

codeobjects



CORADS™

CodeObjects Rapid Application Development Studio

Tutorial 6

Programming

CodeObjects, Inc.

Tutorial 6: Programming

© 2011 CodeObjects, Inc. All rights reserved.

No part of this book may be reproduced, stored in a retrieval system, or transmitted by any means electronic or otherwise without written permission from the publisher.

We have made every effort to make this book as complete and accurate as possible, but no warranty of fitness is implied. The information is provided “as is,” and the publisher assumes no liability for any losses or damages arising from the use of this information. The content included in CORADS Tutorials is intended solely for training purposes, and may or may not depict the specific features in the CORADS package as installed on your system. CodeObjects reserves the right to modify CORADS features at any time.

CORADS is the only software certified for customizing applications provided by CodeObjects. You are allowed to use CORADS only to customize the applications provided by CodeObjects. The CORADS feature set received in your software package is subject to your CORADS license agreement.

CORADS is a trademark of CodeObjects, Inc. WebSphere is a registered trademark of IBM. Other trademarks are the property of their respective owners.

This manual is confidential and proprietary information provided by CodeObjects, Inc. for clients only.

CodeObjects Inc.
1381 McCarthy Blvd.
Milpitas, CA 95035 USA
<http://www.codeobjects.com>

Table of Contents

Getting Started	1
Programming the Tutorial Application	1
Understanding the Java Perspective	1
Introducing Data Interceptors	5
Coding with List of Values.....	8
Implementing a CodeObjects Service	11
Programming with Data Objects	14
Update the Defect's Due Date Using Data Interceptor	27
Detailed Configuration	27
Define the Data Interceptor in the Defect Entity.....	27
Implement the Data Interceptors.....	29
Programmer Notes	34
Wire the Dependencies in the System File.....	35
End Result	35
Due Date Is Editable in the UI	37
Update the Close Date and Closed By Info	38
High Level Implementation.....	38
Implement a Service Exception to Be Used with the Service to Be Written	38
Detailed Configuration for Exception	39
Implement a Data Object to Search for the Employee Based on the Login Name	40
Detailed Configuration.....	40
Implement the Service Interface That Declares the Method Signatures	42
Detailed Configuration.....	42
Implement a Service with Dependencies.....	45
Implement the Service Stub to Write Code.....	47
Programmer Notes	53
Update the System File for Wiring.....	54
Hook up the UI with the Service	54

Detailed Configuration.....	55
Programmer Notes	61
End Result	62
Change to a Different Way.....	63
Update the Close Date and Closed By Info Using UI	
Custom Code.....	64
Detailed Configuration.....	64
Programmer Notes	67
End Result.....	68
Send Mail to the Defect Owner.....	69
Detailed Configuration	70
Add Java Code	73
Programmer Notes	88
End Result	88
Send Email to Defect Owner Automatically	90
Detailed Configuration	90
Programmer Notes	99
End Result	100
Review	100
Appendix: Debugging.....	101
Starting the Process	101
Debugging a Method	102
At a Breakpoint	107

Getting Started

This tutorial focuses primarily on programming. This chapter introduces the concepts that will be discussed in detail in subsequent chapters.

Programming the Tutorial Application

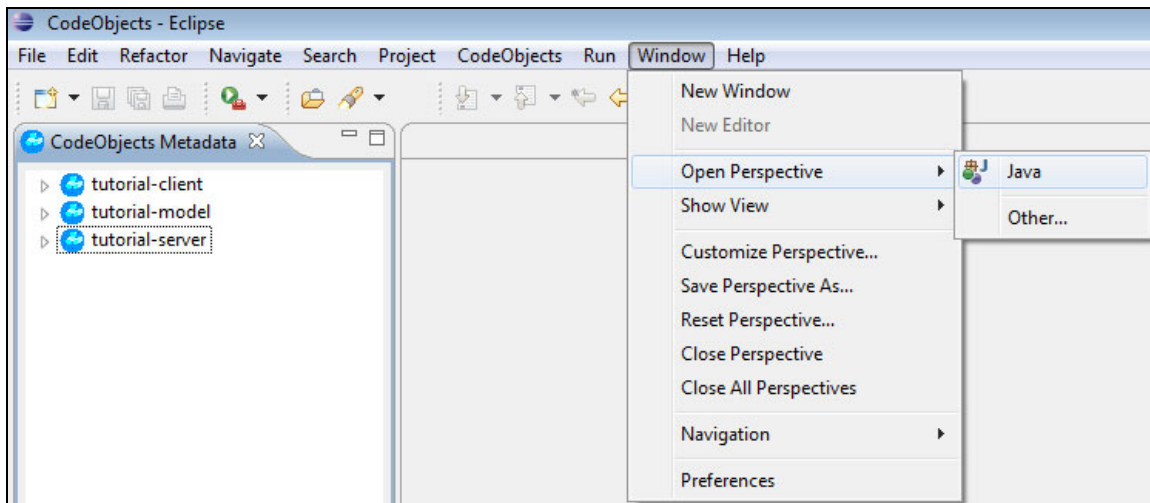
This tutorial focuses on programming, and presents four different tutorial scenarios to demonstrate the use of programming extensions in CORADS.

- Use of a data interceptor
 - You will update the field **DueDate** in **Defect**. Based on the defect's priority, the system will set different values in **DueDate**. This scenario also shows how to use generated enum values for the list of values.
- UI-side custom code
 - When closing a defect, the system will update the **ClosedDate** and **ClosedById** fields to indicate when and who closed the defect.
 - Two approaches to a solution demonstrate two different programming techniques.
 - ◆ Implement a service and let the UI view invoke it.
 - ◆ Implement the solution purely using UI view custom code.
- Create a new button to alert the defect owner of a defect assignment.
 - Implement a button that will send an email. When the email is sent, display an informational message on the top of the UI view to inform the user.
- Instead of relying on a new button, automatically send an email to the defect owner (if assigned) when the user saves the record.

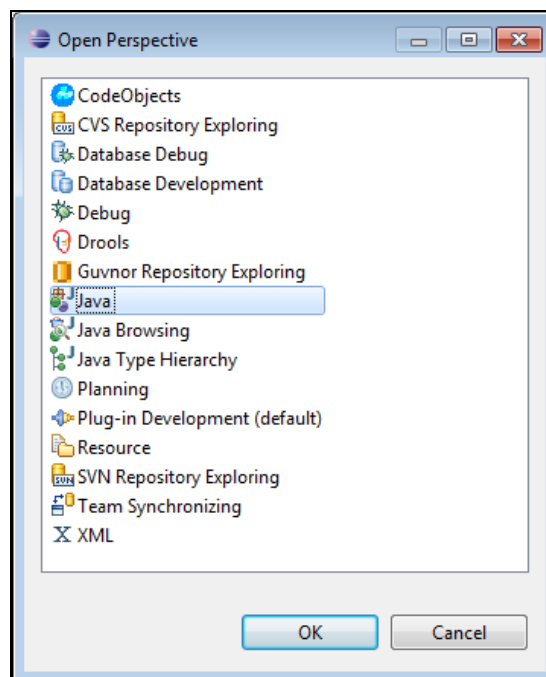
Understanding the Java Perspective

Until now, the tutorials guided your work in Eclipse within the *CodeObjects* perspective. For coding in Eclipse (as opposed to configuring metadata), this tutorial introduces the popular

Java perspective. To access the Java perspective, from the menu, click **Window** → **Open Perspective** → **Java**.

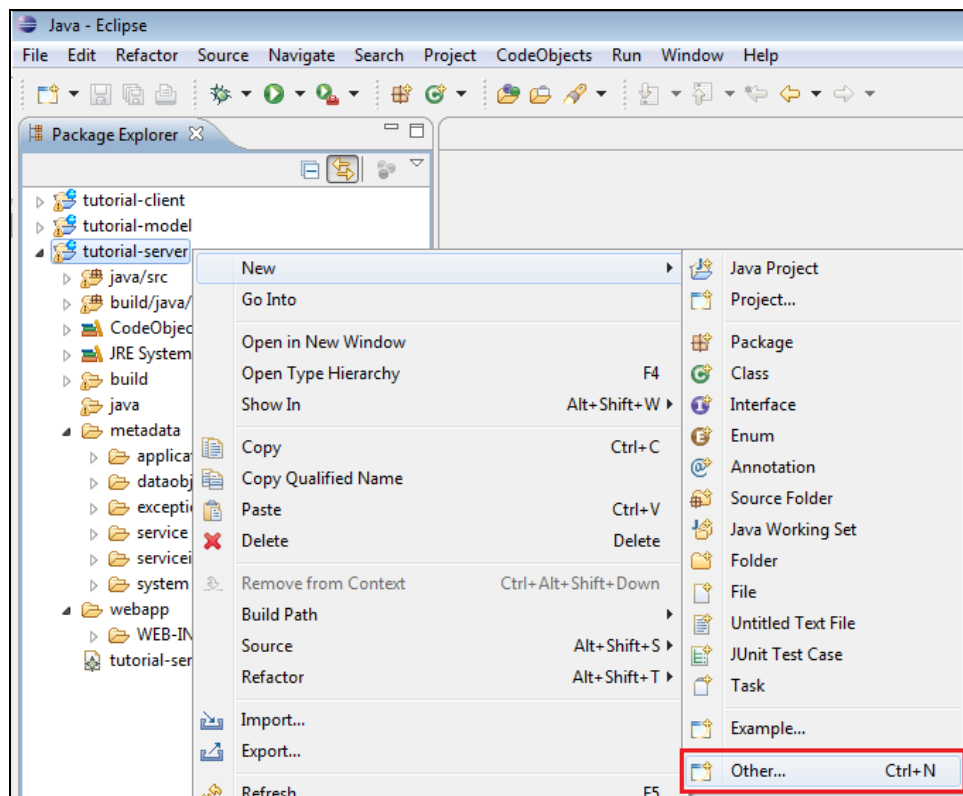


If Java is not available on the menu, click **Other** and in the **Open Perspective** box, click **Java**.

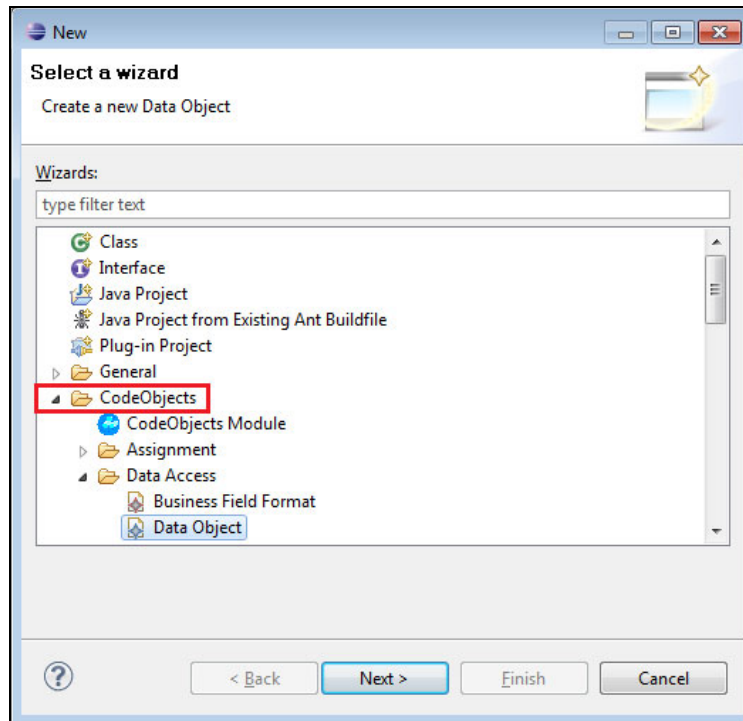


Throughout this tutorial, we may ask you to switch back and forth between the *CodeObjects* perspective (to edit metadata) and the *Java* perspective (to write Java code). You can indeed stay in either perspective. For example, to create a Data Object in *Java* perspective:

1. Click the module in which you intend to create the Data Object metadata, for example **tutorial-server**.
2. Right-click the module name and choose **New** → **Other** to open the **Select a wizard** dialog box.

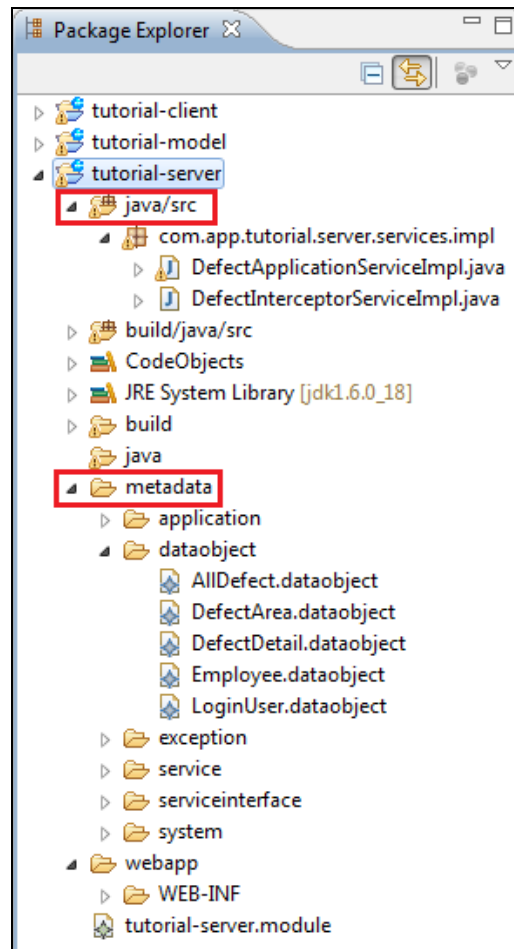


3. Expand the folder **CodeObjects** to see the metadata objects it contains.



4. Click **Next** to see the familiar dialog box for specifying a data object.

Package Explorer view will play an important role in this tutorial. Within **Package Explorer**, you can see the metadata you create (**metadata** folder) and the source code you write (**Java/src** folder).



We suggest that you experiment with Java perspective to get hands-on experience and become familiar with it.

Introducing Data Interceptors

Before proceeding into the next tutorial scenario, we must explain a programming extension in the CodeObjects platform, namely **Data Interceptors**.

What is Data Interceptor? Just like the name *interceptor* implies, it is a mechanism to intercept the data flow (such as before a query or after an update) to perform some application logic.

An Example

As an example from the tutorial, assume that the field **DueDate** is always determined based on the value of **Priority**, and the end user cannot modify it.

- If the Priority is **0-Urgent**, the **DueDate** should be set to today + 2 days.
- If the Priority is **1-High**, the **DueDate** should be set to today + 3 days.
- If the Priority is **2-Medium**, the **DueDate** should be set to today + 4 days
- Other than the above, the **DueDate** should be set to today + 5 days.

With this scenario, we can always intercept the data flow before the update happens. We examine the value of **Priority** and then set the **DueDate** accordingly:

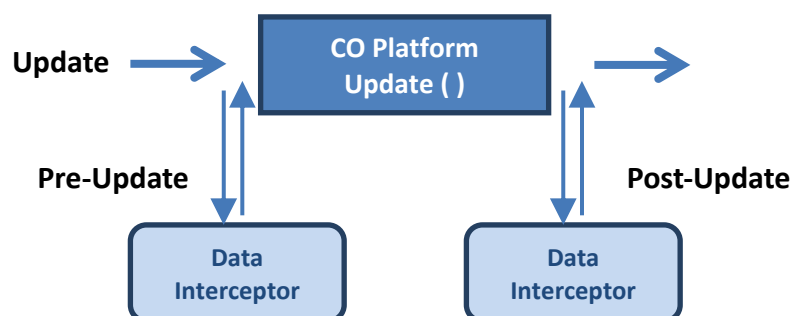
- When Before the update happens (in CORADS, this is called a **pre-update** event.)
- What Examine the **Priority** value and set the **DueDate** accordingly.
- Where The data interceptor implementation.

The following diagrams show the flow schematically. (For completeness, the diagram includes a post-update interceptor, which will be discussed later in this tutorial.)

Normal update data flow



With Data Interceptors



Interceptor Events

The previous example introduces and explains the behavior of pre-update interceptors. What other interceptor events does CodeObjects support?

Operation	Interceptor Event	Description
Query	pre-query	Event before the CodeObjects platform does a query on a given entity
	post-query	Event after the CodeObjects platform finishes a query on a given entity
Insert	pre-insert	Event before the CodeObjects platform inserts an entity record
	post-insert	Event after the CodeObjects platform finishes the insertion of an entity record
Update	pre-update	Event before the CodeObjects platform updates and persists an entity record to the database
	post-update	Event after the CodeObjects platform finishes the update of an entity record to the database
Delete	pre-delete	Event before the CodeObjects platform deletes an entity record
	post-delete	Event after the CodeObjects platform finishes the deletion of an entity record

Interceptor Implementation

You will implement your interceptor as a CodeObjects service. However, you must register and declare your interceptor service at the entity. We will discuss this further shortly.

The following table lists the methods available for implementation. These pass the entity's Java class into the interceptor. For the example of the **Defect** entity, the Java class is Defect.

Operation	Interceptor Event	Handler Method Signature
Query	pre-query	void preQuery(QueryObject queryObject)
	post-query	void postQuery(Defect defect)
Insert	pre-insert	void preInsert(Defect defect)
	post-insert	void postInsert(Defect defect)
Update	pre-update	void preUpdate(Defect curDefect, Defect newDefect)
	post-update	void postUpdate(Defect defect)
Delete	pre-delete	void preDelete(Defect defect)
	post-delete	void postDelete(Defect defect)

Interceptor's Current Restrictions

Within data interceptors, there are a few things you are not allowed to do:

- You cannot set or change the foreign key field's value. Using our Defect entity as an example, you cannot set the foreign key *DefectAreaId*.
- You cannot update the entity's field value in query and delete related events. For example, within your pre-query method, suppose you update the field value. The modified field value will be ignored and will not be persisted or saved to the database.
- The interceptor's method will not be called unless it is registered. When implementing the interceptor, CodeObjects platform will produce the stub Java class for all event handlers even if you do not register the event handling in the entity. However at runtime, the CodeObjects platform will not invoke your interceptor method unless it is registered. For example, even if you implement the *pre-delete* method if you do not register the interceptor in the entity, your implementation of pre-delete will not be invoked by CodeObjects.
- Note that the *post-insert*, *post-update*, and *post-delete* events are designed as a means of event notification and not for data propagation. For example, assume the user modifies the Defect data through the user interface and clicks the update button, which indirectly fires *pre-update* and *post-update* interceptors. You might implement some logic in the *post-update* event to change the content of Defect, but whatever you modified in *post-update* will be ignored. Using our example here, within the *post-update* event handler, you can certainly invoke other services or update other records for your business needs, but if you modify the Defect directly within these event handlers, your modifications will be ignored.

Coding with List of Values

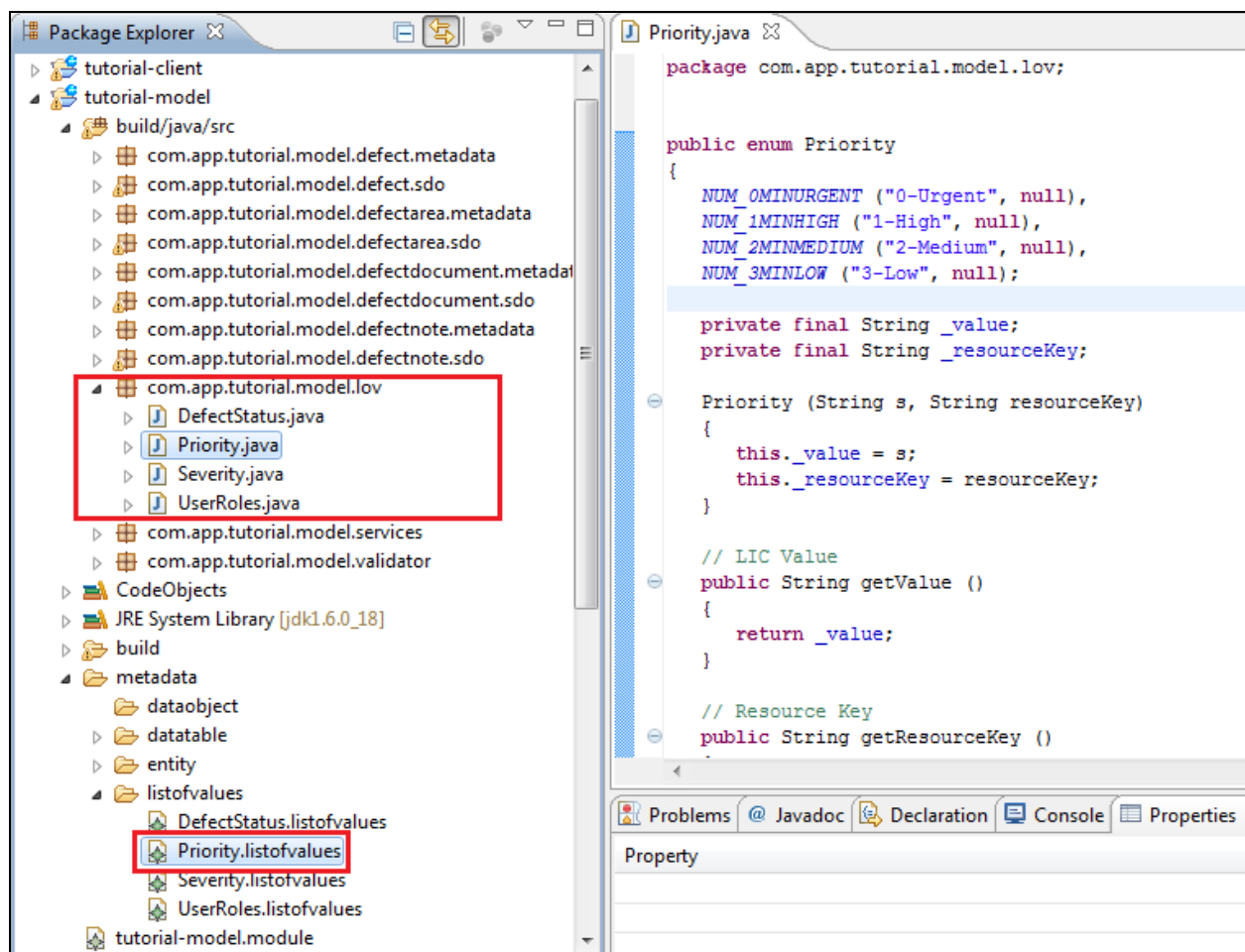
Each *List of Values* metadata generates a Java enum class. Instead of using literal strings (for example, "0-Urgent") in Java code, we recommend that you leverage the generated enum class to avoid mismatches at compile time. For example, if an item in a list of values is changed (such as from "0-Urgent" to "0-Panic"), the enum class will be regenerated with the new value. If your code still refers to the old enum value, which no longer exists, you will encounter a compile time error and thus detect the mismatch.

Getting Started

Take as an example the *Priority* list of values defined in the *tutorial-model* module. CORADS will generate the following Java enum:

```
package com.app.tutorial.model.lov;  
public enum Priority  
{  
    NUM_0MINURGENT ("0-Urgent", null),  
    NUM_1MINHIGH ("1-High", null),  
    NUM_2MINMEDIUM ("2-Medium", null),  
    NUM_3MINLOW ("3-Low", null);  
    ...  
}
```

You can switch to the Package Explorer in Eclipse to view the generated files for the metadata. The generated Java files are located under the *build/java/src* folder.



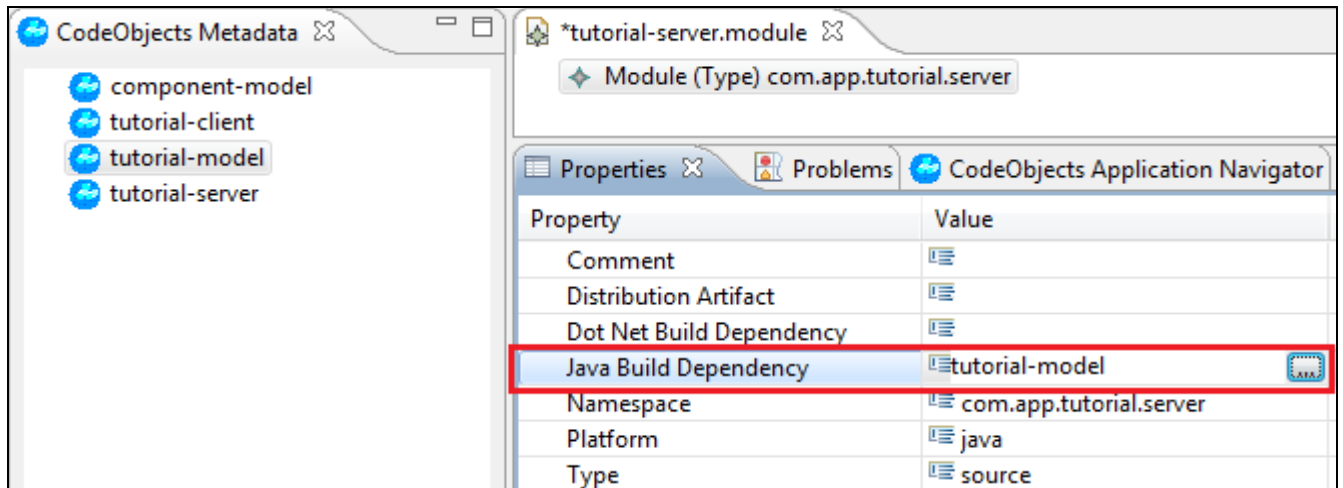
Adding Java Build Dependencies

If you program using the generated enum class, you may need to add a build dependency to it so that the compiler can correctly reference the class definition. For example, assume you will be implementing a service using Java and plan to use the enum class for the *Priority* list of values.

- The service you plan to implement is in the **tutorial-server** module.
- The generated enum of the **Priority** list of values is in the **tutorial-model** module.
- The **tutorial-server** module will depend on the **tutorial-model** module. Therefore, you must include a build dependency on the **tutorial-model** module to the **tutorial-server** module.

To do this:

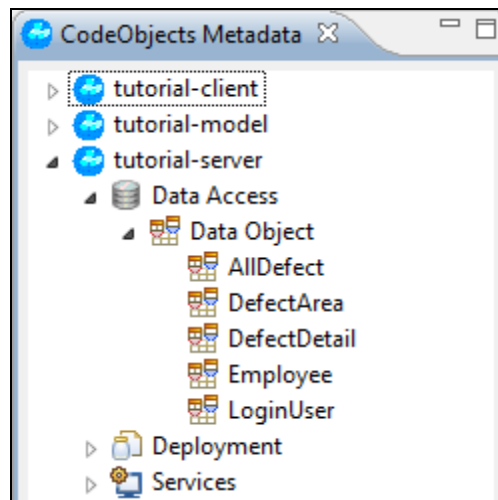
1. In the CodeObjects Metadata Pane, double-click the **tutorial-server** module to display **tutorial-server.module**.
2. Open the property **Java Build Dependency** and add **tutorial-model** module to it. This declares the compile-time dependency on the **tutorial-model** module.



NOTE

CORADS by default calculates the module dependencies based on the metadata references. This example demonstrates how to manually add a programming build dependency, which is beyond the metadata dependencies that CORADS can calculate.

If there are any metadata objects being referenced from other modules, the module dependencies are formed automatically behind the scenes, which frees you from explicitly setting the *Java Build Dependency*. For example, because the data objects defined in the *tutorial-server* module all reference the entities defined in the *tutorial-model* module, CORADS will produce an implicit Java build dependency on the *tutorial-model* module in *tutorial-server* module. Therefore, you do *not* need to explicitly add a Java Build Dependency. However, it is harmless if you should happen to explicitly declare the Java Build Dependency again.



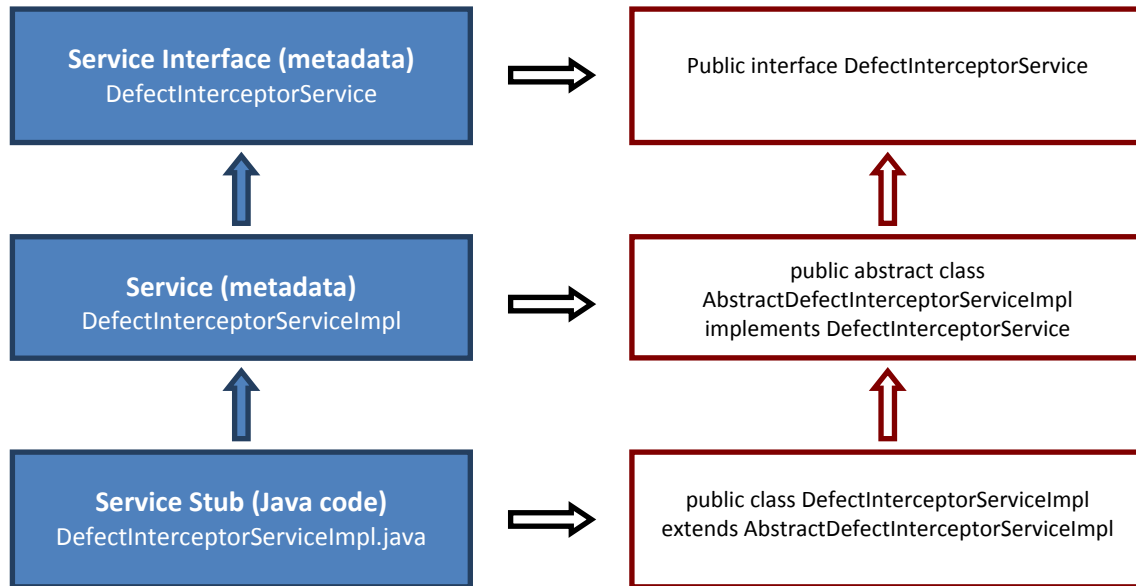
Implementing a CodeObjects Service

A *service* is where you implement your customized application logic. Services are written in modern object-oriented programming languages such as Java. Within the service implementation you can do many things, including interface with legacy systems. In this tutorial, the main theme is to show how you can implement your own service to encapsulate your business logic.

A CodeObjects service primarily encompasses three layers:

- **Service Interface** metadata to define the methods along with their inputs and outputs.
- **Service** metadata to lay out your implementation. For example, you may need to consider which data object you must access in order to complete your business logic. Within the service metadata, you specify which service interface you want to implement.
- **Service Stub** Java classes to implement your business logic.

The following diagram shows the metadata structure and the corresponding Java class inheritance.

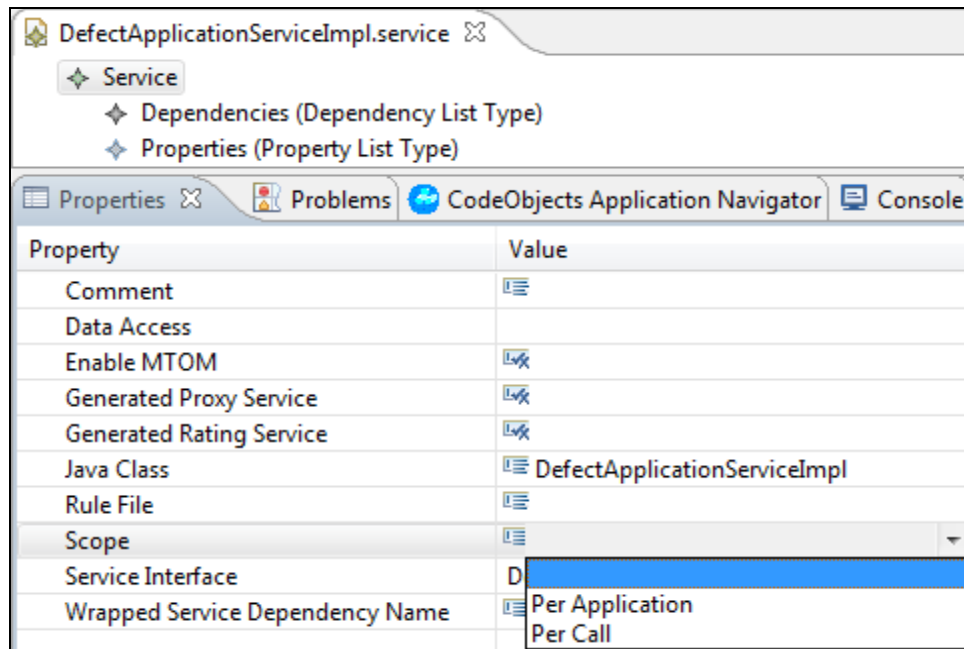


Service Interface	The service interface defines the operations that the service provides and the input and output parameters to these operations. This corresponds to the Java interface as listed to the right in the red box.
Service	The Service metadata represents the manifest that describes an implementation of a Service Interface. As described above, the underlying implementation can be in a host of different technologies. The service metadata also defines the list of dependencies a service implementation might have to other services, as well as any services properties that can be configured at deployment time to modify the service's behavior. This corresponds to the Java abstract class as listed to the right in the red box.
Service Stub	This is the physical Java class that holds your physical implementation. This cannot be any arbitrary Java class—it must inherit from the abstract class produced by CORADS based on your service metadata. This corresponds to the Java concrete class as listed to the right in the red box.

The Life Cycle of a Service

All instances of services are created and managed by the CodeObjects platform at runtime, which frees you from the burden of instantiating the service instances and managing their life cycles yourself. However, it is still very important for you to know the life scope of your service when it comes to the implementation.

The following image shows one of the services that you will implement in this tutorial. In the Property pane for the service, the property *Scope* has two values *Per Application* and *Per Call*.



The property *Scope* determines the life scope of the service instance.

Per Application In this setting, this service is a singleton throughout the entire application boundary. In other words, when you bring up your J2EE server (for example, Apache Tomcat or IBM WebSphere) there will be only one instance of the service. All the concurrent access or multi-threading access to the service will go through the same instance.

With this in mind, it is very important for you to code your service with respect to thread-safety. For example, if you plan to have class level data variables, you must consider the synchronization control among them.

Per Call In this setting this service is created whenever it is invoked from the CodeObjects platform. This setting is primarily designed for batch job services that will be scheduled regularly. When the scheduled time is reached, the specific service will be instantiated and invoked.

Regardless or whether it is Per Application or Per Call, the service is always instantiated by the CodeObjects platform, therefore you have no control in terms of the service construction and service deconstruction.

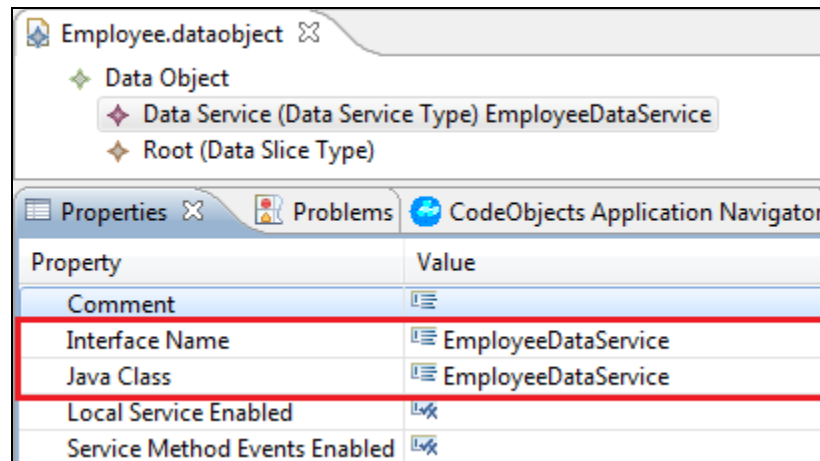
Programming with Data Objects

It is very common for you to want to query some database tables, update the data, and save your changes to the database. Or, you might want to create a brand new record through Java code. This section explains the fundamental programming concepts for dealing with Data Objects.

Understanding What Is Generated from the Data Object

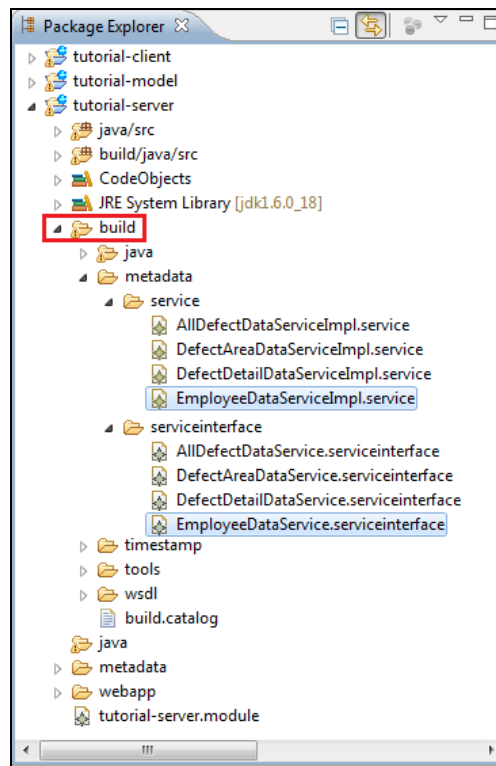
This uses the example **Employee** data object that you created in the previous tutorial.

1. From the **tutorial-server** module, open the **Employee** data object. In the Editor pane, near the top click the node **Data Service (Data Service Type)**. Notice the two properties **Interface Name** and **Java Class**. CORADS uses these two properties to create the service interface metadata.



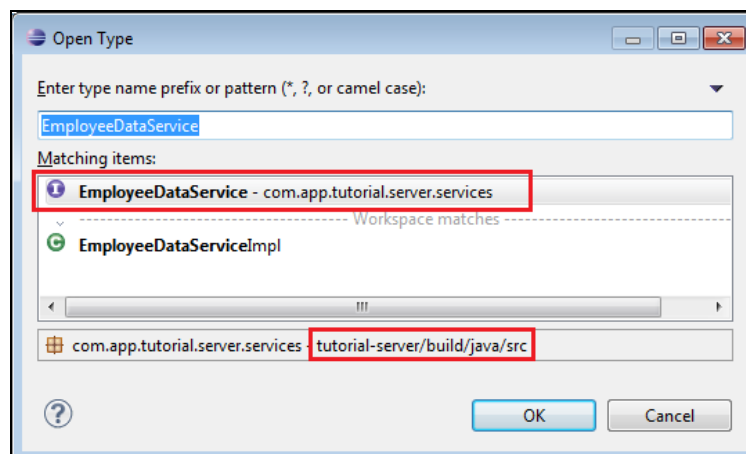
2. Switch to **Package Explorer** in the Java perspective and expand **tutorial-server** → **build** → **metadata** → **service** [and **serviceinterface**]. Notice that in the folder `build\metadata\serviceinterface`, **EmployeeDataService**, a service interface metadata file was generated. Similarly the service metadata **EmployeeDataServiceImpl** (service interface name + "Impl") was generated in the `build\metadata\service` folder.

The following screen image shows the service interface metadata.



Examine the Java interface that was produced from the **EmployeeDataService** service interface metadata. How can you find the corresponding Java class quickly?

1. Ensure that you have set Eclipse to display the **Java Perspective**.
2. Press CTRL+SHIFT+T to open the **Open Type** dialog box. In the text box, enter **EmployeeDataService**. Click the matching data service item (marked in red in the following image.) Note that **tutorial-sever/build/Java/src** from the tutorial-server module is also displayed. Click **OK** to open the EmployeeDataService.



You should see something like the following code to denote the Java programming interface of EmployeeDataService.

```
public interface EmployeeDataService
{
    public DataGraph create() throws DataAccessServiceException;
    public long countEmployee(EmployeeQuery querySpec, Long countUpTo) throws
DataAccessServiceException;
    public long countLoginUsers(long employeeId, LoginUsersQuery querySpec, Long
countUpTo) throws DataAccessServiceException;
    public long countEmployeeByExpression(EmployeeQueryExpression querySpec, Long
countUpTo) throws DataAccessServiceException;
    public long countLoginUsersByExpression(long employeeId,
LoginUsersQueryExpression querySpec, Long countUpTo) throws
DataAccessServiceException;
    public DataGraph queryById(long employeeId) throws
DataAccessServiceException;
    public DataGraph query(EmployeeQuery querySpec) throws
DataAccessServiceException;
    public DataGraph queryExpression(EmployeeQueryExpression querySpec) throws
DataAccessServiceException;
    public DataGraph update(DataGraph data) throws DataAccessServiceException;
}
```

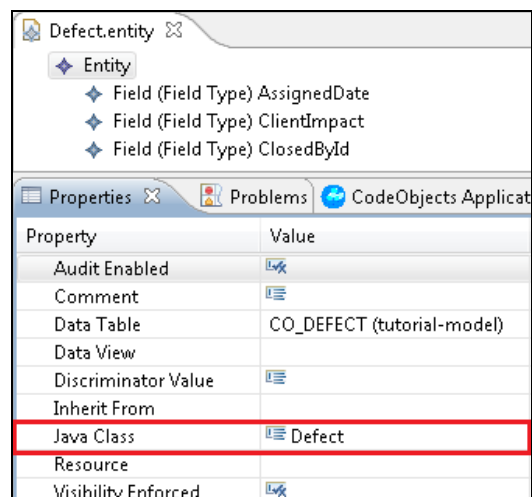
Note the data types DataGraph, EmployeeQuery, and EmployeeQueryExpression being used in the Java interface. These all seem to be related to Employee. What are they, and how are they used? The following few sections explain.

Understanding Java Classes for Entities, Data Slices, and Data Objects

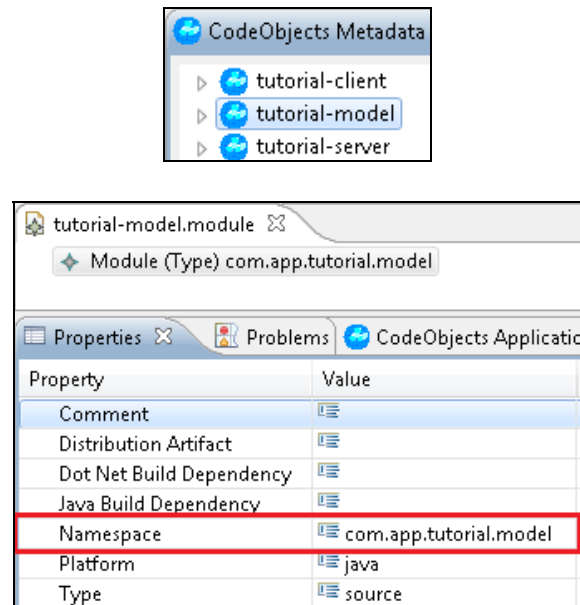
Java Class for the Entity

How can you identify the Java class for an entity?

1. From **tutorial-model** open the **Defect** entity. In the Editor pane at the top, click the node **Entity**. The property Java Class is Defect.



2. Switch to the CodeObjects perspective and double-click **tutorial-model**. Click the top node. The property **Namespace** is **com.app.tutorial.model**.

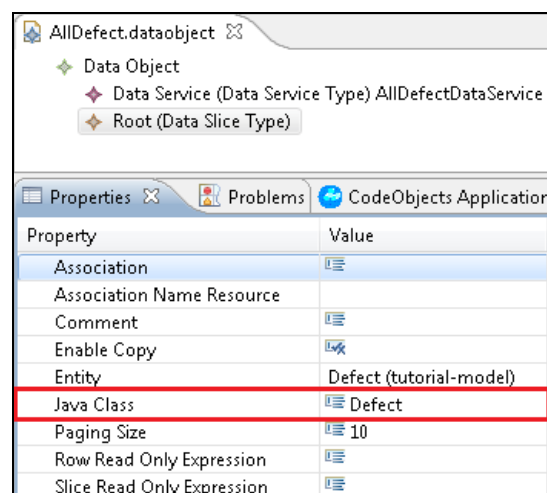


Thus, the Defect entity's fully qualified Java class is **com.app.tutorial.model.defect.sdo.Defect**, which is {namespace of the module} + {entity's class name as all lowercase} + sdo + {entity's class name}.

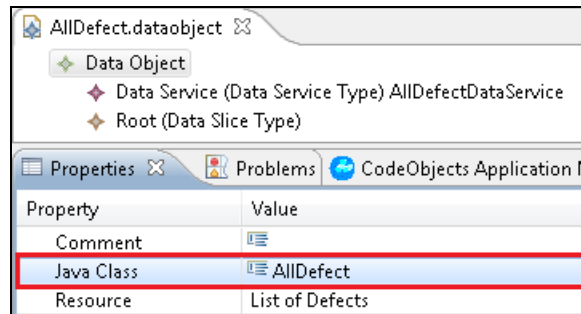
Java Class for a Data Slice

The defect entity is used in the AllDefect data object. To see the Java class for the data slice:

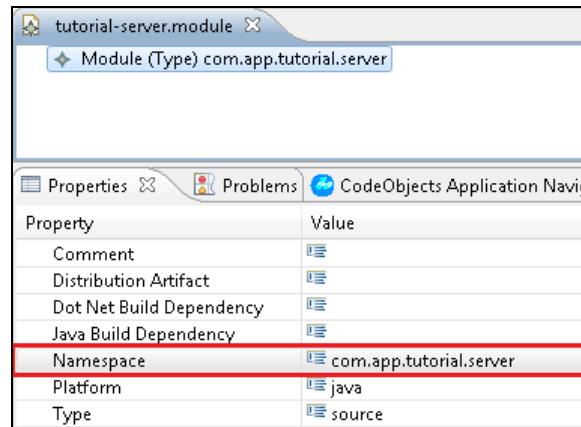
1. From the **tutorial-server** module, open the **AllDefect** data object. Expand the node **Data Object** and click the node **Root (Data Slice Type)**. Notice that the root data slice's property **Java Class** is also set to **Defect**.



- Click the node **Data Object**. The property **Java Class** is **AllDefect**.



- In the CodeObjects perspective, double-click **tutorial-server**, and then in the Editor pane click the node **Module....** The property **Namespace** is **com.app.tutorial.server**.



Thus, the fully qualified Java class for the Defect data slice in the AllDefect data object is **com.app.tutorial.server.dataobject.alldefect.sdo.Defect** which is {namespace of the module} + dataobject + {data object's class name as all lowercase} + sdo + {data slice's class name}.

NOTE

AllDefect is converted to all lowercase to be used in the namespace for the Java class.

The Relationship Between the Entity Class and the Data Slice Class

From the above explanation, there are two Java classes related to Defect.

- `com.app.tutorial.model.defect.sdo.Defect` (Java class for the Defect entity)
- `com.app.tutorial.server.dataobject.alldefect.sdo.Defect` (Java class for the Defect data slice in AllDefect data object)

Examine the Java class of `com.app.tutorial.server.dataobject.alldefect.sdo.Defect`. It inherits from `com.app.tutorial.model.defect.sdo.Defect`.

<code>com.app.tutorial.model.defect.sdo.Defect</code>	This Java class represents the Defect Entity. This is the base class for all Defect data slices.
<code>com.app.tutorial.server.dataobject.alldefect.sdo.Defect</code>	This Java class represents the Defect data slice within the data object, AllDefect. In this class you can access the virtual fields which you define in the data object. Recall that we added a virtual field in AllDefect data object for the defect owner's full name. This virtual field can only be accessed within this class.

One instance of these Java classes represents one database record from CO_DEFECT. For example, when the CodeObjects platform finishes a database query using the AllDefect data object, each of the result records from the CO_DEFECT table will be transformed to an instance of class `com.app.tutorial.server.dataobject.alldefect.sdo.Defect`. If the database query returns 10 defect records, there will be 10 instances of `com.app.tutorial.server.dataobject.alldefect.sdo.Defect`. You use the getter methods to access the field values or use the setter methods to set the field values. There are also getter methods to retrieve the associated data slices. For example, you can use the getter method (`getDefectArea()`) in Defect class to access the associated data slice, DefectArea. From there, you can use getters or setters to access the field values of the DefectArea data record.

Root Object in the Data Object

All these record instances (for example, `com.app.tutorial.server.dataobject.alldefect.sdo.Defect`) are placed within one container. This container is called the *Root* object. The root object's Java class is the class name of the Data Object metadata) + Root. For example, the root object of an AllDefect data object is `com.app.tutorial.server.dataobject.alldefect.sdo.AllDefectRoot`. This root object is the starting point for anything arising from the data object.

SDO Data Type Used in the Generated Data Object

Here we introduce Service Data Object (SDO) programming concepts.

Service Data Object (SDO) is a technology that allows heterogeneous data to be presented and accessed in a uniform way. When programming with data (for example, performing data insertion, modification, or deletion), we program through SDO. SDO is the equivalent of ADO.NET in the Java world. In another words, ADO.NET is to Microsoft.NET as SDO is to Java.

CodeObjects has created strongly typed programming interfaces on top of SDO, which ease data access programming using SDO. Recall from the previous section the explanation of the Java classes created for the entity, data slice, and root object in the data object:

- `com.app.tutorial.model.defect.sdo.Defect`
(Java class for Defect entity)
- `com.app.tutorial.server.dataobject.alldefect.sdo.Defect`
(Java class for Defect data slice in AllDefect data object)
- `com.app.tutorial.server.dataobject.alldefect.sdo.AllDefectRoot`
(Java class for the root object in AllDefect data object)

Understanding DataGraphs in SDO Programming

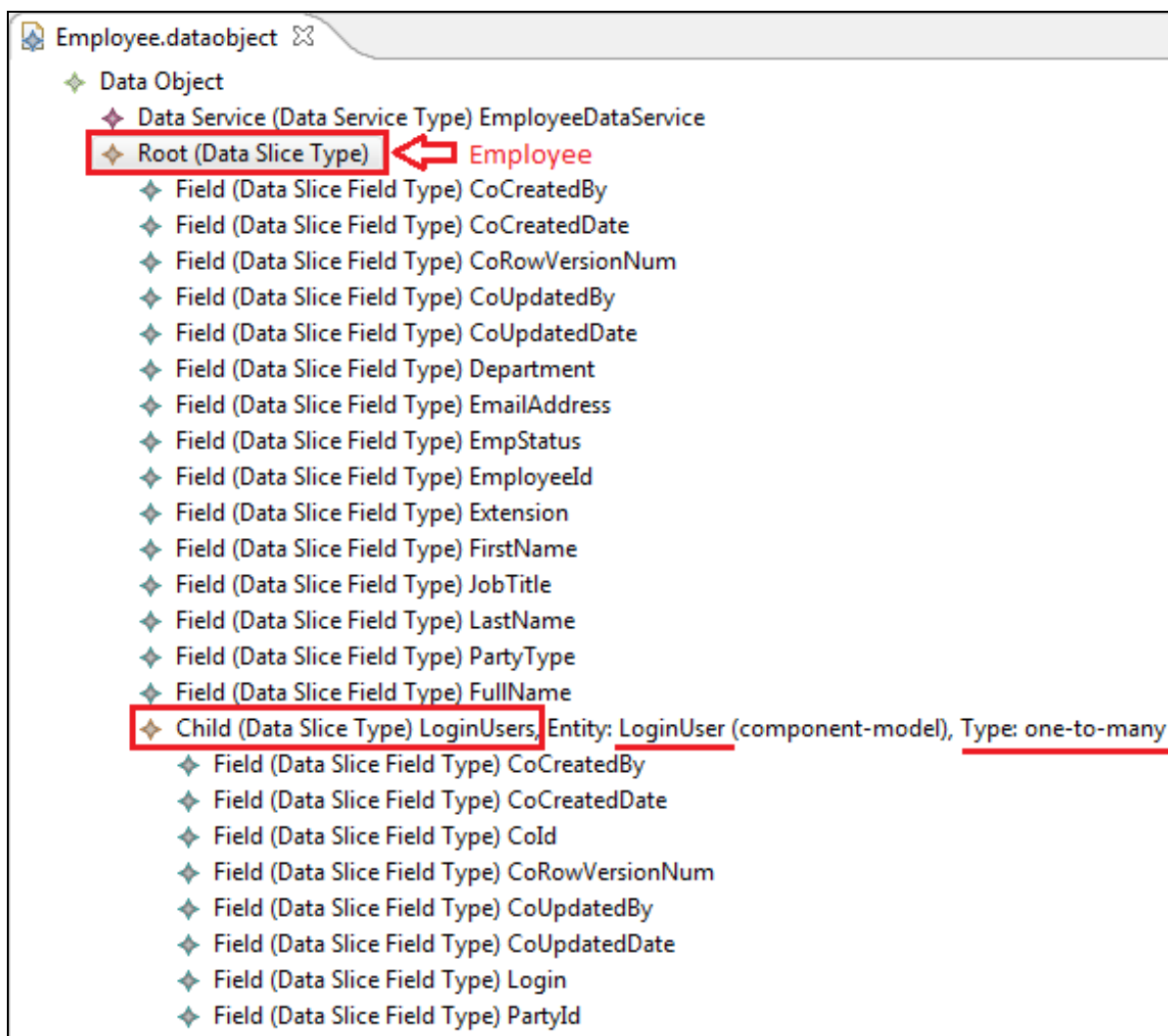
The entire data object is presented using Java class `DataGraph` (class **`commonj.sdo.DataGraph`**) in SDO. `DataGraph` is a container that holds a few important objects listed below:

- **Root object** Every `DataGraph` created in `CodeObjects` contains one and only one root object (example: `com.app.tutorial.server.dataobject.alldefect.sdo.AllDefectRoot` in AllDefect data object). The root object holds all Java data slice objects (such as `com.app.tutorial.server.dataobject.alldefect.sdo.Defect`).
- **Change summary** (`commonj.sdo.ChangeSummary`): Every `DataGraph` contains one and only one change summary that records all the changes made to the `DataGraph`, such as record insertion, deletion, and modification. The change summary is essential to persist your record into the database.

NOTE

You must turn on change tracking in SDO in order for the data graph to track the changes you make to the data object.

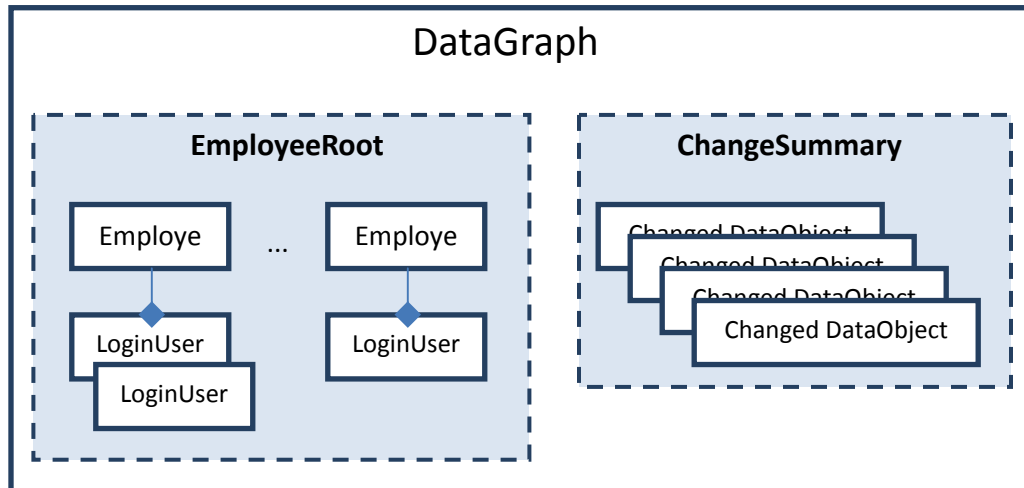
We have been using the AllDefect data object as an example. Examine the *Employee* data object defined in the *tutorial-server* module, to see how a hierarchical data object is presented in a data graph.



The example below shows the structure of the DataGraph holding the employee data.

- There is one and only root object (EmployeeRoot).
 - Within the root object, there are multiple Employee objects.
 - ◆ Below the employee object, there could be many LoginUser objects because it is a one-to-many association.

- There is one and only one ChangeSummary that tracks the changes (such as insertion, deletion, and modification) to the data object.



Sample Code

The following pseudo code segments provide programming examples.

Insert a new employee record

```
// Assume we get DataGraph from EmployeeDataService's create() or query() call
DataGraph data = ...;

// Get the root object
EmployeeRoot root = (EmployeeRoot) data.getRootObject();

// Instantiate a new Employee object. NOTE! This employee object so far is a
// standalone object and has not yet associated with the data graph.
Employee newEmployee = root.createEmployee();

// Set the field values
newEmployee.setFirstName("Scott");
newEmployee.setLastName("Smith");

// Turn on the change summary
data.getChangeSummary().beginLogging();

// Add the new employee object into the root object. After this, this employee
// object is part of the data graph.
root.getEmployee().add(newEmployee);

// Turn off the change summary
data.getChangeSummary().endLogging();

// Invoke the update method to save the record. NOTE!
// Here we assume getEmployeeDataService() will return EmployeeDataService.
...getEmployeeDataService().update(data);
```

Iterate through a list of records

```
// Assume we get DataGraph from EmployeeDataService's query() call
DataGraph data = ...getEmployeeDataService().query(queryobject);

// Fetch the root object
EmployeeRoot root = (EmployeeRoot) data.getRootObject();

// Loop through each employee from the result set
for (Employee employee : root.getEmployee())
{
    System.out.println("Employee " + employee.getFullName());
}
```

Loop through to find a record and delete it

```
// Assume we get DataGraph from EmployeeDataService's query() call
DataGraph data = ...getEmployeeDataService().query(queryobject);

// Fetch the root object
EmployeeRoot root = (EmployeeRoot) data.getRootObject();

// Turn on the change summary to track the removal of the employee
data.getChangeSummary().beginLogging();

// Loop through the records to find "Scott Smith".
// If we do find it, delete it from the list
for (Employee employee : root.getEmployee())
{
    if (employee.getFullName().equals("Scott Smith"))
    {
        root.getEmployee().remove(employee);
        break;
    }
}

// Turn off the change summary
data.getChangeSummary().endLogging();

// Check if we have found and deleted the record by checking
// the change summary size. Save the data graph if there are changes.
if (data.getChangeSummary().getChangedDataObjects().size() > 0)
    this.getEmployeeDataService().update(data);
```

Field Names as Character Strings

When accessing field values, most of the time you will use a getter or setter method. Below is a code snippet from an earlier code sample.

```
// Set the field values
newEmployee.setFirstName("Scott");
newEmployee.setLastName("Smith");
```

However, there could be times that you must represent the field using a literal string. For a literal string representing a field, the first character must always be lowercase rather than upper case, as is used in entity field names. For example:

Entity field	Java getter/setter method	Literal string for the field
FirstName	getFirstName() / setFirstName(...)	"firstName"
LastName	getLastName() / setLastName(...)	"lastName"

Query Object

Now that you understand data graphs, it is time to explain the structure of `EmployeeQuery`, `EmployeeQueryExpression`, based on what you saw from the generated programming interface in the earlier section.

For each data object, the `CodeObjects` platform produces two data structures related to queries. These two structures are intended for you to specify your query criteria. The structures are hierarchical, reflecting exactly the same hierarchy as the data object.

- **Query Object** This is the standard way of setting your query criteria, where you set the field criteria one by one, slice by slice. This is equivalent to setting the query criteria at the user interface.
- **Query Expression** This is the advanced way of setting the query criteria, where you use SQL-like expression language to set the query criteria. Because this is a more advanced topic, we will not explain the SQL expression syntax in detail. Instead we will use an example to give you some ideas for query expressions.

Paging Size

All query objects must set the paging size. This controls the number of returned rows. The consequences of not setting the page size could be disastrous when querying against a big table that has millions of qualified records.

Sort Spec

Optionally you can set the sort spec to control the order of records appearing in the result set.

Sample Code

The following pseudo code segments explain further.

Example 1: Using a query object

```
// Create a query object based on Employee data object
// The paging size is initialized to the same as defined in the data object
EmployeeQuery queryObject = EmployeeQuery.createQueryWithDefaultPaging();

// Change the paging size to be 1000
queryObject.getPagingDetails().setPagingSize(java.math.BigInteger.valueOf(1000))
;

// Set the search criteria to look for field FirstName is "Scott"
queryObject.setFirstName("Scott");

// Set the search criteria to look for field LastName starting with "Sm"
queryObject.setLastName("Sm%");

// Assume we can get EmployeeDataService and we call query() to return the
// data graph
DataGraph data = ...getEmployeeDataService().query(queryObject);
```

Example 2: Using a query object

```
// Create an empty query object based on Employee data object.
// There is no paging set
EmployeeQuery queryObject = EmployeeQuery.createQuery();

// Set the search criteria to look for field FirstName is "Scott"
queryObject.setFirstName("Scott");

// Create a paging detail and set the size
PagingDetailsType pagingDetails = new PagingDetailsType();
pagingDetails.setPagingSize(java.math.BigInteger.valueOf(1000));

// Create a sorting detail based on field LastName using ascending order
// NOTE!! This is the literal string of the field LastName, so it is "lastName"
SortDetailsType sortDetails = new SortDetailsType();
sortDetails.addSortField(new SortField("lastName", SortOrderEnum.ASC));

// Set the paging and sorting details to the query object
queryObject.setPagingDetails(pagingDetails);
queryObject.setSortDetails(sortDetails);

// Assume we can get EmployeeDataService and we call query() to return the
// data graph
DataGraph data = ...getEmployeeDataService().query(queryObject);
```

Example 3: Using a query expression

```
// Create a query object based on Employee data object
// The paging size is initialized to the same as defined in the data object
EmployeeQueryExpression querySpec =
EmployeeQueryExpression.createQueryWithDefaultPaging();

// Set the where clause using SQL-like language.
querySpec.setWhereClause("firstName = 'Scott' AND lastName LIKE 'Sm%'");

// Change the paging size to be 1000
querySpec.getPagingDetails().setPagingSize(java.math.BigInteger.valueOf(1000));

// Assume we can get EmployeeDataService and we call query() to return the
// data graph
DataGraph data = ...getEmployeeDataService().queryExpression(querySpec);
```

Update the Defect's Due Date Using Data Interceptor

This chapter demonstrates the procedure for conditionally updating the Defect Due Date by using a data interceptor.

In this scenario we will update the defect's **DueDate** field based on the defect's **Priority**.

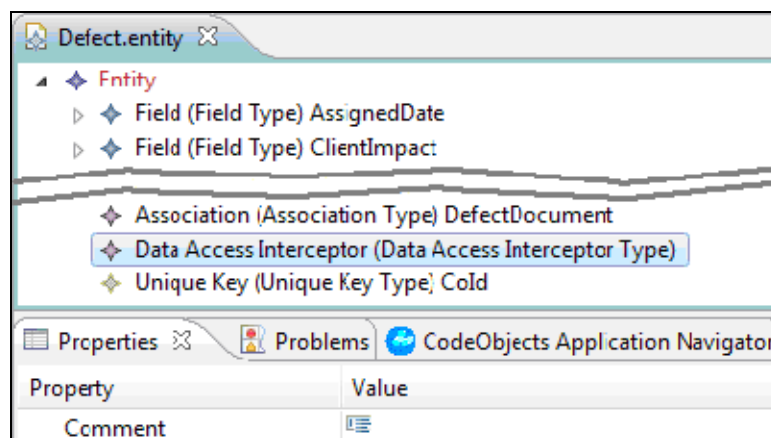
- If the Priority is **0-Urgent**, the **DueDate** should be set to today + 2 days.
- If the Priority is **1-High**, the **DueDate** should be set to today + 3 days.
- If the Priority is **2-Medium**, the **DueDate** should be set to today + 4 days
- If none of the above, the **DueDate** should default to today + 5 days.

We will implement a data interceptor using the **pre-insert** and **pre-update** events to set the **DueDate**.

Detailed Configuration

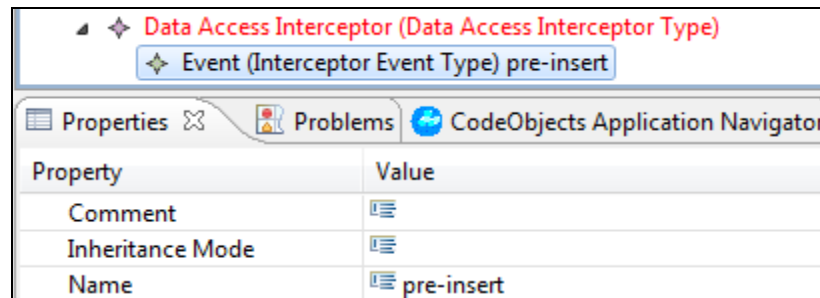
Define the Data Interceptor in the Defect Entity

1. From the **tutorial-model** module, open the **Defect** entity.
2. Right-click the root node **Entity** and choose **New Child** → **Data Access Interceptor** (**Data Access Interceptor Type**).

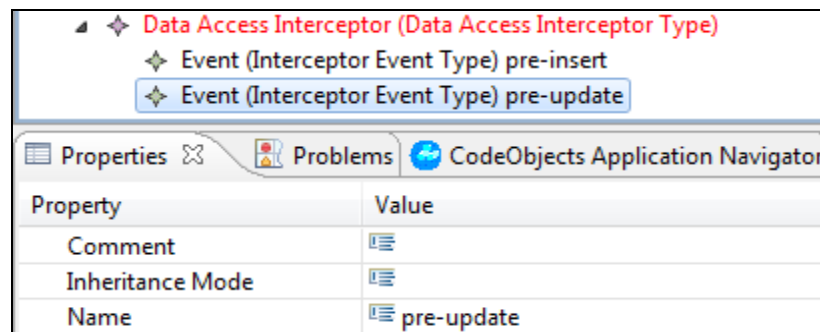


Update the Defect's Due Date Using Data Interceptor

3. Add a child **Event (Interceptor Event Type)** and set the property **Name** to **pre-insert**.

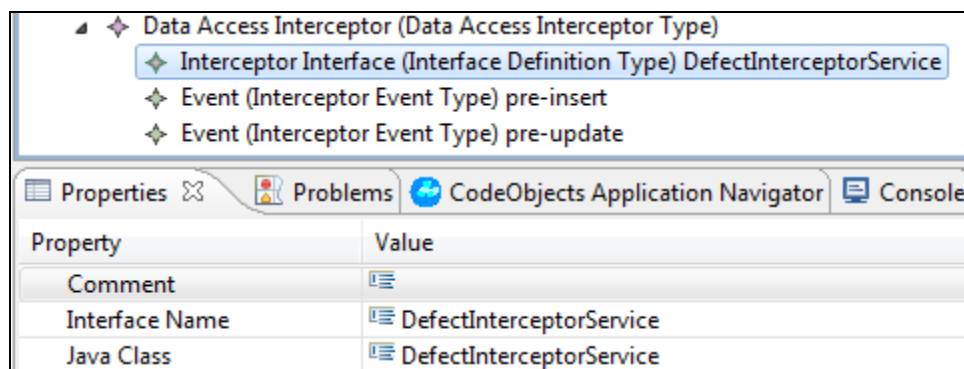


4. Similarly, add another **Event (Interceptor Event Type)** for **pre-update**.



Now, you must define the interface that will implement these two interceptor events (pre-insert and pre-update). The CodeObjects platform will invoke the implementations whenever the data events occur.

5. Add a child **Interceptor Interface (Interface Definition Type)**. Set the property **Interface Name** to **DefectInterceptorService**. Set the property **Java Class** to **DefectInterceptorService**, too.



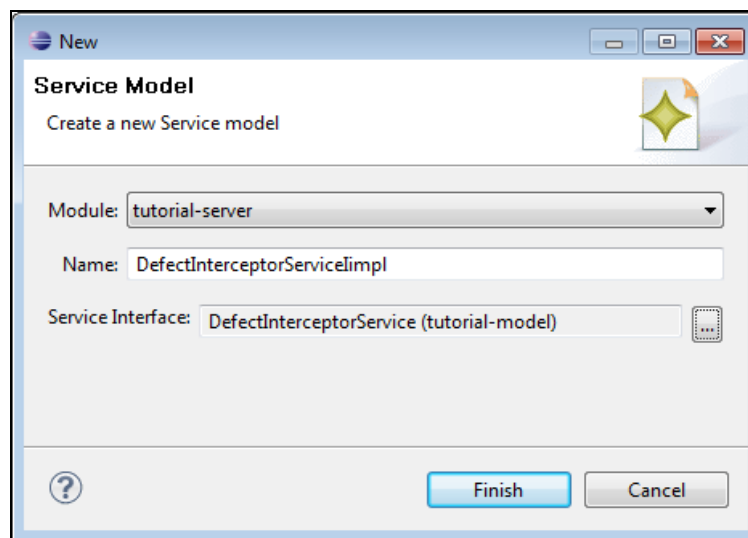
Update the Defect's Due Date Using Data Interceptor

Up to here you have basically registered to the Defect entity events indicating that whenever a defect is about to be inserted or be updated we must invoke theDefectInterceptorService. Save the changes to Defect.entity.

Implement the Data Interceptors

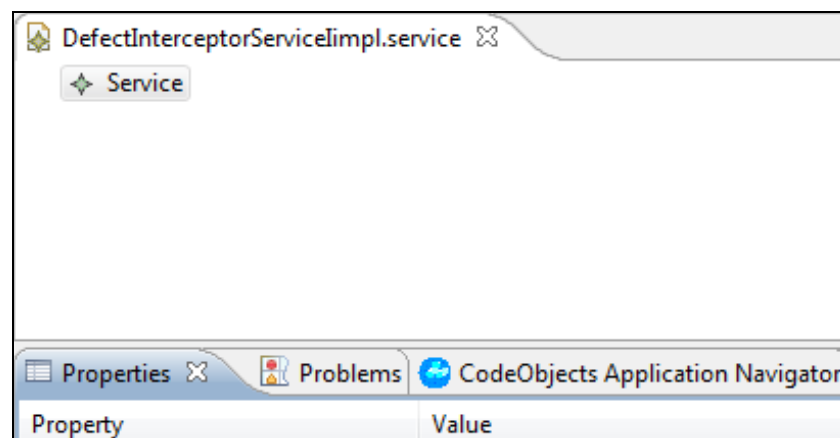
You must create a service implementation to physically handle these two data events: **pre-insert** and **pre-update**. Per the dictates of modular design, all the physical implementations are outside of the **tutorial-model** module.

1. In the CodeObjects Metadata pane, right-click **tutorial-server** and choose **New** → **Services** → **Service** to open the **Service Model** dialog box.



2. Set **Name** to **DefectInterceptorServiceImpl**. Note the naming convention: "...Impl" stands for the implementation. This is what physically implements the code to handle the data interceptor's pre-insert and pre-update events.

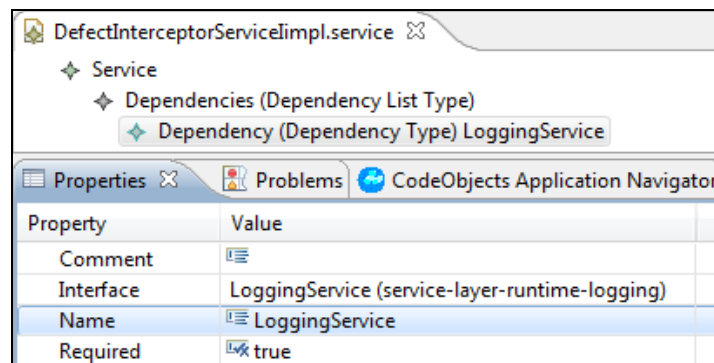
An empty service appears initially like that in the following image.



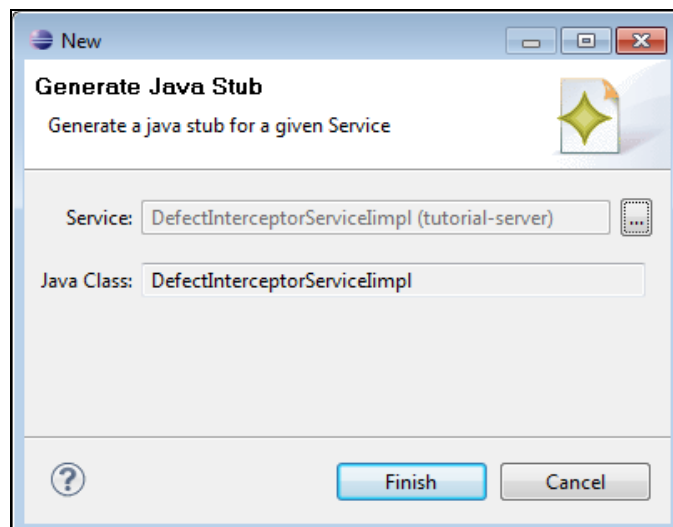
3. Select the root node **Service** and add a **New Child** → **Dependencies (Dependency List Type)**. Also add **to the new node** a **New Child** → **Dependency (Dependency Type)**. Set the properties as listed in the following table:

Property Name	Value
Interface	LoggingService (service-layer-runtime-logging)
Name	LoggingService
Required	True

Doing so will allow you to access all methods defined in the **LoggingService** service interface, which is used for logging within your implementation. This also implicitly defines a **Java Build Dependency** to module **service-layer-runtime-logging**. LoggingService is a standard service implemented by CodeObjects using the popular log4J Java-based logging utility.

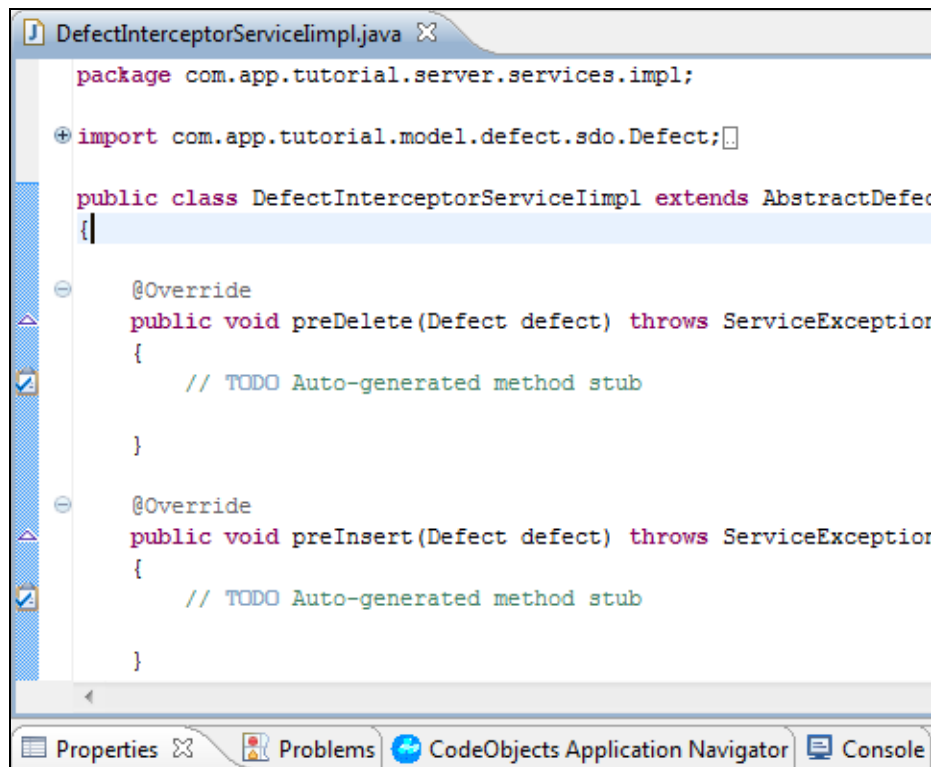


4. Create the service stub class to start to write Java code. In the CodeObjects Metadata Pane, right-click **tutorial-server** and choose **New** → **Services** → **Service Stub Generation** to open the Generate Java Stub dialog box.



5. Set the **Service** to **DefectInterceptorServiceImpl** and click **Finish**.

CORADS creates a stub Java class named **DefectInterfaceServiceImpl.java** for you. This is the physical Java class for the service **DefectInterceptorServiceImpl** that you have defined above. The resulting file should look like the following example:



```
package com.app.tutorial.server.services.impl;

import com.app.tutorial.model.defect.sdo.Defect;

public class DefectInterceptorServiceImpl extends AbstractDefectService {

    @Override
    public void preDelete(Defect defect) throws ServiceException {
        // TODO Auto-generated method stub
    }

    @Override
    public void preInsert(Defect defect) throws ServiceException {
        // TODO Auto-generated method stub
    }
}
```

Notice that the metadata defines only two interceptor events (**pre-insert**, **pre-update**) for the Defect entity, but the system prefilled the signatures of all possible event methods. This does not mean the system will automatically invoke other methods if you do *not* define the events.

Data Interceptor Event	Functions
pre-query	preQuery
post-query	postQuery
pre-insert	preInsert
post-insert	postInsert
pre-update	preUpdate
post-update	postUpdate
pre-delete	preDelete
post-delete	postDelete

Update the Defect's Due Date Using Data Interceptor

6. Copy and paste the following code below to replace the stub class. The code has been pre-written for you. Be aware that the code is based on metadata that has been entered in this tutorial. If you have changed any tutorial metadata names, the necessary code for this will be different per your metadata name.

The following Java code also includes many code comments to explain the programming details. We suggest that you to read the code comments.

```
package com.app.tutorial.server.services.impl;

import java.util.Calendar;

import com.app.tutorial.model.defect.sdo.Defect;
import com.app.tutorial.model.lov.Priority;
import com.codeobjects.platform.data.common.TimeUtil;
import com.codeobjects.platform.data.common.query.QueryObject;
import com.codeobjects.platform.services.runtime.exceptions.ServiceException;

public class DefectInterceptorServiceImpl extends
AbstractDefectInterceptorServiceImpl
{
    @Override
    public void preDelete(Defect defect) throws ServiceException
    {
        // TODO Auto-generated method stub
    }

    @Override
    public void preInsert(Defect defect) throws ServiceException
    {
        // Set the defect's new due date based on the defect's priority.
        if (defect.getPriority() != null)
        {
            Calendar dueDate = this.decideDueDate(defect.getPriority());
            defect.setDueDate(dueDate);
            this._info(String.format("DefectInterceptorServiceImpl: the new due
date is set to %1$tm/%1$te/%1$tY", dueDate));
        }
    }

    @Override
    public void preQuery(QueryObject queryObject)
    {
        // TODO Auto-generated method stub
    }

    @Override
    public void preUpdate(Defect curDefect, Defect newDefect) throws
ServiceException
    {
        // Did the user change the existing priority?
        String curPriority = (curDefect.getPriority() != null) ?
curDefect.getPriority() : "";
    }
}
```

Update the Defect's Due Date Using Data Interceptor

```
String newPriority = (newDefect.getPriority() != null) ?
newDefect.getPriority() : "";

// If the priority is changed, set the defect's new due date
// based on the defect's priority.
if (!curPriority.equals(newPriority))
{
    Calendar dueDate = this.decideDueDate(newDefect.getPriority());
    newDefect.setDueDate(dueDate);
    this._info(String.format("DefectInterceptorServiceImpl: the defect
id(" + newDefect.getCoId() +
                                ") is set to the new due date to
%1$tM/%1$tE/%1$tY", dueDate));
}
}

@Override
public void postDelete(Defect defect)
{
    // TODO Auto-generated method stub
}

@Override
public void postInsert(Defect defect)
{
    // TODO Auto-generated method stub
}

@Override
public void postQuery(Defect defect)
{
    // TODO Auto-generated method stub
}

@Override
public void postUpdate(Defect defect)
{
    // TODO Auto-generated method stub
}

/**
 * To decide the defect's due date based on its priority.
 * <li>For priority '0-Urgent', the due date is 2 days.</li>
 * <li>For priority '1-High', the due date is 3 days.</li>
 * <li>For priority '2-Medium', the due date is 4 days.</li>
 * <li>Otherwise, the due date is set to 5 days.</li>
 *
 * @param defectPriority
 * @return the due date
 */
private Calendar decideDueDate(String defectPriority)
{
    // Use CodeObjects TimeUtil class to retrieve the current date time
    // in UTC.
    Calendar dueDate = TimeUtil.getUTCCalendar();
```

Update the Defect's Due Date Using Data Interceptor

```
// Access the LOV (List Of Values) generated in Java Enum class
if (defectPriority.equals(Priority.NUM_0MINURGENT.getValue()))
    dueDate.add(Calendar.DAY_OF_YEAR, 2);
else if (defectPriority.equals(Priority.NUM_1MINHIGH.getValue()))
    dueDate.add(Calendar.DAY_OF_YEAR, 3);
else if (defectPriority.equals(Priority.NUM_2MINMEDIUM.getValue()))
    dueDate.add(Calendar.DAY_OF_YEAR, 4);
else
    dueDate.add(Calendar.DAY_OF_YEAR, 5);

return dueDate;
}
```

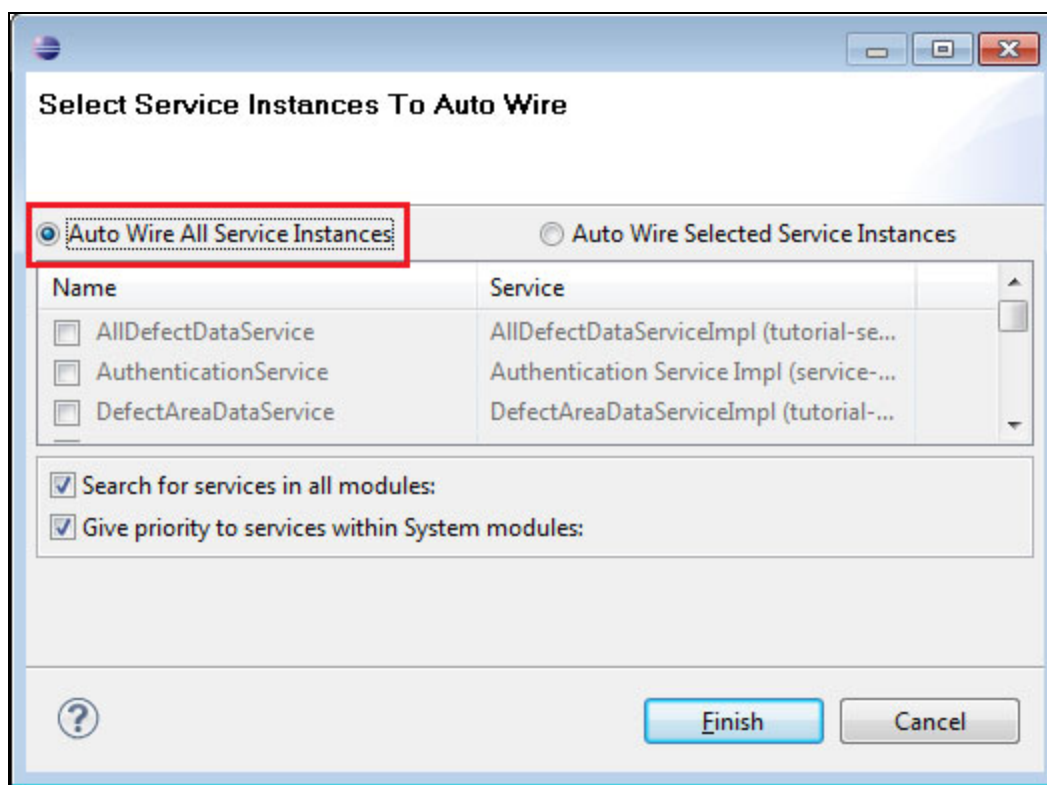
Programmer Notes

Code snippet above	Description
<code>public void preInsert(Defect defect)</code>	The variable defect is the defect record that the user created.
<code>public void preUpdate(Defect curDefect, Defect newDefect)</code>	The variable curDefect is the current defect object before the user's modifications, while the variable newDefect contains all the user's modifications.
<code>defect.getPriority()</code> <code>defect.setDueDate(dueDate)</code>	Access to field values is through calls to the getter methods. Setting field values is done through calls to the setter methods.
<code>this._info(...);</code>	This is a convenience method generated by CORADS in the abstract class to log messages of level INFO. Similarly, there are <code>_debug()</code> , <code>_warn()</code> , and <code>_exception()</code> logging functions defined in the abstract class.
<code>if</code> <code>(defectPriority.equals(Priori</code> <code>ty.NUM_0MINURGENT.getValue())</code> <code>)</code>	Access to the list of values for Priority is done through an enumeration.

Wire the Dependencies in the System File

Because you have created a new service, `DefectInterceptorServiceImpl`, you need to add it to `Tutorial.system` and re-wire.

- From the **tutorial-server** module, open the **Tutorial** system metadata. Click the **Auto Wire** button and select the **Auto Wire All Service Instances** option button so that the system takes care of everything for you. Notice that the system automatically brings in `DefectInterceptorServiceImpl`.



End Result

- Archive and re-deploy the web application. Login as **coadmin** instead of **couser**. This is because only **coadmin** can set the priority, per our previous tutorial.
- Using any existing record, change the priority and save the record (click the **Update** button .) Notice that the **Due Date** is updated per the logic included in the Java code. In this case, the interceptor event **pre-update** is fired.
- Create a new record and set the priority. The due date is populated, too. In this scenario, the interceptor event **pre-insert** is fired.

Update the Defect's Due Date Using Data Interceptor

NOTE

For existing records in the database, the system
does *not* go back to update each record one by one.

The screenshot shows the 'Defect Administration' interface. At the top, there are tabs for 'Defect' and 'Data Administration'. Below the tabs is a section titled 'All Defects' with a 'List of Defects' table. The table has columns: Defect Number, Assigned Date, Owned By, Created By, Closed Date, Due Date, Date Opened, and Priority. The first row shows Defect Number 9, Assigned Date 4/11/2011 10:20 AM, Owned By CO User, Created By couser, Closed Date 4/17/2011 11:18 AM, Due Date 4/11/2011 10:20 AM, Date Opened 4/11/2011 10:20 AM, and Priority 0-Urgent. Below the table is a detailed view for Defect Number 9. The 'Due Date' field is highlighted with a red box and shows '4/17/2011 11:18 / [calendar icon]'. The 'Priority' field is also highlighted with a red box and shows '0-Urgent'. Other fields include 'Last Updated Date' (4/15/2011 11:18 AM), 'Last Updated By' (coadmin), 'Defect Area' (Area 1), 'Severity' (4-Wrong behavior), 'Status' (Open), and 'Sub Status' (Open). The 'Description' field contains 'My 1st tutorial defect.' Below the detailed view is a table with two rows of defects: Defect Number 10 (Created By couser, Due Date 4/13/2011 5:49 PM, Priority 2-Medium) and Defect Number 11 (Created By coadmin, Due Date 4/19/2011 11:18 AM, Priority 2-Medium).

Defect Number	Assigned Date	Owned By	Created By	Closed Date	Due Date	Date Opened	Priority
9	4/11/2011 10:20 AM	CO User	couser	4/17/2011 11:18 AM	4/11/2011 10:20 AM	4/11/2011 10:20 AM	0-Urgent

Defect Number	9	Last Updated Date	4/15/2011 11:18 AM	Last Updated By	coadmin
Date Opened:	4/11/2011 10:20 AM	Defect Area:	Area 1	Severity:	4-Wrong behavior
Due Date:	4/17/2011 11:18 / [calendar icon]	Owned By:	CO User	Priority:	0-Urgent
Closed Date:	[calendar icon]	Closed By:		Status:	Open
Assigned Date:	[calendar icon]	Created By:	couser	Sub Status:	Open
Description:	My 1st tutorial defect.				
Customer Impact:					
Resolution:					

Defect Number	Assigned Date	Owned By	Created By	Closed Date	Due Date	Date Opened	Priority
10			couser			4/13/2011 5:49 PM	2-Medium
11			coadmin		4/19/2011 11:18 AM	4/15/2011 11:18 AM	2-Medium

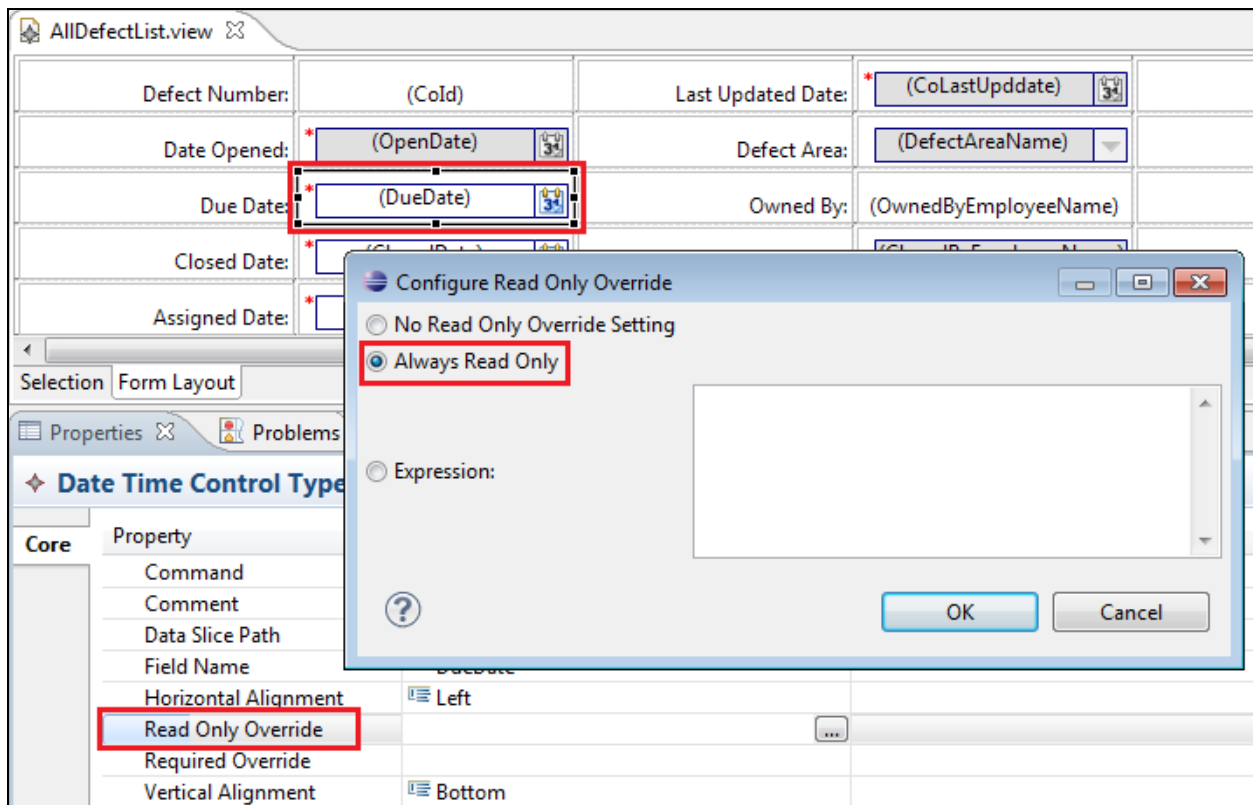
Drill down to the **Defect Detail** view through the **Defect Number** field and set the priority there. You should notice that the **Due Date** is also updated there.

Update the Defect's Due Date Using Data Interceptor

Due Date Is Editable in the UI

In the above screen image, the Due Date is editable for the end user, even if you have set the Due Date in your interceptor. Enforce a rule that Due Date should be read-only for the user and is always determined through the application logic in the interceptor. Go to where this field is displayed in the UI and override the read-only capability. Also do so in **Defect Details** view.

1. Open **AllDefectList** view and switch to **Form Layout**.
2. Click **Due Date** and change the property **Read Only Override** to **Always Read Only**.



Can you set this field to read-only from the entity side? No. After you set read-only to explicitly true, the field is always read-only, even if you want to set the value in your Java code. Archive, redeploy, and test your changes.

Update the Close Date and Closed By Info

This chapter explains how to update closing information fields.

In this chapter, we will demonstrate UI-side custom code to update **ClosedById** and **ClosedDate**.

- When a defect's **Status** is set to **Closed**, **ClosedDate** will update to the current date and time, and **ClosedById** will be set to the employee id of the current login user.

In order to update **ClosedById** field, you must determine how to get the employee id of the current login user. This is the foreign key to the employee record. You will implement a service to do this.

High Level Implementation

First you must implement a service to retrieve the employee id of the login user. For example, given the login "couser", you should find the user's employee id. Then, you must hook up the UI with the service that you implemented. Finally, you will add some UI custom code to set the fields **ClosedDate** and **ClosedById**.

Implement a Service Exception to Be Used with the Service to Be Written

Most modern programming languages include the notion of an *Exception* as a way to denote an error situation. Similarly, you can model such an exception in CORADS to create Exception metadata. Within Exception metadata, you can declaratively denote your Exception (error code, error message, details, and so on.) which later can be associated with your service interface metadata.

All Exception metadata will be converted to Java Exception. For example, if you have an exception metadata **DefectApplicationException**, it will be converted to Java exception.

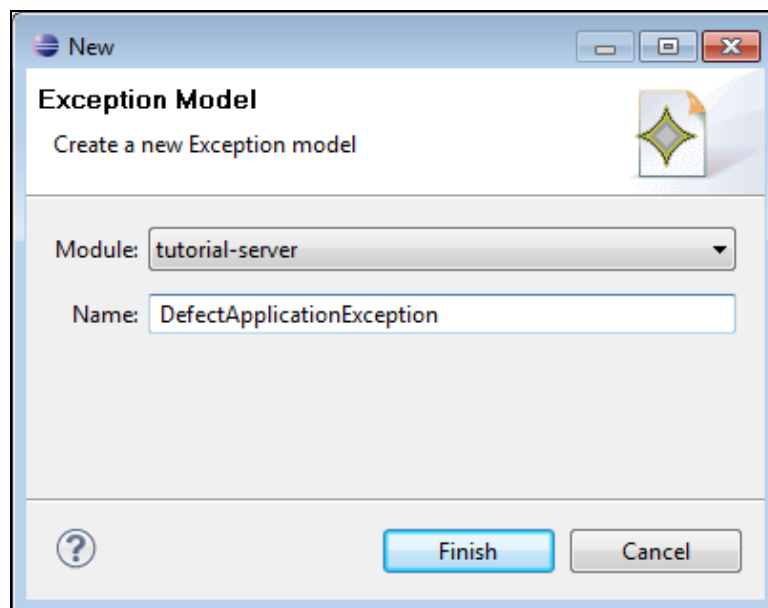
```
public class DefectApplicationException extends ServiceException
...
```

In the above snippet, **ServiceException** is the base class for all generated exception classes.

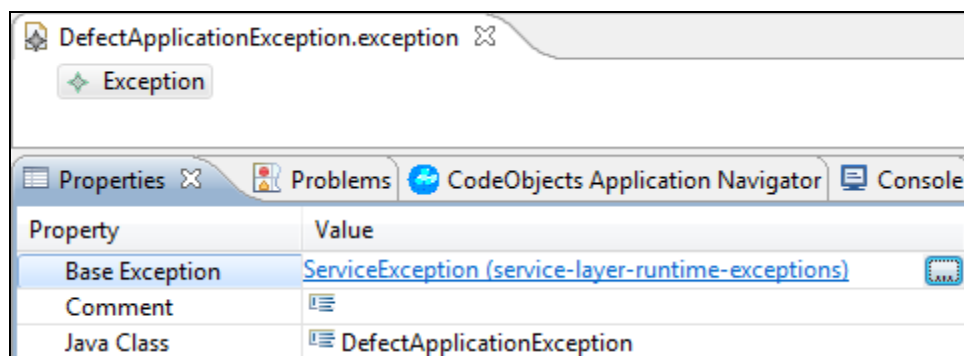
Note that it is not necessary that you always create your own Exception metadata to be used with your service. The guideline here is that if you expect the caller to handle the error from your service, or you want to manifest the list of error codes so that the caller has a clear idea of the cause of the error, we recommend that you have your own Exception metadata. Otherwise, consider using the standard ServiceException directly within your service.

Detailed Configuration for Exception

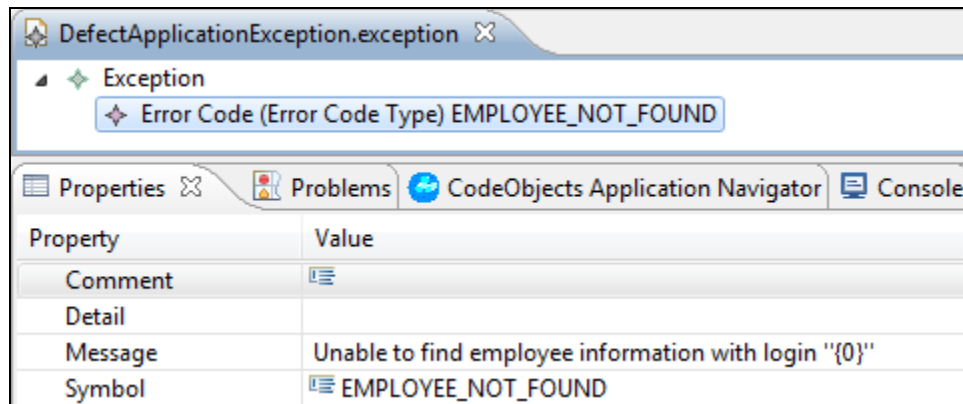
1. Create our own exception **DefectApplicationException**. In the CodeObjects metadata pane, right-click **tutorial-server** and choose **New** → **Services** → **Exception** to open the **Exception Model** dialog box. For **Name**, enter **DefectApplicationException**.



2. The metadata **DefectApplicationException.exception** is created. Set Base Exception to **ServiceException (service-layer-runtime-exceptions)**, which is supposed to be the base exception for any custom exception you create. Set Java Class to **DefectApplicationException**. This can be any name you like, but we recommend that you keep the same name as the metadata name.



3. Add a child node **Error Code (Error Code Type)**. You want to denote a specific error if given the login, the employee's id cannot be found. Set **Message** to **Unable to find employee information with login "{0}"** and **Symbol** to **EMPLOYEE_NOT_FOUND**.



You have successfully created Exception metadata.

Implement a Data Object to Search for the Employee Based on the Login Name

You will create a data object in the **tutorial-server** module, which you will use to find the user's employee id. You can create this data object either in the CodeObjects or Java perspective as explained in the first chapter in this tutorial.

This data object will be based on the **LoginUser** entity defined in the **component-model** module. Recall that in the earlier tutorials that when you created the **Employee** view, you were using **Employee** data object. The **Employee** data object is rooted in the **Employee** data slice with a one-to-many child data slice **LoginUser** entity that stores the user's login id and password. You should not plan to use this data object because you should query directly against the **Login** field in the **LoginUser** entity to search for the record with the matching login name (such as "couser"). Thus, you should create a new data object rooted in the **LoginUser** entity to support your query.

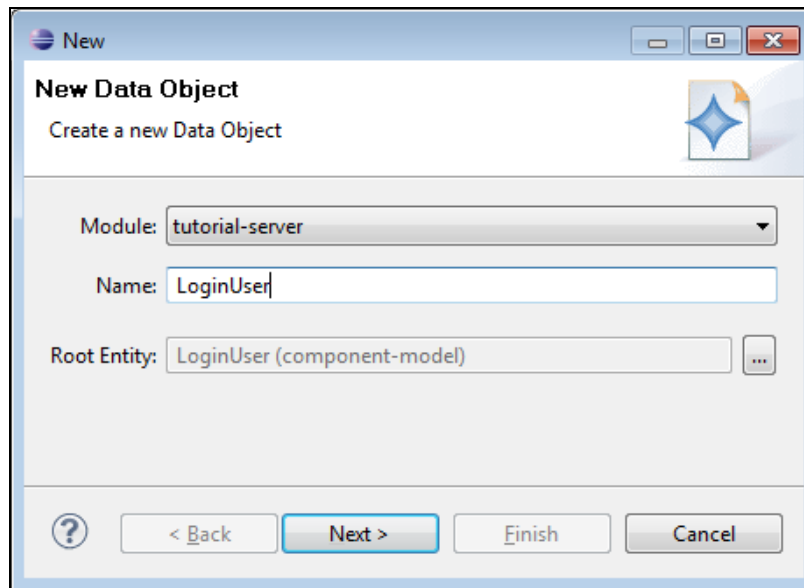
To summarize, the data object that you are about to create will be rooted in the **LoginUser** entity. You will query against it to find the employee record that has a matching login name (Login field in the LoginUser entity) and find its employee id (PartyId field in the LoginUser entity.)

This data object will be used later in your service implementation.

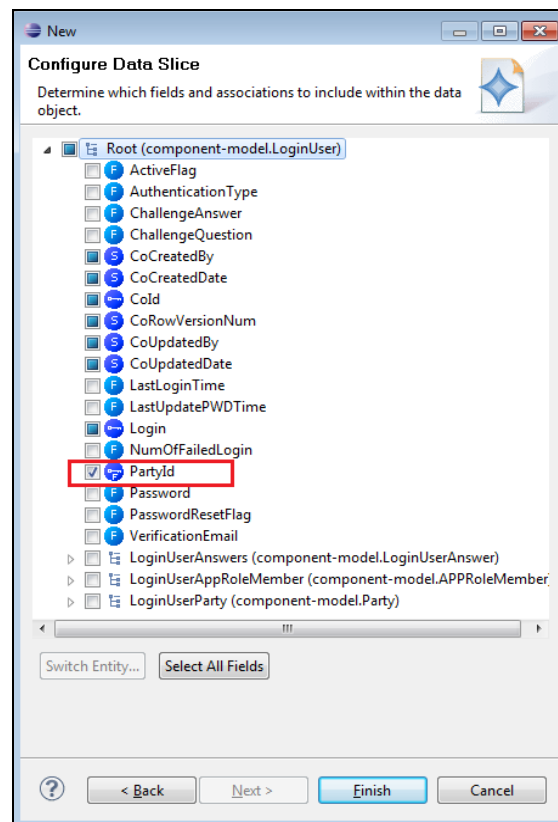
Detailed Configuration

In the **tutorial-server** module, create a data object named **LoginUser**. Make sure that the Root Entity is set to **LoginUser (component-model)**.

1. In the CodeObjects Metadata pane, right-click anywhere and choose **New → Data Access → Data Object**.



2. Click **Next**. Select the **PartyId** check box in addition to the default selected fields, and then click **Finish**.



Implement the Service Interface That Declares the Method Signatures

So far you have created an Exception metadata and a Data Object metadata. It is time to get into the service implementation.

You will start with service interface metadata, then service metadata, then service stub creation, and lastly implement the final Java code.

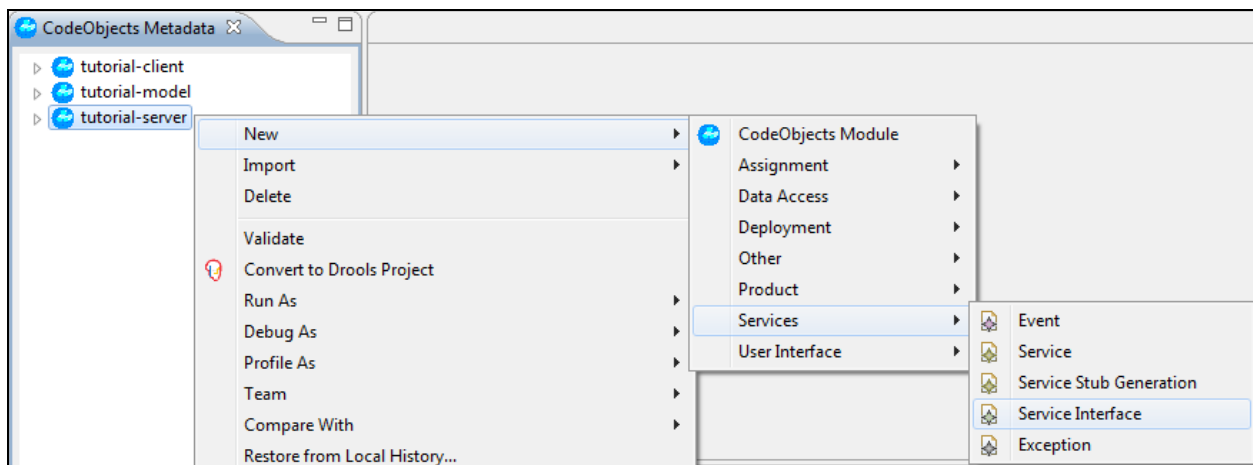
Detailed Configuration

Create a new service interface, **DefectApplicationService**. This service will implement a method that returns the login user's employee id.

You can create a service interface in either the CodeObjects or Java perspective.

To Create a Service Interface in Codeobjects Perspective

1. In the Metadata pane, click **tutorial-server** and choose **New** → **Services** → **Service Interface**.

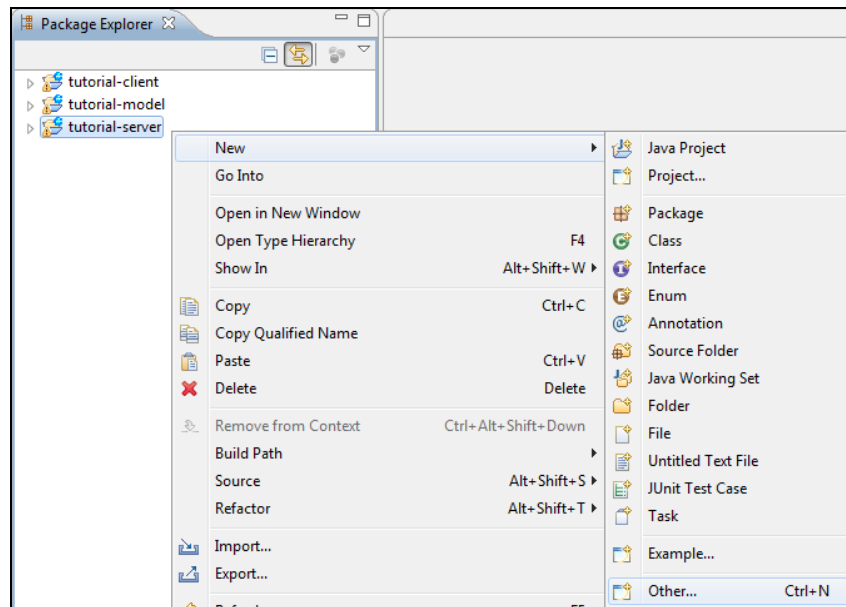


[Continue with step 2 after the description of the equivalent Java perspective step.]

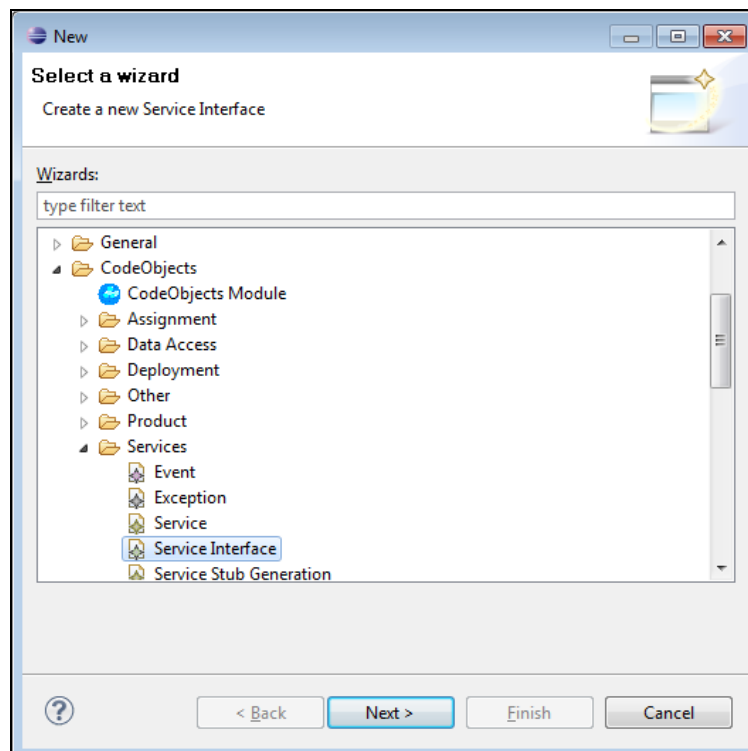
Update the Close Date and Closed By Info

To Create a Service Interface in Java Perspective

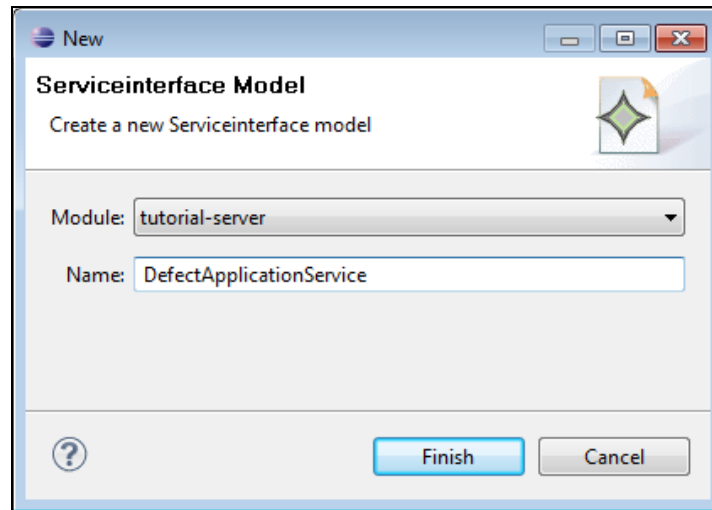
1. Right-click the tutorial-server module and choose **New** → **Other**.



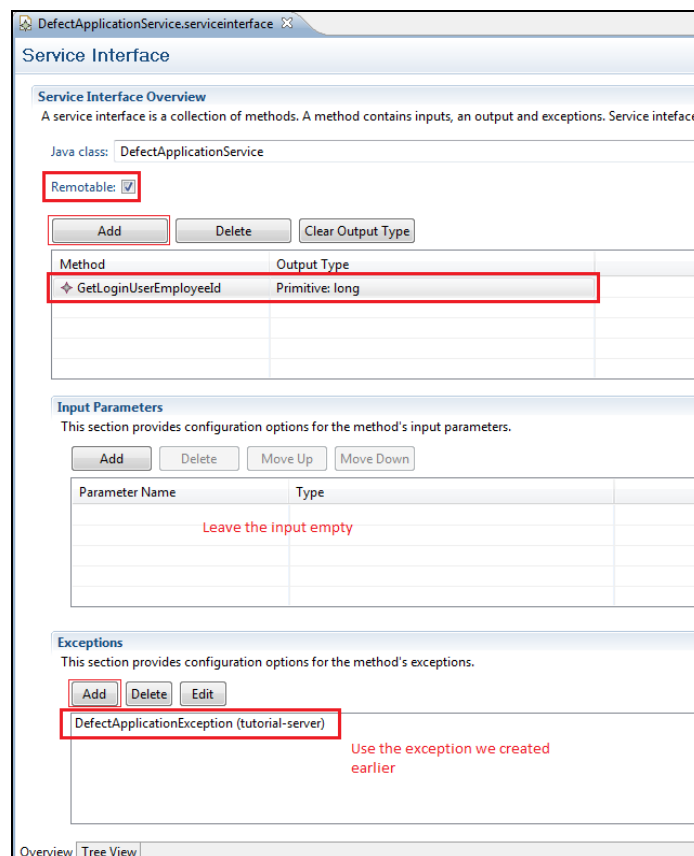
2. In the **Select a wizard** dialog box, under **CodeObjects** → **Services**, click **Service Interface**.



- Set the service **Name** to **DefectApplicationService**. Click **Finish**.



- In the **DefectApplicationService.serviceinterface** panel, click the **Remotable** check box to indicate that this service will be exposed as a web service. Add a **Method** of **GetLoginUserEmployeeId** with *no* input parameter. Set the **Output Type** to **Primitive: long** and under **Exceptions** add **DefectApplicationException**.



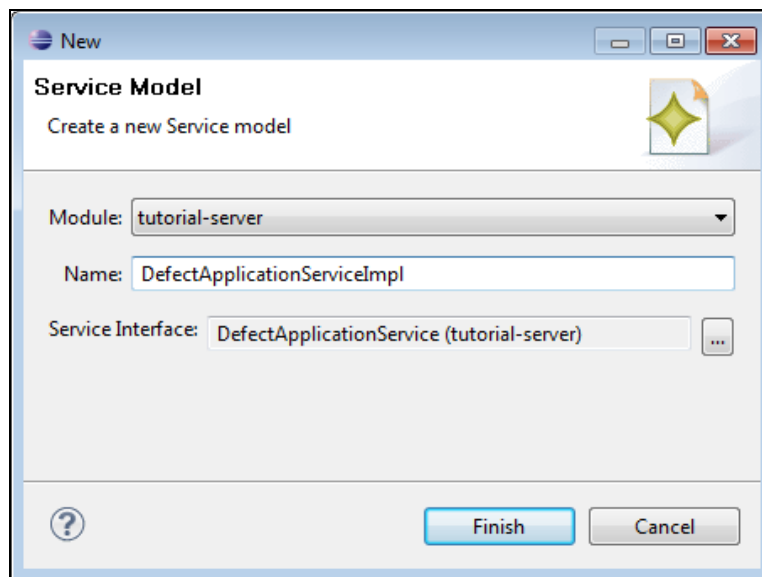
This is equivalent to declaring a Java interface.

```
public interface DefectApplicationService
{
    public long GetLoginUserEmployeeId() throws DefectApplicationException;
}
```

Implement a Service with Dependencies

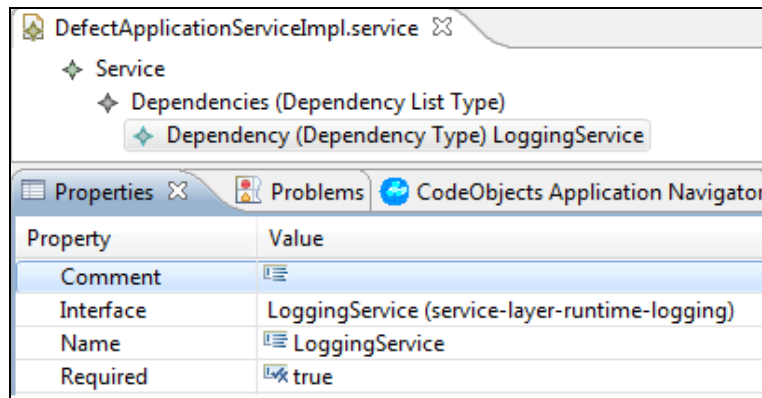
So far you have created a service interface. You must now create a Service to implement the service interface you created.

1. Create a Service in **CodeObjects** perspective or in **Java** perspective. This time choose **Service** instead of **Service Interface**.
2. Within the **Service Model** dialog box, set **Name** to **DefectApplicationServiceImpl** and **Service Interface** to **DefectApplicationService (tutorial-server)**. Click **Finish**. The naming convention of a service is typically Service Interface name + “Impl” keyword, as for example, DefectApplicationService for the service interface and DefectApplicationServiceImpl for the service.



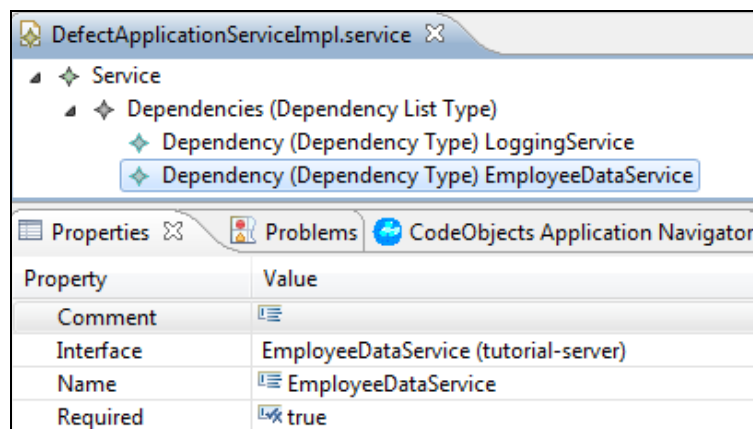
3. For **DefectApplicationServiceImpl.service**, add the child **Dependencies (Dependency List Type)** and add the child **Dependency (Dependency Type)**. Set the properties:
Interface to **LoggingService (service-layer-runtime-logging)**
Name to **LoggingService**
Required to **true**

Here you are adding a dependency to LoggingService (service-layer-runtime-logging). Logging is always a good practice. You will be leveraging this LoggingService from the service-layer-runtime-logging module to conduct application logging.



- To **Dependencies (Dependency List Type)**, add another child **Dependency (Dependency Type)** node. Set the properties:
Interface to EmployeeDataService (tutorial-server)
Name to EmployeeDataService
Required to true

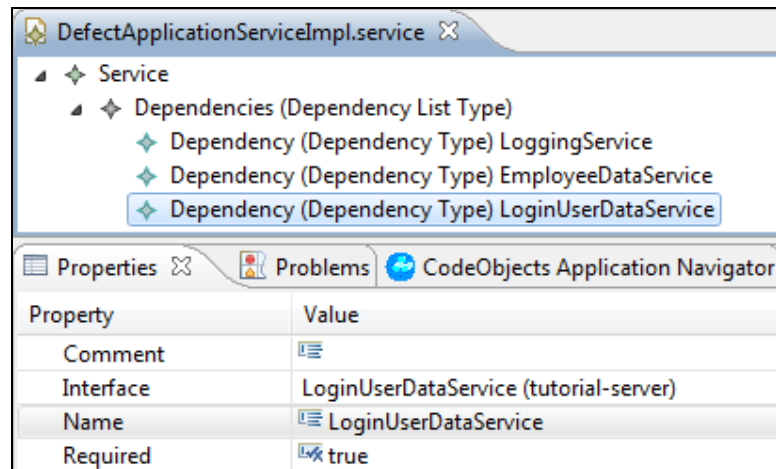
This service is produced from the **Employee** data object in the tutorial-server module. You will use this data object to show how to code the query using a query expression.



- To **Dependencies (Dependency List Type)**, add a third child **Dependency (Dependency Type)** node. Set the properties:
Interface to LoginUserDataService (tutorial-server)
Name to LoginUserDataService
Required to true

Update the Close Date and Closed By Info

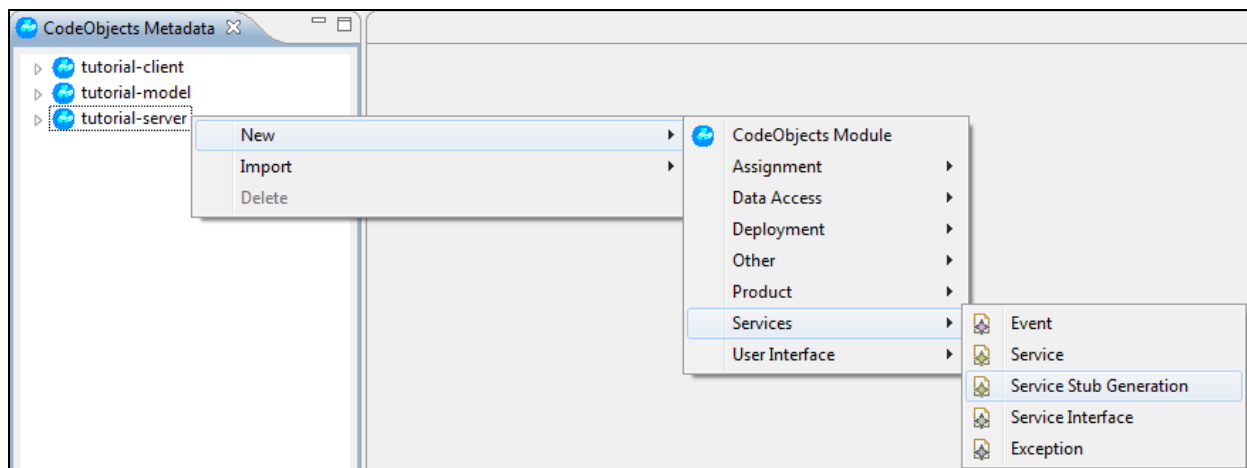
The LoginUserDataService service is produced from the LoginUser data object in the tutorial-server module. You will be using this data object to show how to query using the query object.



Implement the Service Stub to Write Code

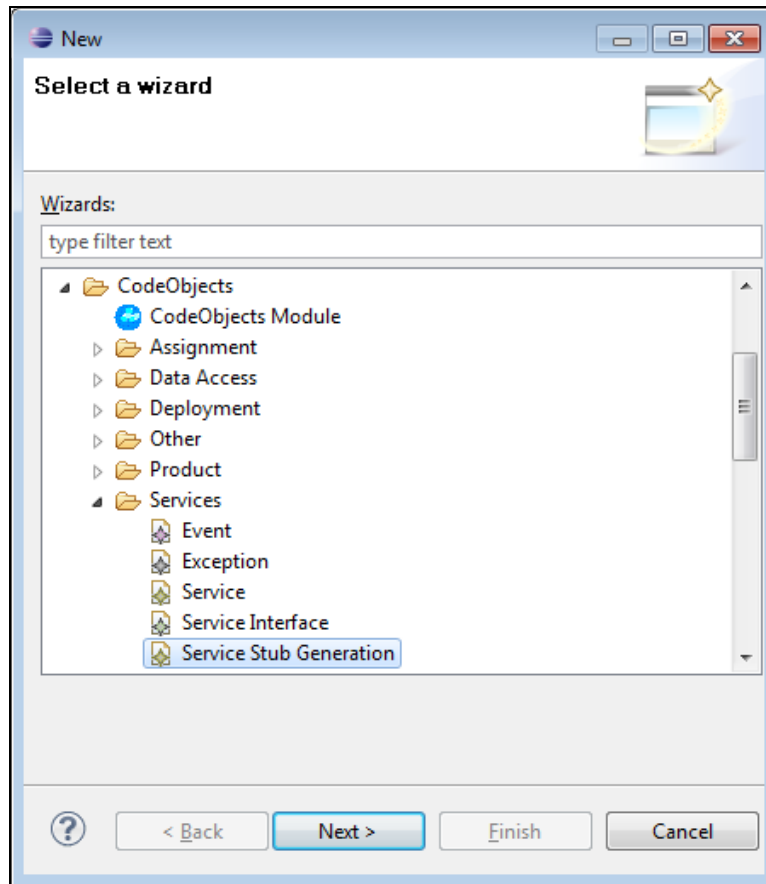
You must now create a service stub to hold the new Java code.

1. Create a Service Stub in the CodeObjects perspective.

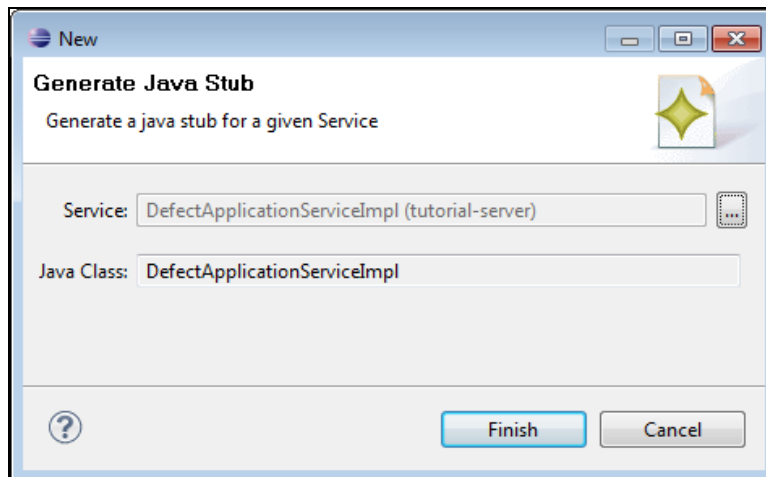


–or–

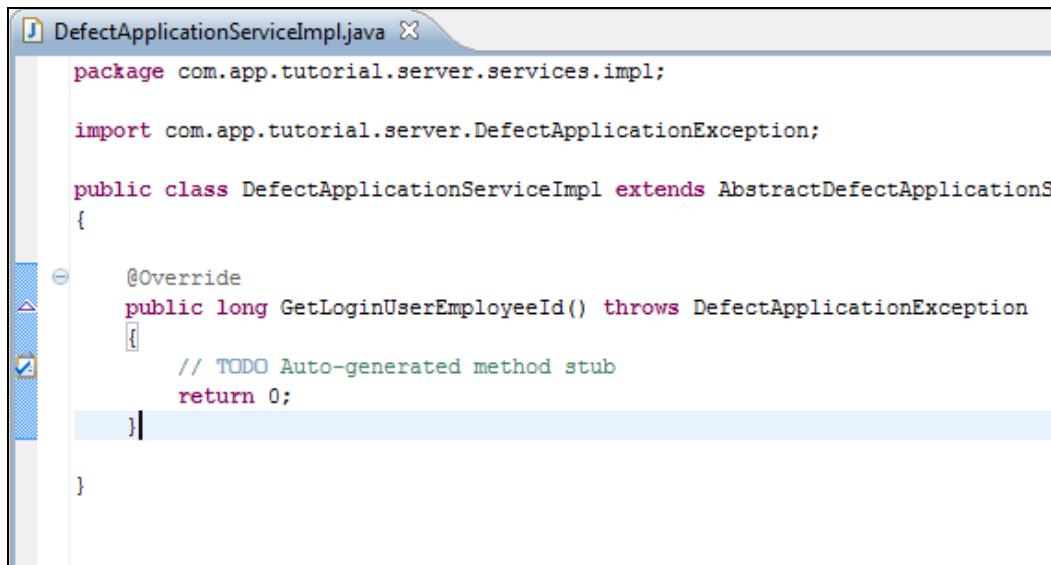
1. Create a Service Stub in the Java perspective.



2. Set **Service** to **DefectApplicationServiceImpl (tutorial-server)** and click **Finish**.



3. The Java stub class is created with an empty **GetLoginUserEmployeeId** method for entering the code.



4. Copy and paste to enter the code. See the code comments for a detailed explanation.

The following code demonstrates two ways to figure out the login user's employee id.

Option 1 as demonstrated in the method **GetLoginUserEmployeeId_UsingLoginUserDataObject()**, use the data object *LoginUser*. You implement this method using a query object.

Option 2 as demonstrated in the method **GetLoginUserEmployeeId_UsingEmployeeDataObject()**, reuse the data object *Employee*. You implement this method using a query expression because you want to filter the parent data slice *Employee* by matching the child data slice *LoginUsers*.

The following code comments out the second option and uses the first option. You can reverse the action to activate the second option to see that code in action.

```
package com.app.tutorial.server.services.impl;

import com.app.tutorial.server.DefectApplicationException;
import
com.app.tutorial.server.dataobject.employee.queryexpression.EmployeeQueryExpress
ion;
import com.app.tutorial.server.dataobject.employee.sdo.EmployeeRoot;
import com.app.tutorial.server.dataobject.loginuser.query.LoginUserQuery;
import com.app.tutorial.server.dataobject.loginuser.sdo.LoginUserRoot;
import com.codeobjects.component.model.employee.sdo.Employee;
import com.codeobjects.component.model.loginuser.sdo.LoginUser;
import com.codeobjects.platform.data.runtime.DataAccessServiceException;
```

```

import commonj.sdo.DataGraph;

public class DefectApplicationServiceImpl extends
AbstractDefectApplicationServiceImpl
{
    /**
     * Demonstrate the programming of query LoginUser or Employee based on the
     * user's login name
     * The user's login name can be found by calling _getUserName() method.
     *
     * Here we demonstrate 2 ways. The 1st way is a simple query against
     * LoginUser data object.
     * The 2nd way is to use query expression using SQL-like language to query.
     *
     * @return the employee id (party id) of the login user
     * @throws DefectApplicationException
     */
    @Override
    public long GetLoginUserEmployeeId() throws DefectApplicationException
    {
        // Class member method _getUserName() is generated by CodeObjects to
        // return the user's login name.
        String loginUser = this._getUserName();

        long employeeId = 0;

        try
        {
            // 1st way: query using LoginUser Data Object. This is to use query
            // object to query.
            employeeId =
            GetLoginUserEmployeeId_UsingLoginUserDataObject(loginUser);

            // 2nd way: query using Employee Data Object. This is to use query
            // expression to query.
            //employeeId =
            GetLoginUserEmployeeId_UsingEmployeeDataObject(loginUser);
        }
        catch (DataAccessServiceException ex)
        {
            throw new DefectApplicationException(ex.getErrorCode(), ex);
        }

        return employeeId;
    }

    /**
     * Demonstrate the use of query object to query LoginUser Data Object
     * @param loginUser the login name
     * @return
     * @throws DefectApplicationException
     * @throws DataAccessServiceException
     */
    private long GetLoginUserEmployeeId_UsingLoginUserDataObject(String
    loginUser)
    throws DefectApplicationException, DataAccessServiceException

```

```

    {
        // First to create a query object with the default paging size set in
        the metadata
        LoginUserQuery querySpec =
        LoginUserQuery.createQueryWithDefaultPaging();

        // Second to set our query criteria at the field level
        querySpec.setLogin(loginUser);

        // Since we do NOT expect more than 1 record, here we set the paging
        size to be 2. If
        // there are more than 1 record returns, we know we have duplicate
        records.
        querySpec.getPagingDetails().setPagingSize(java.math.BigInteger.valueOf(2));

        // Query for the record
        DataGraph data = this.getLoginUserDataService().query(querySpec);

        // Find the root object within the data graph
        LoginUserRoot root = (LoginUserRoot) data.getRootObject();

        // There should be only one record with such a login name. If not, we
        will throw an exception
        if (root.getLoginUser().size() != 1)
        {
            throw new
            DefectApplicationException(DefectApplicationException.EMPLOYEE_NOT_FOUND,
                                     new Object[] {loginUser});
        }

        // There is only 1 record, so the 1st element in the list is what we are
        looking for.
        LoginUser employee = root.getLoginUser().get(0);

        // Return its PartyId which is the employee id
        return employee.getPartyId();
    }

    /**
     * Demonstrate the use of query expression to query Employee Data Object
     * @param loginUser the login name
     * @return
     * @throws DefectApplicationException
     * @throws DataAccessServiceException
     */
    private long GetLoginUserEmployeeId_UsingEmployeeDataObject(String
    loginUser)
    throws DefectApplicationException, DataAccessServiceException
    {
        // First to create a query expression object with the default paging
        size set in the metadata
        EmployeeQueryExpression querySpec =
        EmployeeQueryExpression.createQueryWithDefaultPaging();

```

Update the Close Date and Closed By Info

```
// Set the 'where' clause using SQL-like language. Here we use EXISTS on
LoginUsers data slice within
// Employee Data Object. This SQL is to find the employee who has a
login name = loginUser.
querySpec.setWhereClause(
    String.format("EXISTS (LoginUsers WHERE (Login = '%1$s'))",
loginUser));

// Since we do NOT expect more than 1 record, here we set the paging
size to be 2. If
// there are more than 1 record returns, we know we have duplicate
records.
querySpec.getPagingDetails().setPagingSize(java.math.BigInteger.valueOf(2));

// Query for the record
DataGraph data =
this.getEmployeeDataService().queryExpression(querySpec);

// Find the root object within the data graph
EmployeeRoot root = (EmployeeRoot) data.getRootObject();

// There should be only one record with such a login name. If not, we
will throw an exception
if (root.getEmployee().size() != 1)
{
    throw new
DefectApplicationException(DefectApplicationException.EMPLOYEE_NOT_FOUND,
new Object[] {loginUser});
}

// There is only 1 record, so the 1st element in the list is what we are
looking for.
Employee employee = root.getEmployee().get(0);

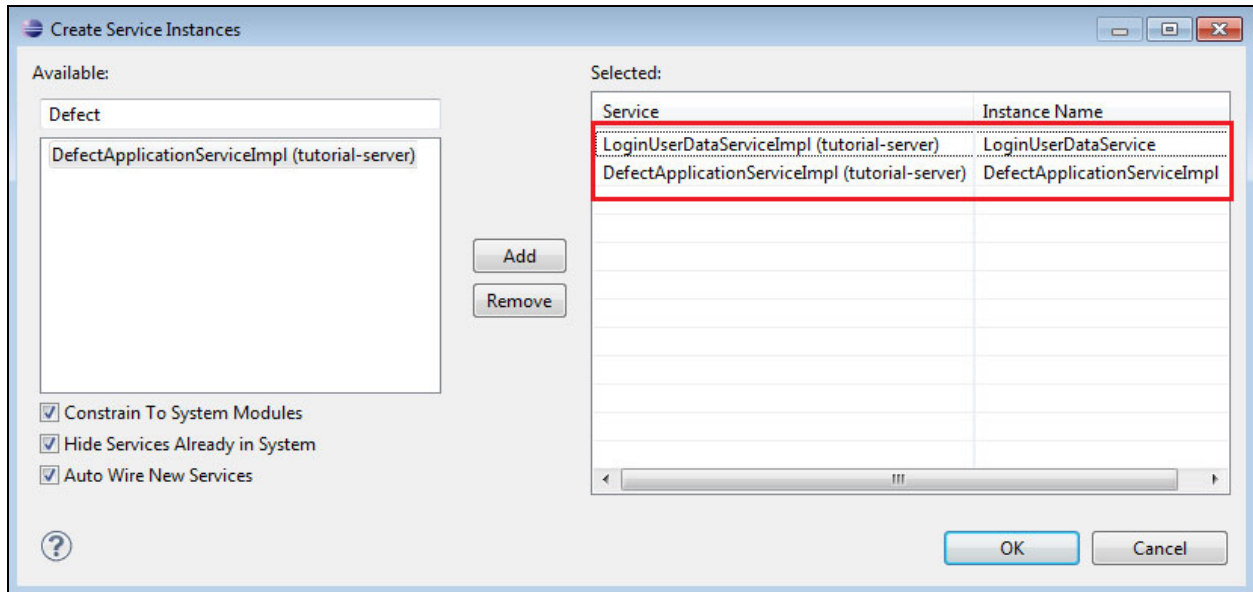
// Return its employee id
return employee.getEmployeeId();
}
}
```

Programmer Notes

Code snippet above	Description
<pre>String loginUser = this._getUserName();</pre>	All the generated service stub Java code will have this method <code>_getUserName()</code>, which returns the user's login name (such as 'couser') who invokes this service at runtime.
<pre>LoginUserQuery querySpec = LoginUserQuery.createQueryWithDefaultPaging(); querySpec.setLogin(loginUser); querySpec.getPagingDetails().setPagingSize(Java.math.BigInteger.valueOf(2));</pre>	The first line calls the static method to create a query object of class <code>LoginUserQuery</code> with the paging size default to whatever is defined in the metadata. It is always a good practice to use this method to initialize the paging detail and change it later as needed. The second line sets the query spec on field <code>Login</code>. The third line overrides the default paging size to 2. We expect there should be one and only one record returned. So we set the page size to 2 because if the return result set has 2 records we know there are 2 of the same login, which is invalid so we would throw out an exception.
<pre>DataGraph data = this.getLoginUserDataService().query(querySpec); LoginUserRoot root = (LoginUserRoot) data.getRootObject(); LoginUser employee = root.getLoginUser().get(0);</pre>	The first line queries the generated service from data object <code>LoginUser</code>. The return data is held within the data graph. The second line gets the root object, which contains all result records. The third line accesses the first <code>LoginUser</code> record. Here we expect that there is only one record returned because the user's login name should be unique. That is, there should not be two users with the same login "couser".
<pre>EmployeeQueryExpression querySpec = EmployeeQueryExpression.createQueryWithDefaultPaging(); querySpec.setWhereClause(String.format("EXISTS (LoginUsers WHERE (Login = '%1\$s'))", loginUser)); querySpec.getPagingDetails().setPagingSize(Java.math.BigInteger.valueOf(2));</pre>	The first line calls the static method to create a query expression object of class <code>EmployeeQueryExpression</code> with the paging size default set to whatever is defined in the metadata. The second line sets the where clause in the query expression. The query expression is using a SQL-like EXIST statement to find the <code>Employee</code> record whose <code>LoginUsers</code> records have a matching login. Note, the details of the query expression are not covered in this tutorial. Refer to the CORADS Developer Reference Guide for the details of query expression syntax. The third line overrides the default paging size to 2. We expect there should be one and only one record returned. We set the page size to be 2, because if the return result set has 2 records, we know that there are 2 of the same login, which is invalid, and thus we need to throw an exception.

Update the System File for Wiring

- In the **tutorial-server** module, open the **Tutorial** system metadata and add the two services that you have created so far, **LoginUserDataServiceImpl** (from the data object LoginUser) and **DefectApplicationServiceImpl**.



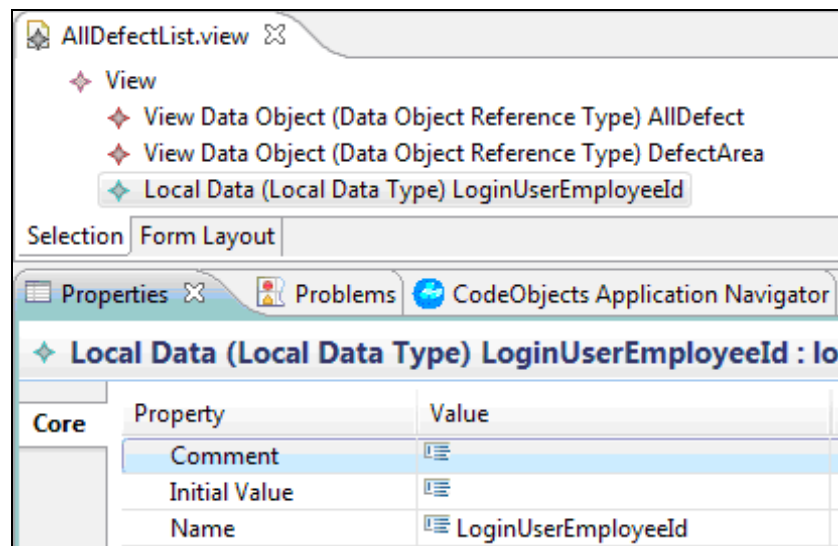
Hook up the UI with the Service

You now have a service that can find the employee id. Hook up the service with your view metadata so that the view can invoke the service to get the employee id of the login user.

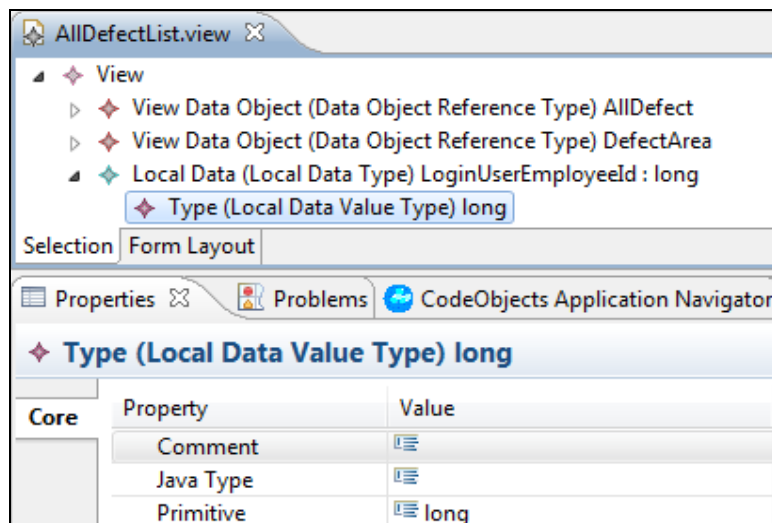
Detailed Configuration

When calling a service, you need a place to hold the output value of the service invocation. The output can either be stored into some field in a data object or be stored into a local data container. The latter is called **Local Data** in View metadata. In this example you will store the output value to local data.

1. In the **tutorial-client** module, open **AllDefectList.view**, and add a child node **Local Data (Local Data Type) LoginUserEmployeeId**. Set the **Name** to **LoginUserEmployeeId**.



2. Create a child node **Type (Local Data Value Type)** and set the property **Primitive** to **long**, which matches with the output of your service method (**GetLoginUserEmployeeId**).



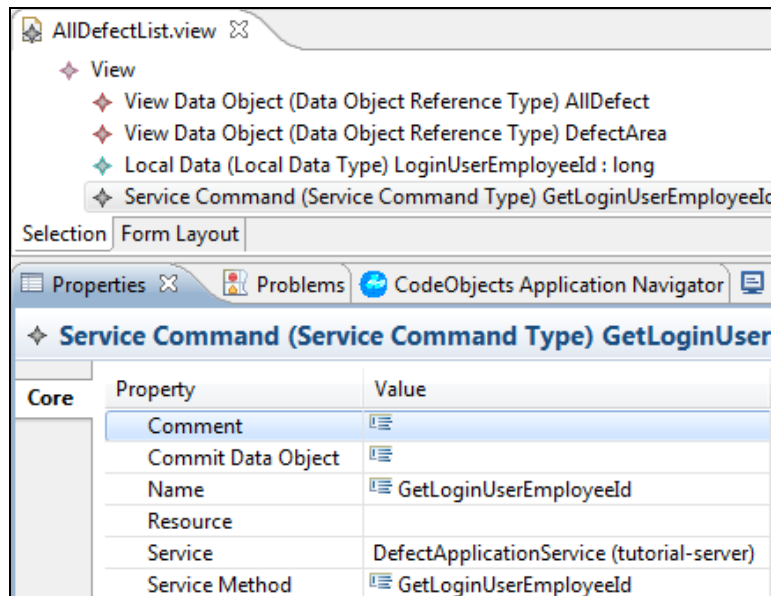
Add a service command to link to the service you implemented.

- To **View** add a child node **Service Command (Service Command Type)** and set the properties:

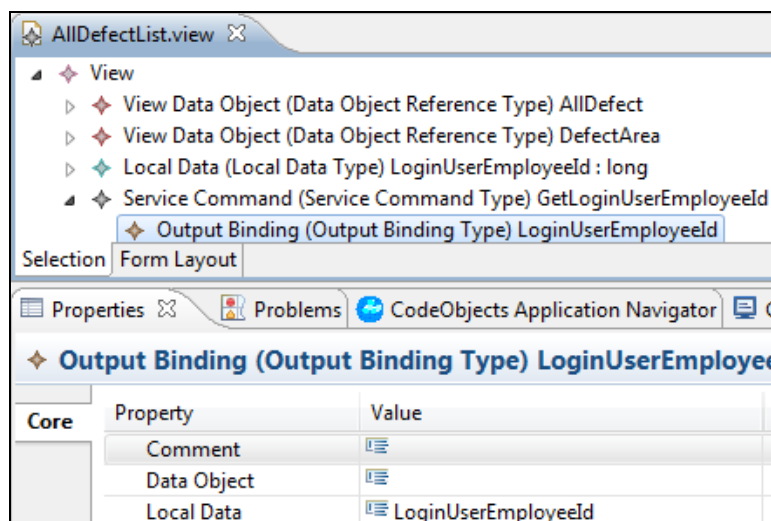
Name to **GetLoginUserEmployeeId**

Service to **DefectApplicationService (tutorial-server)**

ServiceMethod to **GetLoginUserEmployeeId**



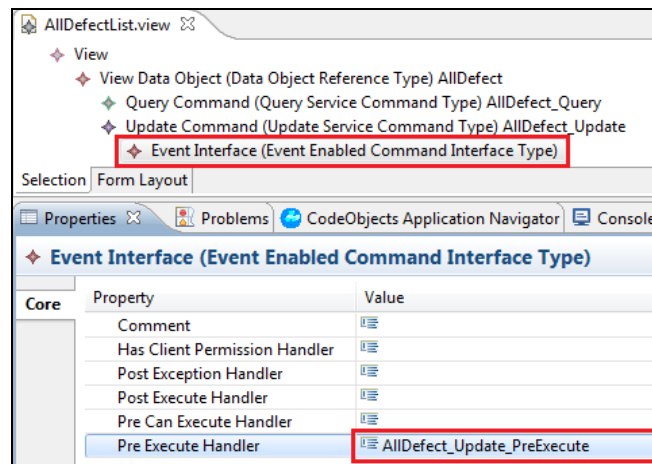
- To **Service Command...** add a child node **Output Binding (Output Binding Type)** and set the **Local Data** property to **LoginUserEmployeeId**, which was created earlier. Here we are saying, "store the output value of the service DefectApplicationService, method GetLoginUserEmployeeId to Local Data LoginUserEmployeeId". You will access this local data later in your custom code.



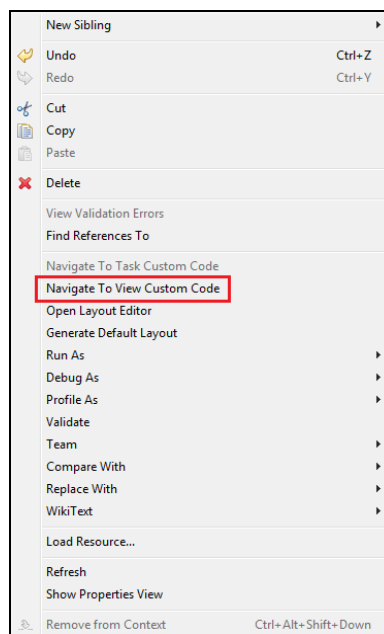
5. You must add a UI event for the update command before you save the record.

Here is the logic of this event: before you update the defect record and save to the database, you should scan the record to see if the user marked the defect as closed and as fixed. If yes, fetch the employee id by calling the service you implemented earlier and set the **ClosedById** field in the record.

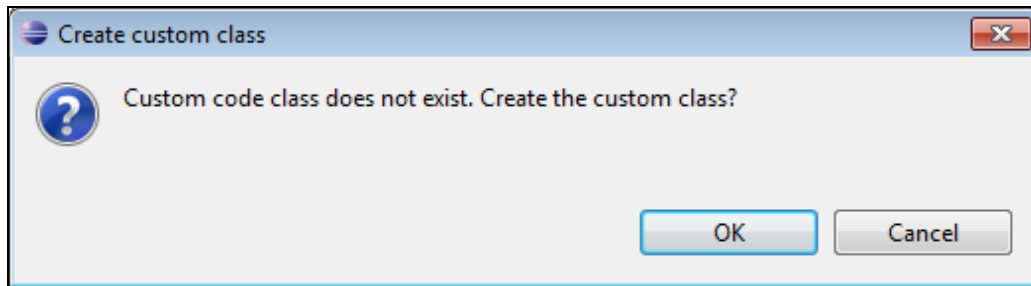
Under AllDefect's **View Data Object** node, right-click **Update Command (Update Service Command Type)**. Add a **New Child → Event Interface (Event Enabled Command Interface Type)** child node, and set the property **Pre Execute Handler** to **AllDefect_Update_PreExecute**. The naming convention is the command name (such as AllDefect_Update) + "_" + event name (such as PreExecute).



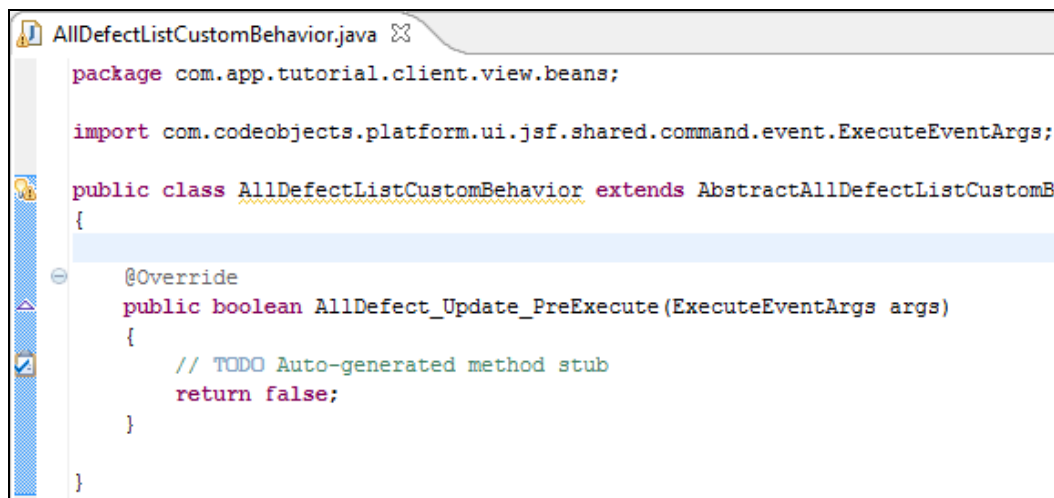
6. In the Explorer pane, right-click the new node **Event Interface** and choose **Navigate to View Custom Code**.



7. CORADS prompts to confirm the creation of the custom Java class. Click **OK**.



The stub Java class is created with the method that you declared for the event handler (**AllDefect_Update_PreExecute**).



Copy and paste the following code. See the code comments for a detailed explanation.

```

package com.app.tutorial.client.view.beans;

import java.util.List;

import com.app.tutorial.model.defect.sdo.Defect;
import com.app.tutorial.model.lov.DefectStatus;
import com.codeobjects.platform.data.common.DataServiceUtil;
import com.codeobjects.platform.data.common.TimeUtil;
import com.codeobjects.platform.ui.jsf.shared.command.event.ExecuteEventArgs;
import commonj.sdo.ChangeSummary;
import commonj.sdo.DataGraph;
import commonj.sdo.DataObject;

public class AllDefectListCustomBehavior extends
AbstractAllDefectListCustomBehavior
{

    public boolean AllDefect_Update_PreExecute(ExecuteEventArgs args)
    
```

Update the Close Date and Closed By Info

```
{
    // First to get the specific ViewModelBean object which contains
    // all strongly typed attributes and methods
    AllDefectListViewModelBean viewModel = (AllDefectListViewModelBean)
args.getViewModel();

    // Get the current defect record where the user is working on
    Defect defect = viewModel.getAllDefectDataModel().getCurrentRow();

    // Get the sdo data graph
    DataGraph data = defect.getDataGraph();

    // If the status is "Closed", check if the user explicitly changed it
    // to "Closed". We do NOT want to blindly do something extra
    if (defect.getStatus().equals(DefectStatus.CLOSED.getValue()) &&
        isDefectStatusChanged(data))
    {
        // Invoke the Service Command which is linked to the service
        // DefectApplicationService
        // When it finishes, per the Service Command metadata, the output
        // is stored in a local data
        viewModel.getServiceCommand_GetLoginUserEmployeeId().execute();

        // Retrieve the employee which is stored in local data
        // "LoginUserEmployeeId"
        long employeeId =
viewModel.getLocalDataManager().getLocalData_LoginUserEmployeeId();

        // Resume the change logging because we want to change 2 fields
        // in our code
        DataServiceUtil.resumeDataGraphLogging(data);

        // The closed data is set to today
        defect.setClosedDate(TimeUtil.getUTCCalendar());

        // Set the closed by id
        defect.setClosedById(employeeId);

        // End the data graph change
        DataServiceUtil.endDataGraphLogging(data);
    }

    return true;
}

/**
 * This function demonstrates how you can go through all the changes the
 * user did to the defect
 * record.
 * @param data the SDO data graph
 * @return
 */
boolean isDefectStatusChanged(DataGraph data)
{
    // Obtain the SDO change summary list which consists all sdo objects
    // which were modified.
```

Update the Close Date and Closed By Info

```
List changedObjects = data.getChangeSummary().getChangedDataObjects();
if (changedObjects == null || changedObjects.size() == 0)
    return false;

// Go through each changed sdo data object to make sure the sdo data
// object is what we are looking for
for (int i = 0; i < changedObjects.size(); i++)
{
    DataObject sdoDataObject = (DataObject) changedObjects.get(i);

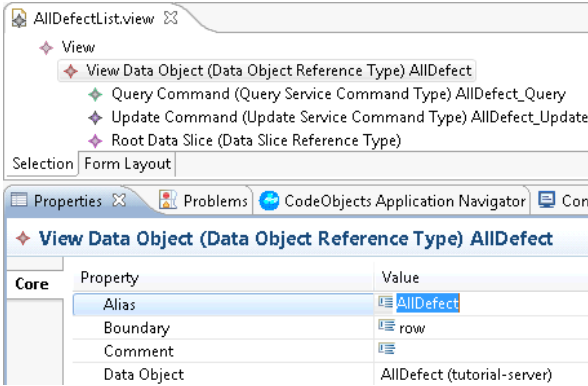
    // Make sure the sdo object is our Defect object
    if (sdoDataObject instanceof Defect)
    {
        // Given the sdo data object, fetch the old data values
        List oldValues =
data.getChangeSummary().getOldValues(sdoDataObject);

        // Iterate through each old value to see if field "Status" is
        // within the list. If yes, this means it was changed.
        for (int ix=0; ix<oldValues.size(); ix++)
        {
            ChangeSummary.Setting setting = (ChangeSummary.Setting)
oldValues.get(ix);
            String propertyName = setting.getProperty().getName();

            // Check if the 'Status' field is changed.
            // NOTE!! within sdo, the property name is always started
            // with lower case, even if the Entity field name starts
            // with capital. For example, field "Status" in Entity will
            // become "status" in sdo. Field "MyStatus" will become
            // "myStatus" in sdo.
            if (propertyName.equals("status"))
                return true;
        }
    }
}

return false;
}
```

Programmer Notes

Code snippet above	Description
<pre>AllDefectListViewModelBean viewModel = (AllDefectListViewModelBean) args.getViewModel();</pre>	Class AllDefectListViewModelBean is the generated programming interface from the view metadata AllDefectList view. From this class you can access the data object, local data, service commands, and so on, programmatically. The class name is named using the convention View's Java Class (such as AllDefectList) + ViewModelBean.
<pre>Defect defect = viewModel.getAllDefectData Model().getCurrentRow();</pre>	Access the data slice record: viewModel.getAllDefectDataModel() accesses the underlying root data slice in AllDefect data object. The term of "...DataModel.." here means the data slice. The naming convention is: the View Data Object's Alias + DataModel for the root data slice and the View Data Object's Alias + the data slice's class name + DataModel for non rooted data slice.  For example: Given the data object AllDefect in AllDefectList view: • To access the root data slice: use viewModel.getAllDefectDataModel() • To access the data slice of DefectArea: use viewModel.getAllDefect_DefectAreaDataModel() • To access the data slice of OwnedByEmployee: use viewModel.getAllDefect_OwnedByEmployeeDataModel() The method call getCurrentRow() accesses the current selected row in UI.
<pre>viewModel.getServiceCommand _GetLoginUserEmployeeId() .execute(); long employeeId = viewModel.getLocalDataMana ger().getLocalData_LoginUs erEmployeeId();</pre>	The first line invokes the service command GetLoginUserEmployeeId. All service command are prefixed with "ServiceCommand_" in the programming interface. This is why it is "getServiceCommand_GetLoginUserEmployeeId". The second line accesses the Local Data LoginUserEmployeeId through the generated local data manager class.

Update the Close Date and Closed By Info

```
DataServiceUtil.resumeData  
GraphLogging(data);
```

```
defect.setClosedDate(TimeU  
til.getUTCCalendar());  
defect.setClosedById(emplo  
yeeId);
```

```
DataServiceUtil.endDataGra  
phLogging(data);
```

The first line resumes the data graph logging. Resuming logging here means continuously adding new changes to the existing change summary.

Calling ChangeSummary's beginLogging() resets all existing changes and starts a new tracking log, which is not desired in this scenario.

End Result

Change the status and save the record. Note that the closed data and closed by fields are changed. Make sure you set **Status** to be **Closed** and **Sub Status** to be **Fixed**.

The screenshot shows the 'Defect Administration' interface. At the top, there are tabs for 'Defect' and 'Data Administration'. Below the tabs is a 'List of Defects' table. The table has columns: Defect Number, Assigned Date, Owned By, Created By, Closed Date, Due Date, and Date Opened. The first row shows Defect Number 9, Assigned Date 4/11/2011 10:20 AM, Owned By CO User, Created By couser, Closed Date 4/19/2011 6:32 PM, Due Date 4/18/2011 11:29 AM, and Date Opened 4/11/2011 10:20 AM. Below the table, there is a form for editing Defect Number 9. The form includes fields for Date Opened, Defect Area, Severity, Due Date, Owned By, Priority, Closed Date, Closed By, Status, Sub Status, Description, Customer Impact, and Resolution. The 'Closed Date' and 'Closed By' fields are highlighted in red, as are the 'Status' and 'Sub Status' dropdowns. The 'Status' dropdown is set to 'Closed' and the 'Sub Status' dropdown is set to 'Fixed'. The 'Description' field contains the text 'My 1st tutorial defect.'.

Defect Number	Assigned Date	Owned By	Created By	Closed Date	Due Date	Date Opened
9	4/11/2011 10:20 AM	CO User	couser	4/19/2011 6:32 PM	4/18/2011 11:29 AM	4/11/2011 10:20 AM
10			couser			4/13/2011 5:49 PM
11			coadmin		4/19/2011 11:18 AM	4/15/2011 11:18 AM

Change to a Different Way

Go back to the file where you implemented **DefectApplicationServiceImpl**, comment out the first way we that used to get the employee id and to use the second way instead. Archive and verify that the result is working.

```
public class DefectApplicationServiceImpl extends
AbstractDefectApplicationServiceImpl
{
    . . . .
    public long GetLoginUserEmployeeId() throws DefectApplicationException
    {
        // Class member method _getUserName() is generated by CodeObjects to
        return the user's login name.
        String loginUser = this._getUserName();

        long employeeId = 0;

        try
        {
            // 1st way: query using LoginUser Data Object. This is to use query
            object to query.
            //employeeId =
            GetLoginUserEmployeeId_UsingLoginUserDataObject(loginUser);

            // 2nd way: query using Employee Data Object. This is to use query
            expression to query.
            employeeId =
            GetLoginUserEmployeeId_UsingEmployeeDataObject(loginUser);
        }
        catch (DataAccessServiceException ex)
        {
            throw new DefectApplicationException(ex.getErrorCode(), ex);
        }

        return employeeId;
    }
}
```

Update the Close Date and Closed By Info Using UI Custom Code

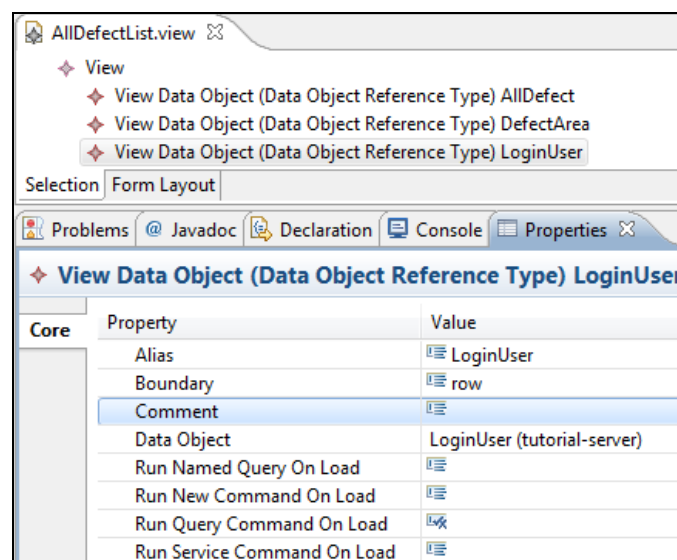
This chapter demonstrates how to use custom UI code to update closing information.

The previous chapter uses an implementation of a service to find the employee id of the login user. Now we want to demonstrate another way of programming, which is to do the same on the client UI side.

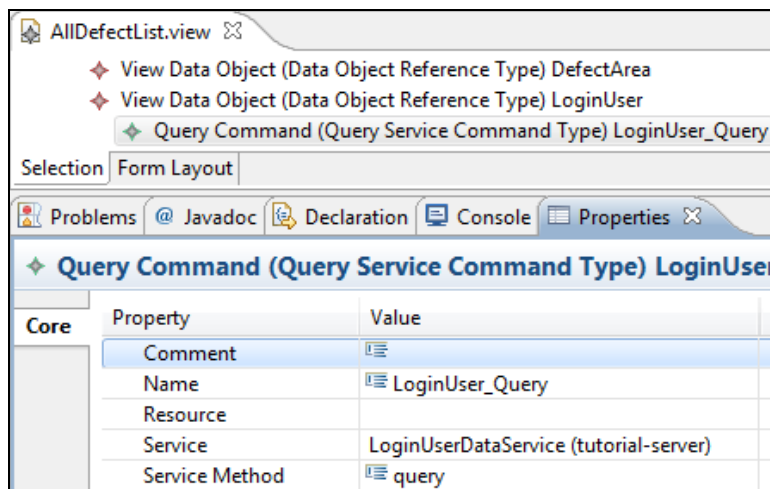
In this chapter, you will associate a data object to the View and do the same query through UI custom code.

Detailed Configuration

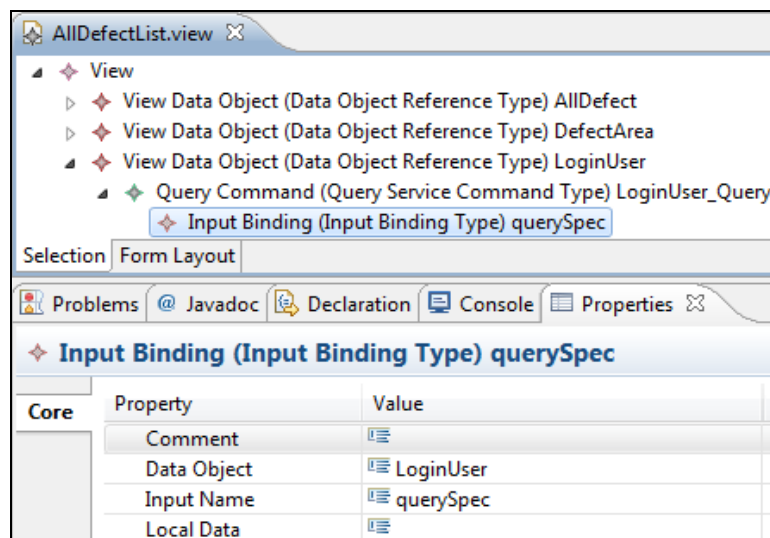
1. Open **AllDefectList.view** and add to the node **View, New Child → View Data Object (Data Object Reference Type)**. Set the properties: **Data Object** to **LoginUser** (**tutorial server**), **Alias** to **LoginUser**, and **Boundary** to **row**. Here you are associating the data object **LoginUser** to the view **AllDefectList**.



2. Create a **New Child** → **Query Command (Query Service Command Type)** and set the properties: **Service** to **LoginUserDataService (tutorial server)**, **Name** to **LoginUser_Query**, and **Service Method** to **query**. Here you are associating the generated data service LoginUserDataService with the data object LoginUser within the AllDefectList view.



3. Create a **New Child** → **Input Binding (Input Binding Type)** and set the properties: **Input Name** to **querySpec**, and **Data Object** to **LoginUser**. Here you define the input for the **query** method in the LoginUser data object.



4. Right-click the new **Input Binding...** node and choose **Navigate to View Custom Code** to see the custom code in **AllDefectListCustomBehavior.java**, which you added earlier.

5. Add the following code snippet to **AllDefectListCustomBehavior.Java**. This new method implements the way to query the **LoginUser** data object to retrieve the employee id of the login user. However, for the following snippet to work, you must add this import statement to the beginning of the file:

```
import com.codeobjects.platform.ui.jsf.shared.bean.ApplicationModelBean;
```

```
/**
 * Find the login user's employee id through calling the query command
 associated with
 * view data object "LoginUser"
 * @param viewModel the AllDefectListViewModelBean object
 * @return the employee id if found, otherwise -1
 */
private long getLoginUserEmployeeId(AllDefectListViewModelBean viewModel)
{
    // Retrieve the login user's login name (e.g. couser) from
ApplicationModelBean
    String loginUser = ApplicationModelBean.getLoginUsername();

    // Set it to the query object in view model
viewModel.getLoginUserQuery().setLogin(loginUser);

    // Execute the query command associated with the view data object
viewModel.getQueryCommand_LoginUser_Query().execute();

    // The return result is at the data model class
(viewModel.getLoginUserDataModel())
    // If there is only 1 record returns, the record will be automatically
    // positioned by UI view model to become the current selected row;
therefore
    // we can access it by using
viewModel.getLoginUserDataModel().getCurrentRow().
    return (viewModel.getLoginUserDataModel().getRowCount() == 1) ?
        viewModel.getLoginUserDataModel().getCurrentRow().getPartyId() :
        -1;
}
```

Update the Close Date and Closed By Info Using UI Custom Code

Programmer Notes

Code snippet above	Description
<code>String loginUser = ApplicationModelBean.getLoginUsername();</code>	Access the static method <code>getLoginUsername()</code> in the CodeObjects platform class <code>ApplicationModelBean</code> to return the user's login name (such as "couser").
<code>viewModel.getLoginUserQuery().setLogin(loginUser);</code>	<code>viewModel.getLoginUserQuery()</code> : Access the current query object for the LoginUser data object. The naming convention for this method is: "get" + View Data Object's Alias name (such as LoginUser) + "Query". <code>setLogin(loginUser)</code> : after we get the query object, set the Login field for the query criteria.
<code>viewModel.getQueryCommand_LoginUser_Query().execute();</code>	Invoke the query command for the data object LoginUser. The naming convention is "getQueryCommand" + " _ " + name of the query command (LoginUser_Query)
<code>viewModel.getLoginUserDataModel().getRowCount()</code>	Access the root data slice of LoginUser data object and get the row count.

Go back to the method in `AllDefectListCustomBehavior.java` where you call the service command. Comment out the call to the service command and add the new code highlighted below in yellow to use the new function.

NOTE

You must add the following import statement to the beginning of the file.

```
import com.codeobjects.platform.ui.jsf.shared.MessageHelper;
```

```
public class AllDefectListCustomBehavior extends
AbstractAllDefectListCustomBehavior
{
    public boolean AllDefect_Update_PreExecute(ExecuteEventArgs args)
    {
        // First to get the specific ViewModelBean object which contains all
        // strongly typed attributes and methods
        AllDefectListViewModelBean viewModel = (AllDefectListViewModelBean)
args.getViewModel();

        // Get the current defect record where the user is working on
        Defect defect = viewModel.getAllDefectDataModel().getCurrentRow();

        // Get the sdo data graph
        DataGraph data = defect.getDataGraph();

        // If the status is "Closed", check if the user explicitly changed it to
        "Closed".
        // We do NOT want to blindly do something extra
```

Update the Close Date and Closed By Info Using UI Custom Code

```
        if (defect.getStatus().equals(DefectStatus.CLOSED.getValue()) &&
            isDefectStatusChanged(data))
        {
            /* Comment out this section in order to exercise the code
            // Invoke the Service Command which is linked to the service
            DefectApplicationService
            // When it finishes, per the Service Command metadata, the output
            // is stored in a local data
            viewModel.getServiceCommand_GetLoginUserEmployeeId().execute();

            // Retrieve the employee which is stored in local data
            "LoginUserEmployeeId"
            long employeeId =
            viewModel.getLocalDataManager().getLocalData_LoginUserEmployeeId();
            */

            // Retrieve the employee id by going through the query command
            //associated with LoginUser view data object
            long employeeId = getLoginUserEmployeeId(viewModel);
            if (employeeId < 0)
            {
                // If not found, use MessageHelper class to display an error
                message on the top of view
                MessageHelper.addError(null, "Unable to identify the login
                user's employee id!");

                // Return false as a way to abort. By doing so, this record will
                NOT get saved to db.
                return false;
            }

            // Resume the change logging because we want to change 2 fields in
            our code
            DataServiceUtil.resumeDataGraphLogging(data);

            // The closed data is set to today
            defect.setClosedDate(TimeUtil.getUTCCalendar());

            // Set the closed by id
            defect.setClosedById(employeeId);

            // End the data graph change
            DataServiceUtil.endDataGraphLogging(data);
        }

        return true;
    }
}
```

End Result

Archive, redeploy, and test your changes. The behavior of the application should not change: the closed date and created by fields should be set when you update the status to Closed and Sub Status to Fixed.

Send Mail to the Defect Owner

This chapter demonstrates how to set a service to send email automatically.

In this chapter you will create a button for sending an email to the defect owner. You will add a new method to the existing service **DefectApplicationService** to implement the email construction and delivery. You will also leverage the out-of-box service **Send Mail Service** implemented by CodeObjects in the **component-send-mail** module to help you to send the mail.

The mail code depends on a template. Before sending the email, you will load the email template and replace the macro fields within the text with the real values before sending the mail out.

Our sample email template contains two macro fields: {DefectOwnerFullName}, {DefectId}

```
Dear {DefectOwnerFullName},
```

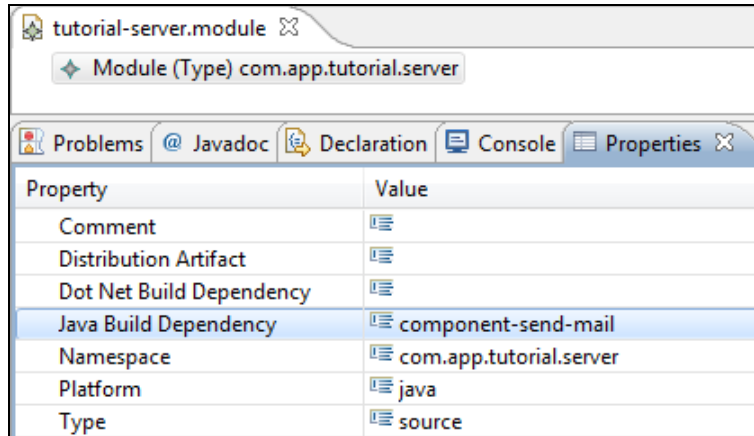
```
Please be advised that there is a new defect #{DefectId} being assigned to you.  
You can connect to the Defect Tracking Application for further details.
```

```
Thanks
```

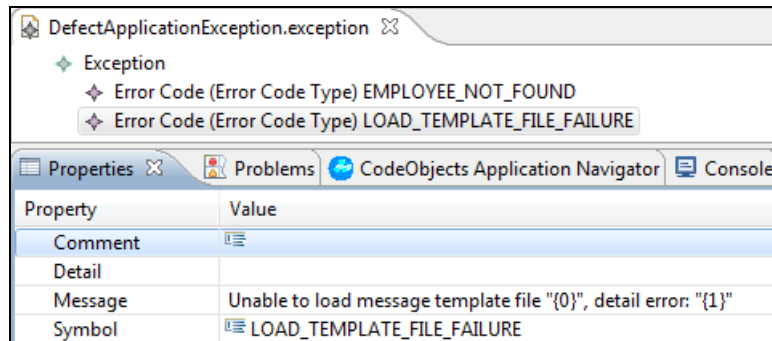
```
Company IT Department
```

Detailed Configuration

1. Open the **tutorial-server** module and add a module dependency to **component-send-mail**, because you will use the prebuilt email service to send the email.



2. To **DefectApplicationException.exception**, add a new exception error code to indicate any failure of loading the email template. You may need to add the **LOAD_TEMPLATE_FILE_FAILURE** resource manually.



3. Open the **DefectApplicationService** service interface metadata and add a new method **SendEmailToDefectOwner**. This method takes as input:

DefectOwnerFullName	the full name of the defect owner
DefectOwnerEmailAddress	the defect owner's email address
DefectId	the defect's id

The first two inputs are of type primitive: string. The final argument is primitive: long.

This method also throws two exceptions: **ServiceException** and **DefectApplicationException**.

This method has no output value. If it encounters a failure, an exception is thrown.

DefectApplicationService.serviceinterface

Service Interface

Service Interface Overview

A service interface is a collection of methods. A method contains inputs, an output and exceptions.

Java class: DefectApplicationService

Remotable: ☒

Add Delete Clear Output Type

Method	Output Type
GetLoginUserEmployeeId	Primitive: long
SendEmailToDefectOwner	

Input Parameters

This section provides configuration options for the method's input parameters.

Add Delete Move Up Move Down

Parameter Name	Type
DefectOwnerFullName	Primitive: string
DefectOwnerEmailAddress	Primitive: string
DefectId	Primitive: long

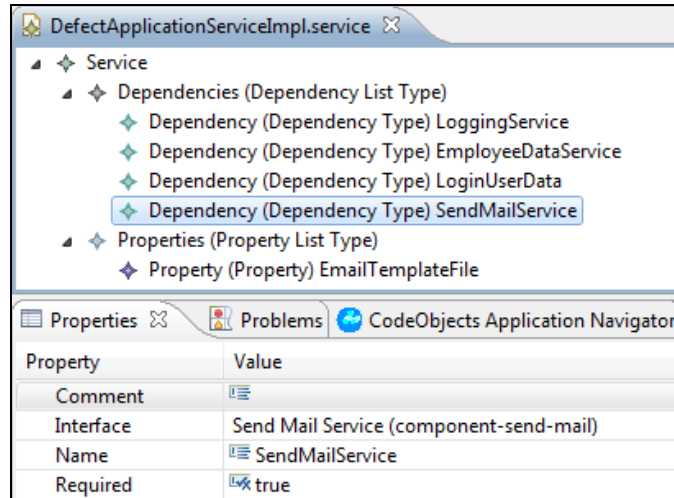
Exceptions

This section provides configuration options for the method's exceptions.

Add Delete Edit

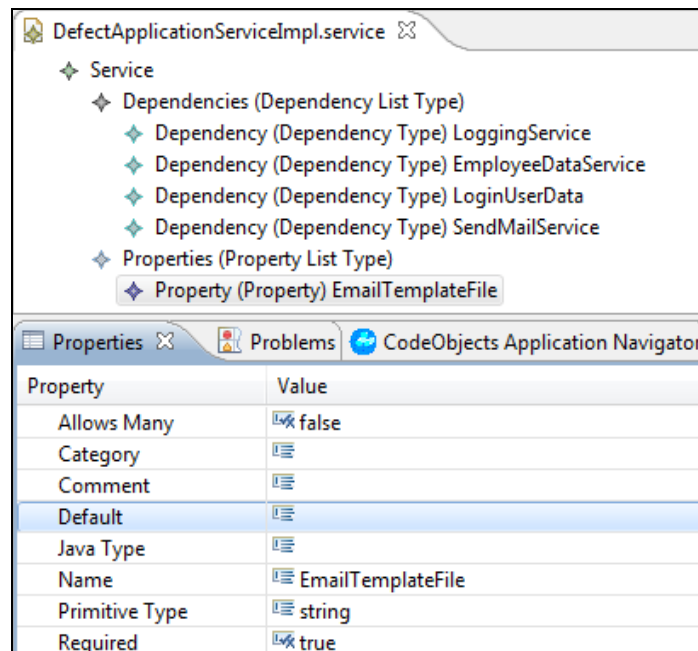
ServiceException (service-layer-runtime-exceptions)
DefectApplicationException (tutorial-server)

4. Open DefectApplicationServiceImpl service to add a dependency on the Send Mail Service from the component-send-mail module.



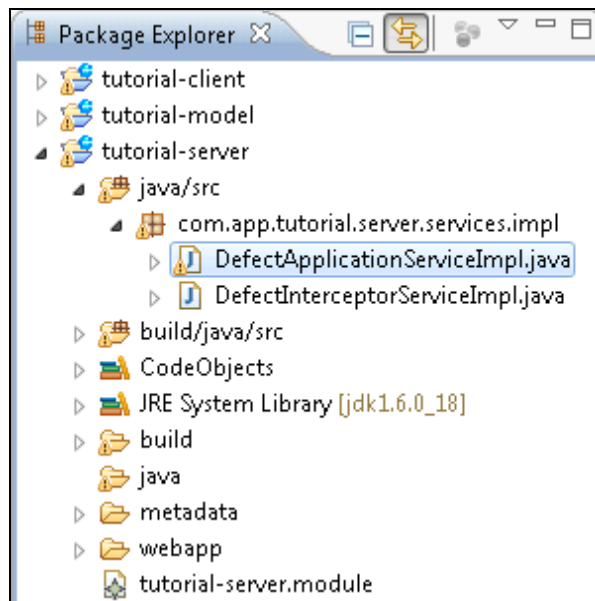
5. At the top of **DefectApplicationServiceImpl**, right-click **Service** and add **New Child** → **Properties (Property List Type)**. To the new node **Properties** add **New Child** → **Property (Property)**. Set these properties:
Name to **EmailTemplateFile**
Primitive Type to **string**
Required to **true**
Allows Many to **false**

This service property is a property that allows you to specify the location of the email template file.



Add Java Code

1. You must add Java code to implement the **SendEmailToDefectOwner** method. Open the service stub class **DefectApplicationServiceImpl.java**:
 - a. Switch to Java perspective.
 - b. Open the **tutorial-server** folder in the Package Explorer view, open the folder **java/src** to locate the file **DefectApplicationServiceImpl.java**, and double-click it to open.



2. You must add a global static variable, `EmailTemplateBody`, to cache the email template. Caching the template rather than re-reading it every time email is sent makes the software more efficient.

```
public class DefectApplicationServiceImpl extends
AbstractDefectApplicationServiceImpl
{
    // Cache the email template body to the memory for efficiency. Since this is
    // a static variable, later we
    // will use the synchronization control to guard it.
    private static String EmailTemplateBody = "";
```

3. Code a function to retrieve the email template. You will use Java synchronization to ensure that only one thread may load the email template's text into memory.

Note that you need to add these import statements to the beginning of the file to support the new method:

```
import java.io.BufferedReader;
```

```
import java.io.FileReader;
import java.io.FileNotFoundException;
import java.io.IOException;
```

```
/**
 * Use synchronization control to protect the setting of the static variable
 * 'EmailTemplateBody'.
 * This function will go to the file system to retrieve the email template
 * and cache in the memory.
 *
 * @param templateFileLocation
 * @return the email template body
 * @throws FileNotFoundException
 * @throws IOException
 */
private static synchronized String getEmailTemplate(String
templateFileLocation)
throws FileNotFoundException, IOException
{
    // If we have the email loaded already, we just return it.
    if (EmailTemplateBody != null && EmailTemplateBody.length() > 0)
        return EmailTemplateBody;

    // Read the email template from the disk
    StringBuffer fileData = new StringBuffer(2048);
    BufferedReader reader = new BufferedReader(new
FileReader(templateFileLocation));
    char[] buf = new char[2048];
    int numRead = 0;
    while((numRead = reader.read(buf)) != -1)
    {
        String readData = String.valueOf(buf, 0, numRead);
        fileData.append(readData);
    }
    reader.close();
    EmailTemplateBody = fileData.toString();

    return EmailTemplateBody;
}
```

4. Implement the **SendEmailToDefectOwner** method. It will get the email template and replace the macro field {DefectOwnerFullname} with the variable DefectOwnerFullname and the macro field {DefectId} with the variable DefectId. After this is done, the code uses the CodeObjects sendMailService module to do the actual mechanics of sending the email.

Note that you must add these import statements to the beginning of the file:

```
import com.codeobjects.component.email.SendMailServiceException;
import com.codeobjects.component.email.services.MailMessage;
import com.codeobjects.platform.services.runtime.exceptions.ServiceException;
```

Send Mail to the Defect Owner

```
/**
 * Send an email to the defect owner. The email body is using the template
 * available on the file
 * system. The template body will be cached into a memory string variable.
 * This service method
 * will also replace the variables within the email template with the real
 * values.
 * Available template variables are
 * <li> {DefectOwnerFullName} : the defect owner's full name</li>
 * <li> {DefectId} : the defect id </li>
 * @param DefectOwnerFullName the defect owner's full name
 * @param DefectOwnerEmailAddress the defect owner's email address
 * @param DefectId the defect id
 * @throws DefectApplicationException
 * @throws ServiceException
 */
public void SendEmailToDefectOwner(String DefectOwnerFullName, String
DefectOwnerEmailAddress, long DefectId)
    throws DefectApplicationException, ServiceException
{
    // First to retrieve the email template body from the cache
    String emailBody = null;
    try
    {
        emailBody = getEmailTemplate(this.getEmailTemplateFile());
        if (emailBody == null || emailBody.length() == 0)
            throw new
DefectApplicationException(DefectApplicationException.LOAD_TEMPLATE_FILE_FAILURE
,
        new Object[] { this.getEmailTemplateFile(), "the
template is empty" });
    }
    catch (Exception ex)
    {
        throw new
DefectApplicationException(DefectApplicationException.LOAD_TEMPLATE_FILE_FAILURE
,
        new Object[] { this.getEmailTemplateFile(), ex.getMessage()
}, ex);
    }

    // Second to replace the variables with the real value
    emailBody = emailBody.replaceAll("\\{DefectOwnerFullName\\}",
DefectOwnerFullName);
    emailBody = emailBody.replaceAll("\\{DefectId\\}",
Long.toString(DefectId));

    // Compose a real email construct
    MailMessage mail = new MailMessage();
    mail.setTo(DefectOwnerEmailAddress);
    mail.setSubject("Defect #" + DefectId + " is now assigned to you.");
    mail.setBody(emailBody);

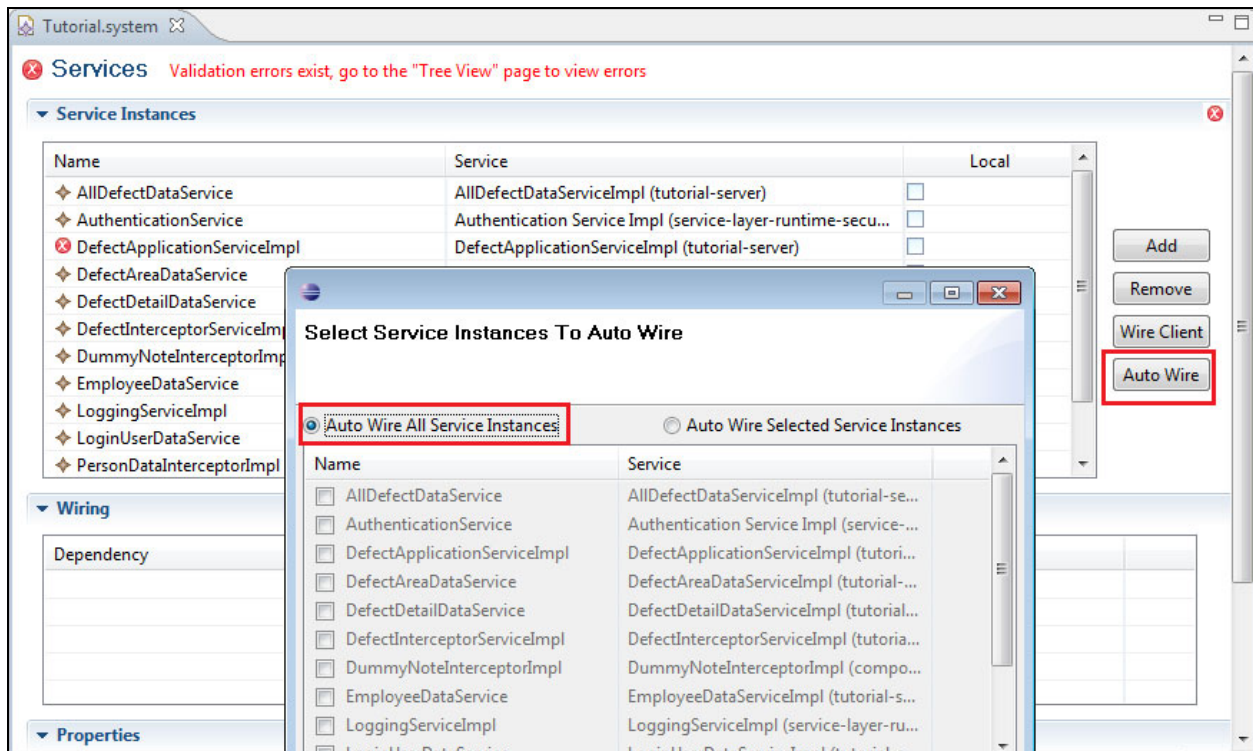
    // Optionally you can set your from: email address and from: display
    name in the code, or
```

Send Mail to the Defect Owner

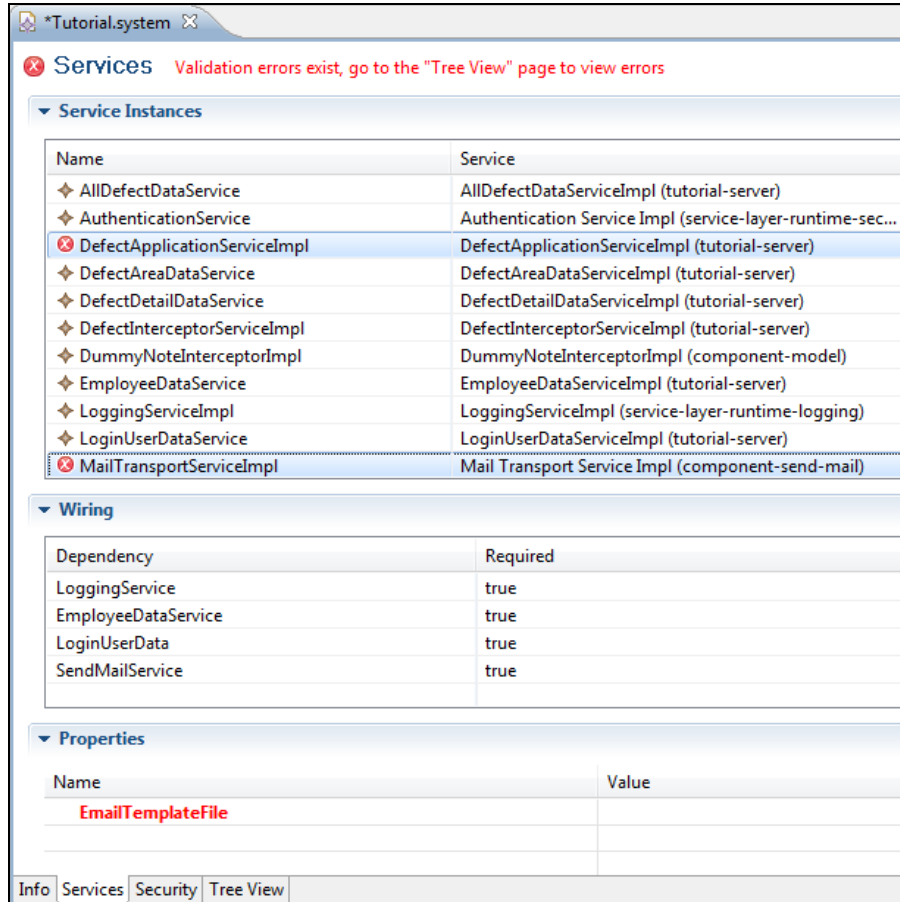
```
// you can set it in the service properties of SendMailService in
Tutorial.system.
// The difference is that if you would like to have a centralized place
for all outbound
// email, you can just configure it at SendMailService's service
properties. Otherwise you
// can set them at each individual calling service like this
DefectApplicationService
    //mail.setFrom("<<Your Email Address>>");
    //mail.setFromName("<<Your Display Name>>");

    // Send the email
    try
    {
        boolean outcome = this.getSendMailService().sendMail(mail);
        if (outcome)
            _info("Successfully sent a defect assignment email to " +
DefectOwnerEmailAddress);
        else
            _error("Failed to send a defect assignment email to " +
DefectOwnerEmailAddress);
    }
    catch (SendMailServiceException ex)
    {
        _error("Failed to send a defect assignment email to " +
DefectOwnerEmailAddress + ", error = " + ex.getMessage());
        _exception(ex);
        throw new ServiceException(ex.getMessage());
    }
}
```

5. Because you added a dependency on the Send Mail Service, do auto wiring. Open the **Tutorial** system file and click the **Auto Wire** button. Auto wire all the service instances.



Note that the service **MailTransportServiceImpl** was added and two services (DefectApplicationServiceImpl and MailTransportServiceImpl) are marked in red to indicate validation errors.



These are because of missing property values:

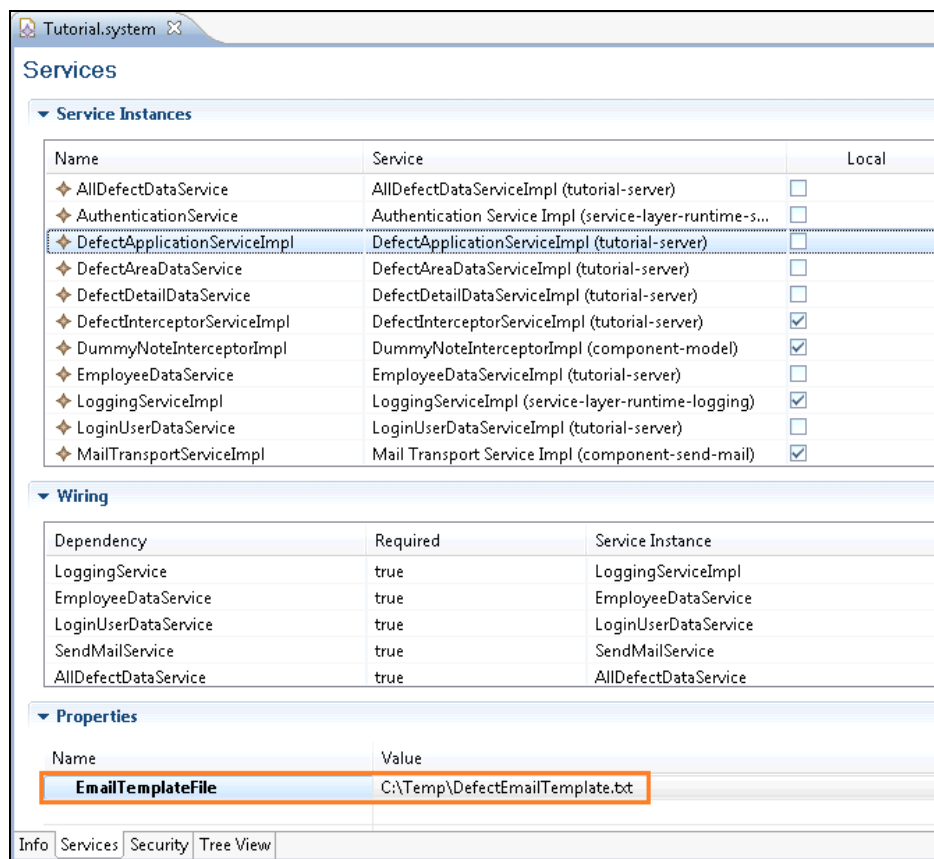
6. In **DefectApplicationServiceImpl.service**, set the property **EmailTemplateFile** to the location of the template in your file.
 - a. Copy the text below and save the email template as **C:\Temp\DefectEmailTemplate.txt**.
 - b. Open **Tutorial.system**. Set the property **EmailTemplateFile** for **DefectApplicationService** to **C:\Temp\DefectEmailTemplate.txt**.

Dear {DefectOwnerFullName},

Please be advised that there is a new defect #{DefectId} being assigned to you. You can connect to the Defect Tracking Application for further details.

Thanks

Company IT Department



For the **MailTransportServiceImpl** service you must set property values based on your local SMTP email server's configuration. Contact your IT staff for the right email server and values. Optionally, you can consider the use of Google's Gmail as your outbound email SMTP server. Below are the sample settings when using Gmail for outbound SMTP email.

Property Name	Property Value
connectionTimeout	120
Debug	false
enableAuthentication	true
enableSSL	true
enableTLS	false
Password	<<Your Password>>
smtpPort	465
smtpServer	smtp.gmail.com
username	<<Your Account>>

You must also set the **from** address for the outbound email. Select **SendMailService** and set the **from** and **fromName** properties to match with your email environment.

The screenshot shows a configuration window with three main sections:

- Service Selection:** A list of services with **SendMailService** highlighted in blue and enclosed in a red box. The list includes:

LoggingServiceImpl	LoggingServiceImpl (service-layer-runtime-logg..
LoginUserDataService	LoginUserDataServiceImpl (tutorial-server)
MailTransportServiceImpl	Mail Transport Service Impl (component-send-...
PersonDataInterceptorImpl	PersonDataInterceptorImpl (component-party-s..
SendMailService	Send Mail Service Impl (component-send-mail)
WebDAVFileStorageServiceImpl	WebDAVFileStorageServiceImpl (component-fil...
- Wiring:** A table showing dependencies:

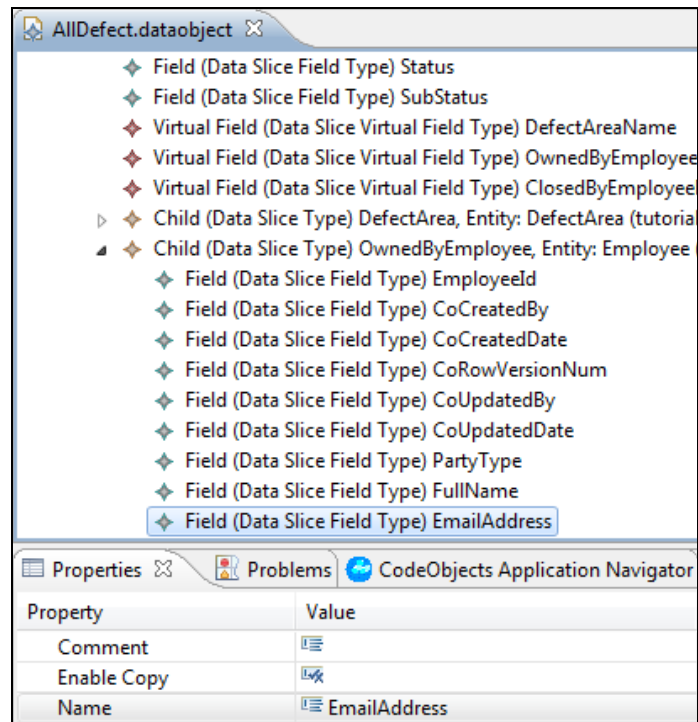
Dependency	Required
mailTransport	true
LoggingService	true
- Properties:** A table for setting properties, with **from** and **fromName** highlighted in a red box:

Name	Value
from	defect.tracking@mydomain.com
fromName	Defect Tracking
replyTo	
replyToName	

7. Add the **EmailAddress** to the **OwnedByEmployee** data slice in the **AllDefect** data object.

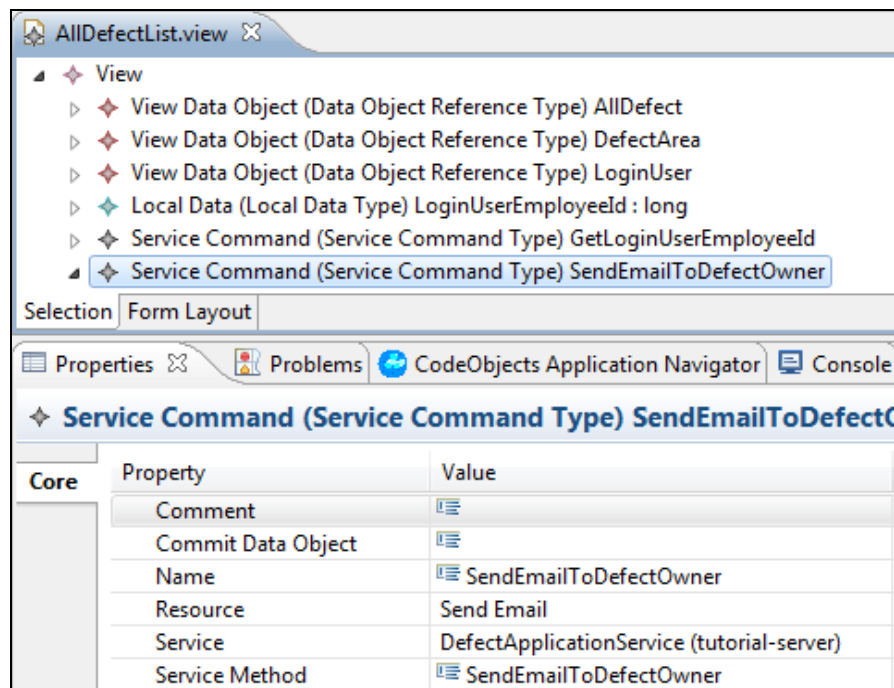
Recall that the service method **SendEmailToDefectOwner** has the input parameters of DefectOwnerFullName, DefectOwnerEmailAddress, and DefectId. DefectId is the field Cold in the Defect entity and DefectOwnerFullName is the **FullName** field in data slice OwnedByEmployee. You just need to provide the source of DefectOwnerEmailAddress. This is the **EmailAddress** field in the Employee entity.

8. In the **tutorial-server** module, open the **AllDefect** data object and add a child node **Field (Data Slice Field Type) EmailAddress** to the data slice **OwnedByEmployee**.

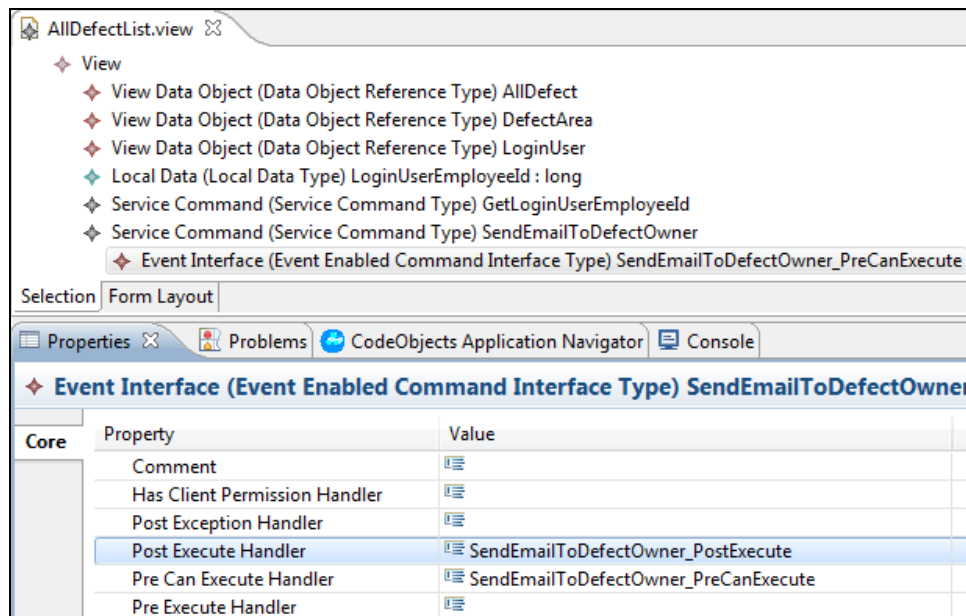


9. In the **tutorial-client** module, open **AllDefectList** view to add a service command. You must link the service that you implemented earlier to your UI so that it can invoke the service.

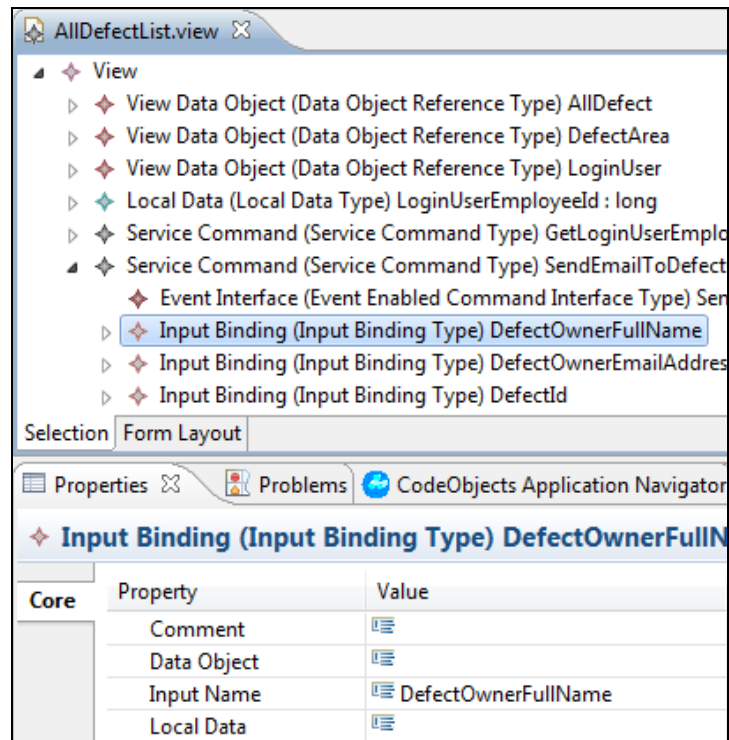
10. To the **View** node, add a child **Service Command (Service Command Type)**. Make sure to add a resource because later you will create a button to invoke this service command. The resource will be used as the button label.



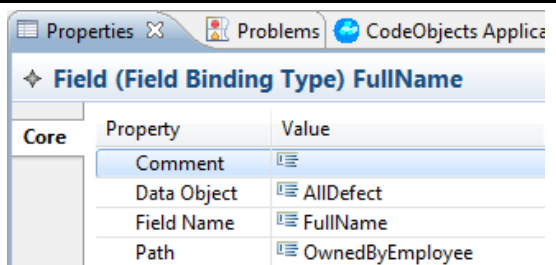
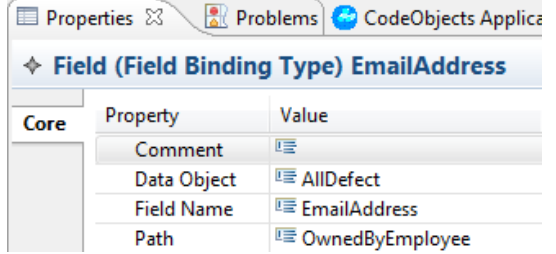
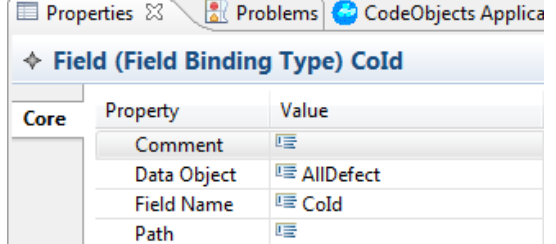
11. To the new node **Service Command**, add a child **Event Interface (Event Enabled Command Interface Type)** and set the event handlers for **Post Execute Handler** and **Pre Can Execute Handler** to **SendEmailToDefectOwner_PostExecute** and **SendEmailToDefectOwner_PreCanExecute** respectively.
 - The Post Execute Handler will implement application logic that displays a status message in the UI after email is sent. The Pre Can Execute Handler implements application logic to disable the button that calls the new service command if the defect is not assigned or has no owner.



12. To the new **SendEmailToDefectOwner** service command, add three child nodes of **Input Binding (Input Binding Type)**. Set the **Input Name** of the new nodes to **DefectOwnerFullName**, **DefectOwnerEmailAddress**, and **DefectId**, respectively.

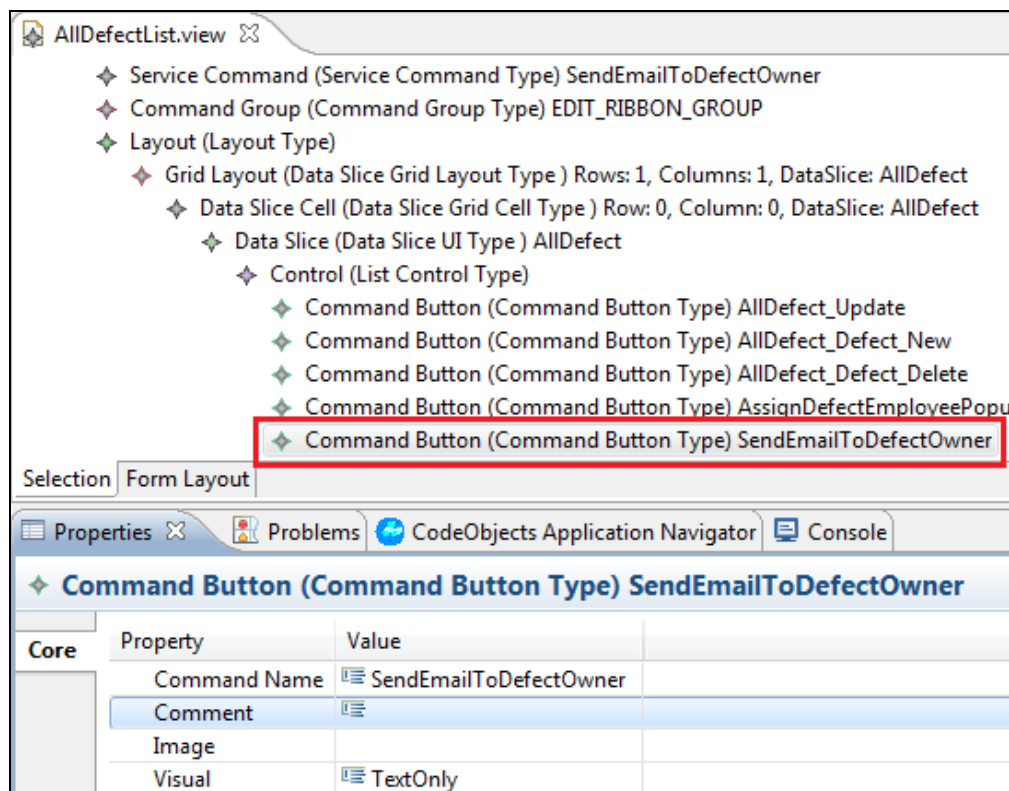


13. For each **Input Binding**, add a child **Field (Field Binding Type)** and set their properties. When you set the field binding types, set the data object first, the path second, and finally the field name. In the first example, you will not see the field name “Full Name” as a choice unless you follow this order.

Input Binding	Input Binding Properties										
- Service Command (Service Command Type) SendEmailToDefectOwner - Event Interface (Event Enabled Command Interface Type) SendEmail Input Binding (Input Binding Type) DefectOwnerFullName Field (Field Binding Type) FullName - Input Binding (Input Binding Type) DefectOwnerEmailAddress - Field (Field Binding Type) EmailAddress - Input Binding (Input Binding Type) DefectId - Field (Field Binding Type) Cold	 <table border="1"> <thead> <tr> <th>Property</th> <th>Value</th> </tr> </thead> <tbody> <tr> <td>Comment</td> <td></td> </tr> <tr> <td>Data Object</td> <td>AllDefect</td> </tr> <tr> <td>Field Name</td> <td>FullName</td> </tr> <tr> <td>Path</td> <td>OwnedByEmployee</td> </tr> </tbody> </table>	Property	Value	Comment		Data Object	AllDefect	Field Name	FullName	Path	OwnedByEmployee
Property	Value										
Comment											
Data Object	AllDefect										
Field Name	FullName										
Path	OwnedByEmployee										
- Service Command (Service Command Type) SendEmailToDefectOwner - Event Interface (Event Enabled Command Interface Type) SendEmail - Input Binding (Input Binding Type) DefectOwnerFullName - Field (Field Binding Type) FullName Input Binding (Input Binding Type) DefectOwnerEmailAddress Field (Field Binding Type) EmailAddress - Input Binding (Input Binding Type) DefectId - Field (Field Binding Type) Cold	 <table border="1"> <thead> <tr> <th>Property</th> <th>Value</th> </tr> </thead> <tbody> <tr> <td>Comment</td> <td></td> </tr> <tr> <td>Data Object</td> <td>AllDefect</td> </tr> <tr> <td>Field Name</td> <td>EmailAddress</td> </tr> <tr> <td>Path</td> <td>OwnedByEmployee</td> </tr> </tbody> </table>	Property	Value	Comment		Data Object	AllDefect	Field Name	EmailAddress	Path	OwnedByEmployee
Property	Value										
Comment											
Data Object	AllDefect										
Field Name	EmailAddress										
Path	OwnedByEmployee										
- Service Command (Service Command Type) SendEmailToDefectOwner - Event Interface (Event Enabled Command Interface Type) SendEmail - Input Binding (Input Binding Type) DefectOwnerFullName - Field (Field Binding Type) FullName - Input Binding (Input Binding Type) DefectOwnerEmailAddress - Field (Field Binding Type) EmailAddress Input Binding (Input Binding Type) DefectId Field (Field Binding Type) Cold	 <table border="1"> <thead> <tr> <th>Property</th> <th>Value</th> </tr> </thead> <tbody> <tr> <td>Comment</td> <td></td> </tr> <tr> <td>Data Object</td> <td>AllDefect</td> </tr> <tr> <td>Field Name</td> <td>Cold</td> </tr> <tr> <td>Path</td> <td></td> </tr> </tbody> </table>	Property	Value	Comment		Data Object	AllDefect	Field Name	Cold	Path	
Property	Value										
Comment											
Data Object	AllDefect										
Field Name	Cold										
Path											

Add a button and link to the service command you created. Clicking the button will make the application send email.

14. Expand the node **Layout (Layout Type)** and its child nodes until you reach **Control (List Control Type)**. Note all the command buttons. Add a new child **Command Button (Command Button Type)**. Set the **Command Name** to **SendEmailToDefectOwner** and **Visual** to **TextOnly**.



15. Right-click the newly added Command Button to view the option to **Navigate to View Custom Code**. This opens **AllDefectListCustomBehavior.java**. Add the following Java code snippet to the file. The snippet below implements the event handlers for Post Execute Handler and Pre Can Execute Handler.

Note that you must add these import statements to the beginning of the file:

```
import
com.codeobjects.platform.ui.jsf.shared.command.event.PostExecuteEventArgs;
import com.app.tutorial.server.dataobject.alldefect.sdo.OwnedByEmployee;
```

```
/**
 * PreCanExecute event handler method for the service command. This service
command is bound by the
```

```
* button "Send Email" on UI, therefore this event handler can control the
enable/disable of the
* button.
*
* The condition we are checking here is that the defect must be saved in DB
(defect.getCoId() != null)
* and the owner is not empty and the owner has email address.
*
* @param args the ExecuteEventArgs
* @return true to enable the service command and false otherwise
*/
public boolean SendEmailToDefectOwner_PreCanExecute(ExecuteEventArgs args)
{
    // First to get the specific ViewModelBean object which contains all
    strongly typed
    // attributes and methods
    AllDefectListViewModelBean viewModel = (AllDefectListViewModelBean)
args.getViewModel();
    Defect defect = viewModel.getAllDefectDataModel().getCurrentRow();
    return (defect != null && defect.getCoId() != null &&
defect.getOwnedById() != null &&
        defect.getOwnedByEmployee().getEmailAddress() != null);
}

/**
 * PostExecute event handler method for the service command. This event
handler will be invoked when
 * the command execution is over. Here we are showing a message to inform
the user the email has
 * been delivered.
 * @param args
 */
public void SendEmailToDefectOwner_PostExecute(PostExecuteEventArgs args)
{
    // First to get the specific ViewModelBean object which contains all
    strongly typed
    // attributes and methods
    AllDefectListViewModelBean viewModel = (AllDefectListViewModelBean)
args.getViewModel();
    OwnedByEmployee defect =
viewModel.getAllDefect_OwnedByEmployeeDataModel().getCurrentRow();
    MessageHelper.addInfo(null, "An email has sent to the defect owner at "
+ defect.getEmailAddress());
}
```

Programmer Notes

Code snippet above	Description
<pre>MessageHelper.addInfo(null, "An email has sent to the defect owner at " + defect.getEmailAddress());</pre>	Use the MessageHelper helper class implemented by CodeObjects to display an info level of message on UI.

End Result

Build and deploy. Make sure that you log in as “coadmin,” because only coadmin can assign the defect. (Recall that you applied the authorization.)

- Notice a new button Send Email is now displayed. This button is enabled only when the defect is assigned, the owner has an email address, and the defect is saved to the database.
 - You can drill down from the field **Owned By** to make sure the owner has an email address.
- Click the **Send Email** button and verify that mail is sent, and that the UI displays the defect owner’s email address.

	Defect Number	Assigned Date	Owned By	Created By	Closed Date	Due Date	Date Opened
▼	9		CO User	couser	4/19/2011 6:32 PM	4/18/2011 11:29 AM	4/11/2011 10:20 AM
▼	10			couser	4/20/2011 6:08 PM		4/13/2011 5:49 PM
▼	11			coadmin	4/20/2011 3:54 PM	4/19/2011 11:18 AM	4/15/2011 11:18 AM

Drill down to the defect owner’s employee record to fill in the email address if it is blank.

Send Mail to the Defect Owner

Defect Data Administration

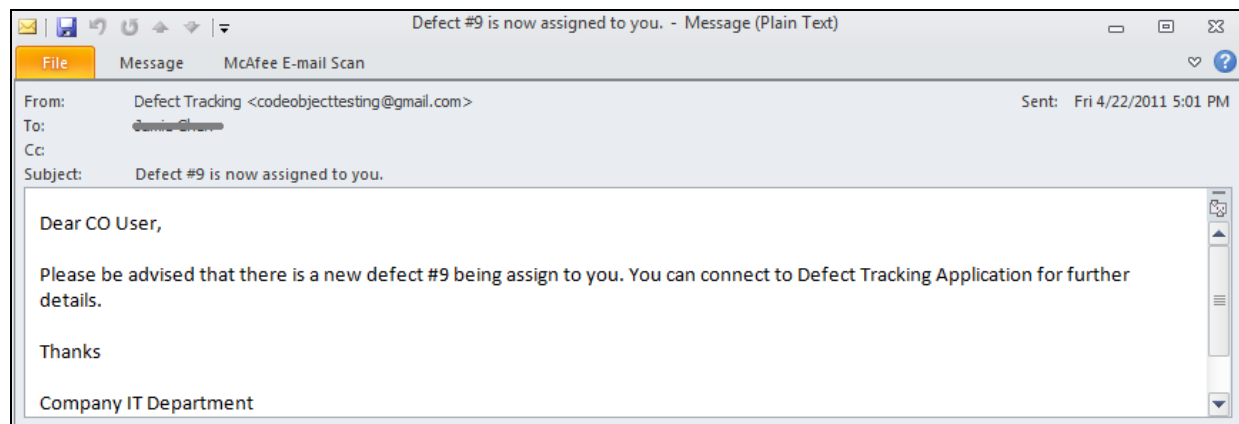
Employee

Employee

	Department	Email Address	Status	Extension	First Name	Title	Last Name
▼	Engineering		Active	123	Scott	Employee	Wombat
▼	Engineering		Active	124	Marissa	Manager	Koala
▼	Engineering		Active	125	Dianne	Employee	Emu
▼	Engineering		Active	126	Peter	Employee	Opal
▲	Support		Active	127	CO	Employee	User

Department: Extension: Last Name: *
 Email Address: First Name:
 Status: Title:

Fill in the email address in **Employee**, save, and then go back to the defect list view. Click the **Send Email** button.



After the email is sent, a message should appear on the top of view to indicate that email delivery was successful.

Defect Data Administration

All Defects

• An email has sent to the defect owner at

List of Defects

	Defect Number	Assigned Date	Owned By	Created By	Closed Date	Due Date	Date Opened
▼	9		CO User	couser	4/19/2011 6:32 PM	4/18/2011 11:29 AM	4/11/2011 10:20 AM
▼	10			couser	4/20/2011 6:08 PM		4/13/2011 5:49 PM
▼	11			coadmin	4/20/2011 3:54 PM	4/19/2011 11:18 AM	4/15/2011 11:18 AM

Send Email to Defect Owner Automatically

This chapter demonstrates configuring the system to send an email to the defect owner automatically.

Previously, you have created a button for sending an email manually. As an advanced technique, this chapter shows how you can send an email automatically.

In general:

- Instead of using the regular update, you should add some update logic to trigger whenever the **OwnedById** field is set or is changed and the owner has an email address.
- Continue with the normal update logic to make sure the update is successful.
- After the normal update is done, invoke the service method you implemented to send an email.
- If delivering a message, inform the client by showing a status message.

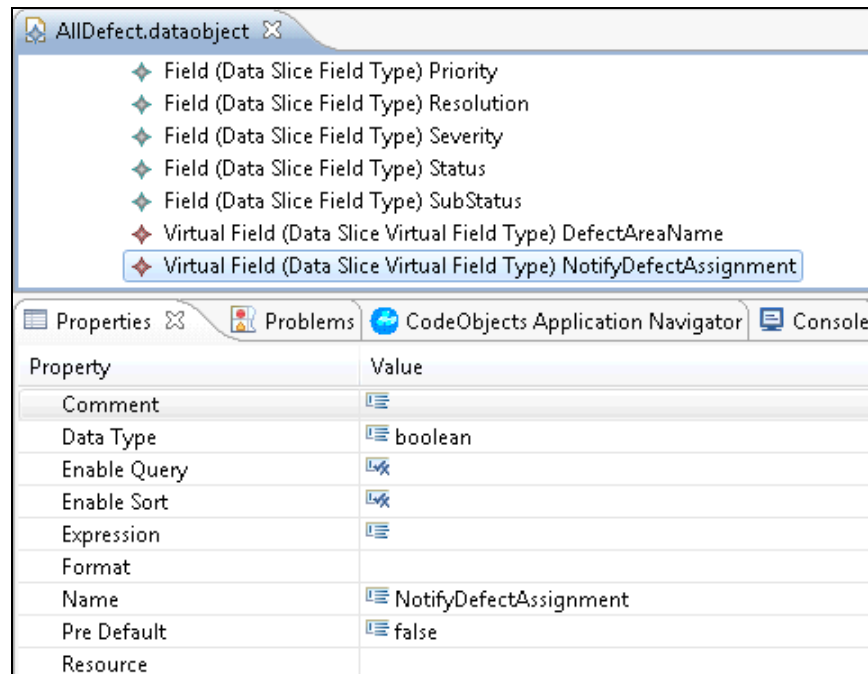
Detailed Configuration

At the UI layer, how does the system know that the email is delivered or that it must display an informational message? Here we introduce a different technique by using a virtual field.

When the UI performs the update:

- The normal update occurs
- The service sends the email, if necessary, and updates a virtual field to reflect the status.
- When control reverts back to the UI, it checks the virtual field value to see if the email happened, and then displays the informational message, as necessary.

1. In **tutorial-server** module, open the **AllDefect** data object. To the Root node, add a child node **Virtual Field (Data Slice Virtual Field Type)**. Set the properties: **Name** to **NotifyDefectAssignment**, **Data Type** to **boolean**, and **Pre Default** to **false**.
This virtual field is used to store the intermediate value from the service to indicate whether the email was sent or not. Note that you leave the property **Expression** empty so this virtual field is not bound to any other field and is purely a place holder with a freely controlled value.



2. In the **tutorial-server** module, open the **DefectApplicationService** service interface metadata and add a new method: **DefectCustomUpdate**. This method is a wrapper method for the original update method for **AllDefectDataService** from the AllDefect data object. It must take same input of a data graph and return the updated data graph.

Send Email to Defect Owner Automatically

When you set the input or the output of a service method to bind to the data object, it produces the data graph in the programming interface.

DefectApplicationService.serviceinterface

Add Delete Clear Output Type

Method	Output Type
GetLoginUserEmployeeId	Primitive: long
SendEmailToDefectOwner	
DefectCustomUpdate	Data Object: AllDefect (tutorial-server)

Input Parameters

This section provides configuration options for the method's input parameters.

Add Delete Move Up Move Down

Parameter Name	Type
AllDefectDataObject	Data Object: AllDefect (tutorial-server)

Exceptions

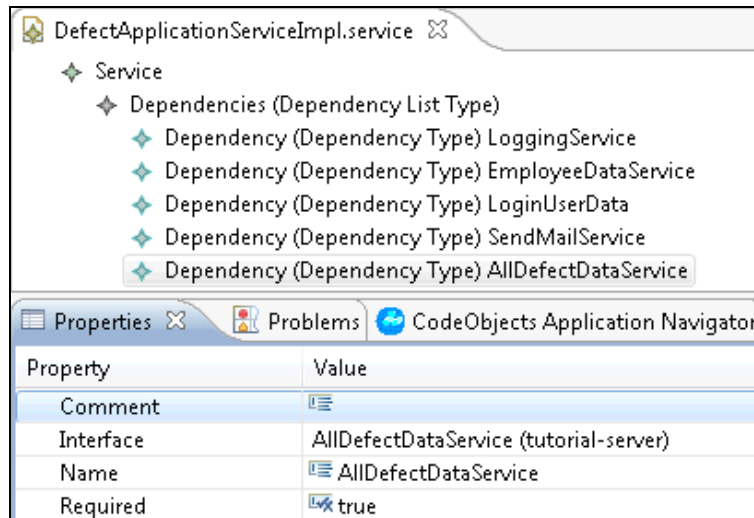
This section provides configuration options for the method's exceptions.

Add Delete Edit

DataAccessServiceException (data-access-layer-runtime)
DefectApplicationException (tutorial-server)
ServiceException (service-layer-runtime-exceptions)

Notice that you bind both the output and input to AllDefect data object. When doing so, the generated Java interface uses SDO DataGraph in the method signature.

- Open the **DefectApplicationServiceImpl** service metadata to add a dependency on **AllDefectDataService**. Set the properties:
Interface to AllDefectDataService (tutorial-server)
Name to AllDefectDataService
Required to true



- Add the following method implementation to **DefectApplicationServiceImpl.java**. The server will determine if the email must be sent or not. If it is sent, the code sets the **NotifyDefectAssignment** virtual field to **true**.

See the detailed code comment for an explanation.

- Copy these imports to the beginning of **DefectApplicationServiceImpl.java**.

```
import com.app.tutorial.server.dataobject.alldefect.sdo.AllDefectRoot;
import java.util.List;
import commonj.sdo.DataObject;
import com.app.tutorial.model.defect.sdo.Defect;
import commonj.sdo.ChangeSummary;
```

```
/** <p>
 * The custom update command for AllDefect data object. This is a type of
wrapper implementation on
 * top of the generated method. This function will scan the change summaries
to find those defects
 * which are assigned with new owner. The virtual field of
'NotifyDefectAssignment' will be set to
 * true as the indication for these defects.
 * </p>
 * <p>
 * Then the generated implementation of update will be called. And then
those defects which has
```

```

    * 'NotifyDefectAssignment' set to true will have an email sent to the
    owner.
    * </p>
    * <p>
    * NOTE! Why don't we just send email before the update? We cannot do send
    email until the update
    * is successful. And after the update is done, all change summary will be
    reset. This is the reason
    * we have such a wrapper method on update.
    * </p>
    * @param AllDefectDataObject the SDO data graph of AllDefect data object
    * @return the saved SDO data graph of AllDefect data object
    * @throws DataAccessServiceException
    * @throws DefectApplicationException
    * @throws ServiceException
    */
    @Override
    public DataGraph DefectCustomUpdate(DataGraph AllDefectDataObject)
    throws DataAccessServiceException, DefectApplicationException,
    ServiceException
    {
        // We need to read the change summary and see if the owner is changed or
        // is assigned.
        // We do NOT want to blindly to send a mail when the OwnedById is not
        // empty. If we do so,
        // the owner will receive an email for any other regular update. We only
        // want to send an
        // email whenever the OwnedById is changed (e.g. the defect is assigned,
        // or is re-assigned).

        // Before we start, let's reset any old value in NotifyDefectAssignment
        // field and start fresh
        AllDefectRoot root = (AllDefectRoot)
        AllDefectDataObject.getRootObject();
        for (int i=0; i<root.getDefect().size(); i++)
        {
            com.app.tutorial.server.dataobject.alldefect.sdo.Defect defect =
            (com.app.tutorial.server.dataobject.alldefect.sdo.Defect)
            root.getDefect().get(i);
            defect.setNotifyDefectAssignment(false);
        }

        // Obtain the SDO change summary list which consists all sdo objects
        // which
        // were modified.
        List changedObjects =
        AllDefectDataObject.getChangeSummary().getChangedDataObjects();
        if (changedObjects != null && changedObjects.size() > 0)
        {
            // Go through each changed sdo data object to make sure the sdo data
            // object
            // is what we are looking for
            for (int i = 0; i < changedObjects.size(); i++)
            {
                DataObject sdoDataObject = (DataObject) changedObjects.get(i);
            }
        }
    }

```

```

        // Make sure the sdo object is our Defect object
        if (sdoDataObject instanceof Defect)
        {
            // Given the sdo data object, fetch the old data values
            List oldValues =
AllDefectDataObject.getChangeSummary().getOldValues(sdoDataObject);

            // Iterate through each old value to see if field
            "OwnedById" is within the list
            // If yes, this means it was changed.
            for (int ix=0; ix<oldValues.size(); ix++)
            {
                ChangeSummary.Setting setting = (ChangeSummary.Setting)
oldValues.get(ix);
                String propertyName = setting.getProperty().getName();

                // Check if the 'OwnedById' field is changed.
                // NOTE!! within sdo, the property name is always
started with lower case,
                // even if the Entity field name starts with capital.
For example, field
                // "Status" in Entity will become "status" in sdo. Field
"MyStatus" will
                // become "myStatus" in sdo.
                if (propertyName.equals("ownedById"))
                {

com.app.tutorial.server.dataobject.alldefect.sdo.Defect defect =
(com.app.tutorial.server.dataobject.alldefect.sdo.Defect) sdoDataObject;

                // OwnedById is not null. Set the virtual field
'NotifyDefectAssignment' to true
                // to indicate we need to send the email after the
update.
                if (defect.getOwnedById() != null)
                    defect.setNotifyDefectAssignment(true);
            }
        }
    }
}

// Calling the real update method
DataGraph resultData =
this.getAllDefectDataService().update(AllDefectDataObject);
root = (AllDefectRoot) resultData.getRootObject();
for (int i=0; i<root.getDefect().size(); i++)
{
    com.app.tutorial.server.dataobject.alldefect.sdo.Defect defect =
        (com.app.tutorial.server.dataobject.alldefect.sdo.Defect)
root.getDefect().get(i);

    // If the virtual field 'NotifyDefectAssignment' is true, let's send
an email
    if (defect.getNotifyDefectAssignment() != null &&

```

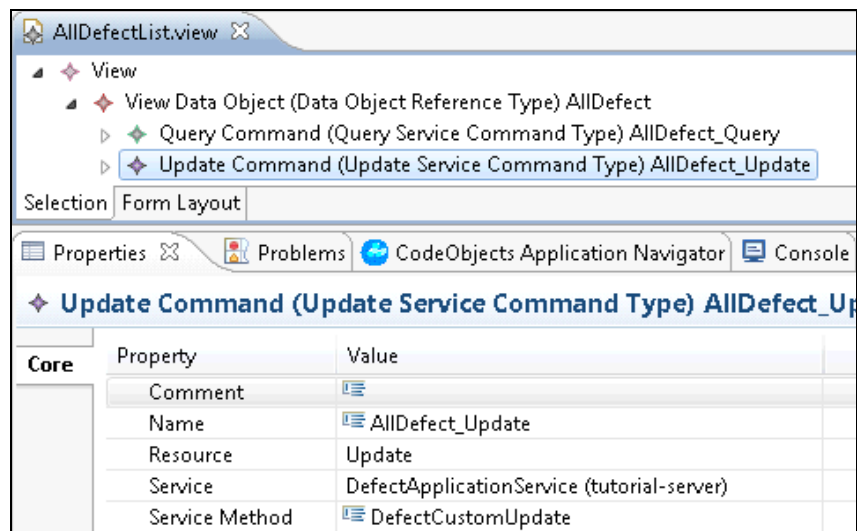
```

        defect.getNotifyDefectAssignment().booleanValue()
    {
        this.SendEmailToDefectOwner(defect.getOwnedByEmployeeName(),
defect.getOwnedByEmployee().getEmailAddress(),
                                defect.getCoId());
    }
    }

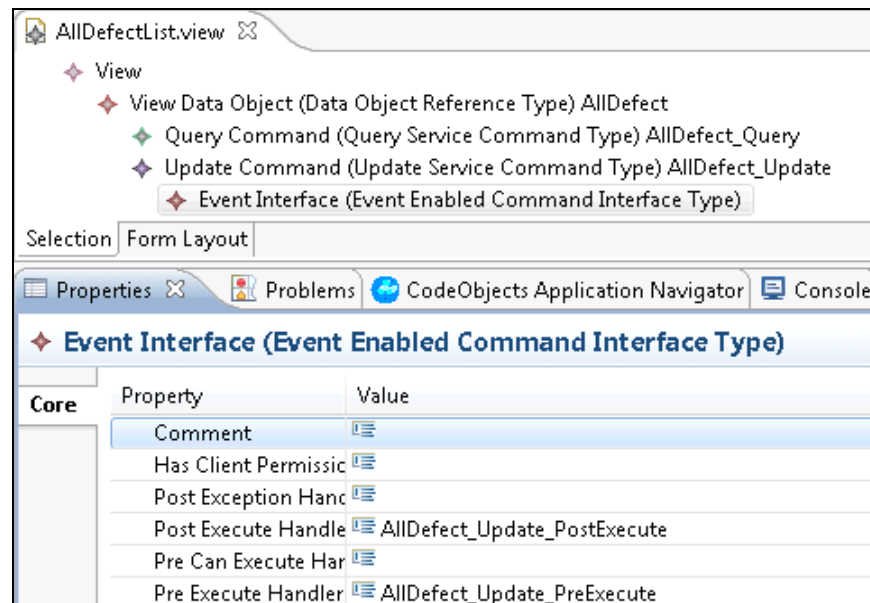
    return resultData;
}

```

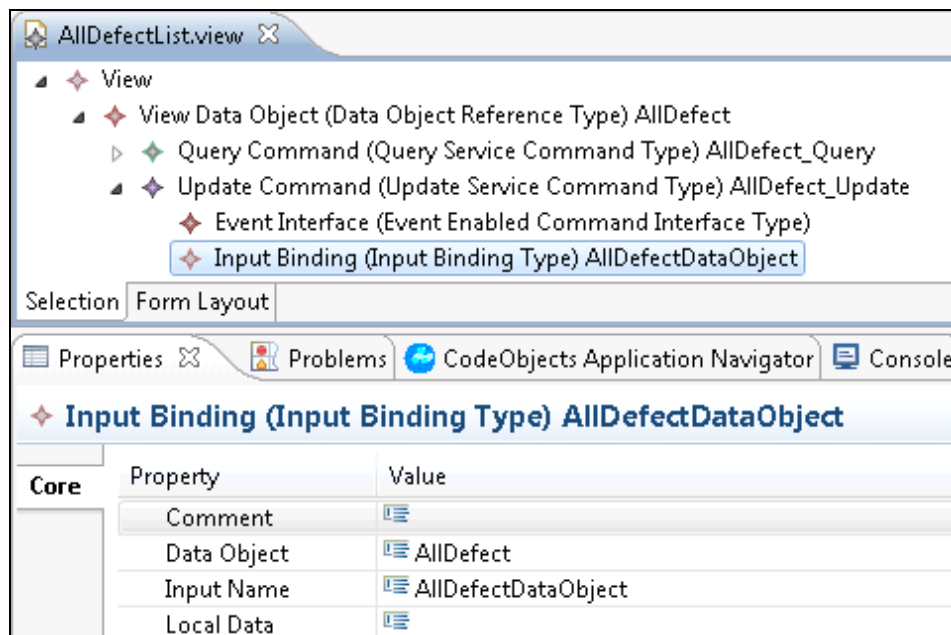
- Alter the **AllDefectList** view to use the new DefectCustomMethod code.
In the **tutorial-client** module, open the **AllDefectList** view metadata. Expand the node **View Data Object (AllDefect)**. Locate the node **Update Command (Update Service Command Type)** and change the property values:
Service to **DefectApplicationService (tutorial-server)**
Service Method to **DefectCustomUpdate**.



- Click the existing child node **Event Interface (Event Enabled Command Interface Type)** and set the **Post Execute Handler** to **AllDefect_Update_PostExecute**. This handler will check the virtual field **NotifyDefectAssignment** and display the informational message, if necessary.



- Because the update method is changed to DefectCustomUpdate, you must also correct the input binding. Click the existing **Input Binding**. Change the property **Input Name** from **data** to **AddDefectDataObject**.



- Return to **AllDefectListCustomBehavior.java**. Add the following code snippet to the file. It implements the Post Execute event handler, which checks the **NotifyDefectAssignment** virtual field and determines whether to show the informational message about the email in the UI.

Send Email to Defect Owner Automatically

```
/**
 * PostExecute event handler for AllDefect_Update command. This is invoked
by the platform after
 * the update command is executed successfully. Here we iterate through each
defect record to
 * examine if we have sent an email alert to the owner by checking the
virtual field of
 * 'NotifyDefectAssignment'. If there are multiple defects being assigned
and multiple email
 * alerts were sent, this function will concatenate the message to show the
user.
 * @param args PostExecuteEventArgs
 */
public void AllDefect_Update_PostExecute(PostExecuteEventArgs args)
{
    // First to get the specific ViewModelBean object which contains all
strongly typed
    // attributes and methods
    AllDefectListViewModelBean viewModel = (AllDefectListViewModelBean)
args.getViewModel();
    StringBuilder messageBuilder = null;
    boolean hasEmailAlertMessage = false;

    // Iterate through each defect to see if they have
NotifyDefectAssignment set to true
    // as an indication that we have sent the email alert. If there are
multiple emails being
    // set, we will concatenate the message as one and display
    for (com.app.tutorial.server.dataobject.alldefect.sdo.Defect defect :
        viewModel.getAllDefectDataModel().getWrappedData())
    {
        if (defect.getNotifyDefectAssignment() != null &&
            defect.getNotifyDefectAssignment().booleanValue())
        {
            // If we have not ever done anything, initiate the message
builder
            if (hasEmailAlertMessage == false)
            {
                messageBuilder = new StringBuilder("Defect assignment email
has been sent to: ");
                hasEmailAlertMessage = true;
            }
            else
                messageBuilder.append(", ");

            messageBuilder.append(defect.getOwnedByEmployee().getEmailAddress());
        }

        // If we have the email alert message, show it on UI
        if (hasEmailAlertMessage)
            MessageHelper.addInfo(null, messageBuilder.toString());
    }
}
```

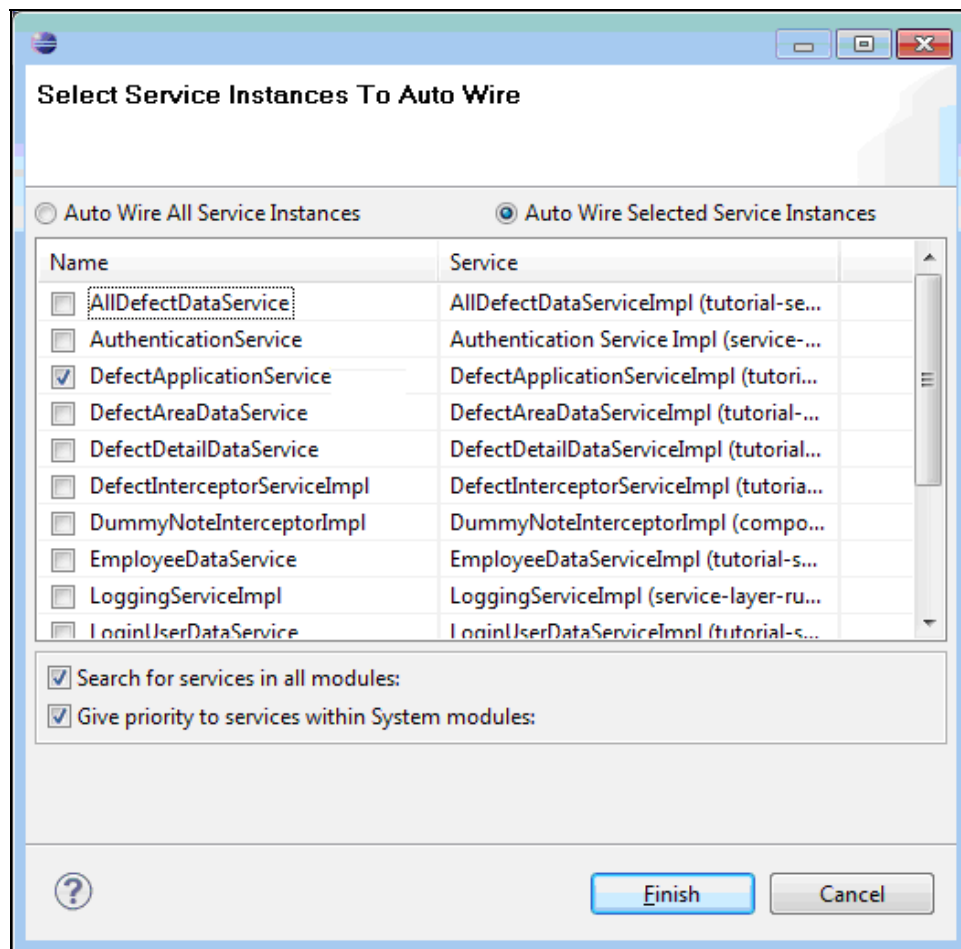
Programmer Notes

Code snippet above	Description
<pre>for (com.app.tutorial.server.dataobject.alldefect.sdo.Defect defect : viewModel.getAllDefectDataModel().getWrappedData ()) { . . . }</pre>	Use the method <code>getWrappedData()</code> to get the underlying data in the data slice Java class so that you can access the <code>NotifyDefectAssignment</code> virtual field.

- In the **tutorial-server** module, open the **Tutorial** system metadata and perform an Auto Wiring.

NOTE

The following image shows which service to Auto Wire. Optionally, you can let CORADS Auto Wire everything.



End Result

Archive and deploy. Login as **coadmin** to assign a defect and make sure that the defect owner has an email address. Click the Update button and verify that the email is sent to the defect owner automatically.

Review

In this tutorial, you:

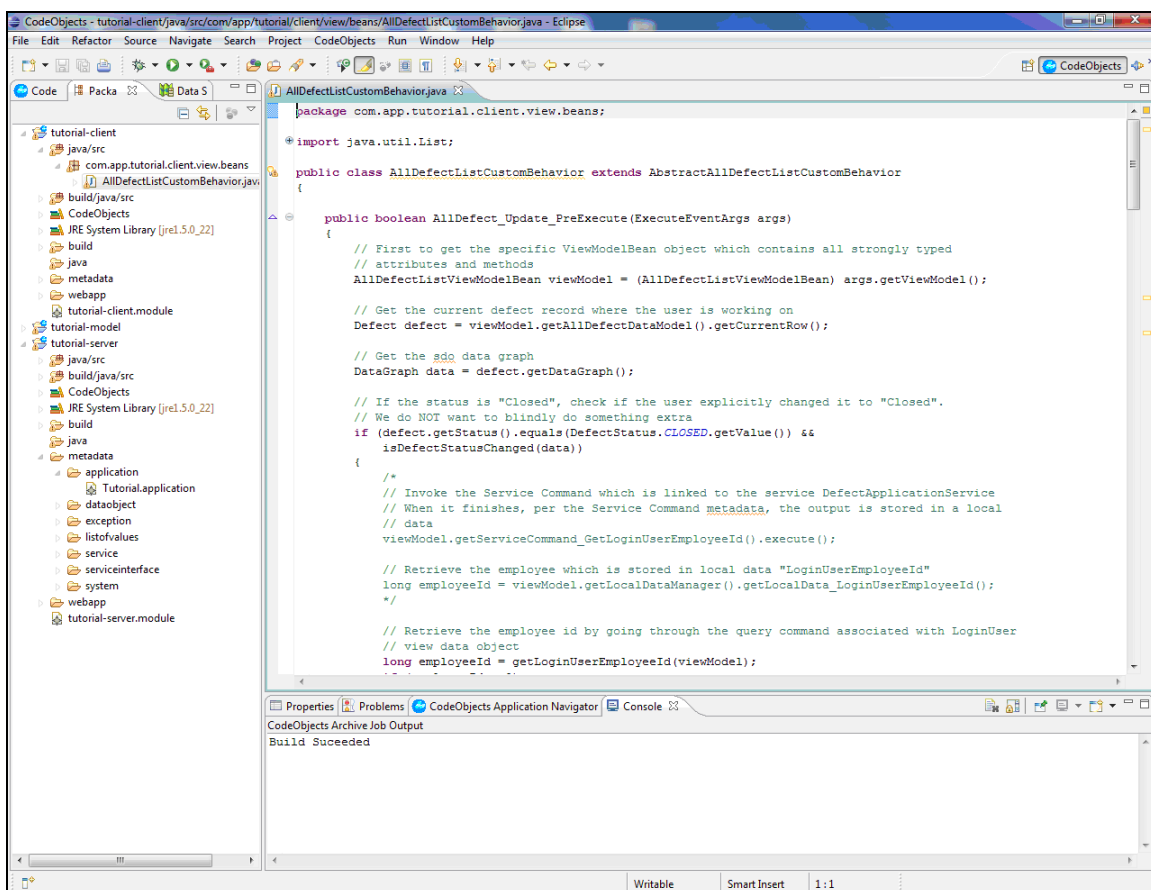
- Updated the defect's due date using a data interceptor
- Updated the close date and closed by information
- Updated the close date and closed by information using custom UI code
- Sent mail to the defect owner manually and automatically.

Appendix: Debugging

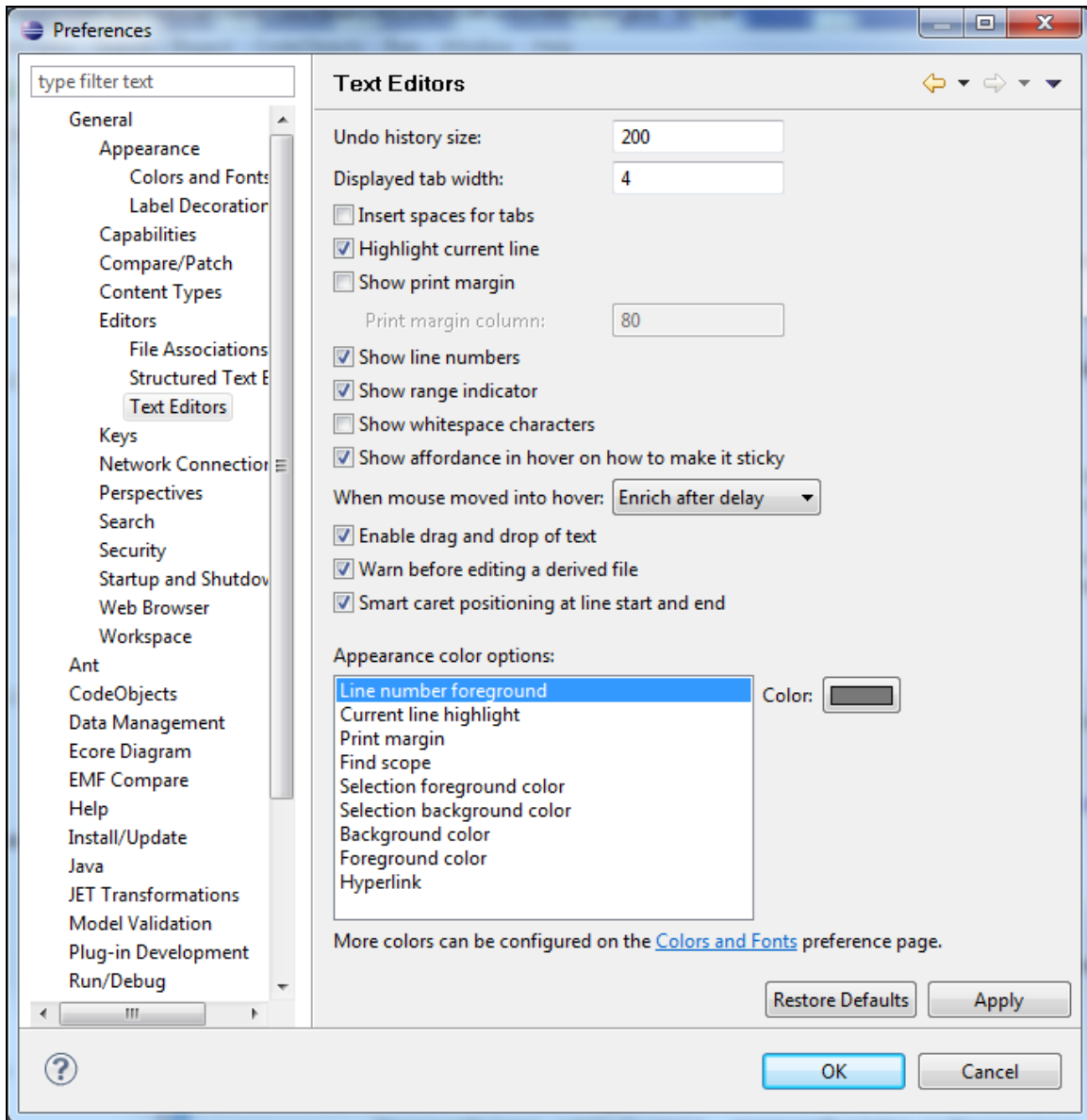
This appendix to the tutorial exercises covers debugging techniques.

Starting the Process

1. Start Tomcat, but *without* using startup.bat. Instead, in a command line window, type: **catalina.bat jpda start**
2. In Eclipse, in the Metadata pane, click the Package Explorer. Open the Java source file that you want to debug. In this example we chose from **tutorial-client** → **java/src** → **com.app.tutorial.client.view.beans** the file **AlldefectListCustomBehavior.java**.



3. If you want line numbers displayed in the source code, from the menu click **Window** → **Preferences** to open the **Preferences** dialog box. Expand **General** → **Editors** → **Text Editors**, select the **Show line numbers** check box, and click **OK**. You only need to do this once.

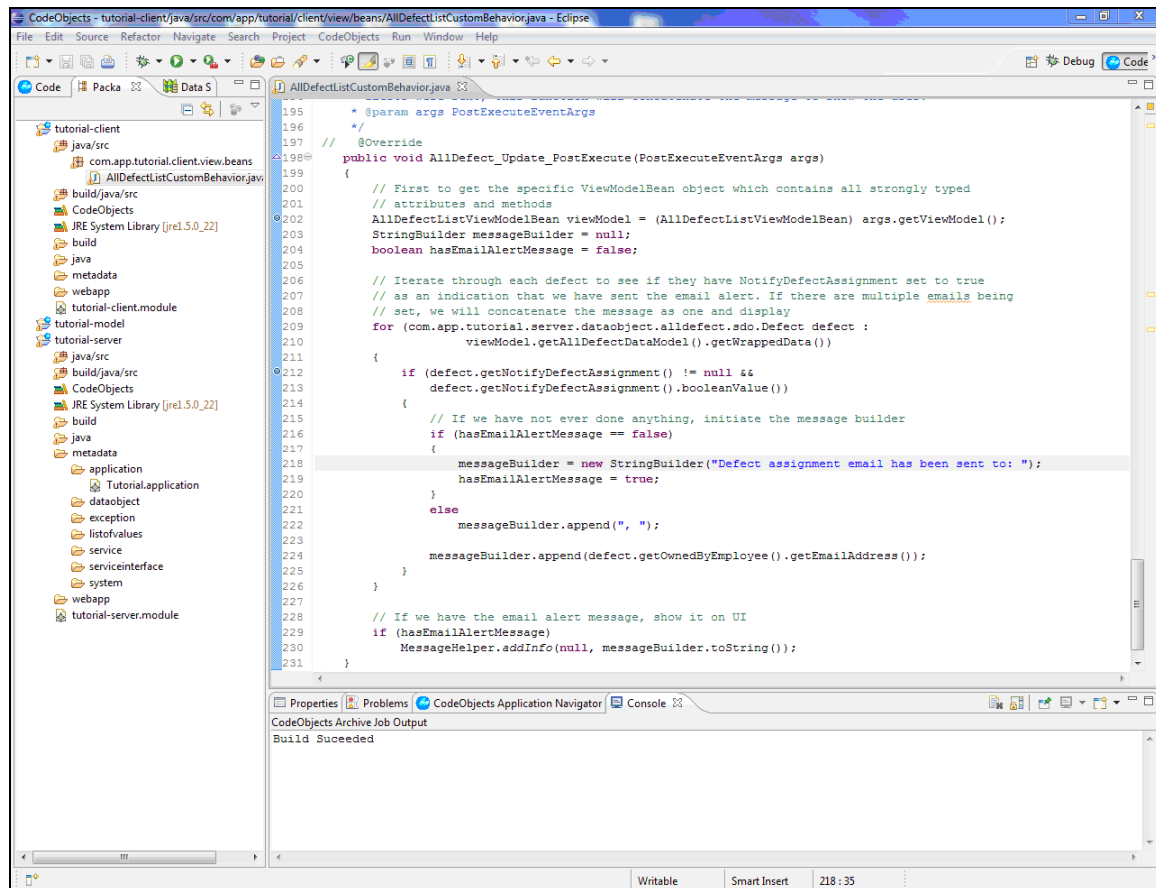


Debugging a Method

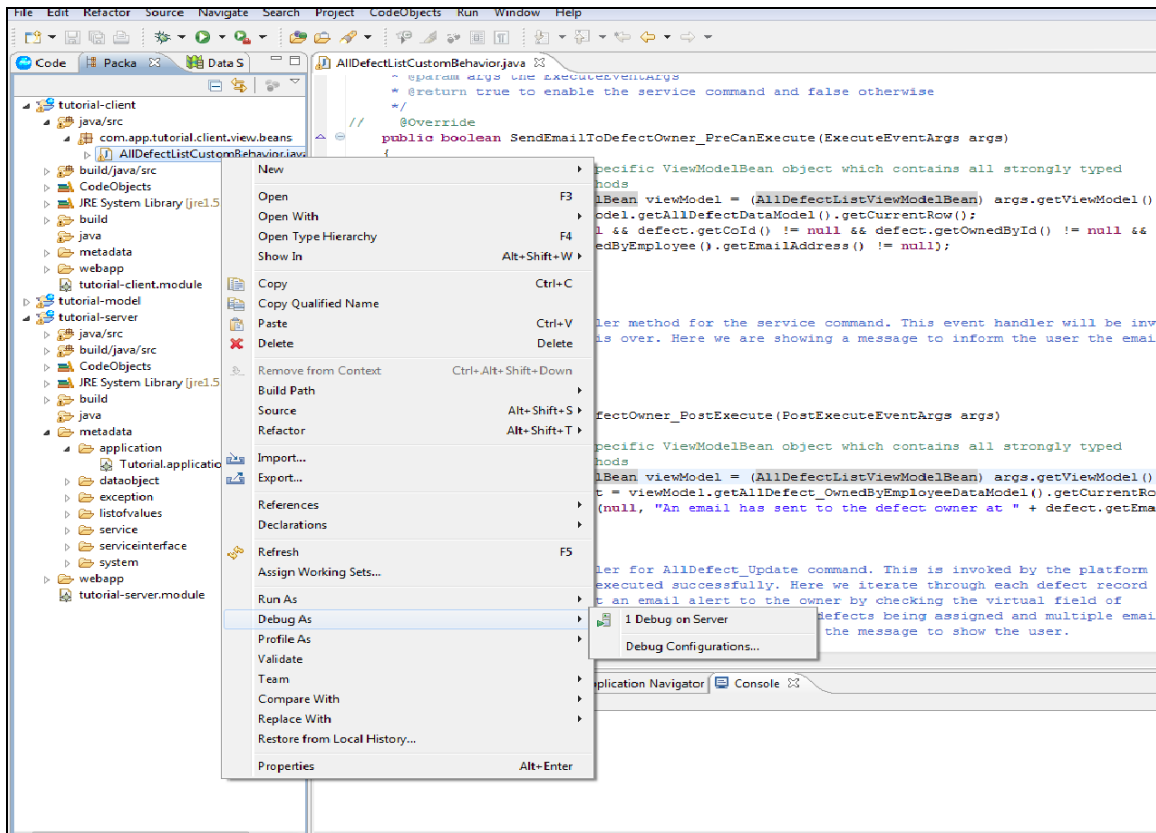
4. Scroll to the method that you want to debug, for example, **AllDefect_Update_PostExecute**. Set a breakpoint: right-click the gray vertical bar adjacent to the source code line of interest and choose **toggle breakpoint**. (The color on your screen may be different.) A small circle appears on the bar, marking the breakpoint. You can also double-click the

Appendix: Debugging

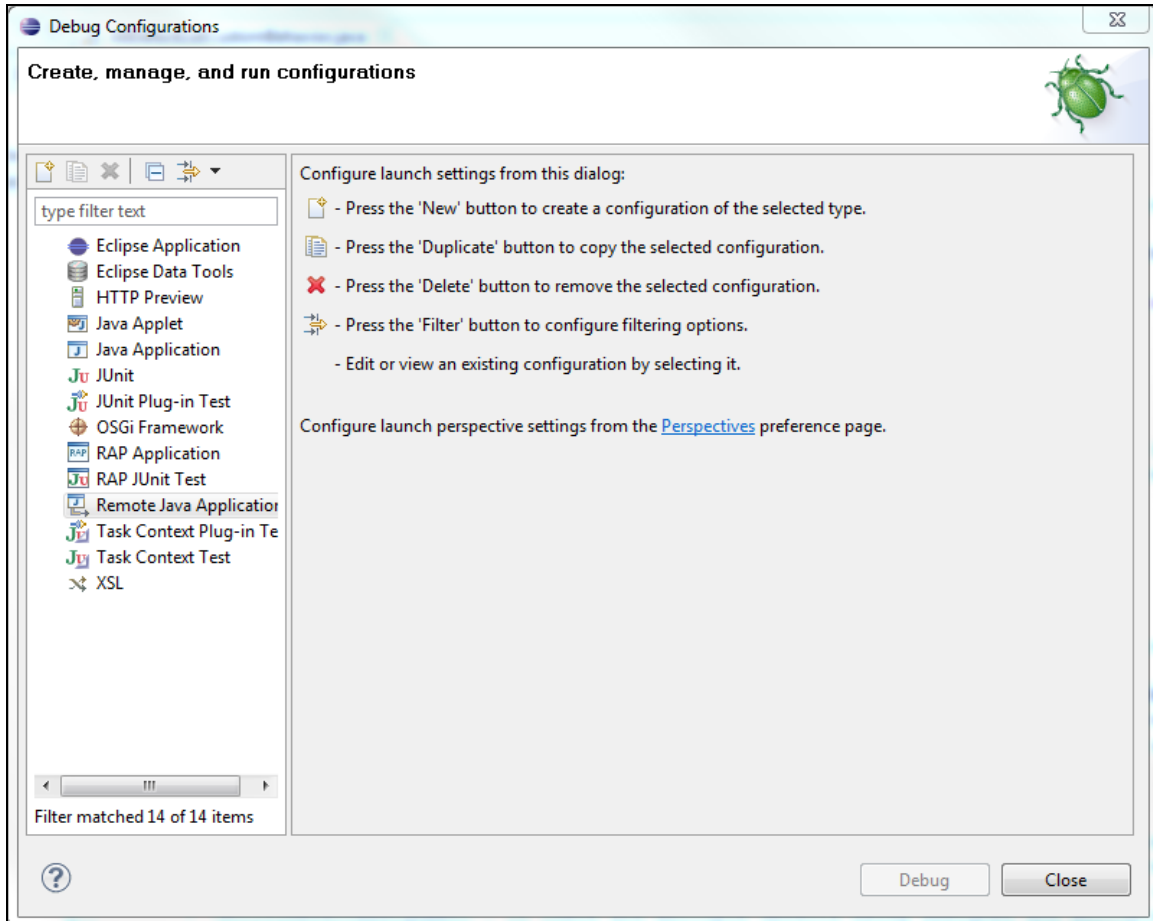
bar to set a breakpoint. In this example there are breakpoints set at lines 202 and 212. Double-click or right-click to toggle the breakpoint off, if desired.



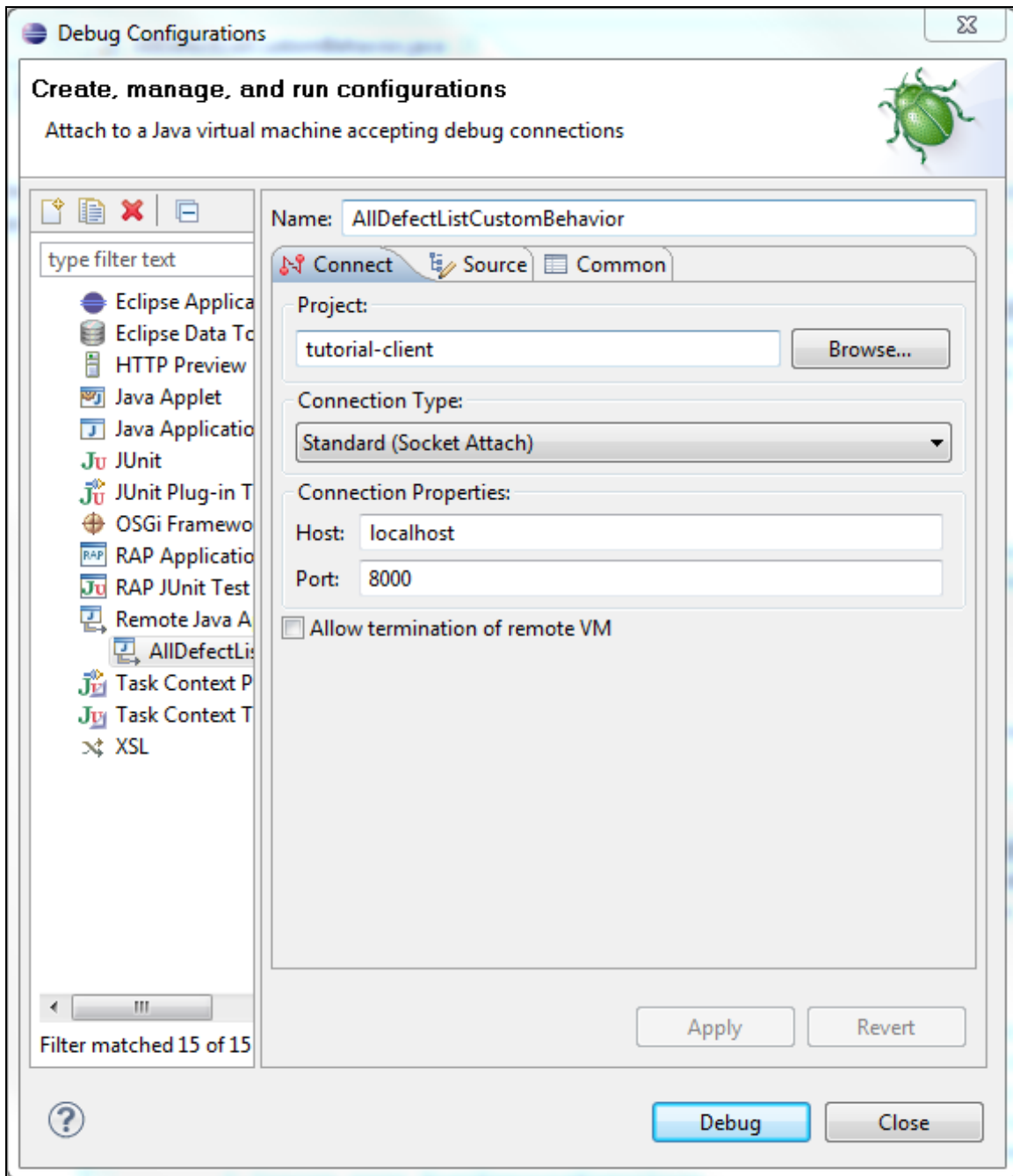
5. In the Metadata pane tree, right-click the java source file and click **Debug As** → **Debug Configurations**.



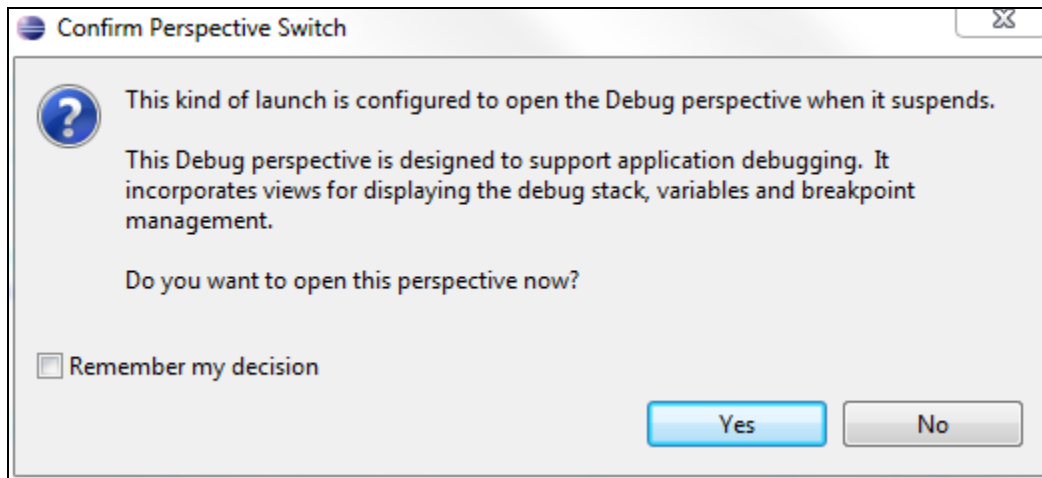
6. The **Debug Configurations** dialog box opens. Right-click **Remote Java Application** and choose **New**.



7. The **Create, manage, and run configurations** dialog box opens. Click the **Debug** button.



8. Run your application. Trigger the condition that causes the code to hit a breakpoint. The first time around, the **Confirm Perspective Switch** dialog box opens. Click **Yes**.



At a Breakpoint

The program is stopped at one of your breakpoints and Eclipse is in the Debug perspective.

9. Click  or press F5 to step into a method.
10. Click  or press F6 to step over a line of source code.
11. Click  or press F7 to step out of the current method.
12. Click  or press F8 to continue the program.
13. Click  to detach the debugger from the running application.

The Variables pane in the Debug view enables you to inspect or alter variable values. The Debug pane in the upper-left corner of the debugger shows the current stack trace.

- Another way to inspect a variable's value is to highlight the variable in the source window, right-click, and choose **Inspect**. A pane opens displaying a tree of variable values.
- Yet another way, is to move the cursor over the variable in the source code to display its value in a ToolTip.