Cúram 8.1.2

Persistence Cookbook

Note

Before using this information and the product it supports, read the information in

Edition

This edition applies to Cúram 8.1, 8.1.1, and 8.1.2.

Contents

1 Developing with the Persistence Infrastructure

Use this information to learn how to use and develop service-layer APIs, and how to customize software that uses the persistence infrastructure. The persistence infrastructure uses façade-layer logic to translate data that is received from a user interface screen into a format suitable for passing into a service-layer API call. It also translates the data that is returned from a service-layer API call into a format suitable for returning to a user interface screen.

1.1 Introduction

Intended Audience

This documented is intended to be read and used by designers and developers of server application functionality which:

- calls service-layer APIs developed using the Persistence Infrastructure; and/or
- is developed as service-layer APIs using the Persistence Infrastructure.

Background

The service layer in "classic" Cúram was developed using an approach which combined:

- "Process classes", which contained processing logic only (i.e. no data); and
- "Struct classes", which contained data only (i.e. no processing logic).

By comparison, a service layer developed using the Persistence Infrastructure contains classes which contain both processing and data.

Thus a service layer developed using the Persistence Infrastructure looks and feels very different to its classic-Cúram counterpart, not only to those designers and developers delivering such a service layer, but also to those designers and developers who must make use of it. Code which calls service-layer APIs is typically either:

- façade-layer logic, responsible for translating the data received from a user interface screen into a format suitable for passing into a service-layer API call, or similarly translating the data returned from a service-layer API call into a format suitable for returning to a user interface screen; or
- server logic in another system, which is designed to re-use the service layer developed using the Persistence Infrastructure.

The purpose of this document is to show developers how to use and develop service-layer APIs, through a series of scenarios and solutions, and how to customize out-of-the-box software that uses the Persistence Infrastructure.

Further Reading

For more information about the classes and interfaces included in the Persistence Infrastructure, see its JavaDoc.

Structure of this document

The scenarios in this cookbook are categorized (according to the task at hand) as follows:

- making calls to service-layer APIs;
- coding service-layer APIs; and
- coding service-layer implementations.

Each of these categories enumerates a number of scenarios, and each scenario describes the problem to be solved and walks through how to "cook up" a solution.

One possible scenario is that you are customizing software provided out-of-the-box. One common reason for doing this is to add attributes to database entities provided out-of-the-box. If this is what you are doing then you may only need to read the following three chapters, after which you may selectively read the rest of this guide as necessary:

- creating a Guice module;
- events;
- using entity context.

There are also chapters covering more advanced topics:

- state transitions; and
- inheritance; and
- adding new searches to existing entities.

1.2 Making calls to service-layer APIs

The scenarios in this section describe how to make calls into service-layer APIs from other code. Typically this "other code" is façade-layer logic.

Whilst service-layer APIs can perform a wide variety of functionality, very typically the overwhelming majority of service-layer API calls are related to the reading or writing of database data. Accordingly, the scenarios in this section are described in terms of database tables.

These scenarios build up a typical façade which controls the

- read;
- insert;
- modification;
- removal;
- cancellation; and
- list

of a data stored on a database table.

You want to read some data from a database table

The problem

You are writing a façade method which needs to:

- retrieve a database row based on its primary key; and

• format the data for return to the user interface, where it will be displayed to the user.

Under classic Cúram, you would have created a call to the generated "entity" method as follows:

```
public class MyFacade {
  // ...
  public SomeEntityDetails viewSomeEntityDetails(
      final SomeEntityKey key) throws AppException,
    InformationalException {

    // create an instance of the return struct
    final SomeEntityDetails someEntityDetails =
        new SomeEntityDetails();

    // objects for reading the database
    final SomeEntity someEntityObj =
        SomeEntityFactory.newInstance();
    final SomeEntityKey someEntityKey = new SomeEntityKey();
    final SomeEntityDtls someEntityDtls;

    // map the key
    someEntityKey.someEntityID = key.someEntityID;

    // do the read
    someEntityDtls = someEntityObj.read(someEntityKey);

    // map the details returned - in this situation the return
    // struct aggregates the generated entity Dtls struct
    someEntityDetails.details = someEntityDtls;

    // return to the client
    return someEntityDetails;
  }
}
```

Figure 1: Façade calling classic Cúram entity to read a database row

How do you read from a database table using a service-layer API (developed using the Persistence Infrastructure)?

The solution

Reading data from a service-layer API (developed using the Persistence Infrastructure) involves writing code using two interfaces, which will be introduced by example:

• the interface for the entity being read; and
• the interface for the entity's Data Access Object ("DAO").

Coding the solution involves these steps:

• create a class variable to hold the DAO;
• create a constructor to request Guice to inject class variables
• use the DAO to retrieve the instance of the entity; and
• access the entity instance to map field values to the client struct.

Create a class variable to hold the DAO

Firstly, you need to create a class member variable for the entity's DAO, and annotate it with @Inject:

```
public class MyFacade {
  // ...

  @Inject
   private SomeEntityDAO someEntityDAO;
```

Figure 2: Creating an injected member variable for a DAO

(The @Inject annotation is provided by Guice, a dependency injector. At runtime, Guice will initialize the someEntityDAO variable to use the configured implementation of SomeEntityDAO. You don't really need to worry about any of this.)

The someEntityDAO object "knows" how to create instances of the entity interface. In this scenario, you'll use the DAO to retrieve the instance of the entity from the database.

Create a constructor to request Guice to inject class variables

Because instances of your class are created outside of Guice's control, you must code an explicit constructor which requests Guice to "inject" class variables (in particular the someEntityDAO variable you created in the previous step):

```
public MyFacade() {
    GuiceWrapper.getInjector().injectMembers(this);
  }
```

Figure 3: Creating a public constructor to inject member variables

If you fail to do this step, then when your application runs you will likely see a NullPointerException when your application attempts to access the someEntityDAO variable.

Use the DAO to retrieve the instance of the entity

In your façade method, code a variable to hold an instance of the entity interface, and set its value by calling.get() on the DAO, passing the key of the database row:

```
// retrieve the instance of the entity
  final SomeEntity someEntity =
 someEntityDAO.get(key.someEntityID);
```

Figure 4: Calling a DAO to get an instance of an entity based on its key

Here, the DAO instance has "dished up" the required instance of the entity interface. someEntity now holds an object which "knows" how to:

- get at data (via "getter" methods); and also
- "do things" with that data (via other methods).

Access the entity instance to map field values to the client struct

Now code calls to the entity "getters" to map fields values to your return struct:

```
// map the details from the entity instance
    someEntityDetails.details.someEntityID = someEntity.getID();
    someEntityDetails.details.name = someEntity.getName();
    someEntityDetails.details.versionNo =
 someEntity.getVersionNo();

    final DateRange dateRange = someEntity.getDateRange();
    someEntityDetails.details.startDate = dateRange.start();
    someEntityDetails.details.endDate = dateRange.end();
```

```
      // ...more mappings
```

Figure 5: Calling getter methods on an entity interface

Points to note:

- Every entity API has a.getID() method, which returns its primary key. There will *not* be a specific getter for the entity's primary key field, e.g. there is no someEntity.getSomeEntityID() method.
- The API for any entity which supports optimistic locking has a.getVersionNo() method.
- Some getters do *not* return primitive types, but instead return objects, e.g. there are no someEntity.getStartDate() or.getEndDate() methods, only a.getDateRange() method which returns a DateRange object which contains a start and end date, but is also capable of date-range processing such as validation and comparison.

You must code a mapping for each field that you need to return to the client. Code-completion in IDEs like Eclipse will help!

Putting it all together

Here's the complete code for this scenario solution:

```
public class MyFacade {
  // ...

  @Inject
  private SomeEntityDAO someEntityDAO;

  public MyFacade() {
    GuiceWrapper.getInjector().injectMembers(this);
  }

  public SomeEntityDetails viewSomeEntityDetails(
      final SomeEntityKey key) throws AppException,
      InformationalException {

    // create an instance of the return struct
    final SomeEntityDetails someEntityDetails =
        new SomeEntityDetails();

    // retrieve the instance of the entity
    final SomeEntity someEntity =
        someEntityDAO.get(key.someEntityID);

    // map the details from the entity instance
    someEntityDetails.details.someEntityID = someEntity.getID();
    someEntityDetails.details.name = someEntity.getName();
    someEntityDetails.details.versionNo =
 someEntity.getVersionNo();

    final DateRange dateRange = someEntity.getDateRange();
    someEntityDetails.details.startDate = dateRange.start();
    someEntityDetails.details.endDate = dateRange.end();
    // ...more mappings

    // return to the client
    return someEntityDetails;
  }
```

```
  // ...
}
```

Figure 6: Complete listing for a façade "view" method

For this first scenario only, here's a side-by-side look at the classic approach vs. the service-layer API approach:

Using a service-layer entity API

```java
public class MyFacade {
  // ...

  @Inject
  private SomeEntityDAO someEntityDAO;

  public MyFacade() {
    GuiceWrapper.getInjector().injectMembers(this);
  }

  public SomeEntityDetails viewSomeEntityDetails(final SomeEntityKey key)
      throws AppException, InformationalException {

    // create an instance of the return struct
    final SomeEntityDetails someEntityDetails = new SomeEntityDetails();

    // retrieve the instance of the entity
    final SomeEntity someEntity = someEntityDAO.get(key.someEntityID);

    // map the details from the entity instance
    someEntityDetails.details.someEntityID = someEntity.getID();
    someEntityDetails.details.name = someEntity.getName();
    someEntityDetails.details.versionNo = someEntity.getVersionNo();

    final DateRange dateRange = someEntity.getDateRange();
    someEntityDetails.details.startDate = dateRange.start();
    someEntityDetails.details.endDate = dateRange.end();
    // ...more mappings

    // return to the client
    return someEntityDetails;
  }
  // ...
}
```
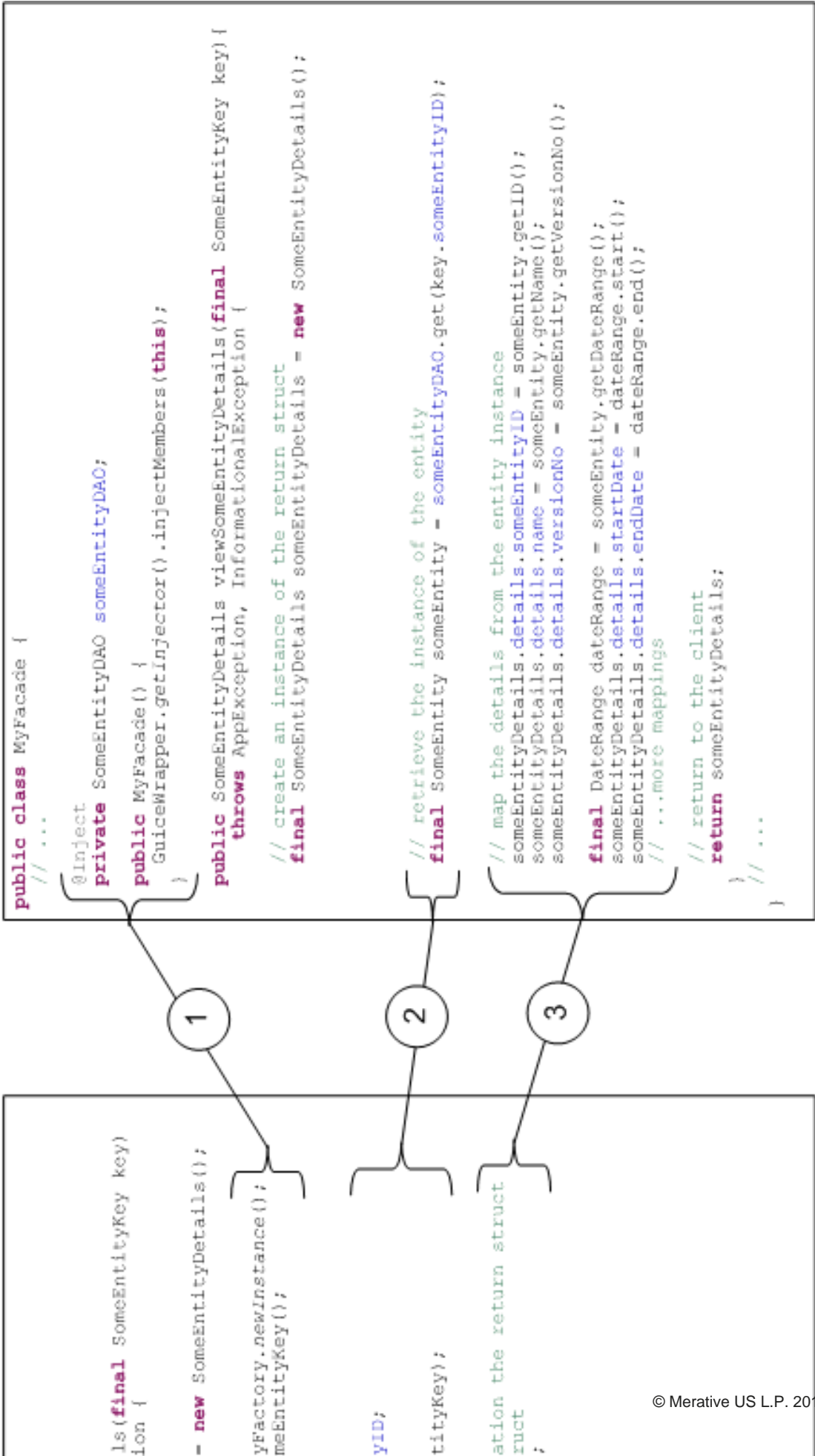
1

2

3

...ram entity

```
ls(final SomeEntityKey key)
ion {

= new SomeEntityDetails();

yFactory.newInstance();
meEntityKey();

yID;

tityKey);

ation the return struct
ruct
```

1. The object which knows how to retrieve instances of the entity. Using the persistence package, the object is called a Data Access Object ("DAO") and is a class member variable initialized by Guice using @Inject. The class constructor requests Guice to initialize this (and any other) class variable(s).
2. The retrieval of the entity from the database uses the DAO.
3. The data held on the entity is mapped to the client struct.

Note that when using service-layer APIs, in general:

- Code to retrieve instances of these APIs is more terse than when using classic Cúram; but
- Code to map entity data to client structs is more verbose (but this is after all one of the main purposes of façade logic).

You want to insert a new row onto a database table

The problem

You are writing a façade method which needs to insert a new row onto a database table.

Under classic Cúram, you would have created a call to the generated "entity" method as follows:

```
// ...
  public SomeEntityKey createSomeEntityDetails(
    final SomeEntityDetails details)
      throws AppException, InformationalException {

    // create an instance of the return struct
    final SomeEntityKey key = new SomeEntityKey();

    // objects for writing to the database
    final SomeEntity someEntityObj =
      SomeEntityFactory.newInstance();
    final SomeEntityDtls someEntityDtls;

    // map the details
    someEntityDtls = details.details;

    // do the insert
    someEntityObj.insert(someEntityDtls);

    // check for informational exceptions
    TransactionInfo.getInformationalManager().failOperation();

    // map the key assigned
    key.someEntityID = someEntityDtls.someEntityID;

    // return to the client
    return key;
  }

  // ...
```

Figure 8: Façade calling classic Cúram entity to create a database row

How do you insert a new row onto a database table using a service-layer API (developed using the Persistence Infrastructure)?

The solution

Coding the solution involves these steps:

- create a class variable to hold the DAO;
- use the DAO to create a new instance of the entity;
- access the entity instance to set field values from the client struct;
- instruct the entity instance to insert itself onto the database; and
- map the entity instance key back to the client (if required).

Create a class variable to hold the DAO

This step is identical to that in above.

In general more than one façade method will require to use the DAO object. Of course, you only need to create the DAO object class member once for the façade class!

Use the DAO to create a new instance of the entity

In your façade method, code a variable to hold an instance of the entity interface, and set its value by calling.newInstance() on the DAO, passing the key of the database row:

```
// create a new entity instance
    final SomeEntity someEntity = someEntityDAO.newInstance();
```

Figure 9: Calling a DAO to create a new instance of an entity

Here, the DAO instance has "dished up" a new instance of the entity interface, which does not (yet) exist on the database. The entity itself takes care of setting its data fields to sensible defaults.

someEntity now holds an object which "knows" how to:

- get at data (via "getter" methods);
- set data (via "setter" methods); and
- "do things" with that data (via other methods).

Access the entity instance to set field values from the client struct

Now code calls to the entity "setters" to map fields values from your input struct:

```
// map the details
    someEntity.setName(details.details.name);

    final DateRange dateRange = new DateRange(
    details.details.startDate,
        details.details.endDate);
    someEntity.setDateRange(dateRange);
    // ...more mappings
```

Figure 10: Calling setter methods on an entity instance

Points to note:

- Often, an entity may have a getter to allow retrieval of a data field, but have no corresponding setter. This is because the entity manages the setting of such fields, and does not allow the field to be set by calling code. Common examples include:
  - the entity's ID;

- the "logical delete" record status; and
- lifecycle state.

- Some setters do *not* take primitive types, but instead take objects, e.g. there are no someEntity.setStartDate() or.setEndDate() methods, only a.setDateRange() method which takes a DateRange object which contains a start and end date.
- When you call a setter on an entity instance, the entity instance will perform any single-field validation on the field being set.

You must code a mapping for each field that you need to populate from the client.

Instruct the entity instance to insert itself onto the database

Once the entity instance has been populated with data supplied by the client, you must code a call for the entity instance to store itself:

```
// do the insert
    someEntity.insert();
```

Figure 11: Calling the insert persistence method on an entity instance

The entity instance will:

- perform additional validation, including:

  - mandatory field validation (i.e. check that all mandatory fields have been set);
  - cross-field validation; and
  - cross-entity validation;
- assign a primary key value; and
- insert its data into the database.

Map the entity instance key back to the client (if required)

Some façade methods require to return back to the client the key of a new row stored.

If required, code a mapping to return the key:

```
// map the key assigned
    key.someEntityID = someEntity.getID();
```

Figure 12: Retrieving the ID of an entity instance

Putting it all together

Here's the complete code for this scenario solution:

```
// ...
  public SomeEntityKey createSomeEntityDetails(
    final SomeEntityDetails details)
      throws AppException, InformationalException {

    // create an instance of the return struct
    final SomeEntityKey key = new SomeEntityKey();

    // create a new entity instance
    final SomeEntity someEntity = someEntityDAO.newInstance();

    // map the details
    someEntity.setName(details.details.name);
```

```
      final DateRange dateRange =
          new DateRange(details.details.startDate,
          details.details.endDate);
      someEntity.setDateRange(dateRange);
      // ...more mappings

      // do the insert
      someEntity.insert();

      // map the key assigned
      key.someEntityID = someEntity.getID();

      // return to the client
      return key;
    }
    // ...
```

Figure 13: Complete listing for a façade "create" method

Note that there is no call to TransactionInfo.getInformationalManager().failOperation() - the entity insert operation takes care of all error handling.

You want to modify a row on a database table

The problem

You are writing a façade method which needs to modify the contents of an existing row on the database.

Under classic Cúram, you would have created a call to the generated "entity" method as follows:

```
// ...
  public void modifySomeEntityDetails(
    final SomeEntityDetails details)
      throws AppException, InformationalException {

    // objects for writing to the database
    final SomeEntity someEntityObj =
      SomeEntityFactory.newInstance();
    final SomeEntityDtls someEntityDtls;

    // map the details
    someEntityDtls = details.details;

    // create an instance of the key
    final SomeEntityKey someEntityKey = new SomeEntityKey();
    someEntityKey.someEntityID = someEntityDtls.someEntityID;

    // do the modify
    someEntityObj.modify(someEntityKey, someEntityDtls);

    // check for informational exceptions
    TransactionInfo.getInformationalManager().failOperation();

  }
```

```
// ...
```

Figure 14: Façade calling classic Cúram entity to modify a database row

How do you modify an existing row on a database table using a service-layer API (developed using the Persistence Infrastructure)?

The solution

The solution draws together elements of processing seen in the earlier scenarios:

- [You want to read some data from a database table on page 10](#); and
- [You want to insert a new row onto a database table on page 16](#).

Coding the solution involves these steps:

- create a class variable to hold the DAO;
- use the DAO to retrieve the instance of the entity;
- access the entity instance to set field values from the client struct; and
- instruct the entity instance to modify its data on the database.

Create a class variable to hold the DAO

This step is identical to that in [You want to read some data from a database table on page 10](#) above.

Use the DAO to retrieve the instance of the entity

This step is identical to that in [You want to read some data from a database table on page 10](#) above.

Access the entity instance to set field values from the client struct

This step is identical to that in [You want to insert a new row onto a database table on page 16](#) above.

It is likely that a façade class will contain both of the following methods:

- a method which insert a new row onto a database table; and
- a method which modifies an existing row on a database table.

For facades which contain both of these kinds of methods, it is likely that the steps to map client data to setters are very similar. Any identical processing should be factored into a common method:

```
// ...
  /**
   * Maps client details to the setters on the service-layer API
   *
   * @param someEntity
   *        the service-layer instance of the entity
   * @param someEntityDtls
   *        the client details to map
   *
   */
  private void setSomeEntityDetails(final SomeEntity someEntity,
      final SomeEntityDtls someEntityDtls) {
```

```
    // map the details
    someEntity.setName(someEntityDtls.name);

    final DateRange dateRange =
      new DateRange(someEntityDtls.startDate,
        someEntityDtls.endDate);
    someEntity.setDateRange(dateRange);
    // ...more mappings
  }
  // ...
```

Figure 15: Factoring out common calls to setter methods

Note that this method cannot be *modeled* as the entity interface argument is not present in the Cúram model; thus this method is private to the Java implementation.

Instruct the entity instance to modify its data on the database

Once the entity instance has been populated with data supplied by the client, you must code a call for the entity instance to store its changes:

```
// do the modify, passing the version number from the client
      someEntity.modify(details.details.versionNo);
```

Figure 16: Calling the modify persistence method on an entity

The entity instance will:

- perform additional validation, including:

  - mandatory field validation (i.e. check that all mandatory fields have been set);
  - cross-field validation; and
  - cross-entity validation;
- modify its data on the database.

> **Important:** For an entity which supports optimistic locking, you must pass the version number held by the client struct. Do not be tempted to use the version number on the entity instance which has been retrieved, as this would render the optimistic lock mechanism useless and allow one user's updates to be overwritten by another user's updates:

```
/** ********** VERY VERY BAD - DO NOT DO THIS! ********** */
      // do the modify, passing the version number from the
 entity
      // instance
      someEntity.modify(someEntity.getVersionNo());
      /** ********** VERY VERY BAD - DO NOT DO THIS! **********
 */
```

Figure 17: Incorrect - bypassing optimistic locking safeguards

Putting it all together

Here's the complete code for this scenario solution:

```
// ...
  public void modifySomeEntityDetails(
    final SomeEntityDetails details)
      throws AppException, InformationalException {
```

```
      // retrieve the instance of the entity
      final SomeEntity someEntity = someEntityDAO
          .get(details.details.someEntityID);

      // set the fields
      setSomeEntityDetails(someEntity, details.details);

      // do the modify, passing the version number from the client
      someEntity.modify(details.details.versionNo);

  }

  /**
   * Maps client details to the setters on the service-layer API
   *
   * @param someEntity
   *          the service-layer instance of the entity
   * @param someEntityDtls
   *          the client details to map
   *
   */
  private void setSomeEntityDetails(final SomeEntity someEntity,
      final SomeEntityDtls someEntityDtls) {

    // map the details
    someEntity.setName(someEntityDtls.name);

    final DateRange dateRange =
      new DateRange(someEntityDtls.startDate,
        someEntityDtls.endDate);
    someEntity.setDateRange(dateRange);
    // ...more mappings
  }

  // ...
```

Figure 18: Complete listing for a façade "modify" method

You want to remove (physically delete) a row from a database table

The problem

You are writing a façade method which needs to remove an existing row from the database.

Under classic Cúram, you would have created a call to the generated "entity" method as follows:

```
// ...
  public void removeSomeEntityDetails(final SomeEntityKey key)
      throws AppException, InformationalException {

    // objects for writing to the database
    final SomeEntity someEntityObj =
      SomeEntityFactory.newInstance();

    // create an instance of the key
    final SomeEntityKey someEntityKey = new SomeEntityKey();
```

```
    someEntityKey.someEntityID = key.someEntityID;

    // do the remove
    someEntityObj.remove(someEntityKey);

    // check for informational exceptions
    TransactionInfo.getInformationalManager().failOperation();

}

// ...
```

Figure 19: Façade calling classic Cúram entity to remove a database row

How do you remove an existing row from a database table using a service-layer API (developed using the Persistence Infrastructure)?

The solution

Coding the solution involves these steps:

- create a class variable to hold the DAO;
- use the DAO to retrieve the instance of the entity;
- instruct the entity instance to remove its data from the database.

Create a class variable to hold the DAO

This step is identical to that in You want to read some data from a database table on page 10 above.

Use the DAO to retrieve the instance of the entity

This step is identical to that in You want to read some data from a database table on page 10 above.

Instruct the entity instance to remove its data from the database

You must code a call for the entity instance to remove its data from the database:

```
// do the remove, passing the version number from the client
    someEntity.remove(key.versionNo);
```

Figure 20: Calling the remove persistence method on an entity

The entity instance will:

- perform cross-entity validation, allowing other entities to veto the removal; and
- remove its data from the database.

For an entity which supports optimistic locking, you must pass the version number held by the client struct. Note that this approach is stricter than the classic Cúram approach which does not require a version number.

> **Important:** Do not be tempted to use the version number on the entity instance which has been retrieved, as this would render the optimistic lock mechanism useless and allow one user's updates to be removed by another user acting on out-of-date data:

```
/** ********** VERY VERY BAD - DO NOT DO THIS! ********** */
    // do the remove, passing the version number from the entity
    // instance
    someEntity.remove(someEntity.getVersionNo());
    /** ********** VERY VERY BAD - DO NOT DO THIS! ********** */
```

Figure 21: Incorrect - bypassing optimistic locking safeguards

Putting it all together

Here's the complete code for this scenario solution:

```
// ...
  public void removeSomeEntityDetails(
     final SomeEntityKeyVersion key)
       throws AppException, InformationalException {

    // retrieve the instance of the entity
    final SomeEntity someEntity =
      someEntityDAO.get(key.someEntityID);

    // do the remove, passing the version number from the client
    someEntity.remove(key.versionNo);

  }

  // ...
```

Figure 22: Complete listing for a façade "remove" method

You want to cancel (logically delete) a row on a database table

The problem

You are writing a façade method which needs to cancel an existing row on the database (i.e. set its "recordStatus" to "Canceled").

Under classic Cúram, you would have created a call to a non-stereotyped "entity" method as follows:

```
// ...
  public void cancelSomeEntityDetails(
     final SomeEntityKeyVersion key)
       throws AppException, InformationalException {

    // objects for writing to the database
    final SomeEntity someEntityObj =
      SomeEntityFactory.newInstance();

    // create an instance of the key/version
    final SomeEntityKeyVersion someEntityKeyVersion =
      new SomeEntityKeyVersion();
    someEntityKeyVersion.someEntityID = key.someEntityID;
    someEntityKeyVersion.versionNo = key.versionNo;

    // do the cancel
    someEntityObj.cancel(someEntityKeyVersion);
```

```
    // check for informational exceptions
    TransactionInfo.getInformationalManager().failOperation();

}

// ...
```

Figure 23: Façade calling classic Cúram entity to cancel a database row

How do you cancel an existing row on a database table using a service-layer API (developed using the Persistence Infrastructure)?

The solution

Coding the solution involves these steps:

- create a class variable to hold the DAO;
- use the DAO to retrieve the instance of the entity;
- instruct the entity instance to cancel its data on the database.

Create a class variable to hold the DAO

This step is identical to that in <u>You want to read some data from a database table on page 10</u> above.

Use the DAO to retrieve the instance of the entity

This step is identical to that in <u>You want to read some data from a database table on page 10</u> above.

Instruct the entity instance to cancel its data on the database

You must code a call for the entity instance to cancel its data on the database:

```
// do the cancel, passing the version number from the client
    someEntity.cancel(key.versionNo);
```

Figure 24: Calling the cancel method on an entity

The entity instance will:

- perform cross-entity validation, allowing other entities to veto the cancellation; and
- cancel its data from the database.

For an entity which supports optimistic locking, you must pass the version number held by the client struct.

> **Important:** Do not be tempted to use the version number on the entity instance which has been retrieved, as this would render the optimistic lock mechanism useless:

```
/** ********** VERY VERY BAD - DO NOT DO THIS! ********** */
    // do the cancel, passing the version number from the entity
    // instance
    someEntity.cancel(someEntity.getVersionNo());
    /** ********** VERY VERY BAD - DO NOT DO THIS! ********** */
```

Figure 25: Incorrect - bypassing optimistic locking safeguards

Putting it all together

Here's the complete code for this scenario solution:

```
// ...
  public void cancelSomeEntityDetails(
    final SomeEntityKeyVersion key)
      throws AppException, InformationalException {

    // retrieve the instance of the entity
    final SomeEntity someEntity =
      someEntityDAO.get(key.someEntityID);

    // do the cancel, passing the version number from the client
    someEntity.cancel(key.versionNo);

  }

  // ...
```

Figure 26: Complete listing for a façade "cancel" method

You want to list all rows of a database table

The problem

You are writing a façade method which needs to:

- retrieve all rows from a database table; and
- format the data for return to the user interface, where it will be displayed to the user.

Under classic Cúram, you would have created a call to the generated "entity" method as follows:

```
// ...
  public SomeEntitySummaryDetailsList listSomeEntityDetails()
      throws AppException, InformationalException {

    // create an instance of the return struct
    final SomeEntitySummaryDetailsList list =
      new SomeEntitySummaryDetailsList();

    // objects for reading the database
    final SomeEntity someEntityObj =
      SomeEntityFactory.newInstance();
    final SomeEntityDtlsList someEntityDtlsList;

    // do the read
    someEntityDtlsList = someEntityObj.readAll();

    // map the details returned
    for (int i = 0; i < someEntityDtlsList.dtls.size(); i++) {
      final SomeEntitySummaryDetails someEntitySummaryDetails =
        new SomeEntitySummaryDetails();
      someEntitySummaryDetails.assign(
    someEntityDtlsList.dtls.item(i));

      list.details.addRef(someEntitySummaryDetails);
    }
```

```
    // return to the client
    return list;
}

// ...
```

Figure 27: Façade calling classic Cúram entity to list all database rows

How do you list all rows from a database table using a service-layer API (developed using the Persistence Infrastructure)?

The solution

Coding the solution involves these steps:

- create a class variable to hold the DAO;
- use the DAO to retrieve all the instances of the entity; and
- iterate the set of entity instances and access these instances to map field values to the client struct.

Create a class variable to hold the DAO

This step is identical to that in You want to read some data from a database table on page 10 above.

Use the DAO to retrieve all the instances of the entity

In your façade method, code a variable to hold a set of instances of the entity interface, and set its value by calling.readAll() on the DAO:

```
// retrieve all the instances of the entity
    final Set<SomeEntity> someEntities = someEntityDAO.readAll();
```

Figure 28: Calling a DAO method to read multiple entity instances

Note that (in this particular example):

- the DAO readAll method returns a Set, typed with the entity interface (SomeEntity); and
- this scenario assumes that the API designer created a readAll method on the DAO (it does not have one by default).

Iterate the set of entity instances and access these instances to map field values to the client struct

Now code a loop which iterates the set retrieved, and maps each instance to the client struct. Note that since a Set is used, the Java 5 syntax for "for" loops can be used:

```
// map the details returned
    for (final SomeEntity someEntity : someEntities) {
       final SomeEntitySummaryDetails someEntitySummaryDetails =
         new SomeEntitySummaryDetails();
       someEntitySummaryDetails.someEntityID = someEntity.getID();
       someEntitySummaryDetails.name = someEntity.getName();

       list.details.addRef(someEntitySummaryDetails);
    }
```

Figure 29: Iterating through multiple entity instances

Putting it all together

Here's the complete code for this scenario solution:

```
// ...
  public SomeEntitySummaryDetailsList listSomeEntityDetails()
      throws AppException, InformationalException {

    // create an instance of the return struct
    final SomeEntitySummaryDetailsList list =
      new SomeEntitySummaryDetailsList();

    // retrieve all the instances of the entity
    final Set<SomeEntity> someEntities = someEntityDAO.readAll();

    // map the details returned
    for (final SomeEntity someEntity : someEntities) {
      final SomeEntitySummaryDetails someEntitySummaryDetails =
        new SomeEntitySummaryDetails();
      someEntitySummaryDetails.someEntityID = someEntity.getID();
      someEntitySummaryDetails.name = someEntity.getName();

      list.details.addRef(someEntitySummaryDetails);
    }

    // return to the client
    return list;
  }
```

Figure 30: Complete listing for a façade "list all" method

Note that the assignment to the someEntities set was shown for clarity only - equivalent terser code is shown below:

```
// ...
  public SomeEntitySummaryDetailsList listSomeEntityDetails()
      throws AppException, InformationalException {

    // create an instance of the return struct
    final SomeEntitySummaryDetailsList list =
      new SomeEntitySummaryDetailsList();

    for (final SomeEntity someEntity : someEntityDAO.readAll()) {
      // map the details returned
      final SomeEntitySummaryDetails someEntitySummaryDetails =
        new SomeEntitySummaryDetails();
      someEntitySummaryDetails.someEntityID = someEntity.getID();
      someEntitySummaryDetails.name = someEntity.getName();

      list.details.addRef(someEntitySummaryDetails);
    }

    // return to the client
    return list;
  }

  // ...
```

Figure 31: Complete listing for a façade "list all" method (terser version)

You want to list all child rows of a database table belonging to some parent row (on another table)

The problem

You are writing a façade method which needs to:

- retrieve all rows from a database table for a given "parent ID"; and
- format the data for return to the user interface, where it will be displayed to the user.

Under classic Cúram, you would have created a call to the generated "entity" method as follows:

```
// ...
  public SomeChildSummaryDetailsList listSomeChildDetails(
      final SomeParentKey key)
      throws AppException, InformationalException {

    // create an instance of the return struct
    final SomeChildSummaryDetailsList list =
      new SomeChildSummaryDetailsList();

    // objects for reading the database
    final SomeChild someChildObj =
 SomeChildFactory.newInstance();
    final SomeChildDtlsList someChildDtlsList;

    // set up the key
    final SomeParentKey someParentKey = new SomeParentKey();
    someParentKey.someParentID = key.someParentID;

    // do the read
    someChildDtlsList =
      someChildObj.searchBySomeParent(someParentKey);

    // map the details returned
    for (int i = 0; i < someChildDtlsList.dtls.size(); i++) {
      final SomeChildSummaryDetails someChildSummaryDetails =
        new SomeChildSummaryDetails();
      someChildSummaryDetails.assign(
    someChildDtlsList.dtls.item(i));

      list.details.addRef(someChildSummaryDetails);
    }

    // return to the client
    return list;
  }
  // ...
```

Figure 32: Façade calling classic Cúram entity to list all child database rows for a given parent

How do you list child rows for a given parent using a service-layer API (developed using the Persistence Infrastructure)?

The solution

Coding the solution involves these steps:

- create a class variable to hold the DAO for the parent entity;
- use the DAO to retrieve the instance of the parent entity;
- call a getter on the parent entity instance to retrieve its set of child entity instances; and
- iterate the set of child entity instances and access these instances to map field values to the client struct.

Create a class variable to hold the DAO

```
@Inject
  private SomeParentDAO someParentDAO;
```

Figure 33: Declaring a variable to hold a DAO for an entity

Use the DAO to retrieve the instance of the parent entity

In your façade method, code a variable to hold an instance of the entity interface, and set its value by calling.get() on the DAO, passing the key of the database row:

```
// retrieve the instance of the parent entity
    final SomeParent someParent =
      someParentDAO.get(key.someParentID);
```

Figure 34: Retrieving an instance of a parent entity

Call a getter on the parent entity instance to retrieve its set of child entity instances

Now code a call to the appropriate getter on the parent entity instance to retrieve its child entity instances:

```
// retrieve all the child instances of the entity for this parent
final Set<SomeChild> someChildren = someParent.getSomeChildren();
```

Figure 35: Calling a getter method on a parent entity instance to retrieve its child entity instances

Iterate the set of child entity instances and access these instances to map field values to the client struct

Now code a loop which iterates the set retrieved, and maps each instance to the client struct:

```
// map the details returned
    for (final SomeChild someChild : someChildren) {
      final SomeChildSummaryDetails someChildSummaryDetails =
        new SomeChildSummaryDetails();
      someChildSummaryDetails.someChildID = someChild.getID();
      someChildSummaryDetails.name = someChild.getName();

      list.details.addRef(someChildSummaryDetails);
    }
```

Figure 36: Iterating through child entity instances

Putting it all together

Here's the complete code for this scenario solution:

```
// ...
  @Inject
  private SomeParentDAO someParentDAO;
```

```
public SomeChildSummaryDetailsList listSomeChildDetails(
    final SomeParentKey key)
    throws AppException, InformationalException {

  // create an instance of the return struct
  final SomeChildSummaryDetailsList list =
    new SomeChildSummaryDetailsList();

  // retrieve the instance of the parent entity
  final SomeParent someParent =
    someParentDAO.get(key.someParentID);

  // retrieve all the child instances of the entity for this
  // parent
  final Set<SomeChild> someChildren =
    someParent.getSomeChildren();

  // map the details returned
  for (final SomeChild someChild : someChildren) {
    final SomeChildSummaryDetails someChildSummaryDetails =
      new SomeChildSummaryDetails();
    someChildSummaryDetails.someChildID = someChild.getID();
    someChildSummaryDetails.name = someChild.getName();

    list.details.addRef(someChildSummaryDetails);
  }

  // return to the client
  return list;
}

// ...
```

Figure 37: Complete listing for a façade "list children" method

Again, here is a briefer version which has no intermediate variable to hold the Set of child entity instances:

```
// ...
  public SomeChildSummaryDetailsList listSomeChildDetails(
      final SomeParentKey key)
      throws AppException, InformationalException {

    // create an instance of the return struct
    final SomeChildSummaryDetailsList list =
      new SomeChildSummaryDetailsList();

    // retrieve the instance of the parent entity
    final SomeParent someParent =
      someParentDAO.get(key.someParentID);

    for (final SomeChild someChild :
 someParent.getSomeChildren()) {
      // map the details returned
      final SomeChildSummaryDetails someChildSummaryDetails =
        new SomeChildSummaryDetails();
      someChildSummaryDetails.someChildID = someChild.getID();
      someChildSummaryDetails.name = someChild.getName();
```

```
      list.details.addRef(someChildSummaryDetails);
    }

    // return to the client
    return list;
  }
  // ...
```

Figure 38: Complete listing for a façade "list children" method (terser version)

Summary

Here is the entire listing for the façade class:

```java
package curam.cookbook.facade.persistence;

import curam.util.persistence.GuiceWrapper;
import curam.util.type.DateRange;

import java.util.Set;

import com.google.inject.Inject;

import curam.cookbook.SomeChild;
import curam.cookbook.SomeEntity;
import curam.cookbook.SomeEntityDAO;
import curam.cookbook.SomeParent;
import curam.cookbook.SomeParentDAO;
import curam.cookbook.facade.struct.SomeChildSummaryDetails;
import curam.cookbook.facade.struct.SomeChildSummaryDetailsList;
import curam.cookbook.facade.struct.SomeEntityDetails;
import curam.cookbook.facade.struct.SomeEntityKeyVersion;
import curam.cookbook.facade.struct.SomeEntitySummaryDetails;
import curam.cookbook.facade.struct.SomeEntitySummaryDetailsList;
import curam.cookbook.sl.entity.struct.SomeEntityDtls;
import curam.cookbook.sl.entity.struct.SomeEntityKey;
import curam.cookbook.sl.entity.struct.SomeParentKey;
import curam.util.exception.AppException;
import curam.util.exception.InformationalException;

public class MyFacade {

  @Inject
  private SomeEntityDAO someEntityDAO;

  public MyFacade() {
    GuiceWrapper.getInjector().injectMembers(this);
  }

  public SomeEntityDetails viewSomeEntityDetails(
    final SomeEntityKey key)
      throws AppException, InformationalException {

    // create an instance of the return struct
    final SomeEntityDetails someEntityDetails =
      new SomeEntityDetails();

    // retrieve the instance of the entity
```

```java
    final SomeEntity someEntity =
      someEntityDAO.get(key.someEntityID);

    // map the details from the entity instance
    someEntityDetails.details.someEntityID = someEntity.getID();
    someEntityDetails.details.name = someEntity.getName();
    someEntityDetails.details.versionNo =
someEntity.getVersionNo();

    final DateRange dateRange = someEntity.getDateRange();
    someEntityDetails.details.startDate = dateRange.start();
    someEntityDetails.details.endDate = dateRange.end();
    // ...more mappings

    // return to the client
    return someEntityDetails;
  }

  public SomeEntityKey createSomeEntityDetails(
    final SomeEntityDetails details)
      throws AppException, InformationalException {

    // create an instance of the return struct
    final SomeEntityKey key = new SomeEntityKey();

    // create a new entity instance
    final SomeEntity someEntity = someEntityDAO.newInstance();

    // map the details
    someEntity.setName(details.details.name);

    final DateRange dateRange =
      new DateRange(details.details.startDate,
        details.details.endDate);
    someEntity.setDateRange(dateRange);
    // ...more mappings

    // do the insert
    someEntity.insert();

    // map the key assigned
    key.someEntityID = someEntity.getID();

    // return to the client
    return key;
  }

  public void modifySomeEntityDetails(
    final SomeEntityDetails details)
      throws AppException, InformationalException {

    // retrieve the instance of the entity
    final SomeEntity someEntity = someEntityDAO
        .get(details.details.someEntityID);

    // set the fields
    setSomeEntityDetails(someEntity, details.details);
```

```
      // do the modify, passing the version number from the client
      someEntity.modify(details.details.versionNo);

    }

    /**
     * Maps client details to the setters on the service-layer API
     *
     * @param someEntity
     *            the service-layer instance of the entity
     * @param someEntityDtls
     *            the client details to map
     *
     */
    private void setSomeEntityDetails(final SomeEntity someEntity,
        final SomeEntityDtls someEntityDtls) {

      // map the details
      someEntity.setName(someEntityDtls.name);

      final DateRange dateRange = new DateRange(
      someEntityDtls.startDate,
          someEntityDtls.endDate);
      someEntity.setDateRange(dateRange);
      // ...more mappings
    }

    public void removeSomeEntityDetails(
      final SomeEntityKeyVersion key)
        throws AppException, InformationalException {

      // retrieve the instance of the entity
      final SomeEntity someEntity =
        someEntityDAO.get(key.someEntityID);

      // do the remove, passing the version number from the client
      someEntity.remove(key.versionNo);

    }

     public void cancelSomeEntityDetails(
      final SomeEntityKeyVersion key)
        throws AppException, InformationalException {

      // retrieve the instance of the entity
      final SomeEntity someEntity =
        someEntityDAO.get(key.someEntityID);

      // do the cancel, passing the version number from the client
      someEntity.cancel(key.versionNo);

    }

    public SomeEntitySummaryDetailsList listSomeEntityDetails()
        throws AppException, InformationalException {

      // create an instance of the return struct
      final SomeEntitySummaryDetailsList list =
```

```
      new SomeEntitySummaryDetailsList();

    // retrieve all the instances of the entity
    final Set<SomeEntity> someEntities = someEntityDAO.readAll();

    // map the details returned
    for (final SomeEntity someEntity : someEntities) {
      final SomeEntitySummaryDetails someEntitySummaryDetails =
        new SomeEntitySummaryDetails();
      someEntitySummaryDetails.someEntityID = someEntity.getID();
      someEntitySummaryDetails.name = someEntity.getName();

      list.details.addRef(someEntitySummaryDetails);
    }

    // return to the client
    return list;
  }

  @Inject
  private SomeParentDAO someParentDAO;

  public SomeChildSummaryDetailsList listSomeChildDetails(
      final SomeParentKey key)
      throws AppException, InformationalException {

    // create an instance of the return struct
    final SomeChildSummaryDetailsList list =
      new SomeChildSummaryDetailsList();

    // retrieve the instance of the parent entity
    final SomeParent someParent =
      someParentDAO.get(key.someParentID);

    // retrieve all the child instances of the entity for this
    // parent
    final Set<SomeChild> someChildren =
      someParent.getSomeChildren();

    // map the details returned
    for (final SomeChild someChild : someChildren) {
      final SomeChildSummaryDetails someChildSummaryDetails =
        new SomeChildSummaryDetails();
      someChildSummaryDetails.someChildID = someChild.getID();
      someChildSummaryDetails.name = someChild.getName();

      list.details.addRef(someChildSummaryDetails);
    }

    // return to the client
    return list;
  }
```

Figure 39: Façade class listing

1.3 Coding service-layer APIs

The scenarios in this section describe how to write service-layer APIs, which may be called from other code such as:

- implementations of other service-layer APIs;
- "classic" Cúram service layers in other components; and/or
- façade layer code.

You want to start writing the API for a new database table

The problem

You identify the need for a new database table and you want to control access to this database table through a service-layer API.

How do you start?

The solution

The interface for interacting with your database table breaks down as follows:

- DAO interface - responsible for describing how to search your database table for rows matching certain criteria ("readmultis"); and
- Entity interface - responsible for describing what calling code can "do" with your entity once row(s) have been retrieved.

Coding the solution involves these steps:

- create an entity interface java file; and
- create an entity DAO interface java file.

Create an entity interface java file

Create a new java file named after your entity, and declare an interface extending StandardEntity:

```
package curam.mypackage;

import curam.util.persistence.StandardEntity;

/**
 * Description of my wonderful new entity.
 */
public interface MyNewEntity extends StandardEntity {
}
```

Figure 40: Creating an entity interface file

The StandardEntity super-interface provides a standard API for all entities, and must be extended by all entity APIs.

As far as callers of your code are concerned, this interface "is" your entity, which is why (by convention) you name the entity interface after your entity. Ensure that the interface is well-commented.

So far your new entity API doesn't do very much, but that'll change during later the scenarios.

Note that not all interfaces need to be public - if the interface does not need to be visible outside of its package then remove the "public" declaration and make the interface "package-private". Typically this can only be done with entities which are not exposed to calling code, e.g. link tables which do not (directly) appear on UI screens. Only make an interface public if it *needs* to be (which is usually the case).

Only include methods in your interface which must be visible to other classes - implementation-only methods will exist only in your implementation class (see 1.4 Coding service-layer implementations on page 50).

Create an entity DAO interface java file

Create another interface in the same java package, named after your entity but suffixed with "DAO", extending StandardDAO (typed with your entity interface):

```
package curam.mypackage;

import curam.util.persistence.StandardDAO;

/**
 * Data access for {@linkplain MyNewEntity}.
 */
public interface MyNewEntityDAO extends StandardDAO<MyNewEntity>
 {
}
```

Figure 41: Creating an entity DAO interface file

The StandardDAO super-interface must be extended by all entity DAO APIs. It provides two DAO API methods "for free":

- newInstance() - creates a new instance of MyNewEntity suitable for inserting onto the database; and
- get(Long id) - retrieves the instance of MyNewEntity with the primary key value specified by id

Your DAO declares that it is responsible for managing MyNewEntity instances by virtue of the type argument to StandardDAO.

In a later scenario you will add additional methods to the DAO interface.

You want to add getters and setters to your entity interface

The problem

Your database table contains a number of data columns. You need to allow callers of your code to:

- get the values held in some of these columns; and
- set the values held in some of these columns.

You also need to support navigation to related entity instances.

In classic Cúram, callers of your code had access full access to each field on the entity Dtls struct, and so there was no need (nor any way) to decide whether a particular field was:

- hidden;
- read-only; or
- read/write.

Regarding navigation, in classic Cúram callers of your code had to perform their own navigation by executing queries on related entities, and seeding those queries with foreign key fields from an entity Dtls struct.

How do you add getters and setters to your entity interface?

The solution

You must code getters and setters on your entity interface, and make an informed decision as to the level of visibility of each field.

For each column on your database table, you must decide:

- whether callers of your entity must be able to read the data - if so you must code a getter method; and
- whether callers of you entity must be able to write the data - if so you must code a setter method; and
- whether access to the column is on a "per-column" basis or whether there is some logical grouping of columns which should be combined into a single object (see the date range example below).

Example

You'll step through an example database table and code getters/setters in your API.

Let's say that the database table MyNewEntity has these columns:

- myNewEntityID - primary key;
- name - String;
- startDate - date;
- endDate - date;
- typeCode - codetable code, specifying the "type" of the entity; and
- myParentEntityID - foreign key to a row on a different database table.

Let's go through the attributes on MyNewEntity and flesh out the entity API.

myNewEntityID

In general, getters and setters for your primary key column are straightforward - you don't write any.

You rarely need to code anything for the primary key of an entity, because each entity already has a getID method (inherited from StandardEntity).

> **Important:** Do not be tempted to write your own getter for the ID:

```
/** ********** VERY VERY BAD - DO NOT DO THIS! ********** */
  /**
   * @return the primary key of MyNewEntity.
   */
  public Long getMyNewEntityID();
```

```
   /** ********** VERY VERY BAD - DO NOT DO THIS! ********** */
```

Figure 42: Incorrect - redundant getter method for entity ID

Similarly, each entity implementation typically takes care of assigning its own primary key, and so callers of the entity API do *not* require a facility to set the primary key themselves.

> **Important:** Do not be tempted to write your own setter for the ID:

```
/** ********** VERY VERY BAD - DO NOT DO THIS! ********** */
  /**
   * @param value
   *          the primary key of MyNewEntity.
   */
  public void setMyNewEntityID(final Long value);

   /** ********** VERY VERY BAD - DO NOT DO THIS! ********** */
```

Figure 43: Incorrect - setter method for entity ID

name

After analysis of requirements, you determine that callers of your API require to both get and set the name column of a database row.

Code a field getter as follows:

```
/**
   * Gets the name.
   *
   * @return the name
   */
  public String getName();
```

Figure 44: Interface declaration for a simple get method

Note that:

- by convention, the method is named get<Fieldname> with the first letter of the field name upper-cased (one exception is that getters that return a boolean value often read better as is<Condition>); and
- the name column holds a String, so the getter must return a String.

Code a field setter as follows:

```
/**
   * Sets the name.
   *
   * @param value
   *          the name
   */
  public void setName(final String value);
```

Figure 45: Interface declaration for a simple set method

Note that:

- by convention, the method is named set<Fieldname>, with the first letter of the field name upper-cased;

- by convention, the variable name of the value passed in is "value";
- the setter returns void; and
- the name column holds a String, so the setter must take a String value.

startDate and endDate

After analysis of requirements, you determine that:

- callers of your API require to get the start and end dates, to compare the range of dates covered with dates supplied by other processing;
- the start date is always set to the current business date when a new row is created; and
- the end date is only set (to a specified date) when the entity enters a state of "closed".

Accordingly, you decide that:

- the start date and end date should be returned to callers as a DateRange "helper" object; and
- callers should *not* be free to set the start and end dates - manipulation of these end dates should be taken care of by specialized methods on the entity (see e.g.).

You require your entity to return a DateRange helper object - rather than coding a getDateRange method, instead it's better to change your API to extend DateRanged:

```
/**
 * Description of my wonderful new entity.
 */
public interface MyNewEntity extends StandardEntity, DateRanged {
```

Figure 46: Extending the DateRanged interface

The DateRanged interface provides your entity with a getDateRange method and also allows access to helper functions which provide commonly-used processing on entities which contain a date range.

typeCode

After analysis, you determine that your entity stores a codetable code describing the "type" of the entity instance.

Create a codetable specifying the permitted values:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<codetables package="curam.mypackage.codetable">
  <codetable
    java_identifier="MYNEWENTITYTYPE"
    name="MYNEWENTITYTYPE"
  >
    <code
      default="false"
      java_identifier="SOMETYPE"
      status="ENABLED"
      value="TYPE1"
    >
      <locale
        language="en"
        sort_order="0"
      >
        <description>Some type</description>
```

```
          <annotation/>
        </locale>
     </code>
     <code
        default="false"
        java_identifier="SOMEOTHERTYPE"
        status="ENABLED"
        value="TYPE2"
     >
        <locale
           language="en"
           sort_order="0"
        >
           <description>Some other type</description>
           <annotation/>
        </locale>
     </code>
   </codetable>
</codetables>
```

Figure 47: Codetable for the type of an entity

The Persistence Infrastructure includes a code generator to generate a class per codetable. These classes provide a type-safe mechanism for passing around an entry from the codetable, and each class is named after its codetable suffixed with the word Entry:

```
package curam.mypackage.codetable.impl;

/**
 * Represents an entry from the
 * {@linkplain curam.mypackage.codetable.MYNEWENTITYTYPE} code
 * table.
 */
public class MYNEWENTITYTYPEEntry extends
    curam.util.type.CodeTableEntry {

  // ...

  /**
   * Private constructor.
   */
  private MYNEWENTITYTYPEEntry(final String code) {
    super(TABLENAME, code);
  }

  /**
   * Gets the
   * {@linkplain
 curam.mypackage.codetable.impl.MYNEWENTITYTYPEEntry}
   * for the specified code value.
   *
   * @param code
   *          the String representation of the code value
 required.
   *
   * @return a
   *          {@linkplain
 curam.mypackage.codetable.impl.MYNEWENTITYTYPEEntry}
   *          representation of the specified code value.
```

```
       *
       * @throws curam.util.exception.AppRuntimeException
       *            if the specified code value is not present in the
       *            {@linkplain
 curam.mypackage.codetable.MYNEWENTITYTYPE}
       *            code table.
       */
  public static
 curam.mypackage.codetable.impl.MYNEWENTITYTYPEEntry get(
      final String code) {
     // ...
  }

  /**
   * The name of the
   * {@linkplain curam.mypackage.codetable.MYNEWENTITYTYPE} table
 _
   * {@value}.
   */
  public static String TABLENAME =
      curam.mypackage.codetable.MYNEWENTITYTYPE.TABLENAME;

  /**
   * Not specified (i.e. blank).
   */
  public static final
 curam.mypackage.codetable.impl.MYNEWENTITYTYPEEntry
      NOT_SPECIFIED = get(null);

  /**
   * TYPE1 en = Some type
   */
  public static final
 curam.mypackage.codetable.impl.MYNEWENTITYTYPEEntry
      SOMETYPE =
      get(curam.mypackage.codetable.MYNEWENTITYTYPE.SOMETYPE);

  /**
   * TYPE2 en = Some other type
   */
  public static final
 curam.mypackage.codetable.impl.MYNEWENTITYTYPEEntry
      SOMEOTHERTYPE =

 get(curam.mypackage.codetable.MYNEWENTITYTYPE.SOMEOTHERTYPE);

}
```

Figure 48: Excerpts from a generated "Entry" class for a codetable

Use of this generated class is preferable to using a String to pass around the value, as (in particular) a String can be constructed with any text whereas the generated class only permits values corresponding to the underlying code table.

Code:

• a getter to return an instance of this generated class; and
• a setter which takes an instance of this generated class:

```
/**
   * Gets the type of this entity instance.
   *
   * @return the type of this entity instance
   */
  public MYNEWENTITYTYPEEntry getType();

  /**
   * Sets the type of this entity instance.
   *
   * @param value
   *          the type of this entity instance
   */
  public void setType(final MYNEWENTITYTYPEEntry value);
```

Figure 49: Getter and setter methods for a codetable-based value

> **Note:** Getter and setter methods do *not* have to be named exactly after their database columns (in this example, the data column typeCode is accessed via methods named getType and setType, *not* getTypeCode and setTypeCode.
>
> In particular, some database column names are abbreviated to comply with database name length constraints, and for these the getter and setter names should not slavishly repeat the abbreviation, e.g. use getSomeVeryVeryLongDatabaseColumnName rather than getSmVyVyLgDbColNm.

myParentEntityID

For foreign keys to related entity instances, in general you should not create getters and setters for the entity ID, but instead code getters and setters which deal with the API of the related entity:

```
/**
   * Gets the parent instance of MyParentEntity.
   *
   * @return the parent instance of MyParentEntity, or null if
 not
   *         yet set
   */
  public MyParentEntity getMyParentEntity();

  /**
   * Sets the parent instance of MyParentEntity.
   *
   * @param value
   *          the parent instance of MyParentEntity
   */
  public void setMyParentEntity(final MyParentEntity value);
```

Figure 50: Interface declaration for getting/setting a related entity instance

Note that this code assumes that the MyParentEntity API has already been coded. If not, you must create a skeletal API:

```
public interface MyParentEntity extends StandardEntity {

}
```

Figure 51: Creating a skeletal API for a related entity

> **Important:** Do not be tempted to expose the related entity ID directly:

```
/** ********** VERY VERY BAD - DO NOT DO THIS! ********** */
  /**
   * @return the foreign key to the parent MyParentEntity
 instance
   */
 public Long getMyParentEntityID();

  /**
   * @param value
   *         the foreign key to the parent MyParentEntity
 instance
   */
 public void setMyParentEntityID(final Long value);

  /** ********** VERY VERY BAD - DO NOT DO THIS! ********** */
```

Figure 52: Incorrect - getting/setting a related ID instead of the related entity

Child instances

Each instance of your entity has a set of associated child entity instances (from a different database table).

If callers of your API require to navigate to these child instances, code a getter which returns a Set, typed with the API of the child entity:

```
/**
   * Gets the MyChildEntity children of this entity instance.
   *
   * @return the MyChildEntity children of this entity instance
   */
  public Set<MyChildEntity> getMyChildren();
```

Figure 53: Interface declaration for getting a set of related entities

Note that this code assumes that the MyChildEntity API has already been coded. If not, you must create a skeletal API:

```
public interface MyChildEntity extends StandardEntity {

}
```

Figure 54: Creating a skeletal API for another related entity

Putting it all together

Here's the complete code for this scenario solution, showing the getters, setters and changes to the interface inheritance hierarchy:

```
package curam.mypackage;

import java.util.Set;

import com.google.inject.ImplementedBy;

import curam.util.persistence.StandardEntity;
import curam.util.type.DateRanged;
```

```java
/**
 * Description of my wonderful new entity.
 */
@ImplementedBy(MyNewEntityImpl.class)
public interface MyNewEntity extends StandardEntity, DateRanged {

  /**
   * Gets the name.
   *
   * @return the name
   */
  public String getName();

  /**
   * Sets the name.
   *
   * @param value
   *            the name
   */
  public void setName(final String value);

  /**
   * Gets the parent instance of MyParentEntity.
   *
   * @return the parent instance of MyParentEntity, or null if
 not
   *            yet set
   */
  public MyParentEntity getMyParentEntity();

  /**
   * Sets the parent instance of MyParentEntity.
   *
   * @param value
   *            the parent instance of MyParentEntity
   */
  public void setMyParentEntity(final MyParentEntity value);

  /**
   * Gets the MyChildEntity children of this entity instance.
   *
   * @return the MyChildEntity children of this entity instance
   */
  public Set<MyChildEntity> getMyChildren();

  /**
   * Gets the type of this entity instance.
   *
   * @return the type of this entity instance
   */
  public MYNEWENTITYTYPEEntry getType();

  /**
   * Sets the type of this entity instance.
   *
   * @param value
   *            the type of this entity instance
```

```
        */
    public void setType(final MYNEWENTITYTYPEEntry value);

}
```

Figure 55: Complete listing for an entity API with getter and setter methods

You want to add persistence methods to your entity interface

The problem

Callers of your entity API need to be able to ask instances of your entity to store data on the database.

In classic Cúram, callers of your code made calls to modeled methods which were generated onto entity "process" classes.

How do you add persistence to your entity interface?

The solution

You must first analyze your requirements and decide which types of database write must be *publicly* supported by your API:

- insert - typically *every* entity API contains an insert() operation, to create a new row on the database;
- modify - typically required if your entity API contains setter methods. You must decide whether the modify requires optimistic lock support;
- cancel - typically required if your entity must allow callers to request that the entity instance be "logically deleted"; and
- remove - (rare) typically required if your entity must allow callers to request that the entity instance be "physically deleted". You must decide whether the remove requires optimistic lock support;

Note that it is quite in order *not* to publish any persistence methods on your entity interface, and instead create your own specialized methods instead.

In practice, entities often combine a mixture of exposing some persistence methods (for what are known as "CRUD" operations) and other specialized methods for business operations such as controlling the change of an entity's state.

Insert

If your entity API contains setter methods, then typically calling code will require an insert method to store new instances of your entity on the database:

```
@Inject
  private MyInsertableEntityDAO myInsertableEntityDAO;

  public void someCallToAnInsert() throws InformationalException
  {
    final MyInsertableEntity myInsertableEntity =
      myInsertableEntityDAO.newInstance();

    // set some field values on the new instance
    myInsertableEntity.setSomeField("some value");
```

```
    myInsertableEntity.setSomeOtherField("some other value");

    // ask the new entity instance to store itself on the
 database
    myInsertableEntity.insert();

  }
```

Figure 56: Sample code calling an entity insert

If your entity API must publish an insert method, change the entity API declaration to extend the Insertable interface:

```
/**
 * This entity supports callers asking it to insert itself.
 */
public interface MyInsertableEntity extends StandardEntity,
    Insertable {

}
```

Figure 57: Extending the Insertable interface

Note that the.insert() method (inherited from Insertable) throws InformationalException, in the case that validation errors are detected.

Modify

If your entity API contains setter methods, then typically calling code will require a.modify method to store changes on the database any changes to field values.

If modify support is required, you must decide whether your API should support:

- an optimistic-lock modify - (common) the modify only succeeds if the version number held by the caller matches that on the database - this mechanism prevents users from over-writing each others' concurrent modifications;
- a non-optimistic-lock modify - (less common) no version number checking is performed; or
- both (rare).

Change the entity API declaration to extend (as appropriate):

- OptimisticLockModifiable; and/or
- Modifiable

e.g.:

```
/**
 * This entity supports callers asking it to modify itself.
 */
public interface MyModifiableEntity extends StandardEntity,
    OptimisticLockModifiable {

}
```

Figure 58: Extending the OptimisticLockModifiable interface

Note that database tables which store historical data (e.g. a history of state changes or other events) typically should not support modify.

Cancel

If your entity supports the concept of logical deletion, then typically calling code will require a.cancel method to logically delete an instance of your entity.

If cancel support is required, change the entity API declaration to extend LogicallyDeleteable:

```
/**
 * This entity supports callers asking it to cancel itself.
 */
public interface MyLogicallyDeleteableEntity extends
    StandardEntity, LogicallyDeleteable {

}
```
Figure 59: Extending the LogicallyDeleteable interface

Note that support for logical deletes requires support for optimistic locking.

Remove

If your entity supports the concept of physical deletion, then typically calling code will require a remove method to physically delete an instance of your entity.

Business tables in Cúram rarely support physical deletion (favoring logical deletion instead). Technical tables (such as link tables) may support physical removal.

If remove support is required, you must decide whether your API should support:

* an optimistic-lock remove - the remove only succeeds if the version number held by the caller matches that on the database - this mechanism prevents one user deleting data containing updates that another user has concurrently made;
* a non-optimistic-lock remove - no version number checking is performed; or
* both.

Change the entity API declaration to extend (as appropriate):

* OptimisticLockRemovable; and/or
* Removable

e.g.:

```
/**
 * This entity supports callers asking it to remove itself.
 */
public interface MyPhysicallyDeleteableEntity extends
    StandardEntity, OptimisticLockRemovable {

}
```
Figure 60: Extending the OptimisticLockRemovable interface

Putting it all together

Typically your entity API will support a number of persistence operations, as evidence by its inheritance hierarchy:

```
/**
    * Description of my wonderful new entity.
    */
```

```
    public interface MyNewEntity extends StandardEntity,
 DateRanged,
        Lifecycle<MyNewEntity.State>, Insertable,
        OptimisticLockModifiable, LogicallyDeleteable {
```

Figure 61: Entity API extending multiple interfaces for persistence

You want to specify searches on your entity

The problem

Instances of your entity need to be retrieved using data other than the primary key of your entity, which may include:

- searches ("readmultis") of your entity, which may return zero or more matches; and/or
- singleton reads ("nsreads") of your entity, which may return zero matches or exactly one match.

In classic Cúram, you would model readmulti and nsread operations on your entity. Callers of your nsread would be expected to handle a RecordNotFoundException (or use the NotFoundIndicator mechanism).

How do you add non-key retrievals to your entity?

The solution

You must code retrievals of your entity on your entity DAO API, not on the entity itself.

A singleton read method must return your entity API, and should specify that null will be returned if no matching entity instance is found:

```
/**
   * Reads the instance with the specified name.
   *
   * @param name
   *          the name to find
   * @return the instance with the specified name, or null if not
   *          found.
   */
  public MyNewEntity readByName(final String name);
```

Figure 62: DAO interface declaration for a singleton read

A search method (which can return zero, one or many instances) must return a collection of your entity API (typically a Set):

```
/**
   * Searches all the instances which have the specified type.
   *
   * @param type
   *          the type to search for
   * @return all the instances which have the specified type, or
 an
   *          empty set if none found.
   */
  public Set<MyNewEntity> searchByType(
      final MYNEWENTITYTYPEEntry type);
```

Figure 63: DAO interface declaration for a search

Your method names must follow the naming standards for modeled entity operations.

Use entity APIs in preference to passing primary keys, e.g. do this:

```
/**
   * Searches all the instances belonging to the specified
 parent.
   *
   * @param myParentEntity
   *          the parent to search for
   * @return all the instances belonging to the specified parent,
 or
   *          an empty set if none found.
   */
  public Set<MyNewEntity> searchByParent(
      final MyParentEntity myParentEntity);
```

Figure 64: DAO interface taking an entity instance as a parameter

not this:

```
/** ********** VERY VERY BAD - DO NOT DO THIS! ********** */
  /**
   * Searches all the instances which have the specified parent
 ID.
   *
   * @param myParentEntityID
   *          the parent ID to search for
   * @return all the instances which have the specified parent
 ID,
   *          or an empty set if none found.
   */
  public Set<MyNewEntity> searchByParentID(
      final Long myParentEntityID);
  /** ********** VERY VERY BAD - DO NOT DO THIS! ********** */
```

Figure 65: Incorrect - DAO interface taking an entity ID value as a parameter

Summary

At this point you have developed the API for your entity and its DAO.

Because entities interact with each other through their APIs, it is possible to develop the service layer APIs for an entire component before commencing implementation. Such an approach allows you to publish the APIs to any interested parties and/or generate navigable JavaDoc for your APIs.

Alternatively, you may wish to create a limited number of entity APIs and proceed to implement these APIs.

1.4 Coding service-layer implementations

The scenarios in this section describe how to implement your service-layer APIs.

You want to start implementing your entity API

The problem

You have created interfaces for your entity and its DAO. You now need to create implementations of these interfaces.

Where do you start?

The solution

You must:

- model your database table in the Cúram model; and
- create the following classes:

  - an adapter for generated data access methods;
  - an implementation for your entity DAO interface; and
  - an implementation for your entity interface.

Model your database table in the Cúram model

You must model your database table in the Cúram model using Cúram's modeling tools.

Ensure that:

- you model a single primary key attribute for the table, which unwinds to a long;
- you model a standard read operation;
- any write operations that you require to model are *standard* write operations (insert, modify and remove);
- if you model a standard insert operation, that it specifies an AUTO_ID strategy (if required);
- if your entity supports optimistic locking (i.e. your entity specifies ALLOW_OPTIMISTIC_LOCKING=yes) and if you model a standard modify operation, that optimistic locking is switched on for this operation (i.e. your modify operation specifies OPTIMISTIC_LOCKING=yes);
- if your entity supports any non-standard read operations, then you model a struct to hold the search criteria and specify the return struct as the full Dtls struct;
- if your entity supports any search operations, then you model a struct to hold the search criteria (you do not need specify any return struct - by default the DtlsList struct will be used); and
- if your entity supports logical deletes, then you model a recordStatus attribute (using the RECORD_STATUS_CODE domain).

Do *not* model:

- any non-stereotyped operations;
- any non-standard write operations;
- any standard write operations which are not required (e.g. remove is only rarely required);
- any read or search operations which return anything other than the full Dtls or DtlsList struct; nor
- any pre- or post- exit points.

Extract and generate your model using the standard command-line tools.

Create an adapter for generated data access methods

You must create an adapter which wraps the generated code for reading, searching and writing database rows.

The Persistence Infrastructure includes a code generator which generates adapter code using information extracted from the Cúram model. To generate a new adapter, add the name of your database table to the file EJBServer/components/<your component>/properties/adapters.properties.

The adapter code generator runs automatically as part of the server build scripts.

Create an implementation for your entity DAO interface

You must create an implementation class for your entity DAO interface.

Create a class in the same package as your DAO interface, and name the class after your entity, suffixed with DAOImpl:

```
package curam.mypackage;

/**
 * Standard implementation of {@linkplain MyNewEntityDAO}.
 */
public class MyNewEntityDAOImpl {
}
```

Figure 66: Creating a DAO implementation file

Your DAO implementation must implement the DAO interface:

```
public class MyNewEntityDAOImpl implements MyNewEntityDAO {
```

Figure 67: Implementing the entity DAO interface

However, if you were to directly implement this interface, you would have to write a huge amount of "plumbing" code. A great deal of plumbing is supplied by StandardDAOImpl, so extend this class, supplying the entity API and the generated Dtls struct for the database table as type parameters:

```
import curam.mypackage.struct.MyNewEntityDtls;
import curam.util.persistence.StandardDAOImpl;

/**
 * Standard implementation of {@linkplain MyNewEntityDAO}.
 */
public class MyNewEntityDAOImpl extends
    StandardDAOImpl<MyNewEntity, MyNewEntityDtls>
    implements MyNewEntityDAO {
```

Figure 68: Extending StandardDAOImpl

Annotate the class with @Singleton:

```
@Singleton
public class MyNewEntityDAOImpl extends
    StandardDAOImpl<MyNewEntity, MyNewEntityDtls>
    implements MyNewEntityDAO {
```

Figure 69: Annotating the DAO implementation as a Singleton

Create a private static variable to hold an instance of your entity adapter:

```
/**
   * Single instance of the entity adapter shared across all DAO
   * implementations.
   */
  private static MyNewEntityAdapter adapter =
    new MyNewEntityAdapter();
```

Figure 70: Declaring a static member variable for the entity adapter

Create a protected constructor which passes the adapter and the class of the entity API to StandardDAOImpl:

```
/**
   * @see StandardDAOImpl
   */
  protected MyNewEntityDAOImpl() {
    super(adapter, MyNewEntity.class);
  }
```

Figure 71: Creating a protected constructor

Use the "Add unimplemented methods" feature in Eclipse to add in the methods you must implement:

```
public MyNewEntity readByName(String name) {
    // TODO Auto-generated method stub
    return null;
  }

  public Set<MyNewEntity> searchByParent(
    MyParentEntity myParentEntity) {
    // TODO Auto-generated method stub
    return null;
  }

  public Set<MyNewEntity> searchByType(
    final MYNEWENTITYTYPEEntry type) {
    // TODO Auto-generated method stub
    return null;
  }
```

Figure 72: Adding unimplemented methods

The implementation of the non-standard singleton readByName calls the adapter to return a Dtls struct (by reading the database), and passes this to a StandardDAOImpl method to create an instance of your entity interface:

```
/**
   * {@inheritDoc}
   */
  public MyNewEntity readByName(final String name) {
    return getEntity(adapter.readByName(name));
  }
```

Figure 73: Implementing a singleton read

The implementation of the readmulti searchByParent calls the adapter to return an array of Dtls structs (by reading the database), and passes this to a StandardDAOImpl method to create set of instances of your entity interface:

```
/**
```

```
    *  {@inheritDoc}
    */
  public Set<MyNewEntity> searchByParent(
    final MyParentEntity myParentEntity) {
    return
 newSet(adapter.searchByParent(myParentEntity.getID()));
  }
```

Figure 74: Implementing a search

The implementation of the readmulti searchByType must translate from the codetable value supplied to the String representation stored on the database:

```
/**
   *  {@inheritDoc}
   */
  public Set<MyNewEntity> searchByType(
    final MYNEWENTITYTYPEEntry type) {
    return newSet(adapter.searchByType(type.getCode()));
  }
```

Figure 75: Implementing a search based on a codetable value

Your implementation of the DAO interface is now complete. However, there is a final important step, which is to specify your DAO implementation as the *default* implementation of the DAO interface.

Open the DAO interface and add an annotation prescribing the default implementation:

```
/**
 * Data access for {@linkplain MyNewEntity}.
 */
@ImplementedBy(MyNewEntityDAOImpl.class)
public interface MyNewEntityDAO extends StandardDAO<MyNewEntity>
 {
```

Figure 76: Specifying the DAO implementation as the default implementation of the DAO interface

If you fail to do this step, then when your application runs you will likely see a NullPointerException when Guice fails to inject instances of your DAO interface:

```
/*
 * This variable will be null if you don't specify the default
 * implementation of MyNewEntityDAO properly...
 */
@Inject
private MyNewEntityDAO myNewEntityDAO;
```

Figure 77: Null pointer exceptions will occur if no default DAO implementation is specified on the DAO interface

Putting it all together

Here's the complete code for the DAO implementation:

```
package curam.mypackage;

import java.util.Set;

import com.google.inject.Singleton;
```

```
import curam.mypackage.struct.MyNewEntityDtls;
import curam.util.persistence.StandardDAOImpl;

/**
 * Standard implementation of {@linkplain MyNewEntityDAO}.
 */
@Singleton
public class MyNewEntityDAOImpl extends
    StandardDAOImpl<MyNewEntity, MyNewEntityDtls> implements
    MyNewEntityDAO {

  /**
   * Single instance of the entity adapter shared across all DAO
   * implementations.
   */
  private static MyNewEntityAdapter adapter =
      new MyNewEntityAdapter();

  /**
   * @see StandardDAOImpl
   */
  protected MyNewEntityDAOImpl() {
    super(adapter, MyNewEntity.class);
  }

  /**
   * {@inheritDoc}
   */
  public MyNewEntity readByName(final String name) {
    return getEntity(adapter.readByName(name));
  }

  /**
   * {@inheritDoc}
   */
  public Set<MyNewEntity> searchByParent(
      final MyParentEntity myParentEntity) {
    return
 newSet(adapter.searchByParent(myParentEntity.getID()));
  }

}
```

Figure 78: Complete listing for an entity DAO implementation

Create an implementation for your entity interface

You must create an implementation class for your entity interface. In this scenario you will only create the skeleton of your implementation class - it will be fleshed-out in later scenarios.

Create a class in the same package as your entity interface, and name the class after your entity, suffixed with Impl:

```
package curam.mypackage;

/**
 * Standard implementation of {@linkplain MyNewEntity}.
 */
public class MyNewEntityImpl
```

```
}
```
Figure 79: Creating an entity implementation file

Your entity implementation must implement the entity interface:

```
public class MyNewEntityImpl implements MyNewEntity {
```
Figure 80: Implementing the entity API

There are a number of common development patterns in the Cúram server layer, and the Persistence Infrastructure comes with a number of helper implementations that implement these patterns.

A common pattern is that an entity:

- stores its data on a single database table;
- supports logical deletes; and
- requires logic for single-field, cross-field and cross-entity validations.

These patterns are implemented by the SingleTableLogicallyDeleteableEntityImpl helper class, so let's base your entity implementation on it:

```
package curam.mypackage;
import curam.mypackage.struct.MyNewEntityDtls;
import

 curam.util.persistence.helper.SingleTableLogicallyDeleteableEntityImpl;

/**
 * Standard implementation of {@linkplain MyNewEntity}.
 */
public class MyNewEntityImpl extends
    SingleTableLogicallyDeleteableEntityImpl<MyNewEntityDtls>
    implements MyNewEntity {
```
Figure 81: Entity implementing extending SingleTableLogicallyDeleteableEntityImpl

SingleTableLogicallyDeleteableEntityImpl provides a standard implementation of these methods:

- insert;
- modify;
- cancel;
- lock;
- getID;
- getRecordStatus; and
- getVersionNo.

Add a protected no-argument constructor:

```
protected MyNewEntityImpl() {
    /* Protected no-arg constructor for use only by Guice */
  }
```
Figure 82: Adding a protected constructor to the entity implementation

Use the "Add unimplemented methods" feature in Eclipse to add in the methods you must implement, and categorize them to aid readability:

```
package curam.mypackage;
```

```java
import java.util.HashMap;
import java.util.Map;
import java.util.Set;

import com.google.inject.Inject;

import curam.message.impl.MYNEWENTITYExceptionCreator;
import
 curam.mypackage.codetable.impl.MYLIFECYCLEENTITYSTATEEntry;
import curam.mypackage.struct.MyNewEntityDtls;
import curam.util.persistence.ValidationHelper;
import curam.util.persistence.helper.CodetableState;
import

 curam.util.persistence.helper.SingleTableLogicallyDeleteableEntityImpl;
import curam.util.persistence.helper.State;
import curam.util.persistence.helper.Transition;
import curam.util.type.DateRange;
import curam.util.type.StringHelper;

/**
 * Standard implementation of {@linkplain MyNewEntity}.
 */
public class MyNewEntityImpl extends
    SingleTableLogicallyDeleteableEntityImpl<MyNewEntityDtls>
    implements MyNewEntity {

  protected MyNewEntityImpl() {
    /* Protected no-arg constructor for use only by Guice */
  }

  /*
   * Field getters
   */
  public String getName() {
    // TODO Auto-generated method stub
    return null;
  }

  public DateRange getDateRange() {
    // TODO Auto-generated method stub
    return null;
  }

  public State getLifecycleState() {
    // TODO Auto-generated method stub
    return null;
  }

  public MYNEWENTITYTYPEEntry getType() {
    // TODO Auto-generated method stub
    return null;
  }

  /*
   * Related-entity getters
   */
  public Set<MyChildEntity> getMyChildren() {
```

```
      // TODO Auto-generated method stub
      return null;
    }

    public MyParentEntity getMyParentEntity() {
      // TODO Auto-generated method stub
      return null;
    }

    /*
     * Setters
     */
    public void setMyParentEntity(MyParentEntity value) {
      // TODO Auto-generated method stub

    }

    public void setName(String value) {
      // TODO Auto-generated method stub

    }

    public void setType(final MYNEWENTITYTYPEEntry value) {
      // TODO Auto-generated method stub
    }

    /*
     * Validation
     */
    public void mandatoryFieldValidation() {
      // TODO Auto-generated method stub

    }

    public void crossFieldValidation() {
      // TODO Auto-generated method stub

    }

    public void crossEntityValidation() {
      // TODO Auto-generated method stub

    }

}
```

Figure 83: Adding unimplemented methods to the entity implementation

Your implementation of the skeletal entity interface is now complete. However, there is a final important step, which is to specify your entity implementation as the *default* implementation of the entity interface.

Open the entity interface and add an annotation prescribing the default implementation:

```
/**
 * Description of my wonderful new entity.
 */
@ImplementedBy(MyNewEntityImpl.class)
public interface MyNewEntity extends StandardEntity, DateRanged,
```

```
        Insertable, OptimisticLockModifiable, LogicallyDeleteable {
```

Figure 84: Specifying the entity implementation as the default implementation of the entity API

If you fail to do this step, then when your application runs you will likely see exceptions when Guice callers of your API attempt to read or create instances of your entity:

```
/*
     * These attempts to construct instances of the entity
 interface
     * will fail if you don't specify the default implementation
 of
     * MyNewEntity properly...
     */
    final long someID = 123;
    final MyNewEntity tryingToRead = myNewEntityDAO.get(someID);

    final MyNewEntity tyringToCreate =
 myNewEntityDAO.newInstance();
```

Figure 85: Exceptions will occur if no default entity implementation is specified on the entity API

You want to implement getters

The problem

You have created a skeletal implementation for your entity. You now need to implement getter methods.

How do you implement getters?

The solution

You must create implementations for your skeletal getter methods created above. Each getter method is responsible for retrieving one or more fields from an underlying Dtls struct and returning a value (either primitive or object) to calling code.

The implementation of your entity has at its heart an instance of a RowManager. The RowManager instance contains a generated Dtls struct and manages the manipulation of this struct.

Getter methods must use the RowManager. getDtls method to get at the Dtls struct. For implementations extending SingleTableEntityImpl (which the example does via SingleTableLogicallyDeleteableEntityImpl), there is a convenience getDtls method which can be used directly as a shorthand.

Our example requires these getters to be implemented:

- getName;
- getDateRange;
- getType;
- getMyParentEntity; and
- getMyChildren.

In general the JavaDoc for your getter implementations can simply inherit from your entity API JavaDoc.

getName

The getter for name is a straight-forward mapping of the name held in the Dlts struct:

```
/**
  * {@inheritDoc}
  */
 public String getName() {
   return getDtls().name;
 }
```

Figure 86: Implementation of a simple get method

getDateRange

The getter for your entity's date range must use the startDate and endDate held on the generated Dtls struct and construct a new DateRange object:

```
/**
  * {@inheritDoc}
  */
 public DateRange getDateRange() {
   return new DateRange(getDtls().startDate, getDtls().endDate);
 }
```

Figure 87: Implementation of a get method which returns a single object representing multiple database column values

getType

The getter for your entity's type must retrieve the relevant MYNEWENTITYTYPEEntry value based on the codetable code String value held in the typeCode field on the Dtls struct:

```
/**
  * {@inheritDoc}
  */
 public MYNEWENTITYTYPEEntry getType() {
   return MYNEWENTITYTYPEEntry.get(getDtls().typeCode);
 }
```

Figure 88: Implementation of a get method which returns a codetable entry value

getMyParentEntity

The getter for a single record must retrieve that related record and return it. However, the getter must check whether the key is currently zero (which is used throughout the server application to signify that a unique ID value has not been set), and if so instead return null.

Create a class member variable for the related record's DAO:

```
@Inject
  private MyParentEntityDAO myParentEntityDAO;
```

Figure 89: Creating a member variable for a related entity's DAO

In the getter, conditionally call the DAO, depending on whether the value of myParentEntityID is zero:

```
/**
  * {@inheritDoc}
  */
 public MyParentEntity getMyParentEntity() {
```

```
    final long myParentEntityID = getDtls().myParentEntityID;

    if (myParentEntityID == 0) {
      return null;
    } else {
      return myParentEntityDAO.get(myParentEntityID);
    }
  }
}
```

Figure 90: Implementing a get method to retrieve a related entity instance

getMyChildren

The getter for a set of related records must call a DAO method to perform a search.

Create a class member variable for the related records' DAO:

```
@Inject
  private MyChildEntityDAO myChildEntityDAO;
```

Figure 91: Creating a member variable for another related entity's DAO

In the getter, call the DAO passing in this object:

```
/**
   * {@inheritDoc}
   */
  public Set<MyChildEntity> getMyChildren() {
    return myChildEntityDAO.searchByParent(this);
  }
```

Figure 92: Implementing a get method to retrieve a set of related entity instances

You must add the searchByParent method to the DAO:

```
/**
 * Data access for {@linkplain MyChildEntity}.
 */
public interface MyChildEntityDAO
  extends StandardDAO<MyChildEntity> {

  /**
   * Searches all the instances belonging to the specified
 parent.
   *
   * @param myNewEntity
   *          the parent to search for
   * @return all the instances belonging to the specified parent,
 or
   *          an empty set if none found.
   */
  public Set<MyChildEntity> searchByParent(
    final MyNewEntity myNewEntity);
```

Figure 93: Adding a search method to the related entity's DAO interface

> **Important:** Do *not* be tempted to take Eclipse's suggestion of using the MyNewEntityImpl class as an argument:

```
/** ********** VERY VERY BAD - DO NOT DO THIS! ********** */
```

```
  /**
   * Searches all the instances belonging to the specified
 parent.
   *
   * @param impl
   *          the parent to search for
   * @return all the instances belonging to the specified parent,
 or
   *          an empty set if none found.
   */
  public Set<MyChildEntity> searchByParent(MyNewEntityImpl impl);
  /** ********** VERY VERY BAD - DO NOT DO THIS! ********** */
```

Figure 94: Incorrect - adding a search method taking the entity implementation as a parameter

(The underlying principle here is that entity and DAO interfaces are allowed to be dependent on other entity and DAO interfaces, but are *not* allowed to be dependent on implementations.)

If an implementation exists for MyChildEntityDAO, then you must implement the new method, and model a new search operation (a readmulti) to retrieve the required records.

Putting it all together

You now have a full set of implemented getter methods. In doing the implementation, you have:

- fleshed out MyNewEntityImpl; and
- added a new method to MyChildEntityDAO.

The full code for these classes is shown below:

MyNewEntityImpl

```
package curam.mypackage;

import java.util.Set;

import com.google.inject.Inject;

import curam.mypackage.struct.MyNewEntityDtls;
import curam.util.exception.InformationalException;
import

 curam.util.persistence.helper.SingleTableLogicallyDeleteableEntityImpl;
import curam.util.type.Date;
import curam.util.type.DateRange;

/**
 * Standard implementation of {@linkplain MyNewEntity}.
 */
public class MyNewEntityImpl extends
    SingleTableLogicallyDeleteableEntityImpl<MyNewEntityDtls>
    implements MyNewEntity {

  @Inject
  private MyParentEntityDAO myParentEntityDAO;

  @Inject
  private MyChildEntityDAO myChildEntityDAO;
```

```java
protected MyNewEntityImpl() {
  /* Protected no-arg constructor for use only by Guice */
}

/*
 * Field getters
 */

/**
 * {@inheritDoc}
 */
public String getName() {
  return getDtls().name;
}

/**
 * {@inheritDoc}
 */
public DateRange getDateRange() {
  return new DateRange(getDtls().startDate, getDtls().endDate);
}

/**
 * {@inheritDoc}
 */
public MYNEWENTITYTYPEEntry getType() {
  return MYNEWENTITYTYPEEntry.get(getDtls().typeCode);
}

/*
 * Related-entity getters
 */
/**
 * {@inheritDoc}
 */
public Set<MyChildEntity> getMyChildren() {
  return myChildEntityDAO.searchByParent(this);
}

/**
 * {@inheritDoc}
 */
public MyParentEntity getMyParentEntity() {
  final long myParentEntityID = getDtls().myParentEntityID;

  if (myParentEntityID == 0) {
    return null;
  } else {
    return myParentEntityDAO.get(myParentEntityID);
  }
}

/*
 * Setters
 */
public void setMyParentEntity(MyParentEntity value) {
  // TODO Auto-generated method stub
```

```
      }

      public void setName(String value) {
        // TODO Auto-generated method stub

      }

      public void setType(final MYNEWENTITYTYPEEntry value) {
        // TODO Auto-generated method stub
      }

      /*
       * Validation
       */
      public void mandatoryFieldValidation() {
        // TODO Auto-generated method stub

      }

      public void crossFieldValidation() {
        // TODO Auto-generated method stub

      }

      public void crossEntityValidation() {
        // TODO Auto-generated method stub

      }

    }
```

Figure 95: Complete listing for an entity implementation with implemented getter methods

MyChildEntityDAO

```
package curam.mypackage;

import java.util.Set;

import curam.util.persistence.StandardDAO;

/**
 * Data access for {@linkplain MyChildEntity}.
 */
public interface MyChildEntityDAO extends
    StandardDAO<MyChildEntity> {

  /**
   * Searches all the instances belonging to the specified
 parent.
   *
   * @param myNewEntity
   *            the parent to search for
   * @return all the instances belonging to the specified parent,
 or
   *            an empty set if none found.
   */
  public Set<MyChildEntity> searchByParent(
```

```
        final MyNewEntity myNewEntity);

}
```

Figure 96: Complete listing for changes made to a related entity DAO arising from implementation of a getter which calls a new search

You want to implement new row defaults

The problem

You have an entity which has one or more fields which require defaulting when new instances are inserted into the database.

How do you specify new row defaults for your entity?

The solution

You must override the setNewInstanceDefaults method and initialize any fields which require defaulting before a new instance is inserted onto the database.

In the example, the initial typeCode of MyNewEntity must be defaulted to "SomeType", and the date range set to start on today's date and no end date specified:

```
/**
   * Defaults:
   * <ul>
   * <li>the type to {@linkplain MYNEWENTITYTYPEEntry#SOMETYPE};
   * and</li>
   * <li>the date range to {@linkplain DateRange#todayOnwards()}.
   * </li>
   * </ul>
   */
  public void setNewInstanceDefaults() {
    setType(MYNEWENTITYTYPEEntry.SOMETYPE);
    setDateRange(DateRange.todayOnwards());
  }
```

Figure 97: Setting default values on new instances of an entity

> **Note:** Be sure to include a call to super.setNewInstanceDefaults().
>
> For example, for logically-deleteable entities, this super implementation defaults the recordStatus to "active".

Note that this implementation of new instance defaults calls a new *private* setter (setDateRange - this setter is not available in the entity API but is local to the entity implementation class. (Recall that you do *not* want callers of your class to be able to set its dates directly.)

The DateRange class contains the convenience method todayOnwards to return a data range that starts on the current business date and has no end date specified.

Create a skeletal implementation of this private setters - you'll flesh it out later:

```
public void setDateRange(DateRange value) {
    // TODO Auto-generated method stub
```

```
}
```
Figure 98: Creating a skeletal implementation of a private setter method

You want to implement setters

The problem

You have created a skeletal implementation for your entity. You now need to implement setter methods.

How do you implement setters?

The solution

You must create implementations for your skeletal setter methods created above. Each setter method is responsible for taking a value (either primitive or object) supplied by calling code and setting one or more fields in an underlying Dtls struct.

Our example requires these setters to be implemented:

- setName;
- setDateRange (private, not present in the entity interface);
- setType; and
- setMyParentEntity.

> **Note:** In general the JavaDoc for your setter implementations can simply inherit from your entity API JavaDoc.
>
> Private setters must detail their own JavaDoc (as there is no API JavaDoc to inherit from).

setName

The setter for the name field maps the value provided to the name field on the Dtls struct. The setter must trim/compress white space and convert any null value passed to an empty string:

```
/**
   * {@inheritDoc}
   */
  public void setName(final String value) {
    getDtls().name = StringHelper.trim(value);
  }
```
Figure 99: Implementation of a simple setter method

The StringHelper class contains the convenience method trim which converts a null to an empty string, trims white space from the ends of genuine strings passed and compresses any contiguous embedded spaces down to a single space.

setDateRange

The setter for the date range field must set *two* values on the underlying Dtls struct:

```
/**
   * Sets the start and end fields from the date range supplied.
   *
```

```
 * @param value
 *          the date range supplied
 */
private void setDateRange(final DateRange value) {
  getDtls().startDate = value.start();
  getDtls().endDate = value.end();
}
```

Figure 100: Implementation of a setter method which sets multiple database column values from one object

setType

The setter for the typeCode database column must convert the state supplied into its codetable code for storage on the database:

```
/**
   * {@inheritDoc}
   */
  public void setType(final MYNEWENTITYTYPEEntry value) {
    getDtls().typeCode = value.getCode();
  }
```

Figure 101: Implementation of a setter which translates an codetable entry to a codetable code String value

setMyParentEntity

The setter for a related record must retrieve the object's ID and store it in the appropriate field on the Dtls struct. A null value must be converted to zero:

```
/**
   * {@inheritDoc}
   */
  public void setMyParentEntity(final MyParentEntity value) {
    final long myParentEntityID;
    if (value == null) {
      myParentEntityID = 0;
    } else {
      myParentEntityID = value.getID();
    }

    getDtls().myParentEntityID = myParentEntityID;
  }
```

Figure 102: Implementation of a setter which sets a related entity

Putting it all together

You now have a full set of implemented setter methods. Here's the code so far:

```
package curam.mypackage;

import java.util.Set;

import com.google.inject.Inject;

import curam.mypackage.struct.MyNewEntityDtls;
import curam.util.exception.InformationalException;
import

 curam.util.persistence.helper.SingleTableLogicallyDeleteableEntityImpl;
```

```
import curam.util.type.Date;
import curam.util.type.DateRange;
import curam.util.type.StringHelper;

/**
 * Standard implementation of {@linkplain MyNewEntity}.
 */
public class MyNewEntityImpl extends
    SingleTableLogicallyDeleteableEntityImpl<MyNewEntityDtls>
    implements MyNewEntity {

  @Inject
  private MyParentEntityDAO myParentEntityDAO;

  @Inject
  private MyChildEntityDAO myChildEntityDAO;

  protected MyNewEntityImpl() {
    /* Protected no-arg constructor for use only by Guice */
  }

  /*
   * Field getters
   */

  /**
   * {@inheritDoc}
   */
  public String getName() {
    return getDtls().name;
  }

  /**
   * {@inheritDoc}
   */
  public DateRange getDateRange() {
    return new DateRange(getDtls().startDate, getDtls().endDate);
  }

  /**
   * {@inheritDoc}
   */
  public MYNEWENTITYTYPEEntry getType() {
    return MYNEWENTITYTYPEEntry.get(getDtls().typeCode);
  }

  /*
   * Related-entity getters
   */
  /**
   * {@inheritDoc}
   */
  public Set<MyChildEntity> getMyChildren() {
    return myChildEntityDAO.searchByParent(this);
  }

  /**
   * {@inheritDoc}
```

```java
  */
public MyParentEntity getMyParentEntity() {
  final long myParentEntityID = getDtls().myParentEntityID;

  if (myParentEntityID == 0) {
    return null;
  } else {
    return myParentEntityDAO.get(myParentEntityID);
  }
}

/*
 * Setters
 */

/**
 * {@inheritDoc}
 */
public void setMyParentEntity(final MyParentEntity value) {
  final long myParentEntityID;
  if (value == null) {
    myParentEntityID = 0;
  } else {
    myParentEntityID = value.getID();
  }

  getDtls().myParentEntityID = myParentEntityID;
}

/**
 * {@inheritDoc}
 */
public void setName(final String value) {
  getDtls().name = StringHelper.trim(value);
}

/**
 * Sets the start and end fields from the date range supplied.
 *
 * @param value
 *          the date range supplied
 */
private void setDateRange(final DateRange value) {
  getDtls().startDate = value.start();
  getDtls().endDate = value.end();
}

/**
 * {@inheritDoc}
 */
public void setType(final MYNEWENTITYTYPEEntry value) {
  getDtls().typeCode = value.getCode();
}

/*
 * Validation
 */
public void mandatoryFieldValidation() {
```

```
         // TODO Auto-generated method stub

     }

     public void crossFieldValidation() {
        // TODO Auto-generated method stub

     }

     public void crossEntityValidation() {
        // TODO Auto-generated method stub

     }

     /**
      * Defaults:
      * <ul>
      * <li>the type to {@linkplain MYNEWENTITYTYPEEntry#SOMETYPE};
      * and</li>
      * <li>the date range to {@linkplain DateRange#todayOnwards()}.
      * </li>
      * </ul>
      */
     public void setNewInstanceDefaults() {
        setType(MYNEWENTITYTYPEEntry.SOMETYPE);
        setDateRange(DateRange.todayOnwards());
     }

}
```

Figure 103: Complete listing for an entity implementation with implemented setter methods

You want to implement single-field validation

The problem

You have created an implementation for your entity setters. You now need to implement single-field validation logic.

How do you implement single-field validation logic?

The solution

Each field setter is responsible for ensuring that the value being set is appropriate. In general, errors arising from single-field validation should be "accumulated" using the InformationalManager, so that callers can be notified of *all* the single-field validation errors found. This is particularly useful to online users who may have entered several fields in error - if single-field validation errors are reported one-by-one then it would be frustrating for the user to be presented with a series of single-error messages instead of a list of all known single-field validation errors.

One important corollary of this is that each field setter should *only* attempt to validate the field being set. It should make no reference to other fields.

For the purposes of single-field validation, a *field* corresponds to the value received by the setter. Generally, there is one setter per underlying database field; however, in cases where database

fields are grouped together (notably with DateRange), it is the object received by the setter which is validated, not the individual underlying database fields. In the case of a setDateRange method, it is the date range which is validated. This single-field validation of the DateRange typically includes start/end date validation which under classic Cúram would have been considered "cross-field" validation.

One other point to note is that the validation of whether *mandatory* fields have been set is deferred to a special "mandatory field validation" method (see below); this is because you cannot guarantee which (if any) setters have been called from calling code.

You must add single-field validation logic to the setters:

- setName;
- setDateRange; and
- setType; and
- setMyParentEntity.

setName

After analyzing requirements, you determine that the setter for the name must validate that the name length is within acceptable bounds.

First create a message catalog:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<messages package="curam.message">
  <message name="ERR_MY_NEW_ENTITY_FV_NAME_EMPTY">
    <locale language="en">
      The name must be specified.
    </locale>
  </message>
  <message name="ERR_MY_NEW_ENTITY_FV_NAME_SHORT">
    <locale language="en">
      The name must be at least %1n characters.
    </locale>
  </message>
  <message name="ERR_MY_NEW_ENTITY_FV_NAME_LONG">
    <locale language="en">
      The name must be no more than %1n characters.
    </locale>
  </message>
</messages>
```

Figure 104: Creating a message catalog with validation error messages

Note that the validation messages for minimum/maximum length take as argument the minimum/maximum lengths permitted, rather than hard-coding these bounds into the messages.

Now code validation logic in the setter and raise errors using the ValidationHelper:

```java
/**
   * Minimum valid name length
   */
  private static final long kMinimumNameLength = 3;

 /**
   * {@inheritDoc}
   */
```

```
public void setName(final String value) {
  getDtls().name = StringHelper.trim(value);

  final long nameLength = getDtls().name.length();
  if (nameLength > 0 && nameLength < kMinimumNameLength) {
    ValidationHelper.addValidationError(
    MYNEWENTITYExceptionCreator
        .ERR_MY_NEW_ENTITY_FV_NAME_SHORT(kMinimumNameLength));
  } else if (nameLength > MyNewEntityAdapter.kMaxLength_name) {
    ValidationHelper.addValidationError(
    MYNEWENTITYExceptionCreator
        .ERR_MY_NEW_ENTITY_FV_NAME_LONG(
      MyNewEntityAdapter.kMaxLength_name));
  }
}
```

Figure 105: Implementing single field validation logic

Note that:

- validation regarding whether the name has been set *at all* will occur during mandatory-field validation; and
- constants for the maximum length of database text columns are automatically generated into the entity adapter. These constants should be used in preference to creating your own, as they will automatically be updated should the length of the database column be customized (by changing the domain definition in the model).

The ValidationHelper class contains the convenience method addValidationError to format an error message and add it to the informational manager. It takes an AppException or CatEntry (shown here). It also has a deprecated overload which takes a String, which can be used as a "quick and dirty" way of writing error messages:

```
/** **** Must be "cleaned up" prior to testing and release ** */
    final long nameLength = getDtls().name.length();
    if (nameLength > 0 && nameLength < kMinimumNameLength) {
      ValidationHelper.addValidationError("Name too short!");
    } else if (nameLength > MyNewEntityAdapter.kMaxLength_name) {
      ValidationHelper.addValidationError("Name too long!");
    }
    /** **** Must be "cleaned up" prior to testing and release **
 */
```

Figure 106: Using ValidationHelper to create temporary error messages

You *must* convert these Strings to message catalog entries prior to testing and release. This facility exists purely to minimize the "switching" you might have to do between editing Java and editing/generating message files that you might otherwise have to do when writing validation logic.

setDateRange

After analyzing requirements, you determine that the date range requires the following validation logic:

- the range is valid (i.e. that the start date is not after the end date); and
- the start date has been specified (but the end date is optional, or, more to the point, whether the end date is required is dependent on the value of other fields).

The first of these is amenable to single-field validation; the second is more appropriate for mandatory-field validation.

Code validation logic to use the standard validation message on DateRange:

```
/**
   * Sets the start and end fields from the date range supplied.
   *
   * @param value
   *           the date range supplied
   */
  private void setDateRange(final DateRange value) {
    getDtls().startDate = value.start();
    getDtls().endDate = value.end();

    value.validateRange();
  }
```

Figure 107: Using DateRange to perform standard validation

The DateRange class contains the convenience method validateRange which validates the start and end dates of the range and raises a standard error message if the start date is after the end date. If you require a specific message, then use DateRange.isValidRange instead.

setType

After analyzing requirements, you determine that the type field has no single-field validation requirements. Mandatory field validation will be required to ensure that the type has been set.

Note that the caller of this method must supply an instance of MYNEWENTITYTYPEEntry, and will fail with a runtime error if it attempts to retrieve an entry value which from a value which is *not* present in the corresponding code table.

setMyParentEntity

After analyzing requirements, you determine that the parent entity ID field has no single-field validation requirements. Mandatory field validation will be required to ensure that the parent entity has been set.

Putting it all together

Here's the entity implementation code with the single-field validation logic:

```
package curam.mypackage;

import java.util.Set;

import com.google.inject.Inject;

import curam.message.impl.MYNEWENTITYExceptionCreator;
import curam.mypackage.struct.MyNewEntityDtls;
import curam.util.persistence.ValidationHelper;
import

 curam.util.persistence.helper.SingleTableLogicallyDeleteableEntityImpl;
import curam.util.type.DateRange;
import curam.util.type.StringHelper;
```

```
/**
 * Standard implementation of {@linkplain MyNewEntity}.
 */
public class MyNewEntityImpl extends
    SingleTableLogicallyDeleteableEntityImpl<MyNewEntityDtls>
    implements MyNewEntity {

  @Inject
  private MyParentEntityDAO myParentEntityDAO;

  @Inject
  private MyChildEntityDAO myChildEntityDAO;

  /**
   * Minimum valid name length
   */
  private static final long kMinimumNameLength = 3;

  protected MyNewEntityImpl() {
    /* Protected no-arg constructor for use only by Guice */
  }

  /*
   * Field getters
   */

  /**
   * {@inheritDoc}
   */
  public String getName() {
    return getDtls().name;
  }

  /**
   * {@inheritDoc}
   */
  public DateRange getDateRange() {
    return new DateRange(getDtls().startDate, getDtls().endDate);
  }

  /**
   * {@inheritDoc}
   */
  public MYNEWENTITYTYPEEntry getType() {
    return MYNEWENTITYTYPEEntry.get(getDtls().typeCode);
  }

  /*
   * Related-entity getters
   */
  /**
   * {@inheritDoc}
   */
  public Set<MyChildEntity> getMyChildren() {
    return myChildEntityDAO.searchByParent(this);
  }

  /**
```

```java
 * {@inheritDoc}
 */
public MyParentEntity getMyParentEntity() {
  final long myParentEntityID = getDtls().myParentEntityID;

  if (myParentEntityID == 0) {
    return null;
  } else {
    return myParentEntityDAO.get(myParentEntityID);
  }
}

/*
 * Setters
 */

/**
 * {@inheritDoc}
 */
public void setMyParentEntity(final MyParentEntity value) {
  final long myParentEntityID;
  if (value == null) {
    myParentEntityID = 0;
  } else {
    myParentEntityID = value.getID();
  }

  getDtls().myParentEntityID = myParentEntityID;
}

/**
 * {@inheritDoc}
 */
public void setName(final String value) {
  getDtls().name = StringHelper.trim(value);

  final long nameLength = getDtls().name.length();
  if (nameLength > 0 && nameLength < kMinimumNameLength) {
    ValidationHelper
        .addValidationError(MYNEWENTITYExceptionCreator

.ERR_MY_NEW_ENTITY_FV_NAME_SHORT(kMinimumNameLength));
  } else if (nameLength > MyNewEntityAdapter.kMaxLength_name) {
    ValidationHelper
        .addValidationError(MYNEWENTITYExceptionCreator
            .ERR_MY_NEW_ENTITY_FV_NAME_LONG(
        MyNewEntityAdapter.kMaxLength_name));
  }
}

/**
 * Sets the start and end fields from the date range supplied.
 *
 * @param value
 *          the date range supplied
 */
private void setDateRange(final DateRange value) {
  getDtls().startDate = value.start();
```

```
      getDtls().endDate = value.end();

      value.validateRange();
    }

  /**
   * {@inheritDoc}
   */
  public void setType(final MYNEWENTITYTYPEEntry value) {
    getDtls().typeCode = value.getCode();
  }

  /*
   * Validation
   */
  public void mandatoryFieldValidation() {
    // TODO Auto-generated method stub

  }

  public void crossFieldValidation() {
    // TODO Auto-generated method stub

  }

  public void crossEntityValidation() {
    // TODO Auto-generated method stub

  }

  /**
   * Defaults:
   * <ul>
   * <li>the type to {@linkplain MYNEWENTITYTYPEEntry#SOMETYPE};
   * and</li>
   * <li>the date range to {@linkplain DateRange#todayOnwards()}.
   * </li>
   * </ul>
   */
  public void setNewInstanceDefaults() {
    setType(MYNEWENTITYTYPEEntry.SOMETYPE);
    setDateRange(DateRange.todayOnwards());
  }

}
```

Figure 108: Complete listing for an entity implementation with implemented single-field validation logic

You want to implement mandatory-field validation

The problem

Your entity is only valid if certain fields have values specified (commonly known as "mandatory" fields).

How do you implement mandatory-field validation logic?

The solution

Each class that implements Validator (which MyNewEntityImpl does via SingleTableEntityImpl) must implement standard methods for validation logic.

(Note that in general, implementation classes may implement Validator but that entity APIs should *not* extend Validator - you do *not* want calling code to be able to call validation methods directly.)

One of these Validator methods is mandatoryFieldValidation, where you must place any logic which detects whether any field value has not been set. It is up to your logic to determine how to detect whether or not a field value is "set" (typically with reference to the defaulted values of the generated Dtls struct).

The persistence infrastructure automatically calls mandatoryFieldValidation prior to any insert or modify operation (but not before a physical remove operation), and fails the operation if any validation errors have been raised. These errors include those raised by setter methods as well as by mandatoryFieldValidation. In particular, processing will not proceed to cross-field or cross-entity validation if any single-field or mandatory-field validation errors have been found.

Logic placed in mandatoryFieldValidation must consider each field on a field-by-field basis; logic which checks one field value against another must instead be placed in cross-field validation. In particular, the persistence infrastructure will prevent any database access occurring during mandatoryFieldValidation.

After analyzing requirements, you determine that in order to be valid your entity must always have the following specified:

- name;
- start date of the date range;
- type; and
- parent entity instance.

You add the following code to implement mandatoryFieldValidation:

```
/**
   * {@inheritDoc}
   */
  public void mandatoryFieldValidation() {
    /*
     * Name cannot be empty
     */
    if (StringHelper.isEmpty(getDtls().name)) {
      ValidationHelper.addValidationError(
      MYNEWENTITYExceptionCreator
          .ERR_MY_NEW_ENTITY_FV_NAME_EMPTY());
    }

    /*
     * Start date must be specified
     */
    getDateRange().validateStarted();

    /*
     * Type must be specified
     */
    if (getType().equals(MYNEWENTITYTYPEEntry.NOT_SPECIFIED)) {
      ValidationHelper
```

```
            .addValidationError("Type must be specified");
      }

      /*
       * Parent entity instance must be specified
       */
      if (getMyParentEntity() == null) {
        ValidationHelper
            .addValidationError(
"Parent entity instance must be specified"
        );
      }

    }
```

Figure 109: Implementing mandatory field validation logic

Note that:

- the fields are tested sequentially, raising validation errors via ValidationHelper, so that all errors are accumulated and reported in one "batch" to calling code;
- the StringHelper class contains the convenience method isEmpty to check whether the string is empty or null;
- MYNEWENTITYTYPEEntry contains the generated constant NOT_SPECIFIED, which is the value returned if a null or empty String is passed to MYNEWENTITYTYPE.get; and
- the DateRange class contains the convenience method validateStarted which raises a standard error message if no start date has been specified. If you require a specialized message, use DateRange.isStarted instead.

You want to implement cross-field validation

The problem

Your entity is only valid if the data in certain groups of fields obeys business rules (commonly known as "cross-field" validation).

How do you implement cross-field validation logic?

The solution

Each class that implements Validator has a crossFieldValidation method where you must place any logic which validates the value in one field against one or more others.

If single-field and mandatory-field validation has succeeded, then the persistence infrastructure automatically calls crossFieldValidation prior to any insert or modify operation (but not before a physical remove operation), and fails the operation if any validation errors have been raised. In particular, processing will not proceed to cross-entity validation if any cross-field validation errors have been found. The persistence infrastructure will prevent any database access occurring during crossFieldValidation.

You want to implement cross-entity validation

The problem

Your entity is only valid if its data obeys business rules with regard to data on other entities (commonly known as "cross-entity" validation).

How do you implement cross-entity validation logic?

The solution

Each class that implements Validator has a crossEntityValidation method where you must place any logic which validates the value in your entity against data on other entities.

If cross-field validation has succeeded, then the persistence infrastructure automatically calls crossEntityValidation after any insert or modify operation (but not after a physical remove operation), and fails the operation if any validation errors have been raised. The persistence infrastructure permits database access occurring during crossEntityValidation, so that your validation logic can retrieve data on other entities required to implement the validation.

1.5 Creating a Guice module

In earlier chapters you saw how Guice's @ImplementedBy annotation is used to designate the default implementation of an interface.

Guice has another more flexible configuration mechanism, namely a Guice *Module* which you code yourself.

Moreover, the configuration that you place in a Guice Module takes precedence over any @ImplementedBy annotations in the code, which allows you to configure Guice to use your custom implementation instead of the default implementation. This may be useful for customisation or testing purposes.

To create your own Guice Module, follow these steps:

• create a class extending AbstractModule; and
• store a row on ModuleClassName.

Create a class extending AbstractModule

Create a class as follows:

```
package curam.mypackage;

import com.google.inject.AbstractModule;

/**
 * Contains my Guice bindings.
 */
public class MyModule extends AbstractModule {

  /**
   * {@inheritDoc}
   */
  @Override
  public void configure() {
    // no explicit bindings
  }
```

```
}
```
Figure 110: Skeleton Guice Module

You can now add new Guice "bindings" to the configure method to override default implementations:

```
@Override
  public void configure() {
    bind(MyNewEntity.class).to(MyCustomNewEntityImpl.class);
  }
```

This configuration will cause Guice to dish up an instances of MyCustomNewEntityImpl instead of the default implementation (MyNewEntityImpl), whenever an MNewEntity interface instance is @Injected.

You will also add configuration code if your application uses events (see).

> **Important:** Each interface can only be bound to a *single* implementation. If the set of runtime Guice modules attempts to bind the same interface more than once, Guice will raise a runtime exception.
>
> As such, code which is delivered to customers should not use this mechanism to bind an interface to an implementation in any situation where the customer should be permitted to specify their own binding for the interface.

Store a row on ModuleClassName

The Persistence Infrastructure reads from a database table named ModuleClassName to identify Guice modules which should be loaded.

You must add a row to this database table with the name of your module. The most straightforward way to do this is to use the Data Manager, by creating a custom DMX file containing the row required:

```xml
        <?xml version="1.0" encoding="UTF-8"?>
<table name="ModuleClassName">
  <column
    name="moduleClassName"
    type="text"
  />
  <row>
    <attribute name="moduleClassName">
      <value>
          curam.mypackage.MyModule
        </value>
    </attribute>
  </row>
</table>
```
Figure 111: DMX file to create a row for your module on ModuleClassName

1.6 Events

The Persistence Infrastructure provides some helper classes which allow you to raise and listen for events. You can define your own events or write listeners for ones that are already defined "out-of-the-box".

Events can be a useful tool in removing an explicit dependency from one class (the event raiser) to another (the event listener). If you require to add another listener to an event, you can do so without having to "open up" the code that raises the event - the event raiser and listener only depend on the event interface, not on each other's implementation.

The Persistence Infrastructure supports:

- an instance of a class raising events declared on an interface;
- zero, one or more "listener" instances wired to listen for these events; and
- special classes of "persistence events" which are automatically raised for all persistence operations, signalling when various standard operations are performed on entities.

To implement events and listeners, follow these steps (described in detail below):

- identify where an event must be raised;
- define the Event interface;
- create an EventDispatcherFactory;
- raise events;
- create an event listener; and
- configure Guice.

After these steps there is a description of how to add a listener for generic persistence events.

Identify where an event must be raised

Let's take as an example a simple class which has a simple method:

```
package curam.mypackage;

public class MyEventSource {

  public void doSomething() {

    // do whatever it is that needs to be done
    System.out.println("Do something!");

  }

}
```

Figure 112: A simple class which performs an action

You decide that events should be raised:

- before doSomething performs its logic (preDoSomething); and
- after doSomething performs its logic (postDoSomething).

Define the Event interface

You must define an interface to contain your event methods.

The event interface will be:

- used by the event source to raise events; and
- implemented by event listeners to listen to and react to events.

Note that we are using the word "interface" loosely here. A very important consideration is whether you might ever change the event interface to create additional methods. If you do, all existing implementations of the interface are forced to implement the new methods. In this case, you are strongly advised to use abstract classes rather than Java interfaces. These classes should provide empty implementations of event methods, rather than declare them abstract, so that newly added methods do not cause compilation problems for existing implementations. This approach also means that you can group many related events together in a single abstract class declaration, knowing that only those methods of interest to a particular customer need to be implemented, since default empty implementations for all methods are inherited.

The event interface is typically public so that class in any code package can listen to its events. The interface can be created as an inner interface, in which case it can simply be named Event without fear of name collision with other event interfaces. Typically your entity implementations are package-protected, and so the event interface should be declared as an inner interface of your entity's public *interface*. However, here for brevity an inner interface is shown declared on the simple class:

```
        package curam.mypackage;

public class MyEventSource {
        public abstract class Event {
    public void preDoSomething(MyEventSource raiser) {
      // intentionally empty
    }

    public void postDoSomething(MyEventSource raiser) {
      // intentionally empty
    }
  }
        public void doSomething() {

    // do whatever it is that needs to be done
    System.out.println("Do something!");

  }

}
```

Figure 113: Defining the Event interface

You must carefully think about the signature of your event methods. The event method is free to pass any number of parameters and/or throw exceptions; note though that the EventDispatcherFactory (see below) ignores any return values, so if you require listeners to return a value then you have to supply your own custom event dispatch logic.

Because each listener is a single instance, typically each event method should pass the instance which *raised* the event, so that the listener can identify the source of the event. In the example

both preDoSomething and postDoSomething take an instance of MyEventSource, namely the instance which raised the event.

Create an EventDispatcherFactory

Your class needs a mechanism for dispatching events to listeners.

Create an instance of EventDispatcherFactory parameterized with your Event interface:

```
        package curam.mypackage;

import com.google.inject.Inject;

import curam.util.persistence.helper.EventDispatcherFactory;

public class MyEventSource {

  public abstract class Event {
    public void preDoSomething(MyEventSource raiser) {
      // intentionally empty
    }

    public void postDoSomething(MyEventSource raiser) {
      // intentionally empty
    }
  }
          @Inject
  private EventDispatcherFactory<Event> dispatcher;
          public void doSomething() {

    // do whatever it is that needs to be done
    System.out.println("Do something!");

  }

}
```

Figure 114: Creating an EventDispatcherFactory

Raise events

You must now raise events at appropriate points in the class's logic. Retrieve an instance of your dispatcher to call the methods on your Event interface:

```
        package curam.mypackage;

import com.google.inject.Inject;

import curam.util.persistence.helper.EventDispatcherFactory;

public class MyEventSource {

  public abstract class Event {
    public void preDoSomething(MyEventSource raiser) {
      // intentionally empty
```

```
        }

        public void postDoSomething(MyEventSource raiser) {
         // intentionally empty
        }

    }

    @Inject
    private EventDispatcherFactory<Event> dispatcher;

    public void doSomething() {
            // notify listeners that something is about to happen
        dispatcher.get(Event.class).preDoSomething(this);
            // do whatever it is that needs to be done
        System.out.println("Do something!");
            // notify listeners that something has just been done
        dispatcher.get(Event.class).postDoSomething(this);
            }
}
```

Figure 115: Raising events

Note how the dispatcher.get method took the class of the Event interface as a parameter. Calling this method returned an event "multiplexer" instance on which any method call will be dispatched to each of the registered listeners for that event interface.

This completes the coding to raise events. You can now move on to create listeners.

Create an event listener

You can create as many event listener classes as you require. These classes can be in any code package and each event listener can react to the event in its own way.

```
package curam.mypackage;

import com.google.inject.Singleton;

@Singleton
final class MyListener implements
    curam.mypackage.MyEventSource.Event {

  protected MyListener() {
    // Protected constructor for use only by Guice
  }

  @Override
  public void preDoSomething(final MyEventSource raiser) {
    System.out
        .println("preDoSomething event was raised from object "
            + raiser);

  }

  @Override
  public void postDoSomething(final MyEventSource raiser) {
    System.out
        .println("postDoSomething event was raised from object "
```

```
                    + raiser);

    }

}
```
Figure 116: Creating an event listener class

Note that the listener class implements the Event interface from the event source, and uses the event methods to respond to the events as required.

You have created a listener class which will listen for events raised from MyEventSource instances. However, in order for these events to be dispatched to your listener instance, you must first perform some Guice configuration.

Configure Guice

You must add code to the configure method of your Guice Module (see) to "wire" your listeners to your events.

Note that no Guice configuration is required to simply declare an event interface and dispatch events on it. Configuration is only required for implementations of the event interface. You need similar configuration for each implementation of the event interface, although this can be split across as many Guice modules as you want. The Multibinder syntax in the figure below ensures that no matter how many implementations and modules you provide, they all end up in the same set of event listeners.

```
@Override
    public void configure() {

      /*
       * Get the listener set
       */
      Multibinder<MyEventSource.Event> myEventListeners =
 Multibinder
          .newSetBinder(binder(), MyEventSource.Event.class);
      /*
       * Add a listener
       */
      myEventListeners.addBinding().to(MyListener.class);

    }
```
Figure 117: Adding wiring

The wiring is now complete, and a call to MyEventSource.doSomething() will result in output resembling the following:

```
preDoSomething event was raised from
  object curam.mypackage.MyEventSource@125d06e
Do something!
postDoSomething event was raised from
  object curam.mypackage.MyEventSource@125d06e
```

Writing listeners for automatic persistence events

The Persistence Infrastructure provides automatically dispatched events for all entity classes. To use these events all you need to do is write event listeners and wire them using Guice, very much as described in the previous section. The event interface for persistence events differs from the previous example in that it is a parameterized abstract class called PersistenceEvent, which takes the name of the entity as a type parameter.

See the Javadoc for the PersistenceEvent class for a complete list of methods. Default empty implementations are provided for all event methods. In the example following, a listener is written which implements just the postInsert method of PersistenceEvent for an entity called MyEntity:

```
package curam.mypackage;

import com.google.inject.Singleton;
import curam.util.persistence.PersistenceEvent;

@Singleton
final class MyListener implements
    PersistenceEvent<MyEntity> {

  protected MyListener() {
    // Protected constructor for use only by Guice
  }

  @Override
  public void postInsert(final MyEntity entity) throws
      InformationalException, AppException {
    // handle the event here
  }

}
```

Figure 118: Creating a persistence event listener class

As for other events, you have to wire your listener implementation in a Guice module:

```
@Override
    public void configure() {

      /*
       * Get the listener set
       */
      Multibinder<PersistenceEvent<MyEntity>> myEventListeners =
          Multibinder.newSetBinder(binder(),
          new TypeLiteral<PersistenceEvent<MyEntity>>() { /
**/ });
      /*
       * Add a listener
       */
      myEventListeners.addBinding().to(MyListener.class);

    }
```

Figure 119: Adding wiring for persistence event listeners

Design Considerations with Events

Some things to think about when defining events or writing listeners for them:

- Like any other class or interface in Java, it is possible to create package-protected event interfaces. This allows you to use Events in your design, without making them freely available to all API clients.
- It is more efficient to implement a listener class as a singleton (either using the @Singleton Guice annotation on the class, or binding the class in singleton scope in the Guice module). Singletons need to be implemented in a thread-safe way, but even if you don't use singletons you should still assume that your listener should be thread-safe, since the safety requirements are imposed by the class which raises events. In short, unless an Event interface is documented as not requiring thread-safe listeners, you should assume thread-safety is required.
- It will rarely be appropriate for your listener methods to modify arguments passed to them. Remember that the same arguments are passed to all listeners, and that furthermore you have no control over the order in which different registered listeners will be called. Changing the contents of a listener method parameter (for instance, calling mutator methods on it) can have negative consequences and cause unexpected results or violate validation requirements. Unless an Event interface documents what can validly be changed, assume nothing can.

Backward compatibility

Previous versions of the Persistence Infrastructure provided event dispatching functionality via the EventDispatcher and StandardEventDispatcher classes. These provided similar functionality but were harder to configure. Their use is now deprecated but they are still supported for backward compatibility. The approach described in this chapter is recommended for all new event handling.

1.7 Using Entity Context

The Persistence Infrastructure allows you to add additional information to any entity instance at runtime. This facility has a number of possible uses which are described later. First we describe the facility and how to use it.

The Problem

You want to attach additional information to an entity instance at runtime, so that it is available to event handlers and other custom code. However, the entity interface itself is not easily customizable.

The Solution

Use Entity Context. Every entity instance allows you to attach additional context information. In fact, you can store a whole variety of different types of information, indexed by the class of the information.

In practice the context information is stored in a ContextContainer which is essentially a Map attached to the entity. The key of the Map is the Java Class of the stored information:

. . .

```
void someMethod(MyEntity entity) {

   // Get the string stored in the entity's context:
   String s = entity.getContextContainer().get(String.class);
   System.out.println(s);

   // Store an updated string in the entity context:
   s += " longer context";
   entity.getContextContainer().put(String.class, s);
}
```

Figure 120: Manipulating entity context

As will be clear from the code above, you can only have one String value stored at any given time in the entity context. It is up to you to make sure that you "own" any class that you use as context on an entity, and that it will not interfere with other customizations. In practice, it may be wise to define your own classes for use as context, rather than using built-in classes such as String.

What if you want to store a set or a list as context? The Java List is a built-in class, and there are no class literals for Lists of your own types, i.e. no List<MyClass>.class. You can use a TypeLiteral as a key in this case, and it will be distinct from Lists of any other type which may also be stored in the entity context:

```
...
  void someMethod(MyEntity entity) {

   // Get the List<MyClass> stored in the entity's context:
   TypeLiteral<List<MyClass>> type =
     new TypeLiteral<List<MyClass>>() { /**/ };
   List<MyClass> list = entity.getContextContainer().get(type);
   System.out.println(list);
}
```

Figure 121: Manipulating parameterized types in context

The ContextContainer class lets you retrieve, set, or remove context information by Class or by TypeLiteral. When you set the contents of the context container (using the ContextContainer.put(Class) method) the previous contents of the context container for that class, if any, are returned.

Customising Inserts using entity context

A common customization pattern is that you want to store additional information on the application database whenever a Cúram entity is inserted. In "classic" Cúram you might have extended the out-of-the-box entity but this is discouraged for code constructed using Persistence Infrastructure because of the undesirable dependencies it creates between custom code and out-of-the-box code. Instead, you'll create a whole separate entity that gets updated in synch with the original.

Let's take a typical use case. A method of a façade class is called by the Cúram client to insert data collected on a UIM page. The façade method gets data from its parameters, and invokes service layer APIs to create a new entity instance and persist it. You want to collect additional information and persist it on a new entity along with the original, using the same primary key value.

The initial steps you will take are as follows, and are the same as described for "classic" Cúram code in the Cúram Server Developer's Guide:

- customize the relevant UIM page to add new fields;
- make corresponding changes to the façade method parameters to add new attributes;
- subclass the façade and override the particular method in question;

The remaining steps are particular to code using the Persistence Infrastructure:

- define a new entity to store the additional attributes;
- store additional attributes collected in the façade as entity context;
- write a listener for insert events on the original entity (as described in the earlier chapter on Events), and have its implementation insert on the new entity using the stored entity context information;
- register the listener.

Here's how that works in practice. In order to keep the program listings concise we assume that you've already declared a new "classic" entity called MyAdditionalEntity and have extended the façade parameters to take the new details.

Here's the original façade:

```
...
public class MyFacade {
  @Inject
  protected MyEntityDAO myEntityDAO;

  public void createMyEntity(final MyEntityDetails details)
 throws
      AppException, InformationalException {
    MyEntity myEntity = myEntityDAO.newInstance();
    setDetails(myEntity, details.dtls);
    myEntity.insert();
  }

  protected void setDetails(final MyEntity e,
      final MyEntityDtls dtls)
      throws AppException, InformationalException {
    e.setFirstname(dtls.firstname);
    e.setSurname(dtls.surname);
  }

}
```

Figure 122: A façade which stores MyEntity

Here's our override of the façade:

```
...
public class MyCustomFacade extends
  curam.custom.facade.base.MyCustomFacade {

  @Override
  public void createMyEntity(final MyEntityDetails details)
      throws AppException, InformationalException {
    MyEntity myEntity = myEntityDAO.newInstance();
    setDetails(myEntity, details.dtls);

    /*
     * Store additional details in entity context
     */
```

```
  myEntity.getContextContainer().put(MyAdditionalEntityDtls.class,
        details.additionalDtls);
    myEntity.insert();
  }

}
```

Figure 123: A façade subclass which uses entity context

Here's our listener for inserts on the original entity. Note the handling when we find that no entity context has been passed. This is a design decision that must be made in each case - do we store blank additional details, or do we store nothing. If we choose to store nothing, then the application must know how to handle the situation later when we retrieve an entity and there are no additional details to be read.

Of course we know that there will always be context if the insert that is occurring was triggered via the façade we've just customized. But we always have to cater for the situation where the insert is occurring on code other than our façade.

```
@Singleton
class MyEntityListener extends PersistenceEvent<MyEntity> {
  /**
   * After MyEntity is inserted, also insert MyAdditionalEntity.
   */
  @Override
  public void postInsert(final MyEntity e) throws AppException,
      InformationalException {

    /*
     * Retrieve the stored details from entity context
     */
    MyAdditionalEntityDtls dtls = e.getContextContainer().get(
        MyAdditionalEntityDtls.class);

    /*
     * Note - don't store null details; on reads, the application
     * must handle having no additional details for a MyEntity
     * instance
     */
    if (dtls != null) {

      /*
       * Use same id as original entity
       */
      dtls.id = e.getID();

      /*
       * Insert additional details
       */
      MyAdditionalEntity additionalEntity =
        MyAdditionalEntityFactory.newInstance();

      additionalEntity.insert(dtls);
    }
  }
}
```

Figure 124: A listener for inserts on MyEntity

Here's how we register our listener:

```
public class MyModule extends AbstractModule {

  @Override
  protected void configure() {

    /*
     * Get the listener set
     */
    Multibinder<PersistenceEvent<MyEntity>> myEventListeners =
      Multibinder.newSetBinder(binder(),
        new TypeLiteral<PersistenceEvent<MyEntity>>() {/**/});

    /*
     * Add a listener
     */
    myEventListeners.addBinding().to(MyEntityListener.class);
  }
}
```

Figure 125: A Guice module to register the listener in the previous listing

In summary, what we've done is to provide a listener which receives insert events for one entity and performs inserts on another, supplemental entity. The data for the supplemental entity was piggybacked on "entity context", and will normally have been provided via a façade. However, it's important to note that this listener pattern works no matter where the insert was invoked from, although you'll find you have to decide how to handle the situation where an insert was performed but no entity context was provided.

Customising Reads using entity context

If you've customized an entity Insert to store additional information, you'll typically want to also customize the Read operation to retrieve the additional attributes. This is very much like the Insert operation in reverse. You'll do the following:

- retrieve additional attributes from entity context and return them from your subclassed façade method;
- write a listener for read events on the original entity and have its implementation read from the new entity, storing the results in entity context for use by the façade;
- register the listener.

Note that in the sample code that follows, the facade and listener classes can be the *same* classes as from our Insert example. We're just looking at different methods. By the same token, if you just have a single Listener class to handle both Insert and Read then you only have to do the Listener registration once. Here's how it all looks in practice. As before, we're assuming that you've already declared a new "classic" entity called MyAdditionalEntity and have extended the façade parameters to take the new details.

Here's the original façade:

```
...
public class MyFacade {
  @Inject
  protected MyEntityDAO myEntityDAO;

  public MyEntityDetails readMyEntity(final MyEntityKey key)
```

```
          throws AppException, InformationalException {

    MyEntityDetails details = new MyEntityDetails();
    MyEntity myEntity = myEntityDAO.get(key.id);
    getDetails(myEntity, details.dtls);
    return details;
  }

  protected void getDetails(final MyEntity myEntity,
      final MyEntityDtls dtls)
      throws AppException, InformationalException {
    dtls.firstname = myEntity.getFirstname();
    dtls.surname = myEntity.getSurname();
  }

}
```

Figure 126: A façade which reads MyEntity

Here's our override of the façade:

```
...
public class MyCustomFacade extends
  curam.custom.facade.impl.MyFacade {

  @Override
  public MyEntityDetails readMyEntity(final MyEntityKey key)
      throws AppException, InformationalException {
    MyEntityDetails details = new MyEntityDetails();
    MyEntity myEntity = myEntityDAO.get(key.id);
    getDetails(myEntity, details.dtls);

    /*
     * Retrieve additional details from entity context
     */
    details.additionalDtls = myEntity.getContextContainer().get(
        MyAdditionalEntityDtls.class);

    return details;
  }

}
```

Figure 127: A façade subclass which uses entity context

Here's our listener for reads on the original entity. Note, we're assuming that there will always be a corresponding record on the new entity. Your design may have to cater for the situation where this is not the case.

```
@Singleton
class MyEntityListener extends PersistenceEvent<MyEntity> {

  /**
   * After MyEntity is read, also read MyAdditionalEntity.
   */
  @Override
  public void postRead(final MyEntity e) throws AppException,
      InformationalException {
    /*
```

```
      * Read additional details from database
      */
    MyAdditionalEntity additionalEntity =
 MyAdditionalEntityFactory
        .newInstance();
    MyAdditionalEntityKey key = new MyAdditionalEntityKey();
    key.id = e.getID();
    MyAdditionalEntityDtls dtls = additionalEntity.read(key);
    /*
      * Store additional details in entity context
      */
    e.getContextContainer().put(MyAdditionalEntityDtls.class,
 dtls);
  }
}
```

Figure 128: A listener for reads on MyEntity

Here's how we register our listener. Note that if you combined the listener from the Insert example and the Read example into a single listener class, you won't need this step. You only register each listener class once:

```
public class MyModule extends AbstractModule {

  @Override
  protected void configure() {

    /*
      * Get the listener set
      */
    Multibinder<PersistenceEvent<MyEntity>> myEventListeners =
      Multibinder.newSetBinder(binder(),
        new TypeLiteral<PersistenceEvent<MyEntity>>() {/**/});

    /*
      * Add a listener
      */
    myEventListeners.addBinding().to(MyEntityListener.class);
  }
}
```

Figure 129: A Guice module to register the listener in the previous listing

In summary, we've created a listener which receives read events for one entity and performs reads on another, supplemental entity. The data from the supplemental entity is piggybacked on "entity context", and is available to a façade method which returns the details to a client.

Customising other operations using entity context

We've shown how to customize entity Insert and Read operations to handle additional data. It is just as easy to handle additional data with other operation types using very similar approaches.

For modifications on an entity, perform the same façade-level customizations, and handle the PersistenceEvent.postModify(ENTITY) event.

There are also persistence events for readmulti, remove, and cancel operations.

1.8 State Transitions

The Persistence Infrastructure provides support for implementing entities which are state machines. These entities each have their own "lifecycle", and the state of a particular entity instance is held in a database column.

Typically, the state of an entity instance may be *retrieved*, but *changes* to the state must be controlled through specialized methods.

This chapter explains how to implement an entity which has a state-based lifecycle.

The problem

Let's take an example: you analyze requirements to determine that your entity should support the following state transitions:
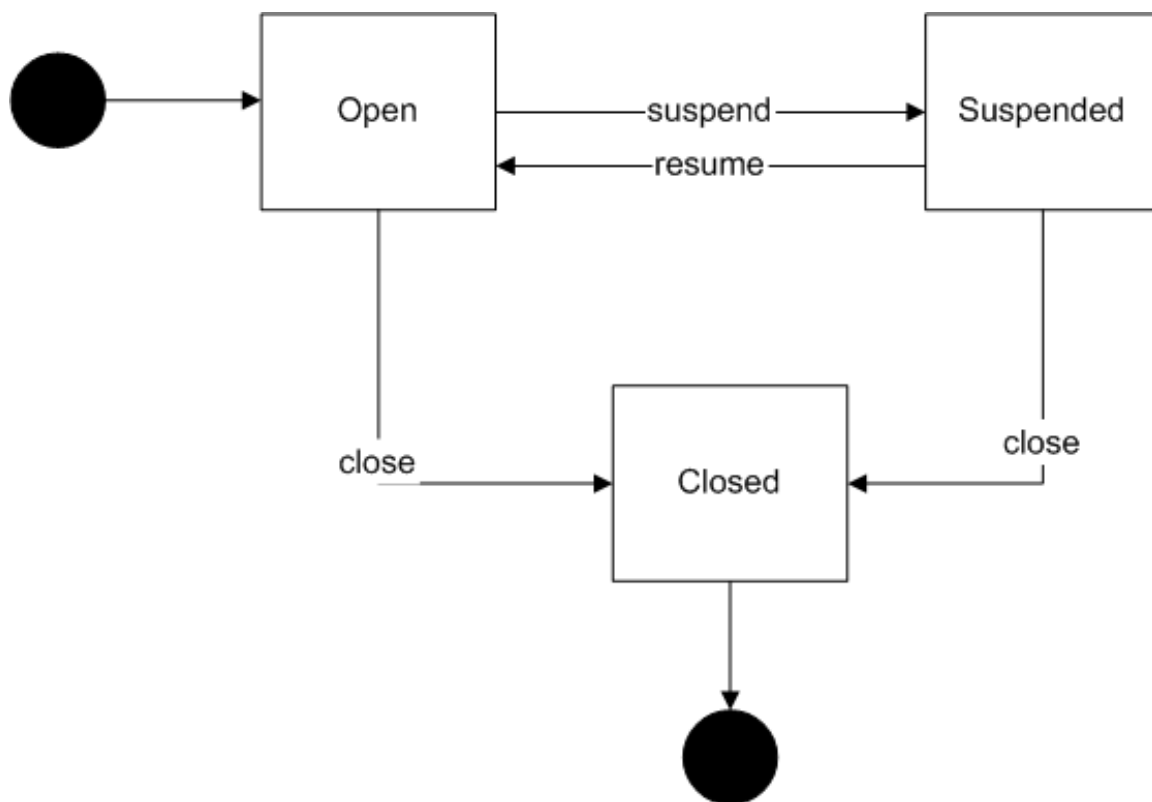

Figure 130: State transition diagram for the example cookbook code

Moreover, each of these transitions has its unique validation, data manipulation and notification requirements.

The solution

You must follow these steps to implement state transitions using the Persistence Infrastructure helper classes:

- Specify states;
- Specify storage mechanism for the state value;
- Identify transition methods;
- Implement getLifecycleState;
- Create a map to hold the permitted states;
- Create an object for each state;
- Create an object for each permitted transition;
- Create a private getter to retrieve the current State;
- Create a private setter to set the current State;
- Create a private helper method to perform a state transition;
- Implement state transition methods;
- Specify the initial state;
- Add state transition validation logic; and
- Override the modify method (if required).

Specify states

Firstly you must identify the possible states of your entity. The possible states are:

- open;
- suspended; and
- closed.

Specify storage mechanism for the state value

Your entity must store its state in some form. A typical storage mechanism is to enumerate the states in a codetable and store the code's String value on a database column.

In this example, you'll enumerate these states in a new codetable called MYLIFECYCLEENTITYSTATE, and present the value as an instance of the generated MYLIFECYCLEENTITYSTATEEntry class.

Create the codetable:

```
<?xml version="1.0" encoding="UTF-8"?>
<codetables package="curam.mypackage.codetable">
  <codetable
    java_identifier="MYLIFECYCLEENTITYSTATE"
    name="MYLIFECYCLEENTITYSTATE"
  >
    <code
      default="false"
      java_identifier="OPEN"
      status="ENABLED"
      value="OPEN"
    >
      <locale
        language="en"
        sort_order="0"
      >
```

```
            <description>Open</description>
            <annotation/>
          </locale>
        </code>
        <code
          default="false"
          java_identifier="SUSPENDED"
          status="ENABLED"
          value="SUSPENDED"
        >
          <locale
            language="en"
            sort_order="0"
          >
            <description>Suspended</description>
            <annotation/>
          </locale>
        </code>
        <code
          default="false"
          java_identifier="CLOSED"
          status="ENABLED"
          value="CLOSED"
        >
          <locale
            language="en"
            sort_order="0"
          >
            <description>Closed</description>
            <annotation/>
          </locale>
        </code>
      </codetable>
    </codetables>
```

Figure 131: Creating a code table file listing the states of an entity

Now mark your entity's interface to extend the Lifecycle interface, parameterized with the data type used to present the state (in this case, MYLIFECYCLEENTITYSTATEEntry):

```
/**
 * Description of my state-machine entity.
 */
@ImplementedBy(MyLifecycleEntityImpl.class)
public interface MyLifecycleEntity extends StandardEntity,
    Lifecycle<MYLIFECYCLEENTITYSTATEEntry>
```

Figure 132: Extending the Lifecycle interface

Identify transition methods

Typically an entity must carefully control its transitions between states. As such, it is often better to create specialized methods for state transitions rather than expose a setState method. Typically the name of each specialized method will reflect the state being transitioned *to*.

Since a state-transition method will modify the entity's data on the database, each such method should take the entity's version number (assuming that the entity supports optimistic locking). Each specialized method is free to specify additional arguments which may be required, e.g.:

- suspend (taking a suspension reason);
- resume (no arguments); and
- close (taking the end date of the entity).

Suspend

```
/**
   * Suspends business pending investigation.
   *
   * Transitions the state to
   * {@linkplain MYLIFECYCLEENTITYSTATEEntry#SUSPENDED}, if it is
   * valid to suspend.
   *
   * @param reason
   *          the reason for suspension
   *
   * @param versionNo
   *          the version number as previously retrieved
   *
   * @throws InformationalException
   *          if the entity is not in a valid state to
transition
   *          to
   *          {@linkplain
MYLIFECYCLEENTITYSTATEEntry#SUSPENDED},
   *          or if any other validation errors are found
   */
      public void suspend(final String reason, final int
versionNo)
      throws InformationalException;
```

Figure 133: Interface declaration of a "suspend" state transition method

Resume

```
/**
   * Resumes business following a suspension investigation
resulting
   * in acquittal.
   *
   * Transitions the state to
   * {@linkplain MYLIFECYCLEENTITYSTATEEntry#OPEN}, if it is
valid
   * to resume business.
   *
   * @param versionNo
   *          the version number as previously retrieved
   *
   * @throws InformationalException
   *          if the entity is not in a valid state to
transition
   *          to {@linkplain MYLIFECYCLEENTITYSTATEEntry#OPEN},
or
   *          if any other validation errors are found
   */
      public void resume(final int versionNo)
```

```
                   throws InformationalException;
```
Figure 134: Interface declaration of a "resume" state transition method

Close

```
/**
   * Ceases business with the agency.
   *
   * Transitions the state to
   * {@linkplain MYLIFECYCLEENTITYSTATEEntry#CLOSED}, if it is
   * valid to cease conducting business.
   *
   * @param endDate
   *          the date on which business with the agency was
 ceased
   *
   * @param versionNo
   *          the version number as previously retrieved
   *
   * @throws InformationalException
   *          if the entity is not in a valid state to
 transition
   *          to {@linkplain
 MYLIFECYCLEENTITYSTATEEntry#CLOSED},
   *          or if any other validation errors are found
   */
      public void close(final Date endDate, final int versionNo)
      throws InformationalException;
```
Figure 135: Interface declaration of a "close" state transition method

Implementations of these methods are free to perform method-specific validations and notifications, e.g. whenever suspend is called, to notify an investigations worker to launch an investigation.

Note that this approach of having specialized methods (e.g. controlling the setting of state and endDate through the close method) is far "cleaner" than an alternative approach of allowing a public setter methods for setEndDate and setState and having complex validation to ensure that whenever the state is modified (by calling code), that the endDate is set.

Implement getLifecycleState

You must implemented a getLifecycleState method, as required by the Lifecycle interface:

```
/**
   * {@inheritDoc}
   */
  public MYLIFECYCLEENTITYSTATEEntry getLifecycleState() {
    return MYLIFECYCLEENTITYSTATEEntry.get(getDtls().state);
  }
```
Figure 136: Implementing getLifecycleState

Create a map to hold the permitted states

Each state will be represented by an instance of the State helper class.

You must create a map to hold your entity's State instances:

```
/**
   * A map of the states for this entity
   */
  private final Map<MYLIFECYCLEENTITYSTATEEntry,
    State<MYLIFECYCLEENTITYSTATEEntry>> states =
    new HashMap<MYLIFECYCLEENTITYSTATEEntry,
    State<MYLIFECYCLEENTITYSTATEEntry>>();
```

Figure 137: A map of permitted states

Create an object for each state

Each permitted state for your class is represented by an instance of the State> helper class. Here you'll use the CodetableState> helper class:

```
/**
   * Actively conducting business with the agency.
   */
  private final State<MYLIFECYCLEENTITYSTATEEntry> OPEN =
    new CodetableState<MYLIFECYCLEENTITYSTATEEntry>(
      states, MYLIFECYCLEENTITYSTATEEntry.OPEN, true, true) {
  };


  /**
   * Business has been suspended pending investigation.
   */
  private final State<MYLIFECYCLEENTITYSTATEEntry> SUSPENDED =
    new CodetableState<MYLIFECYCLEENTITYSTATEEntry>(
      states, MYLIFECYCLEENTITYSTATEEntry.SUSPENDED, true, false)
{
  };


  /**
   * No longer conducting business with the agency.
   */
  private final State<MYLIFECYCLEENTITYSTATEEntry> CLOSED =
    new CodetableState<MYLIFECYCLEENTITYSTATEEntry>(
      states, MYLIFECYCLEENTITYSTATEEntry.CLOSED, false, false) {
  };
```

Figure 138: Creating an object for each permitted state

Each State object is an anonymous class, constructed with:

1. the map to which the object will be added (states);
2. the value used to identify the state object in the map (typically, the code table entry value);
3. whether the entity may be modified when in this state; and
4. whether the entity may be removed when in this state.

> **Note:** There is no automatic processing surrounding the use of the "entity may be modified/ removed" values.
>
> If you require to prevent modifications or removals when your entity is in a particular state, you must override the modify and/or remove methods as appropriate, and in them put validation logic which may make use of calls to State.isModifyAllowed or State.isRemoveAllowed as appropriate.
>
> See below.

Create an object for each permitted transition

Each permitted transition between states is represented by an instance of the Transition helper class.

From the state-transition diagram, you can see that the following transitions are required:

*   from *open* to *closed*;
*   from *open* to *suspended*;
*   from *suspended* back to *open*; and
*   from *suspended* to *closed*.

```
private final Transition<MYLIFECYCLEENTITYSTATEEntry>
    OPEN2CLOSED =
    new Transition<MYLIFECYCLEENTITYSTATEEntry>(
      OPEN, CLOSED) {
  };


  private final Transition<MYLIFECYCLEENTITYSTATEEntry>
    OPEN2SUSPENDED =
    new Transition<MYLIFECYCLEENTITYSTATEEntry>(
      OPEN, SUSPENDED) {
  };


  private final Transition<MYLIFECYCLEENTITYSTATEEntry>
    SUSPENDED2OPEN =
    new Transition<MYLIFECYCLEENTITYSTATEEntry>(
      SUSPENDED, OPEN) {
  };


  private final Transition<MYLIFECYCLEENTITYSTATEEntry>
    SUSPENDED2CLOSED =
    new Transition<MYLIFECYCLEENTITYSTATEEntry>(
      SUSPENDED, CLOSED) {
  };
```

Figure 139: Creating an object for each permitted transition

Each Transition object is an anonymous class, constructed with:

1. the State being exited (i.e. transitioned from); and
2. the State being entered (i.e. transitioned to).

> **Note:** Specifying the set of permitted transitions is typically more straightforward than crafting logic to prevent unsupported transitions from occurring.
>
> You do not need to specify a transition to the initial state - the initial state will be specified in setNewInstanceDefaults (see below).

Create a private getter to retrieve the current State

This method retrieves the State object representing the entity's current state. Note that the method is private, as the State object is not exposed outside of the entity - callers which require to know the entity's state must use getLifecycleState instead.

The relevant State object is retrieved by looking it up in the map of states.

```
/**
     * @return The State object representing the current state
 of
     *         this entity
     */
    private State<MYLIFECYCLEENTITYSTATEEntry> getState() {
     return states.get(getLifecycleState());
     }
```

Figure 140: Creating a private getter to retrieve the current State

Create a private setter to set the current State

This method sets the entity's state value from a State object. Note that the method is private.

```
/**
     * Sets the state codetable code field from the State
 object
     * supplied.
     *
     * @param value
     *         the State supplied
     */
    private void setState(
    final State<MYLIFECYCLEENTITYSTATEEntry> state) {
     getDtls().state = state.getValue().getCode();
     }
```

Figure 141: Creating a private setter to set the current State

Create a private helper method to perform a state transition

You must create a helper method which performs the state transition.

```
/**
     * Transitions this entity to the new state specified.
     *
     * @param newState
     *         the state to transition to
     * @param versionNo
     *         the version number of this entity as previously
     *         retrieved
     * @throws InformationalException
     *          if validation errors occur during the
 transition
```

```
       */
     private void transitionTo(
       final State<MYLIFECYCLEENTITYSTATEEntry> newState,
       final Integer versionNo) throws InformationalException {

       // get the current state of this entity
       final State<MYLIFECYCLEENTITYSTATEEntry> oldState =
         getState();

       // set the field which stores the state value
       setState(newState);

       // validate the state transition
       oldState.transitionTo(newState);

       // update the database, bypassing any pre-modify
 validation
       // in this class
       super.modify(versionNo);
       }
```

Figure 142: Creating a private helper method to perform a state transition

Points to note:

- the validation of whether the transition is permitted is performed by the State.transitionTo method (i.e. in the line oldState.transitionTo(newState); in the figure above). See below for how to add your own validation logic; and
- your entity may have overridden the modify method to add validation to be applied when calling code invokes modify - often this logic is inappropriate to state transitions, and so typically the storage of a state change is accomplished by a call to super.modify (as shown in the figure above) rather then this.modify.

Implement state transition methods

Now you can code the implementations of your specialized state transition methods:

```
/**
   * {@inheritDoc}
   */
  public void close(Date endDate, int versionNo)
    throws InformationalException {
    // store the date of closure
    setEndDate(endDate);

    // transition to "closed"
    transitionTo(CLOSED, versionNo);
  }

  /**
   * {@inheritDoc}
   */
 public void resume(int versionNo) throws InformationalException
 {
    // blank the suspension reason
    setSuspensionReason(null);

    // transition to "open"
```

```
    transitionTo(OPEN, versionNo);
}

/**
 * {@inheritDoc}
 */
public void suspend(String reason, int versionNo)
    throws InformationalException {
  // store the suspension reason
  setSuspensionReason(reason);

  // transition to "suspended"
  transitionTo(SUSPENDED, versionNo);
}
```

Figure 143: Implementing state transition methods

These methods are publicly visible and callable through the entity's interface. Note that in the figure above, additional setter methods (setEndDate and setSuspensionReason) are assumed.

Specify the initial state

You must specify the initial state for new instances of your entity:

```
/**
     * Defaults the state to
     * {@linkplain MYLIFECYCLEENTITYSTATEEntry#OPEN}.
     */
  public void setNewInstanceDefaults() {
  setState(OPEN);
  }
```

Figure 144: Specifying the initial state

> **Note:** If you find that new instances have a *number* of possible initial states, then consider whether:
>
> • calling code should be responsible for creating a new instance of your entity with a default state, and then immediately transitioning it to the required state; or
> • you are trying to force logically different concepts to be stored on the same physical entity, and perhaps should instead consider using inheritance/polymorphism to separate out different behavior.

Add state transition validation logic

The CodetableState and Transition helper classes provide the following standard validation to disallow any transition which is not explicitly specified. For example, the state transition diagram does not permit an entity instance in the closed state to transition to any other state; by default, any attempts to.suspend() an entity instance which is currently closed will result in this error being raised: Cannot transition from 'Closed' to 'Suspended'.

You can add logic (typically to perform validations and/or notifications) to the following places:

• in your State objects:

  • onEnter - this method is called whenever a transition occurs which attempts to enter this State;

- onLeave - this method is called whenever a transition occurs which attempts to leave this State; and

- onUnsupportedTransitionFrom - this method is called whenever an unsupported transition is attempted which attempts to transition to this State from the one specified; by default, the CodetableState helper class raises a default message, but you are free to provider your own validation/notification logic; and

- in your Transition objects:

  - onTransition - this method is called whenever this Transition occurs.

See the Javadoc for the State, CodetableState and Transition helper classes for more information.

For example, if you want your logic to send an email whenever your entity is closed (regardless of whether it was previously open or suspended), override the onEnter method of your CLOSED state:

```
/**
     * No longer conducting business with the agency.
     */
   private final State<MYLIFECYCLEENTITYSTATEEntry> CLOSED =
     new CodetableState<MYLIFECYCLEENTITYSTATEEntry>(
     states, MYLIFECYCLEENTITYSTATEEntry.CLOSED, false, false) {

     @Override
     protected void onEnter() {
     // whenever the entity is closed, send an email
     sendClosureEmail();
     }

     };
```

Figure 145: Adding state transition validation logic

Override the modify method (if required)

If you require logic to prevent modifications to the entity if it is in an inappropriate state, then you must override your entity's modify method:

```
/**
   * {@inheritDoc}
   */
  @Override
  public void modify(Integer versionNo)
    throws InformationalException {

    if (!getState().isModifyAllowed()) {
      ValidationHelper
          .addValidationError(
"You are not allowed to modify this record when it is in this
 state"
);
    }

    super.modify(versionNo);
  }
```

Figure 146: Overriding the modify method

> **Note:** Only explicit calls to your entity's modify method (e.g. through its interface) will hit this logic - state transitions will typically call super.modify directly and thus bypass this logic.

Putting it all together

Here are full listings of the entity interface and implementation example used in this chapter:

```
package curam.mypackage;

import com.google.inject.ImplementedBy;

import
 curam.mypackage.codetable.impl.MYLIFECYCLEENTITYSTATEEntry;
import curam.util.exception.InformationalException;
import curam.util.persistence.Insertable;
import curam.util.persistence.OptimisticLockModifiable;
import curam.util.persistence.StandardEntity;
import curam.util.persistence.helper.Lifecycle;
import curam.util.type.Date;
import curam.util.type.DateRanged;

/**
 * Description of my state-machine entity.
 */
@ImplementedBy(MyLifecycleEntityImpl.class)
public interface MyLifecycleEntity extends StandardEntity,
    DateRanged, Lifecycle<MYLIFECYCLEENTITYSTATEEntry>,
 Insertable,
    OptimisticLockModifiable {

  /**
   * Suspends business pending investigation.
   *
   * Transitions the state to
   * {@linkplain MYLIFECYCLEENTITYSTATEEntry#SUSPENDED}, if it is
   * valid to suspend.
   *
   * @param reason
   *          the reason for suspension
   *
   * @param versionNo
   *          the version number as previously retrieved
   *
   * @throws InformationalException
   *          if the entity is not in a valid state to
 transition
   *          to
   *          {@linkplain
 MYLIFECYCLEENTITYSTATEEntry#SUSPENDED},
   *          or if any other validation errors are found
   */
  public void suspend(final String reason, final int versionNo)
      throws InformationalException;

  /**
```

```
     * Resumes business following a suspension investigation
   resulting
     * in acquittal.
     *
     * Transitions the state to
     * {@linkplain MYLIFECYCLEENTITYSTATEEntry#OPEN}, if it is
   valid
     * to resume business.
     *
     * @param versionNo
     *           the version number as previously retrieved
     *
     * @throws InformationalException
     *           if the entity is not in a valid state to
   transition
     *           to {@linkplain MYLIFECYCLEENTITYSTATEEntry#OPEN},
   or
     *           if any other validation errors are found
     */
    public void resume(final int versionNo)
        throws InformationalException;

    /**
     * Ceases business with the agency.
     *
     * Transitions the state to
     * {@linkplain MYLIFECYCLEENTITYSTATEEntry#CLOSED}, if it is
     * valid to cease conducting business.
     *
     * @param endDate
     *           the date on which business with the agency was
   ceased
     *
     * @param versionNo
     *           the version number as previously retrieved
     *
     * @throws InformationalException
     *           if the entity is not in a valid state to
   transition
     *           to {@linkplain
   MYLIFECYCLEENTITYSTATEEntry#CLOSED},
     *           or if any other validation errors are found
     */
    public void close(final Date endDate, final int versionNo)
        throws InformationalException;

}
```

Figure 147: Lifecycle entity interface example

```
package curam.mypackage;

import java.util.HashMap;
import java.util.Map;

import curam.mypackage.codetable.MYLIFECYCLEENTITYSTATEEntry;
import curam.mypackage.struct.MyLifecycleEntityDtls;
import curam.util.exception.InformationalException;
```

```java
import curam.util.exception.UnimplementedException;
import curam.util.persistence.ValidationHelper;
import curam.util.persistence.helper.CodetableState;
import curam.util.persistence.helper.SingleTableEntityImpl;
import curam.util.persistence.helper.State;
import curam.util.persistence.helper.Transition;
import curam.util.type.Date;
import curam.util.type.DateRange;

/**
 * Standard implementation of {@linkplain MyLifecycleEntity}.
 */
public class MyLifecycleEntityImpl extends
    SingleTableEntityImpl<MyLifecycleEntityDtls> implements
    MyLifecycleEntity {

  protected MyLifecycleEntityImpl() {
    /*
     * Protected no-arg constructor for use only by Guice
     */
  }

  /*
   * Persistence methods
   */
  /**
   * {@inheritDoc}
   */
  @Override
  public void modify(Integer versionNo)
      throws InformationalException {

    if (!getState().isModifyAllowed()) {
      ValidationHelper
          .addValidationError(
"You are not allowed to modify this record when it is in this
 state"
          );
    }

    super.modify(versionNo);
  }

  /*
   * Getters
   */
  /**
   * {@inheritDoc}
   */
  public MYLIFECYCLEENTITYSTATEEntry getLifecycleState() {
    return MYLIFECYCLEENTITYSTATEEntry.get(getDtls().state);
  }

  public DateRange getDateRange() {
    throw new UnimplementedException();
  }

  /*
```

```
 * Setters
 */
private void setEndDate(final Date value) {
  throw new UnimplementedException();
}

private void setSuspensionReason(final String value) {
  throw new UnimplementedException();
}

void sendClosureEmail() {
  throw new UnimplementedException();
}

/*
 * State transitions
 */

/**
 * {@inheritDoc}
 */
public void close(Date endDate, int versionNo)
    throws InformationalException {
  // store the date of closure
  setEndDate(endDate);

  // transition to "closed"
  transitionTo(CLOSED, versionNo);
}

/**
 * {@inheritDoc}
 */
public void resume(int versionNo) throws InformationalException
{
  // blank the suspension reason
  setSuspensionReason(null);

  // transition to "open"
  transitionTo(OPEN, versionNo);
}

/**
 * {@inheritDoc}
 */
public void suspend(String reason, int versionNo)
    throws InformationalException {
  // store the suspension reason
  setSuspensionReason(reason);

  // transition to "suspended"
  transitionTo(SUSPENDED, versionNo);
}

/*
 * State Transitions
 */
```

```java
  /**
   * A map of the states for this entity
   */
 private final Map<MYLIFECYCLEENTITYSTATEEntry,
    State<MYLIFECYCLEENTITYSTATEEntry>>
    states =
      new HashMap<MYLIFECYCLEENTITYSTATEEntry,
      State<MYLIFECYCLEENTITYSTATEEntry>>();

  /**
   * @return The State object representing the current state of
this
   *         entity
   */
 private State<MYLIFECYCLEENTITYSTATEEntry> getState() {
    return states.get(getLifecycleState());
 }

  /**
   * Sets the state codetable code field from the State object
   * supplied.
   *
   * @param value
   *          the State supplied
   */
 private void setState(
      final State<MYLIFECYCLEENTITYSTATEEntry> state) {
    getDtls().state = state.getValue().getCode();
 }

  /**
   * Transitions this entity to the new state specified.
   *
   * @param newState
   *          the state to transition to
   * @param versionNo
   *          the version number of this entity as previously
   *          retrieved
   * @throws InformationalException
   *           if validation errors occur during the transition
   */
 private void transitionTo(
      final State<MYLIFECYCLEENTITYSTATEEntry> newState,
      final Integer versionNo) throws InformationalException {

    // get the current state of this entity
    final State<MYLIFECYCLEENTITYSTATEEntry> oldState =
getState();

    // set the field which stores the state value
    setState(newState);

    // validate the state transition
    oldState.transitionTo(newState);

    // update the database, bypassing any pre-modify validation
in
    // this class
```

```
            super.modify(versionNo);
          }

        /**
         * Actively conducting business with the agency.
         */
        private final State<MYLIFECYCLEENTITYSTATEEntry> OPEN =
            new CodetableState<MYLIFECYCLEENTITYSTATEEntry>(states,
              MYLIFECYCLEENTITYSTATEEntry.OPEN, true, true) {
            };

        /**
         * Business has been suspended pending investigation.
         */
        private final State<MYLIFECYCLEENTITYSTATEEntry> SUSPENDED =
            new CodetableState<MYLIFECYCLEENTITYSTATEEntry>(states,
              MYLIFECYCLEENTITYSTATEEntry.SUSPENDED, true, false) {
            };

        /**
         * No longer conducting business with the agency.
         */
        private final State<MYLIFECYCLEENTITYSTATEEntry> CLOSED =
            new CodetableState<MYLIFECYCLEENTITYSTATEEntry>(states,
              MYLIFECYCLEENTITYSTATEEntry.CLOSED, false, false) {

              @Override
              protected void onEnter() {
                // whenever the entity is closed, send an email
                sendClosureEmail();
              }

            };

        private final Transition<MYLIFECYCLEENTITYSTATEEntry>
          OPEN2CLOSED =
            new Transition<MYLIFECYCLEENTITYSTATEEntry>(OPEN, CLOSED) {
            };

        private final Transition<MYLIFECYCLEENTITYSTATEEntry>
          OPEN2SUSPENDED =
            new Transition<MYLIFECYCLEENTITYSTATEEntry>(OPEN,
    SUSPENDED) {
            };

        private final Transition<MYLIFECYCLEENTITYSTATEEntry>
          SUSPENDED2OPEN =
            new Transition<MYLIFECYCLEENTITYSTATEEntry>(SUSPENDED,
    OPEN) {
            };

        private final Transition<MYLIFECYCLEENTITYSTATEEntry>
          SUSPENDED2CLOSED =
            new Transition<MYLIFECYCLEENTITYSTATEEntry>(SUSPENDED,
    CLOSED) {
            };

        /*
```

```
 * Validation
 */
public void mandatoryFieldValidation() {
  throw new UnimplementedException();
}

public void crossFieldValidation() {
  throw new UnimplementedException();
}

public void crossEntityValidation() {
  throw new UnimplementedException();
}

/**
 * Defaults the state to
 * {@linkplain MYLIFECYCLEENTITYSTATEEntry#OPEN}.
 */
public void setNewInstanceDefaults() {
  setState(OPEN);
}

}
```

Figure 148: Lifecycle entity implementation example

1.9 Inheritance

The Persistence Infrastructure includes support for simple inheritance. This support allows you to:

• specify that an entity interface extends another entity interface; and
• allows you to store the data held (in your base and concrete entity classes) in a number of different ways.

Identifying inheritance

If you are lucky, you will be able to directly identify concepts in your requirements which fall into a natural inheritance hierarchy. Requirements which mention phrases like "X is a Y" / "X is a kind of Y" / "X is a type of Y" are likely candidates for inheritance.

Often, though, you may only discover an inheritance hierarchy during implementation, and you should refactor accordingly. Tell-tale signs include:

• one or more methods whose behavior differs depending on the "type" of the entity instance;
• a need to link each row on a database table (A) to exactly one of *either*:

  • a row on table B; *or*
  • a row on table C (but not both);

• a need to pass around lists of entity instances, which may be made up of instances of entities of more than one type.

It is often a good idea to look out for refactoring opportunities during implementation to take advantage of appropriate object-oriented design techniques.

Entity interface inheritance

Let's take a simple example: You require to store information about Cats and Dogs. You identify that Cats and Dogs have a number of behaviors in common, and so you identify a common Animal interface.

You need to code three interfaces, Cat, Dog and Animal with the Cat and Dog interfaces both extending the Animal interface.

```
package curam.inheritance;

import curam.util.persistence.Insertable;
import curam.util.persistence.OptimisticLockModifiable;
import curam.util.persistence.StandardEntity;
import curam.util.persistence.helper.Named;

public interface Animal extends StandardEntity, Insertable,
    OptimisticLockModifiable, Named {

  public void speak();

}
```
Figure 149: The Animal Interface

```
package curam.inheritance;

public interface Cat extends Animal {

  public int getNumberOfLivesRemaining();

  public void setNumberOfLivesRemaining(final int value);

}
```
Figure 150: The Cat Interface

```
package curam.inheritance;

public interface Dog extends Animal {
  public String getFavouriteTrick();

  public void setFavouriteTrick(final String value);

}
```
Figure 151: The Dog Interface

DAO interfaces

You require to:

- create new Cat instances;
- retrieve a Cat, based on its ID;
- create new Dog instances;
- retrieve a Dog, based on its ID; and
- retrieve a generic Animal, based on its ID (and receive a concrete Cat or Dog instance as appropriate).

The creation and retrieval of Cat and Dog instances is straightforward - create DAO interfaces for Cats and Dogs (you can also include other retrievals too):

```
package curam.inheritance;

import java.util.Set;

import curam.util.persistence.StandardDAO;

public interface CatDAO extends StandardDAO<Cat> {

  public Set<Cat> readAllCats();

}
```

Figure 152: The DAO interface for Cat

```
package curam.inheritance;

import java.util.Set;

import curam.util.persistence.StandardDAO;

public interface DogDAO extends StandardDAO<Dog> {
  public Set<Dog> readAllDogs();

}
```

Figure 153: The DAO interface for Dog

The DAO interface for Animal is slightly different in that callers can *retrieve* a generic Animal based on its ID (and the implementation will be responsible for creating a Cat or Dog object as appropriate), but callers cannot *create* an Animal (all creations must create either a concrete Cat or a concrete Dog).

Use the ReaderDAO interface instead of StandardDAO:

```
package curam.inheritance;

import java.util.Set;

import curam.util.persistence.ReaderDAO;

public interface AnimalDAO extends ReaderDAO<Long, Animal> {

  public Set<Animal> readAllAnimals();

}
```

Figure 154: The read-only DAO interface for Animal

> **Note:** Unlike the Animal/Cat/Dog interfaces, the DAO interfaces for Animal/Cat/Dog do *not* form an inheritance hierarchy.

Deciding on database storage

The Persistence Infrastructure has support for the following data storage options:

- one table per class;
- one table per concrete class; and
- one table for the whole hierarchy.

These options are described in more detail below.

The option you choose will depend on a number of factors:

- the amount of commonality or disparity between the data storage requirements for your classes;
- data retrieval requirements; and
- volumetric and performance concerns.

One table per class

If you choose this option, you will create one physical database table per class (whether abstract or concrete) in your hierarchy.

This option makes use of a *discriminator value* in the form of the attribute Animal.animalType. This attribute stores a String value which will allow the implementation to determine whether a particular Animal is a Cat or a Dog without further reads. This data is denormalized (it can be determined by attempting to read rows on the Cat and Dog tables and seeing which one succeeds), however processing is greatly simplified and performance increase by the use of a discriminator.

This option also assumes that all the tables in the hierarchy share the same key value (animalID). It is possible (though very unwieldy) to allow different key values on the tables; for this example assume that the primary key value of an abstract Animal row is the same as its corresponding concrete Cat or Dog row.

You must provide the following implementation classes (listed in dependency order):

- AnimalImpl;
- CatImpl;
- DogImpl;
- CatDAOImpl;
- DogDAOImpl; and
- AnimalDAOImpl.

These classes are described in detail below. The concrete (Cat and Dog) implementation classes are reasonably straightforward, but the abstract (Animal) classes are more complex.

AnimalImpl

```
package curam.inheritance;
import curam.inheritance.Animal;
import curam.inheritance.struct.AnimalDtls;
import curam.util.persistence.EntityAdapter;
import curam.util.persistence.helper.BasePlusConcreteTableImpl;
import curam.util.type.DeepCloneable;

abstract class AnimalImpl<CONCRETE_ENTITY extends Animal,
  CONCRETE_CLASS_DTLS_STRUCT extends DeepCloneable> extends
    BasePlusConcreteTableImpl<Long, CONCRETE_ENTITY,
    AnimalDtls, CONCRETE_CLASS_DTLS_STRUCT>
    implements Animal {
```

```
  protected AnimalImpl() {
  }

  @Override
  protected void setDiscriminator(final String value) {
    setAnimalType(value);
  }

  @Override
  protected EntityAdapter<Long, AnimalDtls>
 getBaseEntityAdapter() {
    return new AnimalAdapter();
  }

  public String getName() {
    return getBaseRowDtls().name;
  }

  public void setName(final String value) {
    getBaseRowDtls().name = value;
  }

  protected void setAnimalType(final String value) {
    getBaseRowDtls().animalType = value;
  }
}
```

Figure 155: One table per class - implementation of abstract base class

There are a number of important features of this implementation which are explained below.

Class declaration

```
abstract class AnimalImpl<CONCRETE_ENTITY extends Animal,
  CONCRETE_CLASS_DTLS_STRUCT extends DeepCloneable> extends
  BasePlusConcreteTableImpl<Long, CONCRETE_ENTITY, AnimalDtls,
    CONCRETE_CLASS_DTLS_STRUCT>
```

The implementation class extends the helper class BasePlusConcreteTableImpl, which provides support for simple two-level class hierarchies (such as the one in this example).

BasePlusConcreteTableImpl is parameterized with the key type, the concrete entity interface and the Dtls structs used to store the abstract and base rows. AnimalImpl can directly supply two of these parameters (namely Long and AnimalDtls), but the name of the concrete interface and Dtls struct must be specified by the subclass implementations, and so the Animal class takes these types as parameters.

The class is package-protected and marked abstract. In this example the subclasses will be placed in the same code-package; if you require some of your subclasses to be in a different package, you will need to mark your abstract implementation class public.

The class implements the Animal interface; note that the class implements only *some* of the methods required by the interface, leaving others to the subclass implementation, e.g:

- AnimalImpl provides an implementation for getName and setName, as the behavior is identical for all Animal instances; but
- AnimalImpl does not provide an implementation for speak, as the behavior will differ between Cat and Dog instances.

Protected constructor

```
protected AnimalImpl() {
}
```

Store discriminator value

```
@Override
  protected void setDiscriminator(final String value) {
    setAnimalType(value);
  }
```

The class must override the BasePlusConcreteTableImpl.setDiscriminator method to store the discriminator in an appropriate column (in this example the animalType column). A protected setter is used to set the column value.

Base entity adapter

```
@Override
  protected EntityAdapter<Long, AnimalDtls>
 getBaseEntityAdapter() {
    return new AnimalAdapter();
  }
```

The class must override the BasePlusConcreteTableImpl.getBaseEntityAdapter method to provider an entity adapter for retrieving and storing the database row for the base class.

Getters and Setters

The getters and setters make use of the BasePlusConcreteTableImpl.getBaseRowDtls to retrieve the Dtls struct for the base row (in this example an AnimalDtls struct).

CatImpl

One table per class - implementation of concrete class

```
package curam.inheritance;

import curam.inheritance.Cat;
import curam.inheritance.struct.CatDtls;
import curam.test.codetable.ANIMAL_TYPE;

public class CatImpl extends AnimalImpl<Cat, CatDtls>
  implements Cat {

  protected CatImpl() {
  }

  @Override
  protected String getDiscriminatorValue() {
    return ANIMAL_TYPE.CAT;
  }

  @Override
  protected void mapBaseKeyToConcreteDtls() {
    getConcreteRowDtls().animalID = getBaseRowDtls().animalID;
  }

  public int getNumberOfLivesRemaining() {
    return getConcreteRowDtls().numberOfLivesRemaining;
  }

  public void setNumberOfLivesRemaining(final int value) {
    getConcreteRowDtls().numberOfLivesRemaining = value;
  }

  public void speak() {
    System.out.println("Miaow!  My name is " + getName()
        + " and I have " + getNumberOfLivesRemaining()
        + " lives remaining");
  }

  public void setNewInstanceDefaults() {// none required
  }

  public void crossFieldValidation() {// none required
  }

  public void crossEntityValidation() {// none required
  }

  public void mandatoryFieldValidation() {// none required
  }

}
```

Class declaration

```
final class CatImpl extends AnimalImpl<Cat, CatDtls>
  implements Cat {
```

The class:

- extends the AnimalImpl class created above, specifying the Cat interface and CatDtls struct as parameters; and
- implements the Cat interface (which in turn extends the Animal interface).

Constructor

```
protected CatImpl() {
  }
```

The class has a protected constructor, as is the norm for the implementation classes.

Specifying the discriminator value

```
@Override
  protected String getDiscriminatorValue() {
    return ANIMAL_TYPE.CAT;
  }
```

The class must override the BasePlusConcreteTableImpl.getDiscriminatorValue method to specify the discriminator String value which distinguishes Cat instances from other types of Animal.

In this example a code-table constant is used to provide the String value.

Mapping the base key

```
@Override
  protected void mapBaseKeyToConcreteDtls() {
    getConcreteRowDtls().animalID = getBaseRowDtls().animalID;
  }
```

The class overrides the BasePlusConcreteTableImpl.mapBaseKeyToConcreteDtls method, which is called when a new entity instance is stored on the database. Typically, the base row uses the AUTO_ID facility to assign a primary key value on insert, and since (in this example) Animal and Cat share key values, the key value assigned to the Animal.animalID column must also be stored on the Cat.animalID column.

The method makes use of these methods from BasePlusConcreteTableImpl :

- getBaseRowDtls, to access the AnimalDtls row data; and
- getConcreteRowDtls, to access the CatDtls row data.

Getters and Setters

The getters and setters make use of the BasePlusConcreteTableImpl.getConcreteRowDtls method to access the CatDtls row data.

Implementations for the getters and setters for the Animal fields are inherited from AnimalImpl.

speak

```
public void speak() {
    System.out.println("Miaow!  My name is " + getName() +
      " and I have "+ getNumberOfLivesRemaining() +
      " lives remaining");
  }
```

This class must provide an implementation of the Animal.speak method - recall that this method is *not* implemented in AnimalImpl, as the logic differs between CatImpl and DogImpl.

DogImpl

```
package curam.inheritance;

import curam.inheritance.Dog;
import curam.inheritance.struct.DogDtls;
import curam.test.codetable.ANIMAL_TYPE;
```

```
class Dog extends Animal<Dog, DogDtls> implements Dog {

  protected Dog() {
  }

  @Override
  protected String getDiscriminatorValue() {
    return ANIMAL_TYPE.DOG;
  }

  @Override
  protected void mapBaseKeyToConcreteDtls() {
    getConcreteRowDtls().animalID = getBaseRowDtls().animalID;
  }

  public String getFavouriteTrick() {
    return getConcreteRowDtls().favouriteTrick;
  }

  public void setFavouriteTrick(final String value) {
    getConcreteRowDtls().favouriteTrick = value;
  }

  public void speak() {
    System.out.println("Woof!  My name is " + getName()
        + " and I like to " + getFavouriteTrick());
  }

  public void setNewInstanceDefaults() {// none required
  }

  public void crossFieldValidation() {// none required
  }

  public void crossEntityValidation() {// none required
  }

  public void mandatoryFieldValidation() {// none required
  }

}
```

Figure 156: One table per class - implementation of another concrete class

The structure of this class is similar to CatImpl above.

CatDAOImpl and DogDAOImpl

```
package curam.inheritance;

import java.util.Set;

import com.google.inject.Singleton;

import curam.inheritance.Cat;
import curam.inheritance.CatDAO;
import curam.inheritance.struct.CatDtls;
import curam.util.persistence.StandardDAOImpl;
```

```
@Singleton
public class CatDAO extends StandardDAOImpl<Cat, CatDtls>
 implements
    CatDAO {
  private static final CatAdapter adapter = new CatAdapter();

  /**
   * Protected no-arg constructor for use only by Guice
   */
  protected CatDAO() {
    super(adapter, Cat.class);
  }

  public Set<Cat> readAllCats() {
    return newSet(adapter.readAll());
  }

}
package curam.inheritance;

import java.util.Set;

import com.google.inject.Singleton;

import curam.inheritance.Dog;
import curam.inheritance.DogDAO;
import curam.inheritance.struct.DogDtls;
import curam.util.persistence.StandardDAOImpl;

@Singleton
public class DogDAO extends StandardDAOImpl<Dog, DogDtls>
 implements
    DogDAO {
  private static final DogAdapter adapter = new DogAdapter();

  /**
   * Protected no-arg constructor for use only by Guice
   */
  protected DogDAO() {
    super(adapter, Dog.class);
  }

  public Set<Dog> readAllDogs() {
    return newSet(adapter.readAll());
  }

}
```

Figure 157: One table per class - DAO implementations for the concrete classes

The DAO classes for the concrete classes are straightforward DAO implementations.

CatDAOImpl and DogDAOImpl each support the creation of new instances of their respective entities, as well as retrieval of existing instances, by making use of the StandardDAOImpl class.

AnimalDAOImpl

```
package curam.inheritance;
```

```
import java.util.HashMap;
import java.util.Map;
import java.util.Set;

import com.google.inject.Inject;
import com.google.inject.Singleton;

import curam.inheritance.Animal;
import curam.inheritance.AnimalDAO;
import curam.inheritance.CatDAO;
import curam.inheritance.DogDAO;
import curam.inheritance.AnimalDtls;
import curam.test.codetable.ANIMAL_TYPE;
import curam.util.persistence.BaseDAOImpl;
import curam.util.persistence.ReaderDAO;
import curam.util.persistence.RowManager;

@Singleton
public class AnimalDAOImpl extends
    BaseDAOImpl<Long, Animal, AnimalDtls> implements AnimalDAO {

  private static final AnimalAdapter adapter = new
 AnimalAdapter();

  @Inject
  private CatDAO catDAO;

  @Inject
  private DogDAO dogDAO;

  /**
   * Protected no-arg constructor for use only by Guice
   */
  protected AnimalDAO() {
    super(adapter, Animal.class);
  }

  @Override
  protected String getDiscriminator(
      final RowManager<Long, AnimalDtls> rowManager) {
    return rowManager.getDtls().animalType;
  }

  @Override
  protected Map<String, ReaderDAO<Long, ? extends Animal>>
    getConcreteReaderDAOs() {

    final Map<String, ReaderDAO<Long, ? extends Animal>>
      concreteReaderDAOs =
      new HashMap<String, ReaderDAO<Long, ? extends Animal>>();

    concreteReaderDAOs.put(ANIMAL_TYPE.CAT, catDAO);
    concreteReaderDAOs.put(ANIMAL_TYPE.DOG, dogDAO);
    return concreteReaderDAOs;
  }

  public Set<Animal> readAllAnimals() {
    return newSet(adapter.readAll());
```

```
      }
}
```
Figure 158: One table per class - DAO implementation for the abstract class

Class declaration

```
final class AnimalDAOImpl extends
   BaseDAOImpl<Long, Animal, AnimalDtls> implements AnimalDAO
```

The class extends the BaseDAOImpl class, which provides support for reading instances of abstract classes (by calling back to the implementation to decide which concrete class to instantiate). AnimalDAOImpl is responsible for retrieving a Cat or Dog instance, according to the value of the discriminator column, i.e. Animal.animalType.

Adapter

```
private static final AnimalAdapter adapter = new AnimalAdapter();
```

The class contains an adapter variable, as is the norm for DAO implementations.

DAO instances

```
@Inject
  private CatDAO catDAO;

  @Inject
  private DogDAO dogDAO;
```

The class contains injected instances of the DAO interfaces for the concrete classes.

These DAOs will be used to "dish up" the appropriate concrete type when a calling requests to read or search for Animal instances.

Protected constructor

```
/**
   * Protected no-arg constructor for use only by Guice
   */
  protected AnimalDAO() {
    super(adapter, Animal.class);
              }
```

The class contains a protected constructor, as is the norm for DAO implementations. This constructor passes the adapter and the entity class to the super constructor.

Get discriminator value from a row read from the database

```
@Override
  protected String getDiscriminator(
    final RowManager<Long, AnimalDtls> rowManager) {
    return rowManager.getDtls().animalType;
  }
```

You must override the BaseDAOImpl.getDiscriminator method to return the discriminator value from an abstract row read from the database (in this example, the value of Animal.animalID is returned from the row read).

Map discriminator values to DAO instances

```
@Override
  protected Map<String, ReaderDAO<Long, ? extends Animal>>
    getConcreteReaderDAOs() {
    final Map<String, ReaderDAO<Long, ? extends Animal>>
      concreteReaderDAOs =
      new HashMap<String, ReaderDAO<Long, ? extends Animal>>();

    concreteReaderDAOs.put(ANIMAL_TYPE.CAT, catDAO);
    concreteReaderDAOs.put(ANIMAL_TYPE.DOG, dogDAO);
    return concreteReaderDAOs;
  }
```

You must override the BaseDAOImpl.getConcreteReaderDAOs method to return a map of DAOs which can read the concrete instances of your entity.

The persistence infrastructure uses this map to retrieve a Cat or Dog as appropriate, depending on the value of Animal.animalID.

One table per concrete class

If you choose this option, you will create one physical database table for each concrete class, in this example Cat and Dog. The abstract class will have no table of its own; instead, the abstract fields will be replicated on each of the concrete tables.

You must provide the following implementation classes (listed in dependency order):

- AnimalImpl (optional);
- CatImpl;
- DogImpl;
- CatDAOImpl;
- DogDAOImpl; and
- AnimalDAOImpl.

These classes are described in detail below. The concrete (Cat and Dog) implementation classes are reasonably straightforward, but the abstract (Animal) classes are more complex.

AnimalImpl

```
package curam.inheritance;

import curam.inheritance.Animal;
import curam.util.persistence.helper.SingleTableEntityImpl;
import curam.util.type.DeepCloneable;

abstract class AnimalImpl<DTLS_STRUCT extends DeepCloneable>
    extends SingleTableEntityImpl<DTLS_STRUCT> implements Animal
 {

  public void printName() {
    System.out.println("My name is " + getName());
  }

}
```
Figure 159: One table per concrete class - implementation of abstract base class

You may provide this implementation if there is any common behavior between your concrete classes which is identical.

> **Note:** Although the behavior of attribute getters and setters for the base class is conceptually identical for all Animal instances, technically they differ since:
>
> - Cat instances will store their Animal attributes on the Cat table; and
> - Dog instances will store their Animal attributes on the Dog table.
>
> Hence the implementation of Animal getters and setters *cannot* be implemented in a central place.

The class is parameterized with the name of the Dtls struct, to be supplied by the implementing subclass.

The class is package-protected and marked abstract. In this example the subclasses will be placed in the same code-package; if you require some of your subclasses to be in a different package, you will need to mark your abstract implementation class public.

If there is no common implementation logic, you may omit this class, and instead concrete classes will inherit from SingleTableEntityImpl (or some other suitable class) directly.

CatImpl

```
package curam.inheritance;

import curam.inheritance.Cat;
import curam.inheritance.struct.CatDtls;

public class CatImpl extends AnimalImpl<CatDtls> implements Cat {

  protected CatImpl() {
  }

  public int getNumberOfLivesRemaining() {
    return getDtls().numberOfLivesRemaining;
  }

  public void setNumberOfLivesRemaining(final int value) {
    getDtls().numberOfLivesRemaining = value;
  }

  public String getName() {
    return getDtls().name;
  }

  public void setName(String value) {
    getDtls().name = value;
  }

  public void speak() {
    System.out.println("Miaow!  My name is " + getName()
        + " and I have " + getNumberOfLivesRemaining()
        + " lives remaining");
  }

  public void crossFieldValidation() {
```

```
      // none required
  }

  public void crossEntityValidation() {
    // none required
  }

  public void mandatoryFieldValidation() {
    // none required
  }

  public void setNewInstanceDefaults() {
    // none required
  }

}
```

Figure 160: One table per concrete class - implementation of concrete class

Class declaration

```
public class CatImpl extends AnimalImpl<CatDtls> implements Cat {
```

The class:

- extends the AnimalImpl class created above, specifying the CatDtls struct as a parameter; and
- implements the Cat interface (which in turn extends the Animal interface).

Protected constructor

```
protected CatImpl() {
  }
```

The class has a protected constructor, as is the norm for the implementation classes.

Getters and Setters

The getters and setters make use of the regular SingleTableEntityImpl.getDtls method to access the CatDtls row data.

Getters and setters are supplied for both:

- Cat -specific fields; and
- fields common across all Animal types.

speak

```
public void speak() {
    System.out.println("Miaow!  My name is " + getName() +
      " and I have " + getNumberOfLivesRemaining() +
      " lives remaining");
  }
```

This class must provide an implementation of the Animal.speak method.

DogImpl

```
package curam.inheritance;

import curam.inheritance.Dog;
```

```
import curam.inheritance.struct.DogDtls;

public class DogImpl extends AnimalImpl<DogDtls> implements Dog {

  protected DogImpl() {
  }

  public String getFavouriteTrick() {
    return getDtls().favouriteTrick;
  }

  public void setFavouriteTrick(final String value) {
    getDtls().favouriteTrick = value;

  }

  public String getName() {
    return getDtls().name;
  }

  public void setName(String value) {
    getDtls().name = value;
  }

  public void speak() {
    System.out.println("Woof!  My name is " + getName()
        + " and I like to " + getFavouriteTrick());
  }

  public void crossFieldValidation() {
    // none required
  }

  public void crossEntityValidation() {
    // none required
  }

  public void mandatoryFieldValidation() {
    // none required
  }

  public void setNewInstanceDefaults() {
    // none required
  }

}
```

Figure 161: One table per concrete class - implementation of another concrete class

The structure of this class is similar to CatImpl above.

CatDAOImpl and DogDAOImpl

```
package curam.inheritance;

import java.util.Set;

import com.google.inject.Singleton;
```

```java
import curam.inheritance.Cat;
import curam.inheritance.CatDAO;
import curam.inheritance.struct.CatDtls;
import curam.util.persistence.StandardDAOImpl;

@Singleton
public class CatDAOImpl extends StandardDAOImpl<Cat, CatDtls>
    implements CatDAO {
  private static final CatAdapter adapter = new CatAdapter();

  /**
   * Protected no-arg constructor for use only by Guice
   */
  protected CatDAOImpl() {
    super(adapter, Cat.class);
  }

  public Set<Cat> readAllCats() {
    return newSet(adapter.readAll());
  }

}
package curam.inheritance;

import java.util.Set;

import com.google.inject.Singleton;

import curam.inheritance.Dog;
import curam.inheritance.DogDAO;
import curam.inheritance.struct.DogDtls;
import curam.util.persistence.StandardDAOImpl;

@Singleton
public class DogDAOImpl extends StandardDAOImpl<Dog, DogDtls>
    implements DogDAO {
  private static final DogAdapter adapter = new DogAdapter();

  /**
   * Protected no-arg constructor for use only by Guice
   */
  protected DogDAOImpl() {
    super(adapter, Dog.class);
  }

  public Set<Dog> readAllDogs() {
    return newSet(adapter.readAll());
  }

}
```

Figure 162: One table per concrete class - DAO implementations for the concrete classes

The DAO classes for the concrete classes are straightforward DAO implementations.

CatDAOImpl and DogDAOImpl each support the creation of new instances of their respective entities, as well as retrieval of existing instances, by making use of the StandardDAOImpl class.

AnimalDAOImpl

```java
package curam.inheritance;

import java.util.HashSet;
import java.util.Set;

import com.google.inject.Inject;
import com.google.inject.Singleton;

import curam.util.exception.UnimplementedException;

@Singleton
public class AnimalDAOImpl implements AnimalDAO {

  @Inject
  private CatDAO catDAO;

  @Inject
  private DogDAO dogDAO;

  /**
   * Protected no-arg constructor for use only by Guice
   */
  protected AnimalDAOImpl() {
  }

  public Set<Animal> readAllAnimals() {

    final Set<Cat> cats = catDAO.readAllCats();

    final Set<Dog> dogs = dogDAO.readAllDogs();

    final Set<Animal> animals = new HashSet<Animal>(cats.size()
        + dogs.size());
    animals.addAll(cats);
    animals.addAll(dogs);

    return animals;
  }

  public Animal get(final Long id) {
    throw new UnimplementedException();
  }

}
```

Figure 163: One table per concrete class - DAO implementation for the abstract class

Class declaration

```java
public class AnimalDAOImpl implements AnimalDAO {
```

The class does not make use of any superclasses for its implementation.

Adapter

Unlike most DAO implementations, there is no adapter variable because there is no physical Animal database table.

DAO instances

```
@Inject
  private CatDAO catDAO;

  @Inject
  private DogDAO dogDAO;
```

The class contains injected instances of the DAO interfaces for the concrete classes.

These DAOs will be used to delegate searches to.

Protected constructor

```
/**
   * Protected no-arg constructor for use only by Guice
   */
  protected AnimalDAOImpl() {
  }
```

The class contains a protected constructor, as is the norm for DAO implementations.

Performing a search across Animal types

```
public Set<Animal> readAllAnimals() {

    final Set<Cat> cats = catDAO.readAllCats();

    final Set<Dog> dogs = dogDAO.readAllDogs();

    final Set<Animal> animals =
      new HashSet<Animal>(cats.size() + dogs.size());
    animals.addAll(cats);
    animals.addAll(dogs);

    return animals;
  }
```

A search of Animal instances across the Cat and Dog tables is performed by naively delegating the searches and combing the results.

Unsupported - retrieval of an Animal by its ID

```
public Animal get(final Long id) {
    throw new UnimplementedException();
  }
```

> **Important:** It is not possible to retrieve a generic Animal by its ID. This is because the Cat and Dog database tables maintain their own IDs - there is no concept of an animalID as such.
>
> If you require to be able to retrieve a generic Animal by its ID, then do *not* choose to store your data using this "One table per concrete class" method.

One table for the whole hierarchy

If you choose this option, you will create one physical database table to store all types in the hierarchy. The single table, in this example Animal will store all attributes required by any type, with default/null values stored where not applicable to a particular type.

This option requires a *disciminator value* in the form of the attribute Animal.animalType. This attribute stores a String value which will allow the implementation to determine whether a particular Animal is a Cat or a Dog.

You must provide the following implementation classes (listed in dependency order):

- AnimalImpl;
- CatImpl;
- DogImpl;
- CatDAOImpl;
- DogDAOImpl; and
- AnimalDAOImpl.

These classes are described in detail below. The concrete (Cat and Dog) implementation classes are reasonably straightforward, but the abstract (Animal) classes are more complex.

AnimalImpl

```
package curam.inheritance;

import curam.inheritance.Animal;
import curam.inheritance.struct.AnimalDtls;
import curam.util.persistence.helper.SingleTableEntityImpl;

abstract class AnimalImpl extends
 SingleTableEntityImpl<AnimalDtls>
    implements Animal {

  protected AnimalImpl() {
  }

  public String getName() {
    return getDtls().name;
  }

  public void setName(final String value) {
    getDtls().name = value;
  }

}
```

Figure 164: One table for the whole hierarchy - implementation of abstract base class

Class declaration

```
abstract class AnimalImpl extends
 SingleTableEntityImpl<AnimalDtls>
  implements Animal {
```

The implementation class extends the standard class SingleTableEntityImpl, parameterized with the Dtls struct from the single database table (AnimalDtls).

The class is package-protected and marked abstract. In this example the subclasses will be placed in the same code-package; if you require some of your subclasses to be in a different package, you will need to mark your abstract implementation class public.

The class implements the Animal interface; note that the class implements only *some* of the methods required by the interface, leaving others to the subclass implementation, e.g:

- AnimalImpl provides an implementation for getName and setName, as the behavior is identical for all Animal instances; but
- AnimalImpl does not provide an implementation for speak, as the behavior will differ between Cat and Dog instances.

Protected constructor

```
protected AnimalImpl() {
}
```

Getters and Setters

The getters and setters make use of the SingleTableEntityImpl.getDtls to retrieve the Dtls struct for the single row (in this example an AnimalDtls struct).

CatImpl

```
package curam.inheritance;

import curam.inheritance.Cat;
import curam.inheritance.struct.AnimalDtls;
import curam.test.codetable.ANIMAL_TYPE;
import curam.util.persistence.EntityInfo;
import curam.util.persistence.helper.SingleTableEntityImpl;

public class CatImpl extends AnimalImpl implements Cat {

  protected CatImpl() {
  }

  /**
   * {@inheritDoc}
   */
  @Override
  public void setEntityInfo(
      EntityInfo<Long, SingleTableEntityImpl<AnimalDtls>,
      AnimalDtls>
      entityInfo) {
    super.setEntityInfo(entityInfo);

    // check that this object has been constructed with an
    // appropriate row
    if (getID() != null
        && !getDtls().animalType.equals(ANIMAL_TYPE.CAT)) {
      throw new RuntimeException("Expected to be a cat");
    }
  }

  public int getNumberOfLivesRemaining() {
    return getDtls().numberOfLivesRemaining;
  }

  public void setNumberOfLivesRemaining(final int value) {
    getDtls().numberOfLivesRemaining = value;
  }

  public void speak() {
    System.out.println("Miaow!  My name is " + getName()
```

```
        + " and I have " + getNumberOfLivesRemaining()
        + " lives remaining");
  }

  public void crossFieldValidation() {
    // none required
  }

  public void crossEntityValidation() {
    // none required
  }

  public void mandatoryFieldValidation() {
    // none required
  }

  public void setNewInstanceDefaults() {
    getDtls().animalType = ANIMAL_TYPE.CAT;
  }

}
```

Figure 165: One table for the whole hierarchy - implementation of concrete class

Class declaration

```
public class CatImpl extends AnimalImpl implements Cat {
```

The class:

- extends the AnimalImpl class created above. As there is only a single database table, no parameters are required; and
- implements the Cat interface (which in turn extends the Animal interface).

Protected constructor

```
protected CatImpl() {
  }
```

The class has a protected constructor, as is the norm for the implementation classes.

Confirming that the correct type has been retrieved

```
/**
  * {@inheritDoc}
  */
@Override
public void setEntityInfo(
    EntityInfo<Long, SingleTableEntityImpl<AnimalDtls>,
    AnimalDtls>
    entityInfo) {
  super.setEntityInfo(entityInfo);

  // check that this object has been constructed with an
  // appropriate row
  if (getID() != null &&
      !getDtls().animalType.equals(ANIMAL_TYPE.CAT)) {
    throw new RuntimeException("Expected to be a cat");
  }
```

```
    }
```

If the CatDAO is used to retrieve a Cat instance, it is important to check that the Animal row retrieved actually contains the correct discriminator value for a Cat, to guard against client code trying to retrieve a Cat based on a Dog 's ID.

Getters and Setters

The getters and setters make use of the SingleTableEntityImpl.getDtls method to access the AnimalDtls row data.

Implementations for the getters and setters for the Animal fields are inherited from AnimalImpl.

speak

```
public void speak() {
    System.out.println("Miaow!  My name is " + getName() +
        " and I have " + getNumberOfLivesRemaining() +
        " lives remaining");
  }
```

This class must provide an implementation of the Animal.speak method - recall that this method is *not* implemented in AnimalImpl, as the logic differs between CatImpl and DogImpl.

Specifying the discriminator value for new instances

```
public void setNewInstanceDefaults() {
    getDtls().animalType = ANIMAL_TYPE.CAT;
              }
```

When a new Cat is created, its discriminator value must be set. This is done in the setNewInstanceDefaults method.

DogImpl

```
package curam.inheritance;

import curam.inheritance.DogImpl;
import curam.inheritance.struct.AnimalDtls;
import curam.test.codetable.ANIMAL_TYPE;
import curam.util.persistence.EntityInfo;
import curam.util.persistence.helper.SingleTableEntityImpl;

public class DogImpl extends AnimalImpl implements Dog {

  protected DogImpl() {
  }

  /**
   * {@inheritDoc}
   */
  @Override
  public void setEntityInfo(
      EntityInfo<Long, SingleTableEntityImpl<AnimalDtls>,
      AnimalDtls>
      entityInfo) {
    super.setEntityInfo(entityInfo);

    // check that this object has been constructed with an
```

```
      // appropriate row
      if (getID() != null
          && !getDtls().animalType.equals(ANIMAL_TYPE.DOG)) {
        throw new RuntimeException("Expected to be a dog");
      }
    }

  public String getFavouriteTrick() {
    return getDtls().favouriteTrick;
  }

  public void setFavouriteTrick(final String value) {
    getDtls().favouriteTrick = value;
  }

  public void crossFieldValidation() {
    // none required
  }

  public void crossEntityValidation() {
    // none required
  }

  public void mandatoryFieldValidation() {
    // none required
  }

  public void speak() {
    System.out.println("Woof!  My name is " + getName()
        + " and I like to " + getFavouriteTrick());
  }

  public void setNewInstanceDefaults() {
    getDtls().animalType = ANIMAL_TYPE.DOG;
  }

}
```

Figure 166: One table for the whole hierarchy - implementation of another concrete class

The structure of this class is similar to CatImpl above.

CatDAOImpl and DogDAOImpl

```
package curam.inheritance;

import java.util.Set;

import com.google.inject.Singleton;

import curam.inheritance.Cat;
import curam.inheritance.CatDAO;
import curam.inheritance.struct.AnimalDtls;
import curam.test.codetable.ANIMAL_TYPE;
import curam.util.persistence.StandardDAOImpl;

@Singleton
public class CatDAOImpl extends StandardDAOImpl<Cat, AnimalDtls>
    implements CatDAO {
```

```
  private static final AnimalAdapter adapter = new
 AnimalAdapter();

  /**
   * Protected no-arg constructor for use only by Guice
   */
  protected CatDAOImpl() {
    super(adapter, Cat.class);
  }

  public Set<Cat> readAllCats() {
    return newSet(adapter.searchByAnimalType(ANIMAL_TYPE.CAT));
  }

}
package curam.inheritance;

import java.util.Set;

import com.google.inject.Singleton;

import curam.inheritance.struct.AnimalDtls;
import curam.test.codetable.ANIMAL_TYPE;
import curam.util.persistence.StandardDAOImpl;

@Singleton
public class DogDAOImpl extends StandardDAOImpl<Dog, AnimalDtls>
     implements DogDAO {
  private static final AnimalAdapter adapter = new
 AnimalAdapter();

  /**
   * Protected no-arg constructor for use only by Guice
   */
  protected DogDAOImpl() {
    super(adapter, Dog.class);
  }

  public Set<Dog> readAllDogs() {
    return newSet(adapter.searchByAnimalType(ANIMAL_TYPE.DOG));
  }

}
```

Figure 167: One table for the whole hierarchy - DAO implementations for the concrete classes

The DAO classes for the concrete classes are straightforward DAO implementations.

CatDAOImpl and DogDAOImpl each support the creation of new instances of their respective entities, as well as retrieval of existing instances, by making use of the StandardDAOImpl class (parameterized with AnimalDtls, the single database table).

Note that the searches to retrieve e.g. all Cat instances make use of a modeled searchByAnimalType method, as there is no Cat table from which to retrieve all rows. All searches performed by CatDAOImpl should ensure that they return only Cat rows, otherwise an error will be thrown from CatImpl.setEntityInfo.

AnimalDAOImpl

```
package curam.inheritance;

import java.util.HashMap;
import java.util.Map;
import java.util.Set;

import com.google.inject.Inject;
import com.google.inject.Singleton;

import curam.inheritance.struct.AnimalDtls;
import curam.test.codetable.ANIMAL_TYPE;
import curam.util.persistence.BaseDAOImpl;
import curam.util.persistence.ReaderDAO;
import curam.util.persistence.RowManager;

@Singleton
public class AnimalImpl extends
    BaseDAOImpl<Long, Animal, AnimalDtls> implements AnimalDAO {
  private static final AnimalAdapter adapter = new
 AnimalAdapter();

  @Inject
  private CatDAO catDAO;

  @Inject
  private DogDAO dogDAO;

  /**
   * Protected no-arg constructor for use only by Guice
   */
  protected AnimalImpl() {
    super(adapter, Animal.class);
  }

  @Override
  protected String getDiscriminator(
      final RowManager<Long, AnimalDtls> rowManager) {
    return rowManager.getDtls().animalType;
  }

  @Override
  protected Map<String, ReaderDAO<Long, ? extends Animal>>
    getConcreteReaderDAOs() {
    final Map<String, ReaderDAO<Long, ? extends Animal>>
      concreteReaderDAOs =
      new HashMap<String, ReaderDAO<Long, ? extends Animal>>();

    concreteReaderDAOs.put(ANIMAL_TYPE.CAT, catDAO);
    concreteReaderDAOs.put(ANIMAL_TYPE.DOG, dogDAO);
    return concreteReaderDAOs;
  }

  public Set<Animal> readAllAnimals() {
    return newSet(adapter.readAll());
  }
```

```
}
```

Figure 168: One table for the whole hierarchy - DAO implementation for the abstract class

Class declaration

```
public class AnimalImpl extends BaseDAOImpl<Long, Animal,
 AnimalDtls>
   implements AnimalDAO {
```

The class extends the BaseDAOImpl class, which provides support for reading instances of abstract classes (by calling back to the implementation to decide which concrete class to instantiate). AnimalDAOImpl is responsible for retrieving a Cat or Dog instance, according to the value of the discriminator column, i.e. Animal.animalType.

Adapter

```
private static final AnimalAdapter adapter = new AnimalAdapter();
```

The class contains an adapter variable, as is the norm for DAO implementations.

DAO instances

```
@Inject
  private CatDAO catDAO;

  @Inject
  private DogDAO dogDAO;
```

The class contains injected instances of the DAO interfaces for the concrete classes.

These DAOs will be used to "dish up" the appropriate concrete type when a calling requests to read or search for Animal instances.

Protected constructor

```
/**
   * Protected no-arg constructor for use only by Guice
   */
  protected AnimalDAO() {
    super(adapter, Animal.class);
              }
```

The class contains a protected constructor, as is the norm for DAO implementations. This constructor passes the adapter and the entity class to the super constructor.

Get discriminator value from a row read from the database

```
@Override
  protected String getDiscriminator(
    final RowManager<Long, AnimalDtls> rowManager) {
    return rowManager.getDtls().animalType;
  }
```

You must override the BaseDAOImpl.getDiscriminator method to return the discriminator value from an abstract row read from the database (in this example, the value of Animal.animalID is returned from the row read).

Map discriminator values to DAO instances

```
@Override
  protected Map<String, ReaderDAO<Long, ? extends Animal>>
    getConcreteReaderDAOs() {
    final Map<String, ReaderDAO<Long, ? extends Animal>>
      concreteReaderDAOs =
      new HashMap<String, ReaderDAO<Long, ? extends Animal>>();

    concreteReaderDAOs.put(ANIMAL_TYPE.CAT, catDAO);
    concreteReaderDAOs.put(ANIMAL_TYPE.DOG, dogDAO);
    return concreteReaderDAOs;
  }
```

You must override the BaseDAOImpl.getConcreteReaderDAOs method to return a map of DAOs which can read the concrete instances of your entity.

The persistence infrastructure uses this map to retrieve a Cat or Dog as appropriate, depending on the value of Animal.animalID.

1.10 Adding New Searches to Existing Entities

Cúram ships with a number of entities which have service layers implemented using the Persistence Infrastructure.

Cúram recognizes that in certain circumstances, customers may wish to add additional read SQL (select statements) to the Cúram-shipped database entities behind PI-based service layer code, to retrieve data in new ways using existing Cúram-shipped database columns and/or columns on a custom database table.

Cúram supports a choice of approaches that allow you to implement new searches, described below.

> **Important:** Cúram does *not* support the addition of write SQL (insert/update/delete statements) to the Cúram-shipped database entities behind PI-based service layer code (as the invocation of such SQL would bypass the very service layer code that exists to protect the integrity of such data).

Approach 1

In the custom model package structure, model an extension entity which extends the Curam-shipped entity (if such an extension does not already exist).

In the extension entity, model a stereotyped retrieval operation (read/readmulti/nsread/nsmulti/ ns). The retrieval operation must return the full generated Dtls struct for the Curam-shipped entity (or the corresponding DtlsList struct for multi operations); moreover, any hand-crafted SQL for the operation must correctly populate every field in the return struct, including versionNo (if present). Note that hand-crafted is free to join to custom database tables if necessary to filter results (but not to return data from custom database tables).

In the custom code package structure, create a hand-crafted custom DAO interface/ implementation to house the new search operations. Note that unlike standard DAO interface/ implementations, your hand-crafted classes will *not* extend PI-supplied infrastructure classes.

In your custom DAO interface, declare your new search methods.

In your custom DAO implementation, implement your new search methods. The methods will delegate to the generated code for your custom entity extension. Note that there is no generated adapter support for operations contributed by extension classes, and so your implementation will need to provide the exception wrapping and struct mapping traditionally performed by the generated adapters.

In your client code which requires to execute your custom search, inject an instance of your new custom DAO interface and use your new search methods to return instances of the Curam-shipped interface for the entity's service layer class. You may access the entity's data via the accessor (getter) methods on the service layer class, including any derived data, and access any side-saddle tables using the entity's context, just as you would for instances returned by the Curam-shipped DAO interface.

(Optional) If you find that your client code ends up having to inject instances of both the Curam-shipped DAO interface and your new custom DAO interface, you might consider mimicking some or all of the Curam-shipped DAO methods on your new custom interface. The implementation of these mimicked methods may delegate to the Curam-shipped DAO implementation. Curam does not recommend that you allow your new custom DAO interface to extend the Curam-shipped DAO interface, nor that you allow your new custom DAO implementation to subclass the Curam-shipped DAO implementation, as to do so may present future upgrade difficulties.

Approach 2

In the custom model package structure, model an extension entity which extends the Curam-shipped entity (if such an extension does not already exist).

In the extension entity, model a stereotyped retrieval operation (read/readmulti/nsread/nsmulti/ns). The retrieval operation is free to return any data that it requires, including data joined from custom database tables, and to use any suitable return struct (i.e. the restrictions in Approach 1 do not apply here)..

In your client code which requires to execute your custom search, invoke the generated DAL code directly. Note that:

- you must *not* invoke any database write methods directly from your client code;
- derived data which might ordinarily be provided by a service-layer class will not be available; and
- data held on custom side-saddle table will only be available via a separate call to the generated DAL code for that custom side-saddle table.

Notices

Permissions for the use of these publications are granted subject to the following terms and conditions.

Applicability

These terms and conditions are in addition to any terms of use for the Merative website.

Personal use

You may reproduce these publications for your personal, noncommercial use provided that all proprietary notices are preserved. You may not distribute, display or make derivative work of these publications, or any portion thereof, without the express consent of Merative

Commercial use

You may reproduce, distribute and display these publications solely within your enterprise provided that all proprietary notices are preserved. You may not make derivative works of these publications, or reproduce, distribute or display these publications or any portion thereof outside your enterprise, without the express consent of Merative.

Rights

Except as expressly granted in this permission, no other permissions, licenses or rights are granted, either express or implied, to the publications or any information, data, software or other intellectual property contained therein.

Merative reserves the right to withdraw the permissions granted herein whenever, in its discretion, the use of the publications is detrimental to its interest or, as determined by Merative, the above instructions are not being properly followed.

You may not download, export or re-export this information except in full compliance with all applicable laws and regulations, including all United States export laws and regulations.

MERATIVE MAKES NO GUARANTEE ABOUT THE CONTENT OF THESE PUBLICATIONS. THE PUBLICATIONS ARE PROVIDED "AS-IS" AND WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, AND FITNESS FOR A PARTICULAR PURPOSE.

Merative or its licensors may have patents or pending patent applications covering subject matter described in this document. The furnishing of this documentation does not grant you any license to these patents.

Information concerning non-Merative products was obtained from the suppliers of those products, their published announcements or other publicly available sources. Merative has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-Merative products. Questions on the capabilities of non-Merative products should be addressed to the suppliers of those products.

Any references in this information to non-Merative websites are provided for convenience only and do not in any manner serve as an endorsement of those websites. The materials at those websites are not part of the materials for this Merative product and use of those websites is at your own risk.

Privacy policy

The Merative privacy policy is available at https://www.merative.com/privacy.

Trademarks

Merative ™ and the Merative ™ logo are trademarks of Merative US L.P. in the United States and other countries.

IBM®, the IBM® logo, and ibm.com® are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide.

Adobe™, the Adobe™ logo, PostScript™, and the PostScript™ logo are either registered trademarks or trademarks of Adobe™ Systems Incorporated in the United States, and/or other countries.

Oracle and Java are registered trademarks of Oracle and/or its affiliates.

The registered trademark Linux® is used pursuant to a sublicense from the Linux Foundation, the exclusive licensee of Linus Torvalds, owner of the mark on a worldwide basis.

Microsoft™, Windows™, and the Windows™ logo are trademarks of Microsoft™ Corporation in the United States, other countries, or both.

UNIX™ is a registered trademark of The Open Group in the United States and other countries.

Other company, product, and service names may be trademarks or service marks of others.