Polymorphic Associations Library

Table of Contents

This module (`isis-module-poly`) provides a set of helpers to support the definition of polymorphic associations; that is: relationships from one persistent entity to another by means of a (Java) interface.

Note that an library has has been superceded by a design pattern that does not require any supporting classes. Rather than emit events on the event bus, the design pattern suggests defining an SPI interface to be implemented.

Modules that use the SPI pattern (as opposed to this module) include Alias subdomain and Classification subdomain.

Rationale

Persistable polymorphic associations are important because they allow decoupling between classes using the dependency inversion principle; module dependencies can therefore by kept acyclic. This is key to long-term maintainability of the codebase (avoiding the big ball of mud anti-pattern).

While JDO/DataNucleus has several built-in strategies to support polymorphic associations, none allow both a persisted reference to be arbitrarily extensible as well as supporting foreign keys (for RDBMS-enforced referential integrity). The purpose of this module is therefore to provide helper classes that use a different approach, namely the "table of two halves" pattern.

Table of Two Halves Pattern

The "table of two halves" pattern models the relationship tuple itself as a class hierarchy. The supertype table holds a generic polymorphic reference to the target object (leveraging Apache Isis' Bookmark Service) while the subtype table holds a foreign key is held within the subtype.

It is quite possible to implement the "table of two halves" pattern without using the helpers provided by this module; indeed arguably there's more value in the demo application that accompanies this module (discussed below) than in the helpers themselves. Still, the helpers do provide useful structure to help implement the pattern.

Demo Application

This module has a comprehensive demo fixtures that demonstrates four different polymorphic associations:

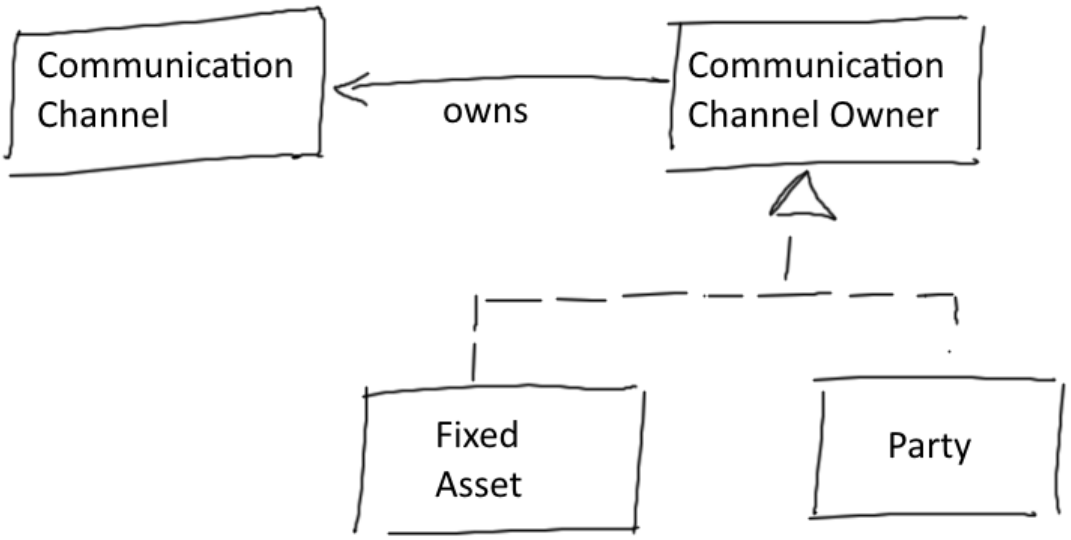
- 1-to-1 and n-to-1: a `CommunicationChannel` may be owned by a `CommunicationChannelOwner`.
- 1-to-n: a `Case` may contain multiple `CaseContent`s
- 1-to-1: a `Case` may have a primary `CaseContent`.

The `CommunicationChannel` and `Case` are regular entities, while `CommunicationChannelOwner` and `CaseContent` are (Java) interfaces.

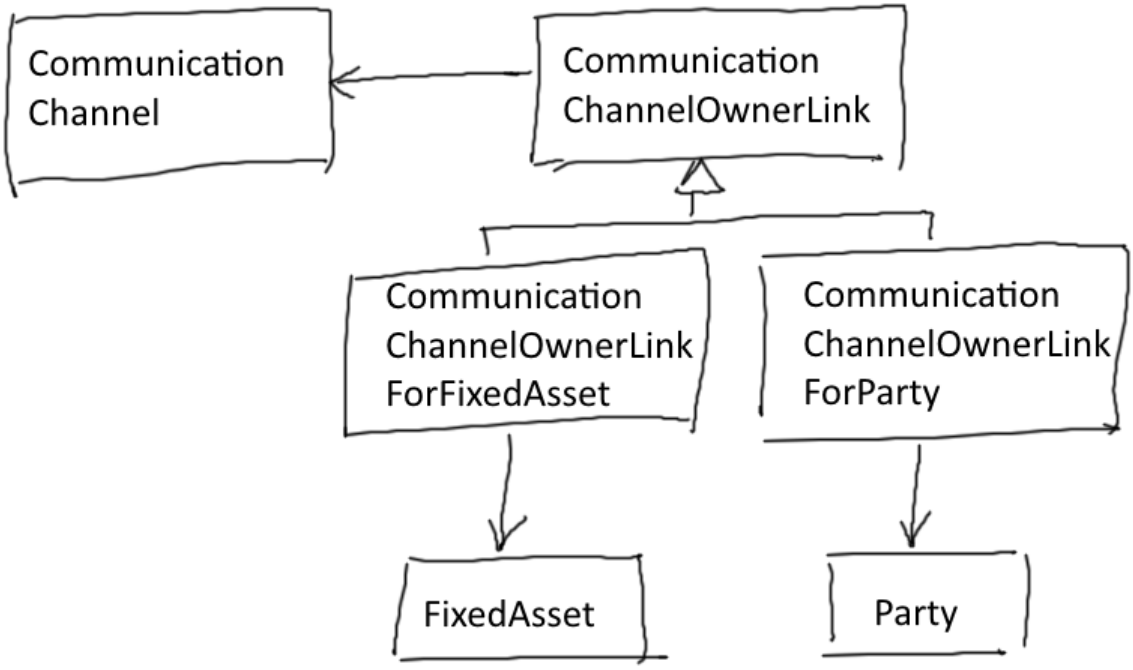
The demo app also has two entities, `FixedAsset` and `Party`, that both implement each of these interfaces. Each `FixedAsset` may "own" a single `CommunicationChannel`, while a `Party` may "own" multiple `CommunicationChannel`s. Meanwhile both `FixedAsset` and `Party` can be added as the "contents" of multiple `Case`s, and either can be used as a `Case's "primary content".

Communication Channel

The following UML diagram shows the "logical" polymorphic association between `CommunicationChannel` and its owning `CommunicationChannelOwner`:

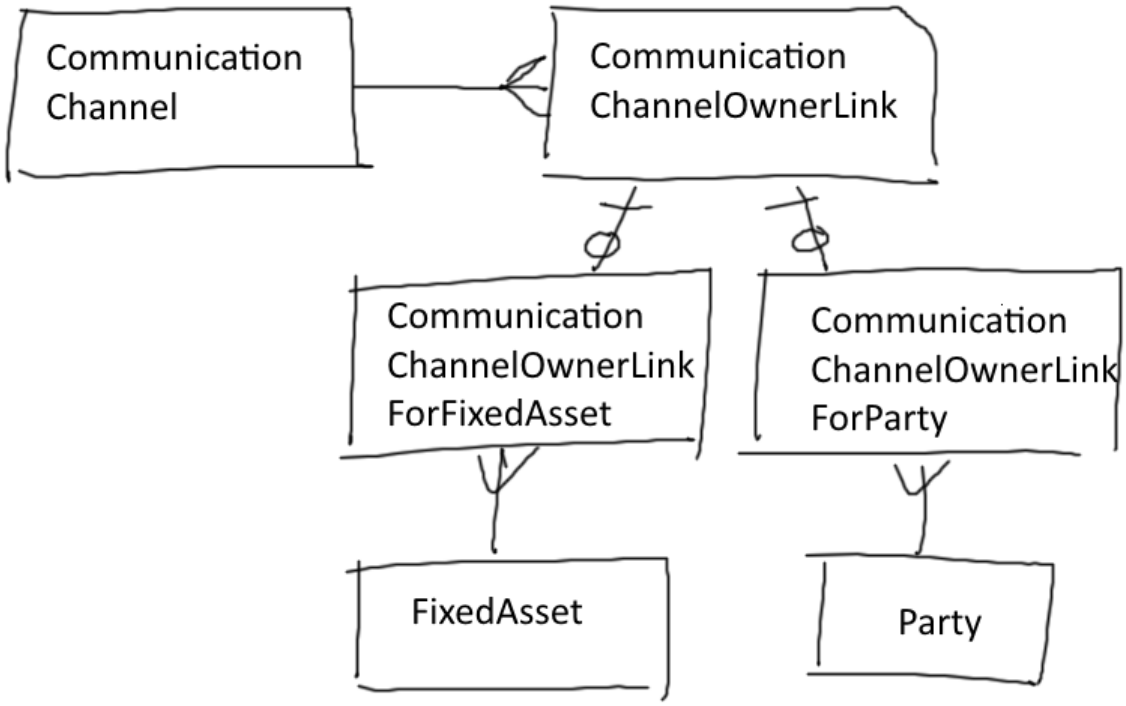


This is realized using the following entities:



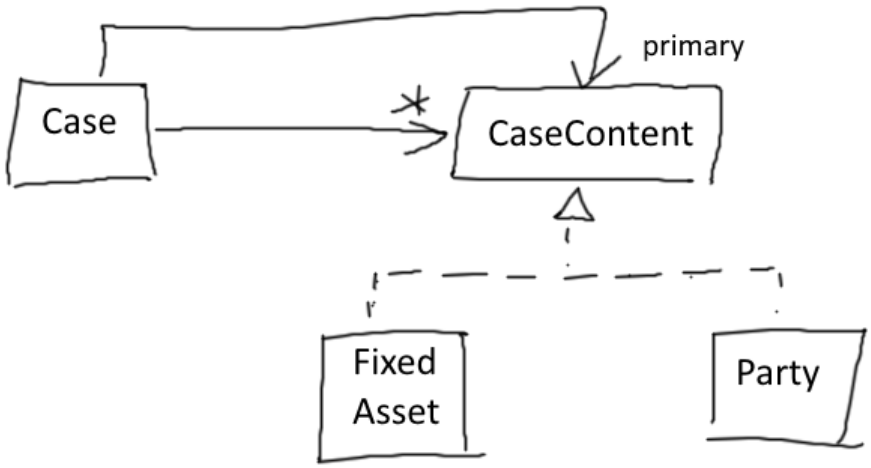
Here the `CommunicationChannelOwnerLink` is a persistent entity that has subtypes for each of the implementations of the `CommunicationChannelOwner` interface, namely `CommunicationChannelOwnerLinkForFixedAsset` and `CommunicationChannelOwnerLinkForParty`. This inheritance hierarchy can be persisted using any of the [standard strategies](http://www.datanucleus.org/products/datanucleus/jdo/orm/inheritance.html) supported by JDO/DataNucleus.

In the demo application the NEW_TABLE strategy is used, giving rise to these tables:



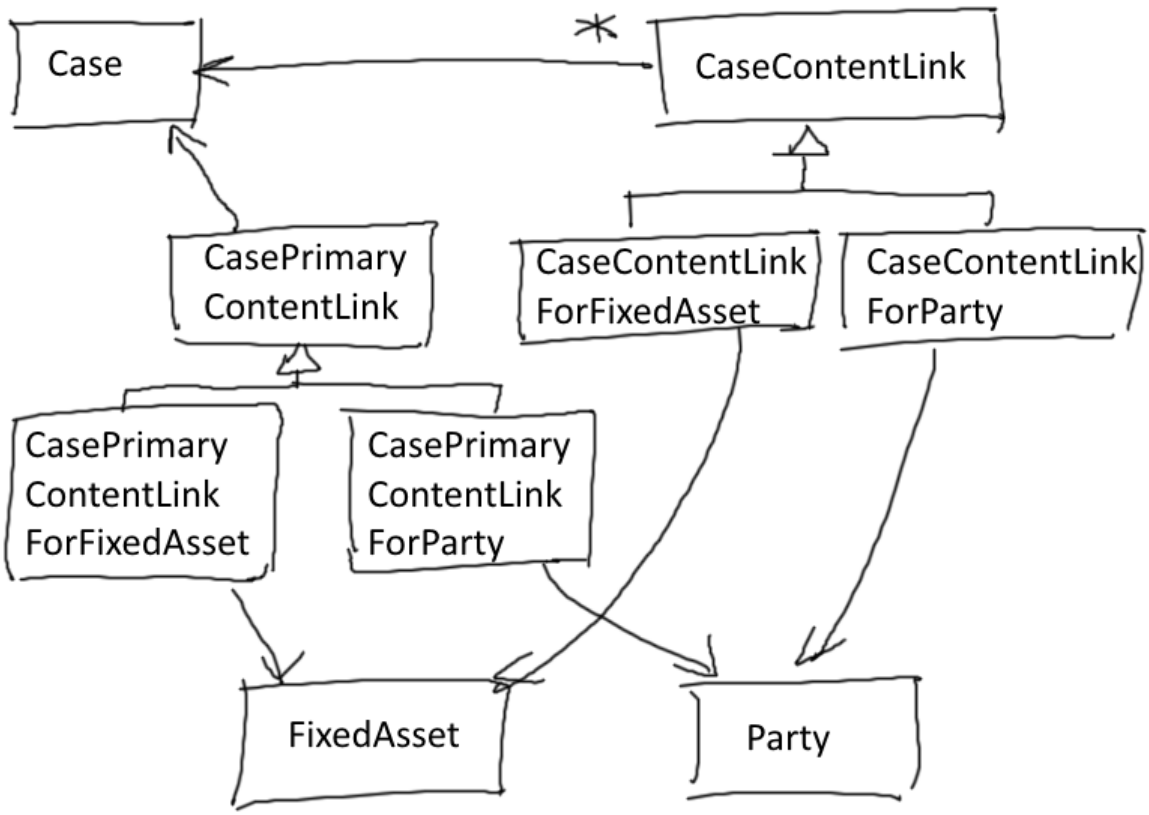
Case Contents

The following UML diagram shows the (two) "logical" polymorphic assocations between `Case` and its `CaseContent`s:



Note how `Case` actually has *two* polymorphic associations: a 1:n to its "contents", and a 1:1 to its "primary content".

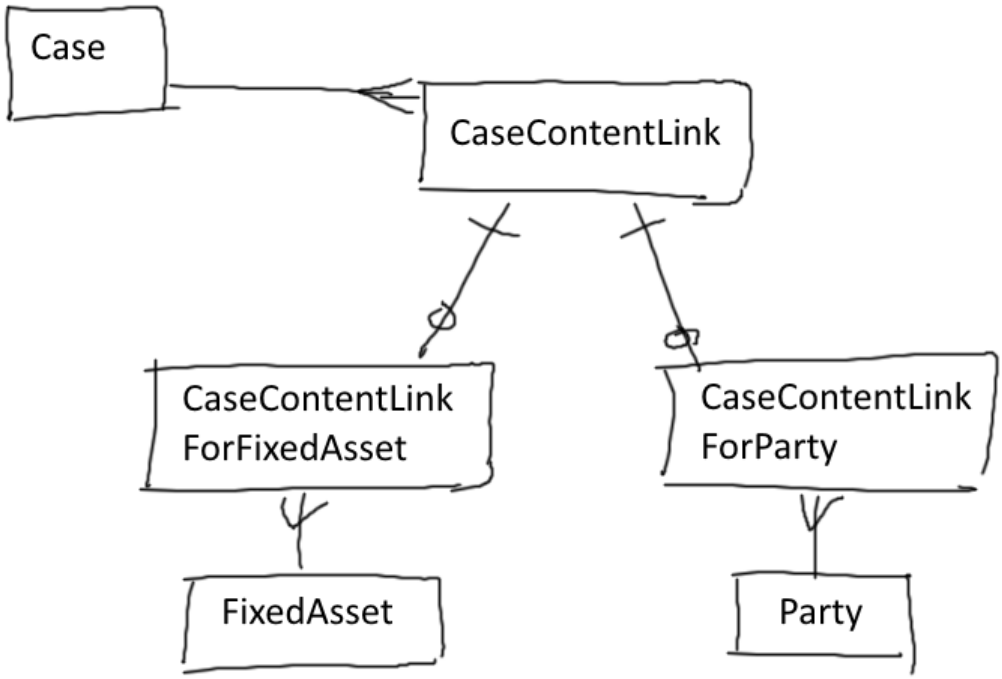
This is realized using the following entities:



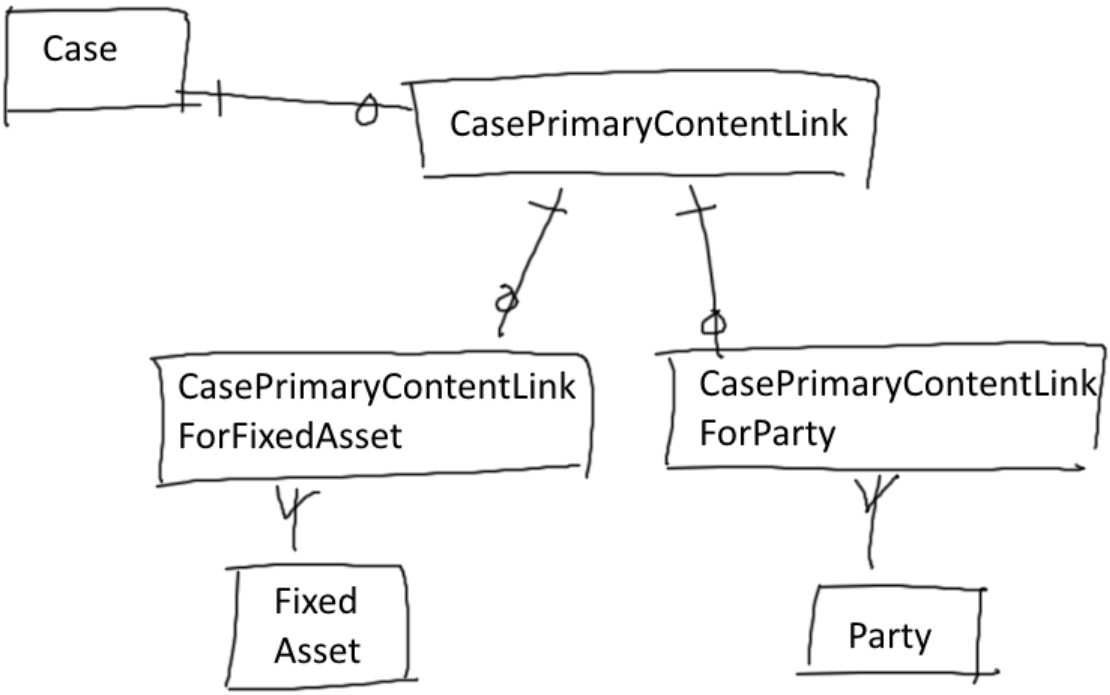
Here the `CaseContentLink` is a persistent entity that (as for communication channels) has subtypes

for each of the implementations of the `CaseContent` interface. But because `Case` actually has two associations to `CaseContent`, there is also a further `CasePrimaryContentLink` persistent entity, again with subtypes.

In the demo application the NEW_TABLE strategy is used for both, giving rise to these tables for the "case content" association:



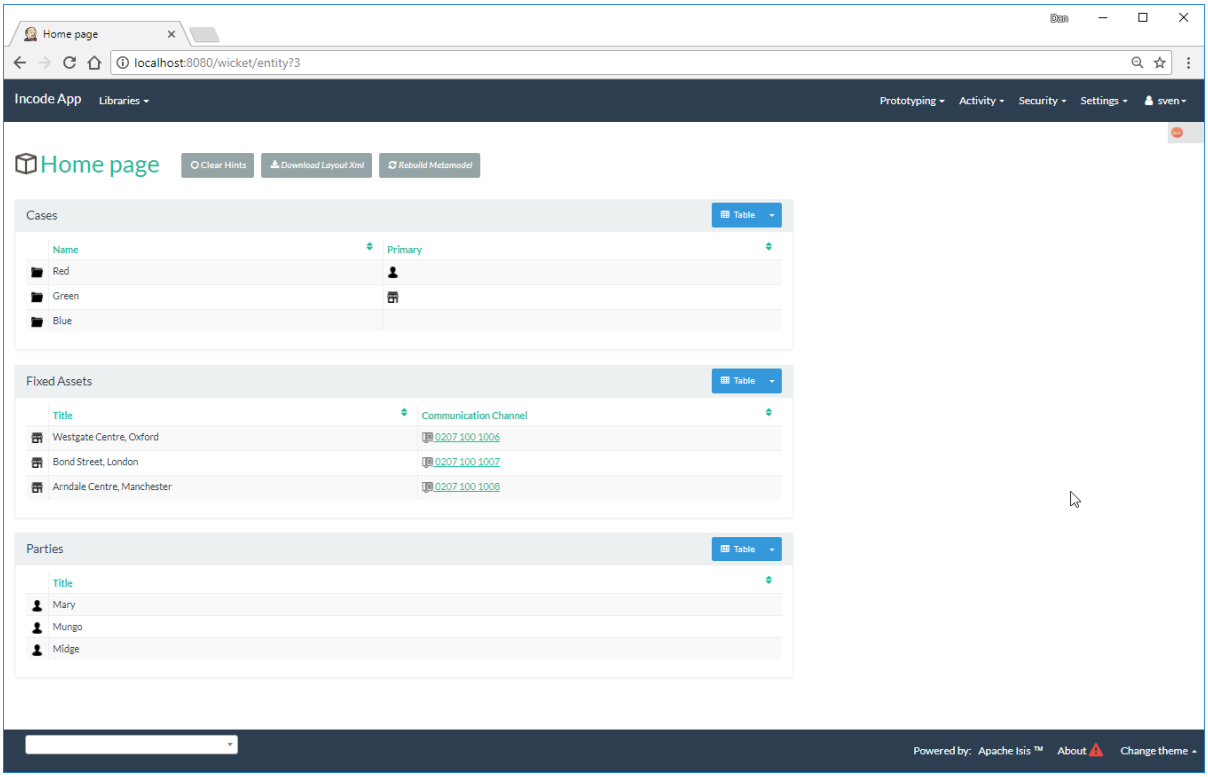
and also to these for the "primary content" association:


Case
CasePrimaryContentLink
CasePrimaryContentLink ForFixedAsset
CasePrimaryContentLink ForParty
Fixed Asset
Party

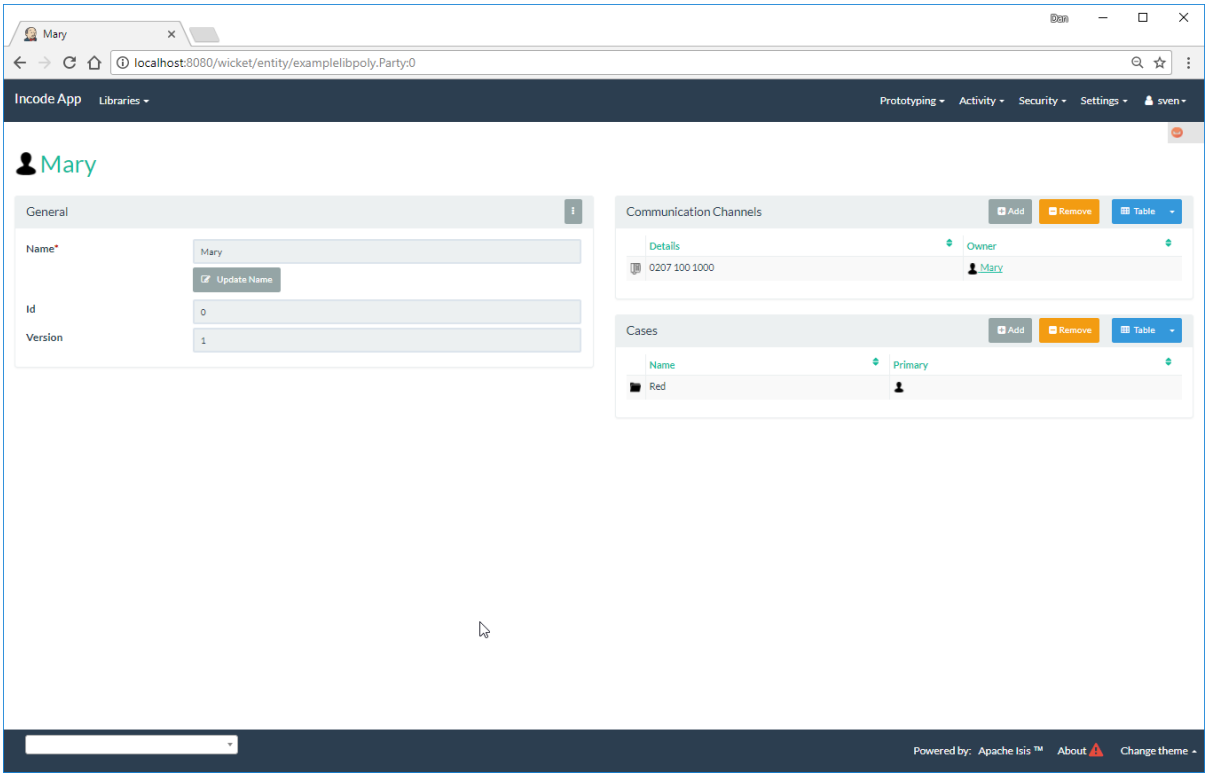
Screenshots

The module's functionality can be explored by running the quickstart with example usage using the `org.incode.domainapp.example.app.modules.ExampleDomLibPolyAppManifest`.

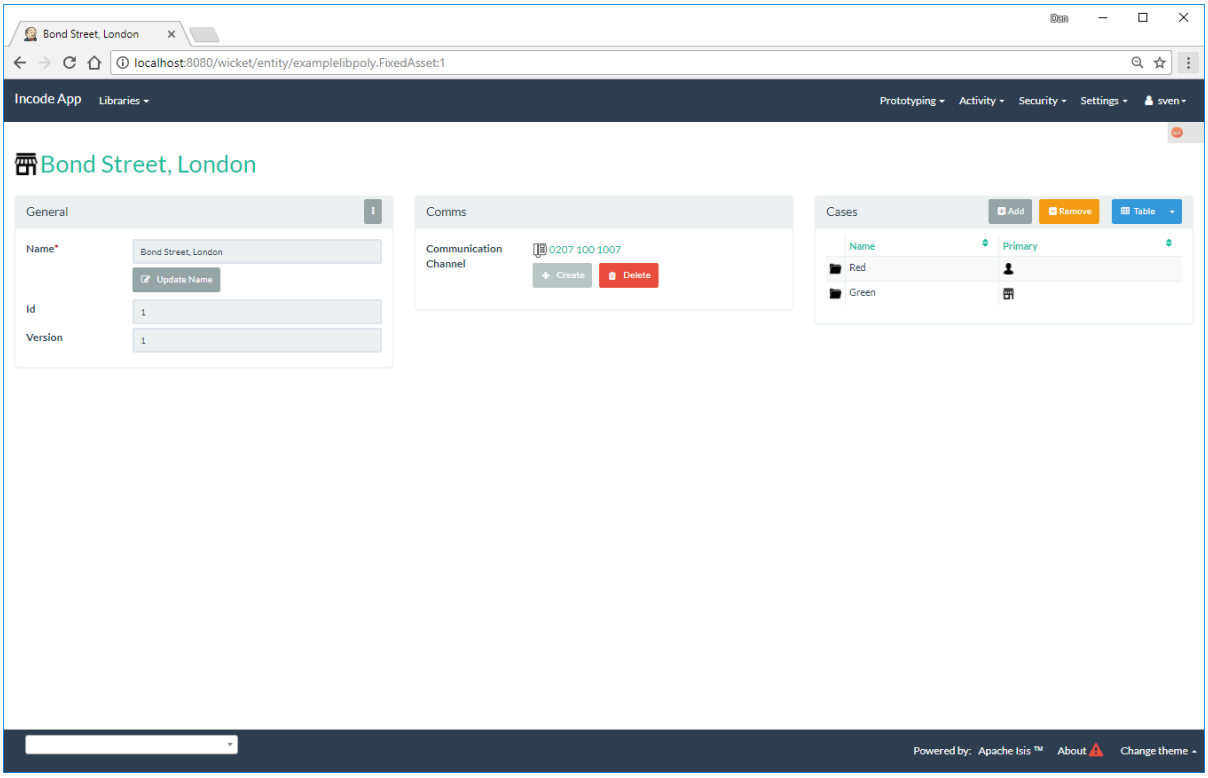
The home page when the app is run shows 3 parties, 3 fixed assets which between them have 9 communication channels. There are also 3 cases and the parties and fixed assets are variously contained within:



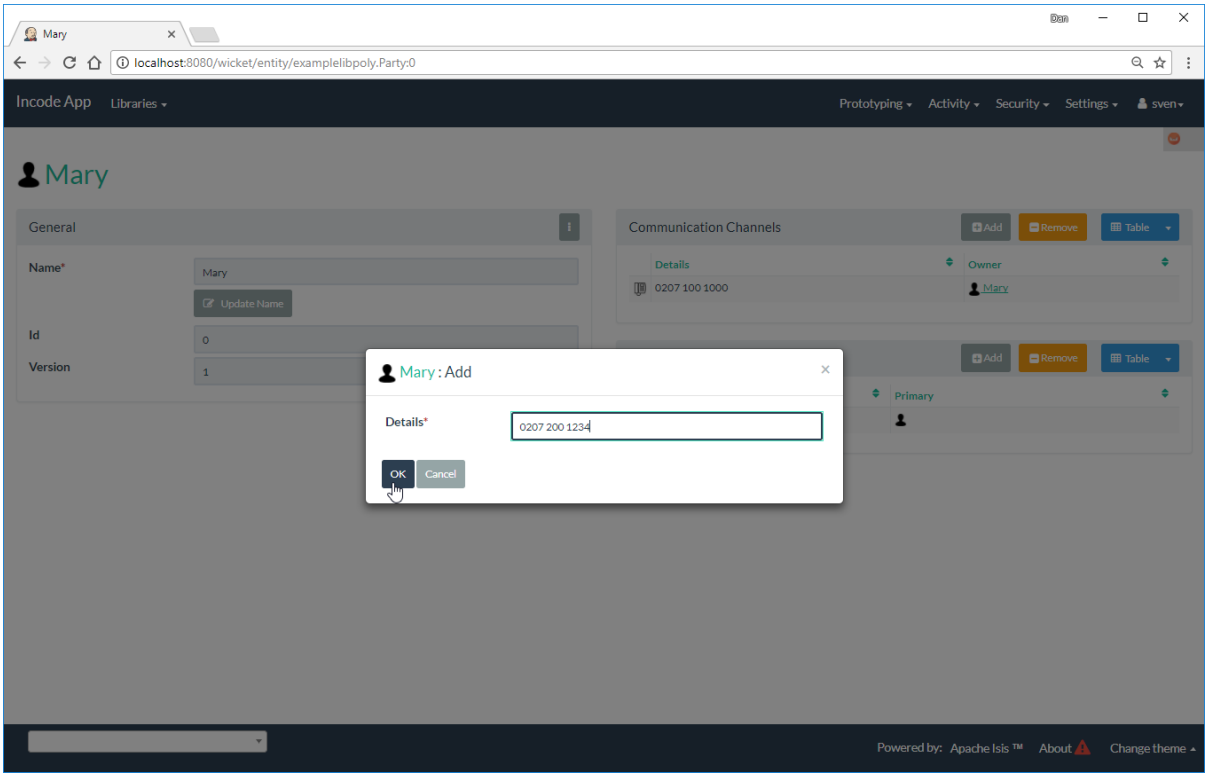
If we navigate to the `Party` entity, we can see that it shows a collection of `CommunicationChannel`s that the party owns, and also a collection of the `Case`s within which the party is contained:



The `FixedAsset` entity is similar in that it also has a collection of `Case`s. However, in our demo app we have a business rule that the fixed asset can own only a single `CommunicationChannel`.

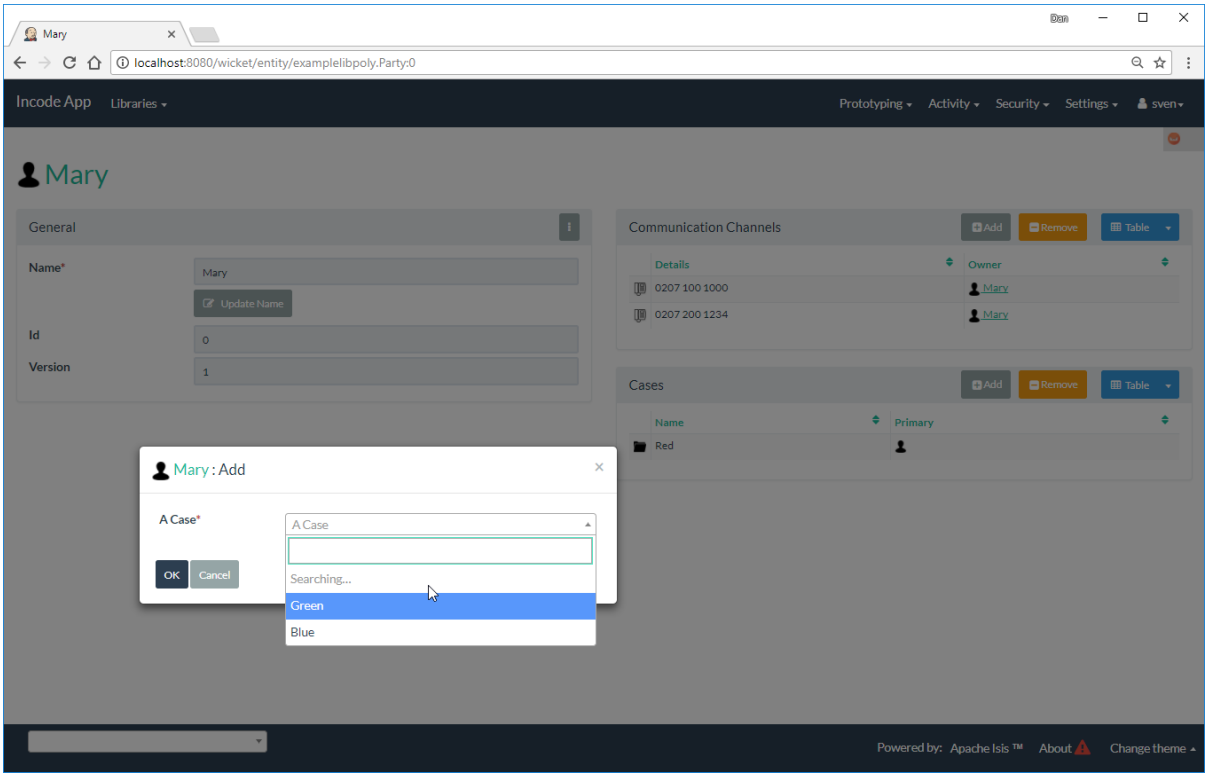


On the `Party` entity we can add (and remove) `CommunicationChannel`s:



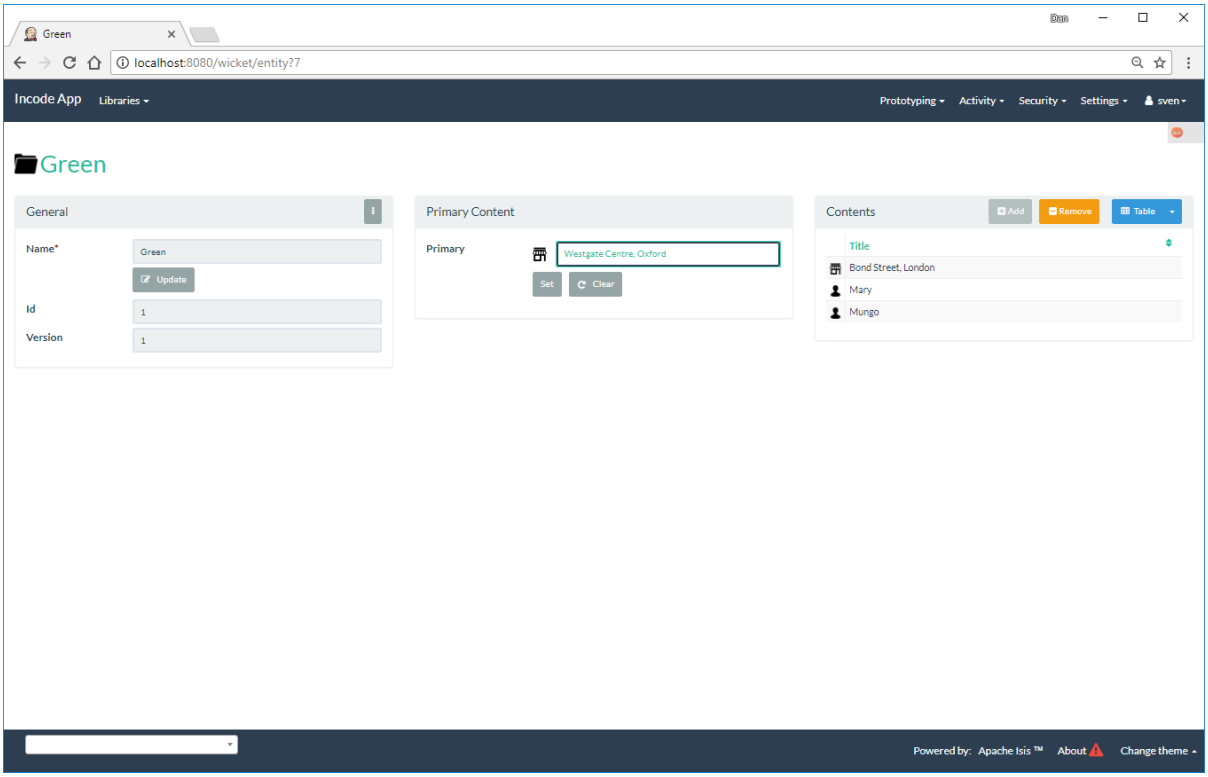
In this demo app, because communication channels are *not* shared by entities, this will actually create and persist the corresponding `CommunicationChannel`.

We can also add (or remove) from `Case`s:

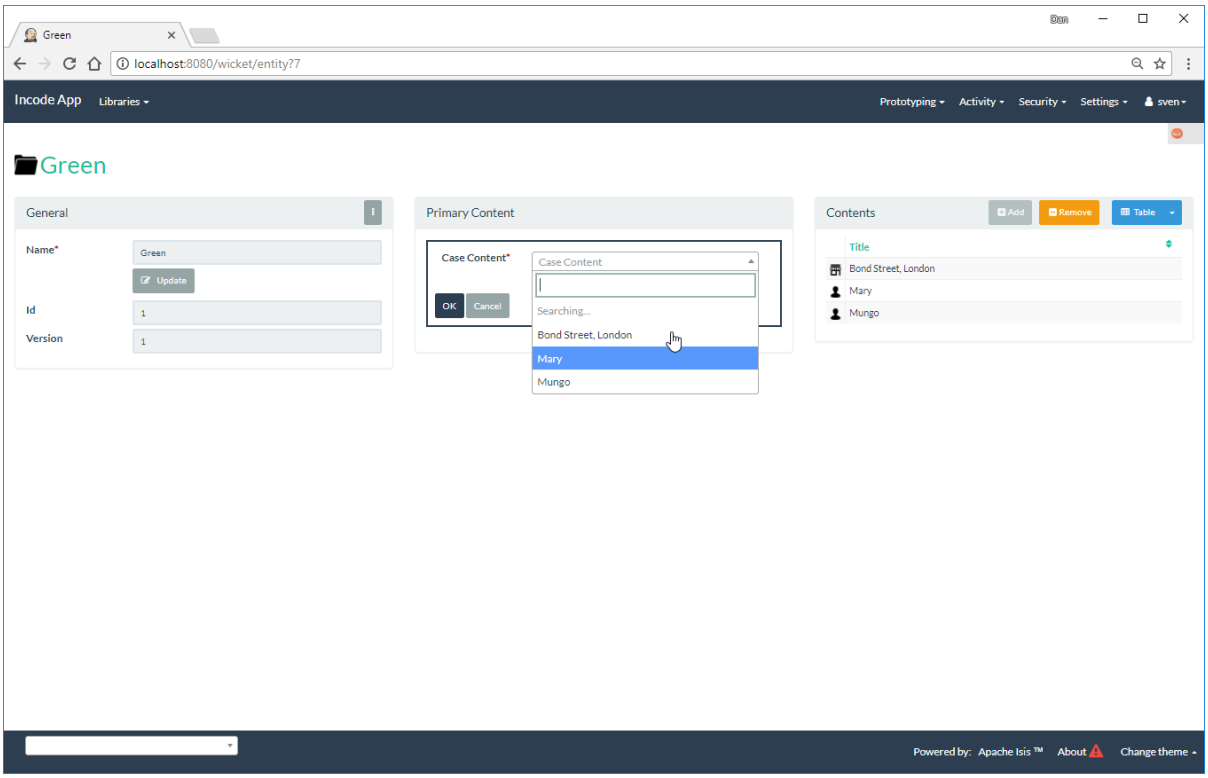


Here the rule is slightly different: the `Case` already exists and so the party is merely associated with an existing case.

From the `Case` entity's perspective, we can see its contents and also its primary content:



As might be expected, we have an action to set (or clear) the primary content:



Design

The key design idea is to leverage Isis' event bus service to determine which concrete subtype should be created and persisted to hold the association.

- when the association needs to be created, an event is posted to the event bus
- the subscriber updates the event with the details of the subtype to be persisted
- if no subscriber updates the event, then the association cannot be created and an exception is thrown.

The helper classes provided by this module factor out some of the boilerplate relating to this design, however there is (necessarily) quite a lot of domain-specific code. What's important is understanding the design and how to replicate it.

The recipe for the pattern is:

Table 1. Recipe

#	Step	Example
1	Create an interface for the target of the association	* `CommunicationChannelOwner` * `CaseContent`
2	Create a persistent entity corresponding to the association	* `CommunicationChannelOwnerLink` + for the `CommunicationChannel`/"owner" association * `CaseContentLink` + for `Case`/"contents" association * `CasePrimaryContentLink` + for `Case`/"primary content" association
3	Create an "instantiate event". We suggest using a nested static class of the link entity:	* `CommunicationChannelOwnerLink.InstantiateEvent` * `CaseContentLink.InstantiateEvent` * `CasePrimaryContentLink.InstantiateEvent`
4	Create a corresponding repository service for that link persistent entity:	* `CommunicationChannelOwnerLinks` * `CaseContentLinks` * `CasePrimaryContentLinks`
5	Create a subtype for each implementation of the target interface:	* `CommunicationChannelOwnerLinkForFixedAsset` and `CommunicationChannelOwnerLinkForParty` * `CaseContentLinkForFixedAsset` and `CaseContentLinkForParty` * `CasePrimaryContentLinkForFixedAsset` and `CasePrimaryContentLinkForParty`

#	Step	Example
6	Create a subscriber to the event for each implementation of the target interface. We suggest using a nested static class of the subtype.	* `CommunicationChannelOwnerLinkForFixedAsset. InstantiateSubscriber` and `CommunicationChannelOwnerLinkForParty. InstantiateSubscriber` * `CaseContentLinkForFixedAsset. InstantiateSubscriber` and `CaseContentLinkForParty. InstantiateSubscriber` * `CasePrimaryContentLinkForFixedAsset. InstantiateSubscriber` and `CasePrimaryContentLinkForParty. InstantiateSubscriber`

API and Usage

The module itself consist of the following classes:

- `PolymorphicAssociationLink` - an abstract class from which to derive the `*Link` entity
- `PolymorphicAssociationLink.InstantiateEvent` - a superclass for the "instantiate event"
- `PolymorphicAssociationLink.Factory` - a utility class that broadcasts the event and persists the link using the requested subtype

Let's look at each in more detail, relating back to the "communication channel owner" association in the demo app.

PolymorphicAssociationLink

A link is in essence a tuple between two entities. One of these links is direct "subject"; the other is the polymorphic reference. The `PolymorphicAssociationLink` class is intended to be used base class for all `*Link` entities (step 2 in the pattern recipe), and defines this structure:

```
public abstract class PolymorphicAssociationLink<
                            S, P, L extends PolymorphicAssociationLink<S, P, L>>
        implements Comparable<L> {

    protected PolymorphicAssociationLink(final String titlePattern) { ... }

    public abstract S getSubject();
    public abstract void setSubject(S subject);

    public abstract String getPolymorphicObjectType();
    public abstract void setPolymorphicObjectType(final String polymorphicObjectType);

    public abstract String getPolymorphicIdentifier();
    public abstract void setPolymorphicIdentifier(final String polymorphicIdentifier);

    public P getPolymorphicReference() { ... }
    public void setPolymorphicReference(final P polymorphicReference) { ... }

    public int compareTo(final PolymorphicAssociationLink other) { ... }
}
```

The subclass is required to implement the `subject`, `polymorphicObjectType` and the `polymorphicIdentifier` properties; these should delegate to the "concrete" properties.

For example, the `CommunicationChannelOwnerLink` looks like:

```
public abstract class CommunicationChannelOwnerLink
        extends PolymorphicAssociationLink<
                    CommunicationChannel, CommunicationChannelOwner,
                    CommunicationChannelOwnerLink> {

    public CommunicationChannelOwnerLink() {
        super("{polymorphicReference} owns {subject}");
    }

    public CommunicationChannel getSubject() {
        return getCommunicationChannel();
    }
    public void setSubject(final CommunicationChannel subject) {
        setCommunicationChannel(subject);
    }

    public String getPolymorphicObjectType() {
        return getOwnerObjectType();
    }
    public void setPolymorphicObjectType(final String polymorphicObjectType) {
        setOwnerObjectType(polymorphicObjectType);
    }

    public String getPolymorphicIdentifier() {
        return getOwnerIdentifier();
    }
    public void setPolymorphicIdentifier(final String polymorphicIdentifier) {
        setOwnerIdentifier(polymorphicIdentifier);
    }

    @lombok.Getter @lombok.Setter
    private CommunicationChannel communicationChannel;      ①

    @lombok.Getter @lombok.Setter
    private String ownerObjectType;                         ①

    @lombok.Getter @lombok.Setter
    private String ownerIdentifier;                         ①
}
```

① JDO persisted properties

Thus, the abstract properties defined by `PolymorphicAssociationLink` just delegate to corresponding persisted (JDO annotated) properties in `CommunicationChannelOwnerLink`.

Also note the pattern passed to the constructor; this is used to generate a title.

PolymorphicAssociationLink.InstantiateEvent

The `PolymorphicAssociationLink.InstantiateEvent` is the base class to derive an instantiate event type for each polymorphic association (step 3 in the pattern recipe). Having derived event classes means that the event subscribers need only receive the exact events that they care about.

The `InstantiateEvent` has the following structure:

```
public abstract static class InstantiateEvent<
                                S, P, L extends PolymorphicAssociationLink<S, P, L>>
        extends java.util.EventObject {

    protected InstantiateEvent(
            final Class<L> linkType,
            final Object source,
            final S subject,
            final P polymorphicReference) { ... }

    public S getSubject() { ... }
    public P getPolymorphicReference() { ... }

    public Class<? extends L> getSubtype() { ... }
    public void setSubtype(final Class<? extends L> subtype) { ... }
}
```

Any subclass is required to take the last three parameters in its constructor; the event is instantiated reflectively by `PolymorphicAssociationLink.Factory`.

For example, the `CommunicationChannelOwnerLink.InstantiateEvent` is simply:

```
public static class InstantiateEvent
        extends PolymorphicAssociationLink.InstantiateEvent<
                    CommunicationChannel, CommunicationChannelOwner,
                    CommunicationChannelOwnerLink> {

    public InstantiateEvent(
            final Object source,
            final CommunicationChannel subject,
            final CommunicationChannelOwner owner) {
        super(CommunicationChannelOwnerLink.class, source, subject, owner);
    }
}
```

PolymorphicAssociationLink.Factory

The final class `PolymorphicAssociationLink.Factory` is responsible for broadcasting the event and then persisting the appropriate subtype for the link. It has the following structure:

```
public static class Factory<S,P,L extends PolymorphicAssociationLink<S,P,L>,
                            E extends InstantiateEvent<S,P,L>> {

    public Factory(
            final Object eventSource,
            final Class<S> subjectType,
            final Class<P> polymorphicReferenceType,
            final Class<L> linkType, final Class<E> eventType) { ... }

    public void createLink(final S subject, final P polymorphicReference) { ... }

}
```

Unlike the other two classes, the factory is not subclassed. Instead, it should be instantiated as appropriate. Typically this will be in a repository service for the `*Link` entity (step 4 in the pattern recipe).

For example, with the communication channel example the `Factory` is instantiated in the `CommunicationChannelOwnerLinks` repository service:

```
public class CommunicationChannelOwnerLinks {

    PolymorphicAssociationLink.Factory<
            CommunicationChannel,
            CommunicationChannelOwner,
            CommunicationChannelOwnerLink,
            CommunicationChannelOwnerLink.InstantiateEvent> linkFactory;

    @PostConstruct
    public void init() {
        linkFactory = container.injectServicesInto(
                new PolymorphicAssociationLink.Factory<>(
                        this,
                        CommunicationChannel.class,
                        CommunicationChannelOwner.class,
                        CommunicationChannelOwnerLink.class,
                        CommunicationChannelOwnerLink.InstantiateEvent.class
                ));

    }

    public void createLink(
            final CommunicationChannel communicationChannel,
            final CommunicationChannelOwner owner) {
        linkFactory.createLink(communicationChannel, owner);
    }
}
```

Note that it is necessary to inject services into the factory (`container.injectServicesInto(…)`).

Completing the Pattern

The helper classes provided by this module are actually only used by the "subject" domain entity (or the containing package for said entity); steps 1 through 4 in the pattern recipe. But what about the implementation for an entity (such as `FixedAsset`) that wishes to be used in such a polymorphic association, ie the final steps 5 and 6?

Step 5 of the pattern requires a subtype of the `*Link` entity specific to the subtype to be reference. For example, for `FixedAsset` this looks like:

```
public class CommunicationChannelOwnerLinkForFixedAsset
        extends CommunicationChannelOwnerLink {

    @Override
    public void setPolymorphicReference(final CommunicationChannelOwner polyReference)
{
        super.setPolymorphicReference(polyReference);
        setFixedAsset((FixedAsset) polyReference);
    }

    // JDO persisted property
    private FixedAsset fixedAsset;

}
```

where the inherited `setPolymorphicReference(…)` method is overridden to also populate the JDO persisted property (`fixedAsset` in this case).

And, finally, step 6 defines a subscriber on the instantiate event. We recommend this is a nested static class of the `*Link` subtype, and so:

```
public class CommunicationChannelOwnerLinkForFixedAsset
                        extends CommunicationChannelOwnerLink {

    @DomainService(nature = NatureOfService.DOMAIN)
    public static class InstantiationSubscriber extends AbstractSubscriber {

        @Programmatic
        @Subscribe
        public void on(final CommunicationChannelOwnerLink.InstantiateEvent ev) {
            if(ev.getPolymorphicReference() instanceof FixedAsset) {
                ev.setSubtype(CommunicationChannelOwnerLinkForFixedAsset.class);
            }
        }
    }
}
```

The thing to note is that although there are quite a few steps (1 through 4, in fact) to make an association polymorphic, the steps to then reuse that polymorphic association (steps 5 and 6) are really rather trivial.

Some quick asides

The demo application has a couple of other interesting implementation details - not to do with polymorphic associations - but noteworthy nonetheless.

Use of event bus for cascade delete

With the `Case` class there is a "case contents" and a "primary case content"; the idea being that the primary content should be one in the "contents" collection.

If the case content object that happens to be primary is dissociated from the case, then a `CaseContentContributions.RemoveFromCaseDomainEvent` domain event is broadcast. A subscriber listens on this to delete the primary case link:

```
public class CasePrimaryContentSubscriber extends AbstractSubscriber {

    @Subscribe
    public void on(final CaseContentContributions.RemoveFromCaseDomainEvent ev) {
        switch (ev.getEventPhase()) {
            case EXECUTING:
                final CasePrimaryContentLink link =
                            casePrimaryContentLinks.findByCaseAndContent(
                                                ev.getCase(), ev.getContent());
                if(link != null) {
                    container.remove(link);
                }
                break;
        }
    }
}
```

Contributed properties for collections of an interface type

It (currently) isn't possible to define (fully abstract) properties on interfaces, meaning that by default a collection of objects implementing an interface (eg `Case's "caseContents" collection) would normally only show the icon of the object; not particularly satisfactory.

However, Isis **does** support the notion of contributed properties to interfaces. The demo application uses this trick for the "caseContents" in the `CaseContentContributions` domain service:

```
public class CaseContentContributions {

    @Action( semantics = SemanticsOf.SAFE )
    @ActionLayout( contributed = Contributed.AS_ASSOCIATION )
    @PropertyLayout( hidden = Where.OBJECT_FORMS )
    public String title(final CaseContent caseContent) {
        return container.titleOf(caseContent);
    }
}
```

Moreover, this trick contributes to all implementations (`FixedAsset` and `Party`).

There is however a small gotcha, in that we only want this contributed property to be viewed on tables. The `@Property(hidden=Where.OBJECT_FORMS)` ensures that it is not shown anywhere else.

How to configure/use

Classpath

Update your classpath by adding this dependency in your dom project's `pom.xml`:

```
<dependency>
    <groupId>org.isisaddons.module.poly</groupId>
    <artifactId>isis-module-poly-dom</artifactId>
    <version>1.16.1</version>
</dependency>
```

Check for later releases by searching Maven Central Repo.

For instructions on how to use the latest `-SNAPSHOT`, see the contributors guide.

Bootstrapping

In the `AppManifest`, update its `getModules()` method, eg:

```
@Override
public List<Class<?>> getModules() {
    return Arrays.asList(
            ...
            org.isisaddons.module.poly.PolyModule.class,
            ...
    );
}
```

Known issues

None known at this time.

Dependencies

Maven can report modules dependencies using:

```
mvn dependency:list -o -pl modules/lib/poly/impl -D excludeTransitive=true
```

which, excluding Apache Isis itself, returns no direct compile/runtime dependencies.