spans 1 or more of these widths. Thus, a column with a span of 12 is the full width, one with a span of 6 is half the width, one with a span of 4 is a third of the width, and so on.

When specifying the span of a column, Bootstrap also allows a size to be specified (`XS`, `SM`, `MD`, `LG`). The size determines the rules for responsive design. Apache Isis defaults to `MD` but this can be overridden. It is also possible to specify multiple size/spans for a given column.

The grid classes provided by Apache Isis reside in the `org.apache.isis.applib.layout.grid.bootstrap3` package, and consist of:

- `BS3Grid`

  Consists of a number of `BS3Row`s.

  This class is the concrete implementation of `Grid` interface, discussed previously. As such, it extends the `Grid.Visitor` to iterate over all of the `Row`s of the grid.

- `BS3Row`

  A container of `BS3Col`s. This element is rendered as `<div class="row">`.

- `BS3Col`

  A container of almost everything else. A column most commonly contains properties (grouped

into `FieldSet`s, described above) or collections (specified by `CollectionLayoutData`, also above). However, a `Col` might instead contain a `BS3TabGroup` (described below) in order that the object members is arranged into tabs.

It is also possible for a `Col` to contain the object's title/icon (using `DomainObjectLayoutData`) or indeed arbitrary actions (using `ActionLayoutData`).

Finally, a `BS3Col` can also contain other `BS3Row`s, allowing arbitrarily deep hierarchies of containers as required.

This element is rendered as, for example, `<div class="col-md-4">` (for a size `MD`, span of 4).

- `BS3TabGroup`

  A container of `BS3Tab`s.

- `BS3Tab`

  A container of `BS3Row`s, which will in turn contain `BS3Col`s and thence ultimately the object's members.

There are also two close cousins of `Col`, namely `ClearFixVisible` and `ClearFixHidden`. These map to Bootstrap's responsive utility classes, and provide greater control for responsive designs.

As you can probably guess, the `BS3Grid` is the top-level object (that is, it is JAXB `@XmlRootElement`); this is the object that is serialized to/from XML.

All of these classes also allow custom CSS to be specified; these are added to the CSS classes for the corresponding `<div>` in the rendered page. The `application.css` file can then be used for application-specific CSS, allowing arbitrary fine-tuning of the layout of the page.

Chapter 4. Schema

Most applications need to integrate with other apps in the enterprise. To facilitate such integration scenarios, Apache Isis defines a number of standard XSD schemas:

- the command schema, which captures the *intention* of a user to invoke an action or edit a property
- the interaction execution schema, which captures the actual execution of an action invocation/property edit
- the changes schema, which captures which objects have been created, updated or deleted as the result of an execution of an action invocation/property edit
- the action memento invocation schema (deprecated in `1.13.0`, replaced by either "cmd" or "ixn"), which allows action invocations to be captured and reified.

These each use XSD types defined by the common schema (most notably the `oidDto` complex type which identifies a domain object).

The (non-ASF) Isis addons' command and Isis addons' publishmq modules uses these schemas to reify corresponding applib objects (`Command`, `Interaction.Execution` and `PublishedObjects`), either to persist or publishing using an Apache ActiveMQ message queue.

The sections below discuss these schemas in more detail.

4.1. Command

The command ("cmd") schema defines the serialized form of the *intention* to invoke an action or to edit a property.

Mixin actions are represented as regular actions on the mixed-in object. In other words, the fact that the actual implementation of the action is defined by a mixin is an implementation detail only.

4.1.1. `commandDto`

The `commandDto` root element is defined as:

```
<xs:schema targetNamespace="http://isis.apache.org/schema/cmd"                ①
           elementFormDefault="qualified"
           xmlns:xs="http://www.w3.org/2001/XMLSchema"
           xmlns="http://isis.apache.org/schema/cmd"
           xmlns:com="http://isis.apache.org/schema/common">

    <xs:import namespace="http://isis.apache.org/schema/common"               ②
               schemaLocation="../common/common-1.0.xsd"/>

    <xs:element name="commandDto">                                            ③
        <xs:complexType>
            <xs:sequence>
                <xs:element name="majorVersion" type="xs:string"              ④
                        minOccurs="1" maxOccurs="1" default="1"/>
                <xs:element name="minorVersion" type="xs:string"
                        minOccurs="1" maxOccurs="1" default="1"/>

                <xs:element name="transactionId" type="xs:string"/>           ⑤
                <xs:element name="user" type="xs:string"/>                    ⑥
                <xs:element name="targets" type="com:oidsDto"/>               ⑦
                <xs:element name="member" type="memberDto"/>                  ⑧
            </xs:sequence>
        </xs:complexType>
    </xs:element>
    ...
</xs:schema>
```

① the command schema has a namespace URI of "http://isis.apache.org/schema/cmd". Although URIs are not the same as URLs, you will find that the schemas are also downloadable from this location.

② uses complex types defined in the "common" schema.

③ definition of the `commandDto` root element. The corresponding XML will use this as its top-level element.

④ each instance of this schema indicates the version of the schema it is compatible with (following semantic versioning)

⑤ unique identifier for the transaction in which this command is created. The transaction Id is used to correlate to the interaction that executes the command, and to any changes to domain objects occurring as a side-effect of that interaction.

⑥ the name of the user who created the command (whose intention it is to invoke the action/edit the property).

⑦ the target object (or objects) to be invoked. As of `1.13.0`, a bulk action will create multiple commands, each with only a single target, but a future version of the framework may also support a single bulk command against this multiple targets (ie all-or-nothing).

⑧ the `memberDto`, defined below, the captures the action/property and arguments/new value.

The `CommandDto` DTO corresponding to the `commandDto` root element can be marshalled to/from XML

using the `CommandDtoUtils` class.

4.1.2. `memberDto` and subtypes

The `memberDto` complex type is an abstract type representing the intention to either invoke an action or to edit a property. The `actionDto` and `propertyDto` are the concrete subtypes:

```
<xs:schema targetNamespace="http://isis.apache.org/schema/cmd" ...>
    ...
    <xs:complexType name="memberDto" abstract="true">                       ①
        </xs:element>
        <xs:sequence>
            <xs:element name="memberIdentifier" type="xs:string"/>          ②
            <xs:element name="logicalMemberIdentifier" type="xs:string">    ③
        </xs:sequence>
        <xs:attribute  name="interactionType" type="com:interactionType"/>  ④
    </xs:complexType>

    <xs:complexType name="actionDto">                                       ⑤
        <xs:complexContent>
            <xs:extension base="memberDto">
                <xs:sequence>
                    <xs:element name="parameters" type="paramsDto"/>
                </xs:sequence>
            </xs:extension>
        </xs:complexContent>
    </xs:complexType>

    <xs:complexType name="propertyDto">                                     ⑥
        <xs:complexContent>
            <xs:extension base="memberDto">
                <xs:sequence>
                    <xs:element name="newValue" type="com:valueWithTypeDto"/>
                </xs:sequence>
            </xs:extension>
        </xs:complexContent>
    </xs:complexType>
</xs:schema>
```

① the `memberDto` is an abstract type. Its primary responsibility is simply to identify the member (action or property).

② the formal identifier (fully qualified class name + member name) of the member being interacted with (action or property).

③ the "logical" formal identifier (object type, as per `@DomainObject(objectType=)`, + member name) of the member being interacted with (action or property).

④ the `interactionType` attribute indicates whether the member is an action or a property.

⑤ the `actionDto` complex type captures the set of parameters (also including the argument values) with which to invoke the action. The `paramsDto` type is defined below.

⑥ the `propertyDto` complex type captures the new value (possibly `null`) to set the property to.

In general the `logicalMemberIdentifier` should be used in preference to the `memberIdentifier` because will not (necessarily) have to change if the class is moved during a refactoring.

Note also that there is a corresponding `memberExecutionDto` complex type in the "ixn" schema that is for the actual execution (capturing metrics about its execution and also the return value if an action invocation).

4.1.3. Ancillary types

The schema also defines a small number of supporting types:

```
<xs:schema targetNamespace="http://isis.apache.org/schema/cmd" ...>
    ...
    <xs:complexType name="paramsDto">                                    ①
        <xs:sequence minOccurs="0" maxOccurs="unbounded">
            <xs:element name="parameter" type="paramDto"/>
        </xs:sequence>
    </xs:complexType>

    <xs:complexType name="paramDto">                                     ②
        <xs:complexContent>
            <xs:extension base="com:valueWithTypeDto">
                <xs:attribute name="name" use="required" type="xs:string"/>
            </xs:extension>
        </xs:complexContent>
    </xs:complexType>
</xs:schema>
```

① the `paramsDto` is simply the list of parameter/arguments.

② the `paramDto` complex type essentially combines a parameter with its corresponding argument: a named value that has a type. It extends the `valueWithTypeDto` complex type taken from the "common" schema.

4.2. Interaction Execution

The interaction ("ixn") schema defines the serialized form of an action invocation or a property edit. In fact, it actually defines a call-graph of such executions for those cases where the `WrapperFactory` is used to execute sub-actions/property edits.

Each execution identifies the target object, the member to invoke, and the arguments. It also captures metrics about the execution, and the result of the execution (eg return value of an action invocation).

Mixin actions are represented as regular actions on the mixed-in object. In other words, the fact that the actual implementation of the action is defined by a mixin is an implementation detail only.

4.2.1. interactionDto

The interactionDto root element is defined as:

```
<xs:schema targetNamespace="http://isis.apache.org/schema/ixn"                ①
           elementFormDefault="qualified"
           xmlns:xs="http://www.w3.org/2001/XMLSchema"
           xmlns="http://isis.apache.org/schema/ixn"
           xmlns:cmd="http://isis.apache.org/schema/cmd"
           xmlns:com="http://isis.apache.org/schema/common">

    <xs:import namespace="http://isis.apache.org/schema/common"               ②
               schemaLocation="../common/common-1.0.xsd"/>
    <xs:import namespace="http://isis.apache.org/schema/cmd"
               schemaLocation="../cmd/cmd-1.0.xsd"/>

    <xs:element name="interactionDto">                                        ③
        <xs:complexType>
            <xs:sequence>
                <xs:element name="majorVersion" type="xs:string"              ④
                            minOccurs="0" maxOccurs="1" default="1"/>
                <xs:element name="minorVersion" type="xs:string"
                            minOccurs="0" maxOccurs="1" default="0"/>

                <xs:element name="transactionId" type="xs:string"/>           ⑤
                <xs:element name="execution" type="memberExecutionDto"/>      ⑥
            </xs:sequence>
        </xs:complexType>
    </xs:element>

</xs:schema>
```

① the interaction schema has a namespace URI of "http://isis.apache.org/schema/ixn". Although URIs are not the same as URLs, you will find that the schemas are also downloadable from this location.

② uses complex types defined in the "common" schema and also the "cmd" schema

③ definition of the interactionDto root element. The corresponding XML will use this as its top-level element.

④ each instance of this schema indicates the version of the schema it is compatible with (following semantic versioning)

⑤ unique identifier for the transaction in which this interaction is being executed. The transaction Id is used to correlate back to the command that represented the intention to perform this execution, as well as to any changes to domain objects that occur as a side-effect of the interaction.

⑥ the top-level memberExecutionDto, defined below, either an action invocation or edit of a property.

The InteractionDto DTO corresponding to the interactionDto root element can be marshalled

to/from XML using the `InteractionDtoUtils` class.

4.2.2. `memberExecutionDto`

The `memberExecutionDto` complex type is an abstract type representing either the invocation an action or the editing of a property. It corresponds to the `memberDto` of the "cmd" schema; some elements are copied directly:

```
<xs:schema targetNamespace="http://isis.apache.org/schema/ixn" ... >
    ...
    <xs:complexType name="memberExecutionDto" abstract="true">                ①
        <xs:sequence>
            <xs:element name="sequence" type="xs:int"/>                       ②
            <xs:element name="target" type="com:oidDto"/>                     ③
            <xs:element name="memberIdentifier" type="xs:string"/>            ④
            <xs:element name="logicalMemberIdentifier" type="xs:string"/>     ⑤
            <xs:element name="user" type="xs:string"/>                        ⑥
            <xs:element name="title" type="xs:string"/>                       ⑦
            <xs:element name="metrics" type="metricsDto"/>                    ⑧
            <xs:element name="threw" type="exceptionDto"                      ⑨
                        minOccurs="0" maxOccurs="1"/>
            <xs:element name="childExecutions" minOccurs="0" maxOccurs="1"> ⑩
                <xs:complexType>
                    <xs:sequence>
                        <xs:element name="execution" type="memberExecutionDto"
                                    minOccurs="0" maxOccurs="unbounded"/>
                    </xs:sequence>
                </xs:complexType>
            </xs:element>
        </xs:sequence>
        <xs:attribute  name="interactionType" type="com:interactionType"/>  ⑪
    </xs:complexType>
    ...
</xs:schema>
```

① the `memberExecutionDto` is an abstract type

② uniquely identifies this execution within the transaction. Can be combined with `transactionId` to create a unique identifier (across all other interaction executions and also changed objects events) of this particular interaction execution.

③ the target object, corresponding to one of the elements of the `targets` element of the `memberDto`

④ the member identifier; corresponds to `memberIdentifier` of the `member` element of the `memberDto`

⑤ the *logical* member identifier; corresponds to `logicalMemberIdentifier` of the `member` element of the `memberDto`

⑥ the user executing the action invocation/property edit; corresponds to the `user` element of the `memberDto`

⑦ the current "human-friendly" title of the target object

⑧ the set of metrics captured for this execution, of type `metricsDto` defined below.

⑨ if the action invocation/property edit threw an exception, then this is captured here.

⑩ if any sub-actions or sub-edits were performed via the `WrapperFactory`, then these are captured in the `childExecutions` element.

⑪ the `interactionType` attribute indicates whether the member is an action or a property (similar attribute exists for the "cmd" schema).

In general the `logicalMemberIdentifier` should be used in preference to the `memberIdentifier` because will not (necessarily) have to change if the class is moved during a refactoring.

The `actionInvocationDto` and `propertyEditDto` are the concrete subtypes:

```
<xs:schema targetNamespace="http://isis.apache.org/schema/ixn" ... >
    ...
    <xs:complexType name="actionInvocationDto">                          ①
        <xs:complexContent>
            <xs:extension base="memberExecutionDto">
                <xs:sequence>
                    <xs:element name="parameters" type="cmd:paramsDto"/>  ②
                    <xs:element name="returned"                          ③
                                type="com:valueWithTypeDto"
                                minOccurs="0" maxOccurs="1"/>
                </xs:sequence>
            </xs:extension>
        </xs:complexContent>
    </xs:complexType>
    ...
    <xs:complexType name="propertyEditDto">                              ④
        <xs:complexContent>
            <xs:extension base="memberExecutionDto">
                <xs:sequence>
                    <xs:element name="newValue"                          ⑤
                                type="com:valueWithTypeDto"/>
                </xs:sequence>
            </xs:extension>
        </xs:complexContent>
    </xs:complexType>
    ...
</xs:schema>
```

① the `actionInvocationDto` inherits from `memberExecutionDto`. It corresponds to the similar `actionDto` complex type of the "cmd" schema

② the `parameters` element captures the parameter and argument values; for the top-level execution it is a direct copy of the corresponding `parameters` element of the `actionDto` complex type of the "cmd" schema.

③ the `returned` element captures the returned value (if not void). It is not valid for both this element and the inherited `threw` element to both be populated.

④ the `propertyEditDto` inherits from `memberExecutionDto`. It corresponds to the similar `propertyDto`

complex type of the "cmd" schema

⑤ the `newValue` element captures the new value; for the top-level execution it is a direct copy of the corresponding `newValue` element of the `propertyDto` complex type of the "cmd" schema.

4.2.3. Ancillary types

The schema also defines a small number of supporting types:

```
<xs:schema targetNamespace="http://isis.apache.org/schema/ixn" ... >
    ...
    <xs:complexType name="metricsDto">                                    ①
        <xs:sequence>
            <xs:element name="timings" type="com:periodDto"/>
            <xs:element name="objectCounts" type="objectCountsDto"/>
        </xs:sequence>
    </xs:complexType>

    <xs:complexType name="objectCountsDto">                               ②
        <xs:sequence>
            <xs:element name="loaded" type="com:differenceDto"/>
            <xs:element name="dirtied" type="com:differenceDto"/>
        </xs:sequence>
    </xs:complexType>

    <xs:complexType name="exceptionDto"/>                                 ③
        <xs:sequence>
            <xs:element name="message" type="xs:string"/>
            <xs:element name="stackTrace" type="xs:string"/>
            <xs:element name="causedBy" type="exceptionDto" minOccurs="0" maxOccurs="
1"/>
        </xs:sequence>
    </xs:complexType>
</xs:schema>
```

① the `metricsDto` captures the time to perform an execution, and also the differences in various object counts.

② the `objectCountsDto` complex type is the set of before/after differences, one for each execution; the framework tracks number of objects loaded (read from) the database and the number of objects dirtied (will need to be saved back to the database). Together these metrics give an idea of the "size" of this particular execution.

③ the `exceptionDto` complex type defines a structure for capturing the stack trace of any exception that might occur in the course of invoking an action or editing a property.

The changes schema also provides metrics on the number of objects loaded/changed, but relates to the entire interaction rather than just one (sub)execution of an interaction.

4.3. Changes

The changes ("chg") schema defines the serialized form identifying which objects have been created, updated or deleted as the result of invoking an action or editing a property. It also captures a number of other metrics counts (number of objects loaded, number of object properties modified), useful for profiling.

An instance of the DTO (corresponding to this schema) is used within the `PublisherService` SPI, identifying changed objects that are to be published (as per `@DomainObject#publishing()` or equivalent).

4.3.1. `changesDto`

The `changesDto` root element is defined as:

```
<xs:schema targetNamespace="http://isis.apache.org/schema/chg"
①
           elementFormDefault="qualified"
           xmlns:xs="http://www.w3.org/2001/XMLSchema"
           xmlns="http://isis.apache.org/schema/chg"
           xmlns:com="http://isis.apache.org/schema/common">

    <xs:import namespace="http://isis.apache.org/schema/common"
②
               schemaLocation="../common/common-1.0.xsd"/>

    <xs:element name="changesDto">
③
        <xs:complexType>
            <xs:sequence>
                <xs:element name="majorVersion" type="xs:string"
④
                            minOccurs="0" maxOccurs="1" default="1"/>
                <xs:element name="minorVersion" type="xs:string"
                            minOccurs="0" maxOccurs="1" default="0"/>

                <xs:element name="transactionId" type="xs:string"/>
⑤
                <xs:element name="sequence" type="xs:int"/>
⑥
                <xs:element name="completedAt" type="xs:dateTime" minOccurs="0"
maxOccurs="1"/> ⑦
                <xs:element name="user" type="xs:string"/>
⑧
                <xs:element name="objects" type="objectsDto"/>
⑨
            </xs:sequence>
        </xs:complexType>
    </xs:element>
    ...
</xs:schema>
```

① the changes schema has a namespace URI of "http://isis.apache.org/schema/chg". Although URIs are not the same as URLs, you will find that the schemas are also downloadable from this location.

② uses complex types defined in the "common" schema.

③ definition of the `changesDto` root element. The corresponding XML will use this as its top-level element.

④ each instance of this schema indicates the version of the schema it is compatible with (following semantic versioning)

⑤ unique identifier for the transaction in which this interaction is being executed. The transaction Id is used to correlate back to the command that represented the intention to perform this execution, as well as to the interaction that executes said command.

⑥ uniquely identifies this set of changes within the interaction. Can be combined with `transactionId` to create a unique identifier (across all other changed object events and also any interaction executions) of this particular set of changed objects.

⑦ the date/time that the transaction that dirtied this objects completed

⑧ the user that executed the (top-level) action invocation/property edit.

⑨ identifies the objects that have changed.

The `ChangesDto` DTO corresponding to the `changesDto` root element can be marshalled to/from XML using the `ChangesDtoUtils` class.

4.3.2. `objectsDto`

The `objectsDto` complex type actually identifies the objects created, updated or deleted. It also captures additional metrics counters:

```
<xs:schema targetNamespace="http://isis.apache.org/schema/chg" ... >
    ...
    <xs:complexType name="objectsDto">
        <xs:sequence>
            <xs:element name="loaded" type="xs:int"/>
①
            <xs:element name="created" type="com:oidsDto"/>
②
            <xs:element name="updated" type="com:oidsDto"/>
            <xs:element name="deleted" type="com:oidsDto"/>
            <xs:element name="propertiesModified" type="xs:int"/>
③
        </xs:sequence>
    </xs:complexType>
</xs:schema>
```

① the number of objects that were loaded, in total, by the interaction.

② the identities of the objects that were, respectively, created, updated or deleted within the transaction.

③ the number of objects' properties changed, in total, by the interaction.

The interaction schema also provides metrics on the number of objects loaded/changed, but is more granular, each figure relating to a single (sub-)execution within an interaction.

4.4. Action Invocation Memento

The "aim" schema defines the serialized form (or memento) of an action invocation.

This schema has been removed in `1.13.0`, replaced with ixn.xsd (for action invocations/property edits) and with cmd.xsd (commands, ie the *intention* to invoke an action/edit a property).

The remaining content on this page describes how `CommandContext` works up to v1.12.x. However, as of `1.13.0` the `CommandContext` uses its own `cmd.xsd` schema).

Action invocations are captured (in memory rather than in serialized form) when the end-user invokes the action "through" the UI, by way of the CommandContext service. Using the `ActionInvocationMementoDtoUtils` utility class, a service can instantiate `ActionInvocationMementoDto` which can then be serialized to/from using the same `ActionInvocationMementoDtoUtils` class.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema targetNamespace="http://isis.apache.org/schema/aim"
①
           elementFormDefault="qualified"
           xmlns:xs="http://www.w3.org/2001/XMLSchema"
           xmlns="http://isis.apache.org/schema/aim"
           xmlns:common="http://isis.apache.org/schema/common">

    <xs:import namespace="http://isis.apache.org/schema/common"
②
               schemaLocation="http://isis.apache.org/schema/common/common-1.0.xsd"/>

    <xs:element name="actionInvocationMementoDto">
③
        <xs:complexType>
            <xs:sequence>
                <xs:element name="metadata">
                    <xs:complexType>
                        <xs:sequence>
                            <xs:element name="transactionId" type="xs:string"/>
④
                            <xs:element name="sequence" type="xs:int"/>
⑤
                            <xs:element name="timestamp" type="xs:dateTime"/>
⑥
                            <xs:element name="target" type="common:oidDto"/>
⑦
                            <xs:element name="targetClass" type="xs:string"/>
⑧
                            <xs:element name="targetAction" type="xs:string"/>
⑨
                            <xs:element name="actionIdentifier" type="xs:string"/>
⑩
                            <xs:element name="user" type="xs:string"/>
⑪
                            <xs:element name="title" type="xs:string"/>
⑫
                        </xs:sequence>
```

```
                    </xs:complexType>
                </xs:element>
                <xs:element name="payload">
                    <xs:complexType>
                        <xs:sequence>
                            <xs:element name="parameters">
⑬
                                <xs:complexType>
                                    <xs:sequence maxOccurs="unbounded">
                                        <xs:element name="param" type="paramDto"/>
                                    </xs:sequence>
                                    <xs:attribute name="num" use="required" type=
"xs:int"/>
                                </xs:complexType>
                            </xs:element>
                            <xs:element name="return" type="common:valueDto"
⑭
                                        minOccurs="0" maxOccurs="1"/>
                        </xs:sequence>
                    </xs:complexType>
                </xs:element>
            </xs:sequence>
        </xs:complexType>
    </xs:element>

    <xs:complexType name="paramDto">
⑮
        <xs:sequence>
            <xs:element name="value" type="common:valueDto"/>
⑯
        </xs:sequence>
        <xs:attribute name="parameterName" use="required" type="xs:string"/>
⑰
        <xs:attribute name="parameterType" use="required" type="common:valueType"/>
        <xs:attribute name="null" use="optional" type="xs:boolean"/>
    </xs:complexType>
</xs:schema>
```

① the aim schema has a namespace URI of "http://isis.apache.org/schema/aim". Although URIs are not the same as URLs, you will find that the schemas are also downloadable from this location.

② reuses the common schema

③ definition of the `actionInvocationMementoDto` complex type. This consists of metadata (the transaction identifier, the target object, the initiating user) and the payload (the action parameter/arguments, the return value if known).

④ the unique transaction Id (a guid) allocated by the framework for each and every transaction

⑤ a sequence number within the transaction. It is possible for there to be more than one action invocation to be

⑥ when the action was invoked

⑦ target object, as an OID (using `oidDto` from the common schema)

⑧ fully qualified class name of the target object, for information only

⑨ name of the action, for information only

⑩ Javadoc style unique identifier for the action.

⑪ User that invoked the action

⑫ title of the target object, for information only

⑬ Collection of parameter/arguments, defined in terms of the `paramDto` complex type (discussed just below)

⑭ The return value of the action, if known (and not void)

⑮ The `paramDto` defines both an action parameter and its corresponding argument values

⑯ The value of the parameter, in other words an argument value

⑰ Metadata about the parameter itself: its name, type, optionality.

As of `1.11.0` through `1.12.2` this schema is not used directly by the framework; in particular `Command#setMemento(…)` sets a similar but less formal XML structure. This may change in the future.

4.5. Common Schema

The "common" schema defines a number of complex types that are used by other higher-level schemas.

4.5.1. `oidDto`

The `oidDto` complex type captures an object's type and its identifier. This is basically a formal XML equivalent to the `Bookmark` object obtained from the `BookmarkService`.

Although simple, this is an enormously powerful concept, in that it represents a URI to any domain object managed by a given Apache Isis application. With it, we have the ability to lookup any arbitrary object. Further discussion and examples can be found here.

The `oidDto` complex type is defined as:

```
<xs:schema targetNamespace="http://isis.apache.org/schema/common"
①
          elementFormDefault="qualified"
          xmlns="http://isis.apache.org/schema/common"
          xmlns:xs="http://www.w3.org/2001/XMLSchema">

    <xs:complexType name="oidDto">
②
        <xs:sequence/>
        <xs:attribute name="type" type="xs:string"/>
③
        <xs:attribute name="id" type="xs:string"/>
④
        <xs:attribute name="objectState" type="bookmarkObjectState"/>
    </xs:complexType>

    <xs:simpleType name="bookmarkObjectState">
⑤
        <xs:restriction base="xs:string">
            <xs:enumeration value="persistent"/>
            <xs:enumeration value="transient"/>
            <xs:enumeration value="viewModel"/>
        </xs:restriction>
    </xs:simpleType>

    <xs:complexType name="oidsDto">
⑥
        <xs:sequence>
            <xs:element name="oid" type="oidDto" minOccurs="1" maxOccurs="unbounded"/>
        </xs:sequence>
    </xs:complexType>
    ...
</xs:schema>
```

① the common schema has a namespace URI of "http://isis.apache.org/schema/common". Although URIs are not the same as URLs, you will find that the schemas are also downloadable from this location.

② the `oidDto` complex type defines the unique identifier for any domain object: its type, and an identifier. The `objectState` attribute can usually be omitted (indicating a persistent object)

③ the object type, corresponding to either the `@DomainObject#objectType()` attribute, or to the (JDO) `@PersistenceCapable` annotation (`schema` and/or `table` attributes), or to the (JDO) `@Discriminator` annotation. If none is specified, then the fully qualified class name will be used.

④ the object identifier (aka primary key), converted to string form.

⑤ the `bookmarkObjectState` enumerates the possible persistence states of the referenced object. In previous versions of the schema the attribute was defaulted to "persistent"; the "persistent" state is assumed if the attribute is omitted.

⑥ Models a list of OIDs. This is used by the "cmd" schema to represent the intention to perform a

bulk actions (against a number of selected objects).

In previous versions of the schema the object type and object identifers of `oidDto` were modelled as an element rather than an attribute. The element form can still be used, but is deprecated.

The `oidDto` complex type is used in a number of places by the framework:

- first, as a means of serializing JAXB view model/DTOs (annotated with `@XmlRootElement`), that reference domain entities.

  These references are serialized instead into OIDs

- second, as references to the target of a command representing the *intention* to invoke an action or edit a property, as described by the "cmd" (command) schema.

  They are also used to represent references to any action arguments/properties that take domain object entities/view models.

- third, as references to the target of an interaction capturing the actual execution of an action invocation or property edit, as described by the "ixn" (interaction) schema.

4.5.2. `valueDto` etc

The common schema also defines two types representing values: the `valueDto` complex type, the `valueType` simple type and the `valueWithTypeDto` complex type:

```
<xs:schema targetNamespace="http://isis.apache.org/schema/common" ... >
    ...
    <xs:complexType name="valueDto">                                    ①
        <xs:choice minOccurs="0" maxOccurs="1">
            <xs:element name="string" type="xs:string"/>
            <xs:element name="byte" type="xs:byte"/>
            <xs:element name="short" type="xs:short"/>
            ...
            <xs:element name="timestamp" type="xs:dateTime"/>
            <xs:element name="enum" type="enumDto"/>
            <xs:element name="reference" type="oidDto"/>
        </xs:choice>
    </xs:complexType>

    <xs:simpleType name="valueType">                                    ②
        <xs:restriction base="xs:string">
            <xs:enumeration value="string"/>
            <xs:enumeration value="byte"/>
            <xs:enumeration value="short"/>
            ...
            <xs:enumeration value="enum"/>
            <xs:enumeration value="reference"/>
            <xs:enumeration value="void"/>                              ③
        </xs:restriction>
    </xs:simpleType>

    <xs:complexType name="valueWithTypeDto">                            ④
        <xs:complexContent>
            <xs:extension base="valueDto">
                <xs:attribute name="type" use="required" type="valueType"/>
                <xs:attribute name="null" use="optional" type="xs:boolean"/>
            </xs:extension>
        </xs:complexContent>
    </xs:complexType>
    ...
</xs:schema>
```

① Intended to hold any valid value, eg of an argument to an action or a new value of a property.

② Enumerates the full set of types understood by the framework; note that these also include references to entities or view models, and to enums.

③ Not valid to be used as the parameter type of an action; can be used as its return type.

④ Inherits from `valueDto`, capturing both a value and its corresponding type. Used for the return value of action invocations, and for the new value in property edits.

These type definitions are just building blocks, also used within the action iInvocation memento schema. The first, `valueDto` is The second, `valueType`, enumerates the different types of vales, eg of a formal parameter to an action.

4.5.3. Ancillary types

The common schema also defines a number of ancillary types, used either by the common schema itself (see above) or by the "cmd" and "ixn" schemas.

```
<xs:schema targetNamespace="http://isis.apache.org/schema/common" ... >
    ...
    <xs:complexType name="enumDto">                                     ①
        <xs:sequence>
            <xs:element name="enumType" type="xs:string"/>
            <xs:element name="enumName" type="xs:string"/>
        </xs:sequence>
    </xs:complexType>

    <xs:complexType name="periodDto">                                   ②
        <xs:sequence>
            <xs:element name="startedAt" type="xs:dateTime"/>
            <xs:element name="completedAt" type="xs:dateTime"
                        minOccurs="0" maxOccurs="1"/>
        </xs:sequence>
    </xs:complexType>

    <xs:complexType name="differenceDto">                               ③
        <xs:sequence/>
        <xs:attribute name="before" type="xs:int"/>
        <xs:attribute name="after" type="xs:int"/>
    </xs:complexType>

    <xs:simpleType name="interactionType">                              ④
        <xs:restriction base="xs:string">
            <xs:enumeration value="action_invocation" />
            <xs:enumeration value="property_edit" />
        </xs:restriction>
    </xs:simpleType>
</xs:schema>
```

① Models an instance member of an enum (eg `Color.RED`).

② Captures a period of time, eg for capturing metrics/timings.

③ Captures a pair of numbers representing a difference. Used for example to capture metrics (number objects modified before and after).

④ Whether this command/interaction with a member is invoking an action, or editing a property. Used by both the "cmd" and "ixn" schemas.