

DocNet Basic Tutorial

Getting Started with the eHF

Imprint

InterComponentWare
Industriestraße 41
69190 Walldorf
Tel.: +49 (0) 6627 385 0
Fax.: +49 (0) 6627 385 199

© Copyright 2009 InterComponentWare AG. All rights reserved.

Document version: 0.9
Document Language: en (US)
Product Name: eHealth Framework
Product Version: eHF 2.9
Last Change: 26.10.2009
Editorial Staff: Douglas Begg

Contents

1 Introduction	1
1.1 Document Overview	1
1.2 Prerequisites	1
1.3 Integrated Development Environment	2
1.4 Document Conventions	3
2 The DocNet Application	4
2.1 Setting up the First DocNet Module	4
2.1.1 Create the Project.....	5
2.1.2 Maven Eclipse:Eclipse.....	7
2.1.3 Import into Eclipse.....	7
2.1.4 Project Directory Structure.....	8
2.2 DocNet Application Summary	9
3 Model and Generate	10
3.1 Creating the DocNet Model	11
3.2 The HSQL Database	24
3.2.1 Starting a HSQL Database.....	24
3.2.2 Shutting Down a HSQL Database.....	25
3.3 Building DocNet	25
3.4 The Generated Module Architecture	27
3.4.1 General Module Architecture Stack.....	27
3.4.2 The Object/CRUD Service Stack.....	29
3.4.3 The Domain Service Stack.....	31
3.5 The Java Classes	32
3.5.1 The Persistence Layer.....	33
3.5.2 The Service Layer.....	34
3.5.3 External API.....	34
3.6 Spring Configuration	34
3.7 Configuration Files	35
3.8 Testing	36
3.8.1 Spring & JUnit Tests.....	36
3.8.2 The Default Module Test.....	36
3.8.3 Creating our First Service Layer JUnit Tests.....	37
3.9 Model and Generate Summary	47
4 Creating an API	48
4.1 Service Adapter Layer Creation with the eHF Profile	48
4.2 Modeling an Internal Service Adapter API	48
4.3 The Generated Service Adapter Layer	50
4.4 Creating our First Service Adapter Layer JUnit Tests	50
4.5 Creating an API Summary	55
5 Adding Constraints	56
5.1 Defining Constraints in the Model	56

5.2 Expanding the JUnit Tests to Include Constraints	58
5.3 Expanding the ProgramCrudServiceAdapterTest for Constraints	59
5.3.1 ValidationException.....	61
5.4 Expanding the AppointmentServiceAdapterTest for Constraints	62
5.5 Constraints Summary	63
6 Adding a Custom Method	65
6.1 Defining a Custom Method in the Model	65
6.2 Custom Methods and the Generator	68
6.3 Implementing our Custom Method Logic	69
6.4 Testing the loadByName Custom Method	70
6.5 Returning More than One Program	70
6.6 Custom Method Summary	71
7 DocNet Assembly	72
7.1 So What is an Assembly?	72
7.2 Project Artifacts	72
7.3 Creating a DocNet Assembly	73
7.3.1 Maven Eclipse:Eclipse.....	74
7.3.2 Import into Eclipse.....	75
7.4 Incorporate DocNet Module into DocNet Assembly	75
7.4.1 Parameterized Configuration Files.....	77
7.4.2 First DocNet Assembly Tests.....	77
7.5 DocNet Assembly Summary	79
8 Web Services	80
8.1 Does DocNet Already Define Any Web Services?	80
8.1.1 Build & Deploy DocNet.....	80
8.1.2 The Predefined DocNet Web Services.....	81
8.2 Configuring Web Services	82
8.2.1 Module Web Service Configuration.....	82
8.2.2 Assembly Web Service Configuration.....	83
8.3 Default Web Services	84
8.4 Defining a Custom Web Service	85
8.4.1 Modeling External Transfer Objects (XTOs).....	85
8.4.2 Implementing a Custom Web Service.....	86
8.4.3 Testing the DocnetModuleServiceXtoAdapter.....	88
8.5 Testing Custom Web Service	91
8.5.1 Authorization Detour.....	91
8.5.2 Build & Deploy DocNet.....	92
8.5.3 Creating and Sending SOAP Requests.....	93
8.6 Web Services Summary	98
9 Summary & Outlook	99
9.1 Record	99
9.2 eHF Code System	100
9.3 eHF Document	100
9.4 Security	101

HSQL Database Configuration	103
Spring Contexts	105
A A Module's Standard Spring Context	105
B A Module's Test Spring Context	108

1 Introduction

The eHealth Framework (eHF) is a powerful platform for the development of healthcare solutions. InterComponentWare AG (ICW) has incorporated its extensive experience in developing and deploying solutions into the eHealth Framework, which represents the foundation of the Java Platform Enterprise Edition (Java EE) development at ICW.

The ICW eHealth Framework consists of reusable software components, development tools, as well as architectural guidelines and conventions defining a full software-development and product lifecycle. From the perspective of a partner, the framework provides services and infrastructure capabilities for integrating applications within an eHF-based solution.

The DocNet Tutorial is an introductory tutorial for developers who are new to ICW's eHealth Framework. It is aimed at both developers who wish to further develop the eHF itself, and those that want to build an application based upon it.

If you are completely new to the eHF, then we recommend that you first go and read the eHF White Paper. It will provide you with a good introduction into the architecture, technology and services of the eHF. The benefit you will gain from this tutorial will be greatly enhanced by having read the white paper first.

1.1 Document Overview

The purpose of this tutorial is to illustrate the full lifecycle of developing a new module for the eHF. This module could then later either be incorporated into the eHF itself or be the basis of a new application built on top of the eHF.

Through following this tutorial you will be provided with an introduction into the following:

1. Installation of the Integrated Development Environment (IDE) used at ICW.
2. All the steps involved in setting up and developing a new eHF based module, including:
 - Project setup.
 - Data Model design and code generation – including looking behind the scenes at what is generated for you and implementing constraints.
 - Defining web services.

1.2 Prerequisites

To be able to fully follow this tutorial it is assumed that you have experience/knowledge of certain technologies. These technologies, and the level of knowledge required, are listed in [Table 1](#).

Technology	Required Knowledge
Java, Java Enterprise http://java.sun.com ↗	eHF is based on Java EE. So we expect you to be very familiar with the development of Java Enterprise applications.
Spring http://www.springsource.org ↗	Spring is a Java/JEE application framework. A basic understanding of Spring's configuration mechanism is expected, although an explanation of the specific configuration files we use will be given. As the eHF uses Spring heavily, extensive knowledge about

Technology	Required Knowledge
	dependency injection, Spring configuration, Aspect Oriented Programming and transaction management is required if you want to build your own eHF based applications.
Hibernate http://www.hibernate.org ↗	Hibernate is a framework for object/relational persistence. A basic understanding of Hibernate is expected.
Maven http://maven.apache.org ↗	Maven is a tool for building Java projects. A basic understanding of build tools would be expected, but extensive knowledge of Maven itself is not required. The required targets are detailed throughout the tutorial.
openArchitectureWare (oAW) http://www.eclipse.org/gmt/oaw ↗	openArchitectureWare is a generator framework for model driven software development. No extensive knowledge is required, as a short introduction will be given in the tutorial.
UML http://www.omg.org ↗	UML is a modeling and specification language. A good understanding of UML class diagrams is expected, although the basic steps of modeling with TopCased will be detailed in the tutorial.
Eclipse http://www.eclipse.org ↗	Eclipse is platform for developing software. A good understanding of working with Eclipse is expected.
JUnit http://junit.org ↗	JUnit is a framework for regression tests. A good understanding of JUnit is expected.
HSQldb http://hsqldb.org ↗	HSQldb is a relational database engine. A good understanding of databases in general is expected.

Table1. Used Technologies and Required Knowledge

1.3 Integrated Development Environment

The eHealth Framework is complemented by an integrated development environment (IDE). It is based on the popular Eclipse project and equipped with useful plug-ins it eases and improves the development process.

Besides the common Eclipse capabilities, several plug-ins are provided to facilitate interoperability with other frameworks and components used within the eHealth Framework. As such, while not a requirement for developing an eHF module, it is recommended to install the IDE before proceeding further with this tutorial as it will be used, and referred to, throughout.

Details of the IDE, the installer and setup instructions can be found in the IDE Installation Manual.



Note: Use of the ICW IDE is assumed.

For the purposes of this tutorial it is assumed that the developer has a full installation of the IDE (Eclipse, additional Eclipse Plugins, infrastructure tools like Apache, Maven repository etc) as described in the installation manual.

1.4 Document Conventions





Mark up	Explanation	Example
italic	Windows, Dialogs, Sections, Forms	Open the window: <i>Options</i>
bold	Buttons, Fields, Tabs, Check boxes, Options, Drop-down-Menus	The Button next lets you navigate through the wizard
blue with arrow	Link	Click on http://www.java.com ↗ The website will be opened.
bold and italic	Emphasis	This information is <i>only</i> available on the vendor's page.
	Caution	CAUTION: Save your input.
	Note	Note: This view shows only data according to the installation process.
	Example	Example: The following is a little example. ;-)
	Path	Path: Click File > save as
<<Variable>>	Variable	The installation directory will be shown<< INSTALL DIRECTORY>>.
[ctrl+alt]	Shortcut	[Shift] or [Shift+Enter]

Table2. Document Conventions

2 The DocNet Application

Throughout this tutorial we are going to build a small example application called "DocNet".

For many chronic illnesses there are special programs that patients can be placed on, to provide them with a structured form of health care to better assist their recovery. An example of one such program would be Alcoholic's Anonymous.

In this tutorial we will develop a small system for the management of these so called "programs".

To keep within the scope of the tutorial the application itself will be kept fairly simple, but it will serve as a good platform through which to introduce the main aspects of the eHF.

In this chapter we will show you the basic set up of a new module. Then in the section [Model & Generate](#) on page 10 we will introduce our domain model and show you the benefits of model driven software development (MDSD) with the eHF generator. Subsequent chapters introduce further aspects of the eHF, particularly advanced modeling options.

2.1 Setting up the First DocNet Module

The eHF itself is composed of several modules (authorization, code system, document, etc). For our DocNet project we are going to develop an additional DocNet module, which will utilize various eHF modules. This new application module and the basic eHF modules will later be assembled together to form our eHF based DocNet application (detailed in the [DocNet Assembly](#) on page 72 chapter).

(Almost) all eHF modules share in common, that they are based on the same generic architecture, shown in [Figure 1](#). We will introduce this architecture stack later in the tutorial. For the moment, we will simply give you an idea of the benefits of this architecture.

The shown architecture addresses concerns such as security, transaction handling, best practice service stack (Service Adapter Layer, Service Layer and Persistence Layer) and so on. This is provided to you in a very simple manner. All you have to do is "draw" your domain object model as a UML class diagram. Once completed a model driven generator generates a new custom eHF module with all the appropriate aspects of this architecture for you. Allowing you to concentrate on implementing your own business logic, rather than having to worry about boiler plate code.

In particular, security aspects (a very important aspect in health care applications) will be introduced transparently for you. Further important out-of-the-box features of the newly generated modules are automatic transaction management, general CRUD (create, read, update, delete) operations for your domain model, and web service interfaces.

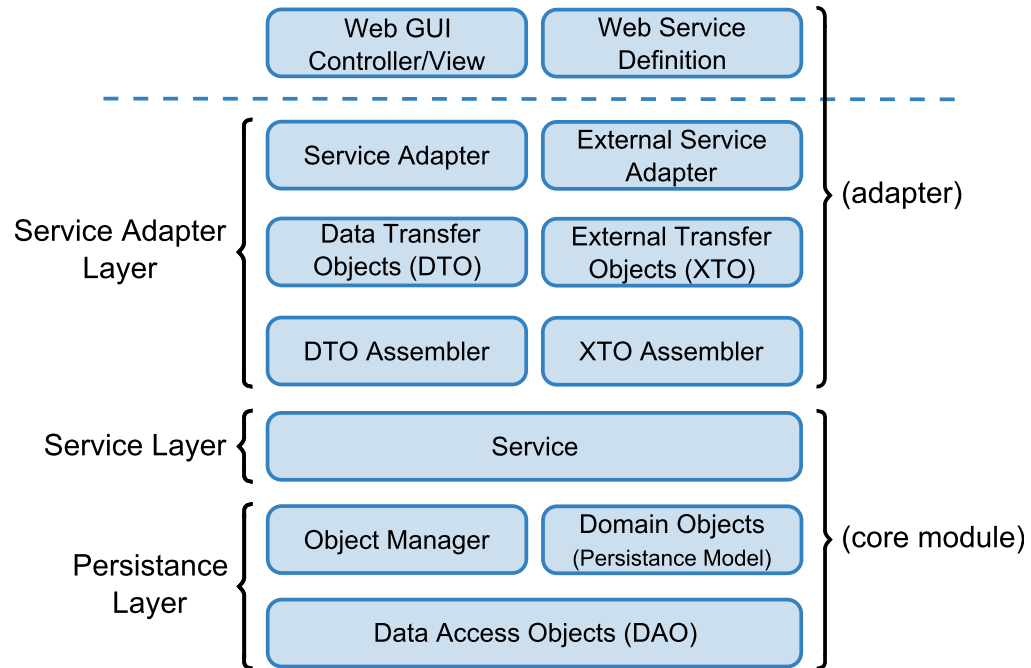


Figure 1: Standard eHF Service Module Architecture

However, before we can begin development of our DocNet application module we first need to setup the project structure for this new application module. To do this we use the Maven Generator Plug-in (<http://maven.apache.org/maven-1.x/plugins/genapp/>), in conjunction with a custom designed eHF module template.

The *Setting up an eHF Project How-To* guide provides full details on this process, as well as fully explaining the subsequent folder structure created and the various configuration properties available to the project afterwards. Don't worry though we'll cover the main points required in this chapter as well.

Later on in the chapter [Model and Generate](#) on page 10 we will actually create our domain object model and run the generator.

2.1.1 Create the Project

In this section we will create the basic application module project in our Eclipse workspace. For this, we use the *maven genapp plugin* in conjunction with a custom eHF module project template. We can then import the newly created project into our Eclipse workspace.

The custom module project template is stored, along with other custom templates, in the *ehf-genapp-maven-plugin*. If you installed the ICW IDE, then this should already be installed in your Maven installation. If it is not installed, then simply copy the jar file from the Maven repository to your maven plugins folder (under a default ICW IDE install location this is `C:\ICW_IDE\maven\plugins`).

1. Open a console window, and navigate to your Eclipse `<<workspace>>` folder. (default location is `C:\ICW_IDE\workspace`)
2. Run the `maven genapp` plug-in in conjunction with the *ehf-module-template* by entering the following command:

```
maven genapp -Dmaven.genapp.template="ehf-module-template"
```

3. Answer the eight questions, as shown in the following output listing.



Note: The project's root directory does not need to exist beforehand.

If the project's root directory does not already exist, then the genapp plug-in will simply create it for you.

```

|_ \V/ |__ _Apache__ ___
| | \V/ | / _` \ V / -_) ' \ ~ intelligent projects ~
|_| | \_\_,_| \/\_\_|_|_| v. 1.1

WARNING: No pom file was found, assuming default settings!

build:start:

genapp:
Please specify the project root directory: [C:\ICW_IDE\workspace]
docnet
Please specify an id for your application: [ehf-module]
docnet
Please specify a groupId for your application: [ehf]
docnet-basic
Please specify a name for your application: [eHF Module]
DocNet
Please specify the package for your application: [com.icw.ehf]
com.mycompany.docnet
Please specify a module name for your application: [module]
docnet
Please specify a name for the database schema of your application. Use only [A-Z0-9_]*, e.g. EHF_MODULE: [EHF_MODULE]
EHF_DOCNET
Please specify the version of eHF you would like to use: [SNAPSHOT]
X.X.X

```

Once completed Maven has generated a basic eHF module project called *docnet* in our Eclipse workspace.

The eight inputs we needed to provide for the maven genapp plugin were:

1. **project root directory: docnet** - this is the root directory where our new module will be located. This can either be a relative or absolute path - in our case relative. Maven will create this directory for us if it doesn't already exist.
2. **application ID: docnet** - this is used by Maven to define the `artifactId` tag in the `project.xml` file of the module. This is the name that will be used by Maven when building the project artifacts (i.e. `.jar` files etc).
3. **application group ID: docnet-basic** - this is propagated in the project description. It is also used by Maven to define the group folder, via the `groupId` tag in the `project.xml` file of the module, that the final artifacts will be stored in within the Maven repository.
4. **application name: DocNet** - this is simply a short descriptive name. It is used by Maven to define the `name` tag in the `project.xml` file of the module.

5. **application package: com.mycompany.docnet** - this is the top level java package that we will be using throughout our code.
6. **module name: docnet** - this is the name of the module and its main identifier used throughout the module. It is used throughout the various configuration files when referring to the module, and also for the naming of fragment files that will be integrated into parent files. Don't worry too much about these files just now, at the moment we just need to worry about the name of the module. It should not contain any prefixes, and if the module is to ultimately be part of the eHF, it should NOT contain the **ehf-**prefix.
7. **database schema: EHF_DOCNET** - this is the name of the database schema to be used for your module.
8. **ehf version: X.X.X** - this is the version of the eHF that should be used in the module. The value entered here will be used as the version for the default dependency definitions to other eHF modules in the project's `project.xml` file. Replace X.X.X with the version of the eHF that you are currently working with.

Note: The Project Object Model



The `project.xml` file referred to throughout the last section is the Project Object Model (POM) in Maven 1.1 (the version used throughout this tutorial). The POM is an XML file that contains information about the project and configuration details used by Maven to build the project.

For more information on the POM see the Maven web site - <http://maven.apache.org/guides/introduction/introduction-to-the-pom.html> ↗

2.1.2 Maven Eclipse:Eclipse

Maven provides various plug-ins to setup projects for use in different IDEs. For an Eclipse based IDE we can run `maven eclipse:eclipse`. This will then create the Eclipse specific `.classpath` and `.project` configuration files.

1. Open a console window, and navigate to the `<<workspace>>/docnet` folder, this is the project folder that the Maven genapp plug-in has created for us previously in section [Create the Project](#) on page 5 .
2. Run the eclipse plug-in by entering the following command:

```
maven eclipse:eclipse
```

Maven will list a vast number of actions that it carries out (most likely too quickly for you to be able to read), however hopefully at the end of it all you should see "BUILD SUCCESSFUL".

2.1.3 Import into Eclipse

Next we need to import this project into Eclipse.

1. From within Eclipse select the menu **File -> Import...**

2. In the Import window that appears expand the **General** folder and select **Existing Projects into Workspace** and then click **Next >**.
3. In the next window of the wizard, for the **root directory** use the **Browse** button to select the <<workspace>> folder. Docnet should then be listed in the list of available projects. Make sure it is selected and then click **Finish**.

The project "docnet" will now be listed in the package explorer of Eclipse.

Once the project is imported into Eclipse you should then see the following project directory structure:

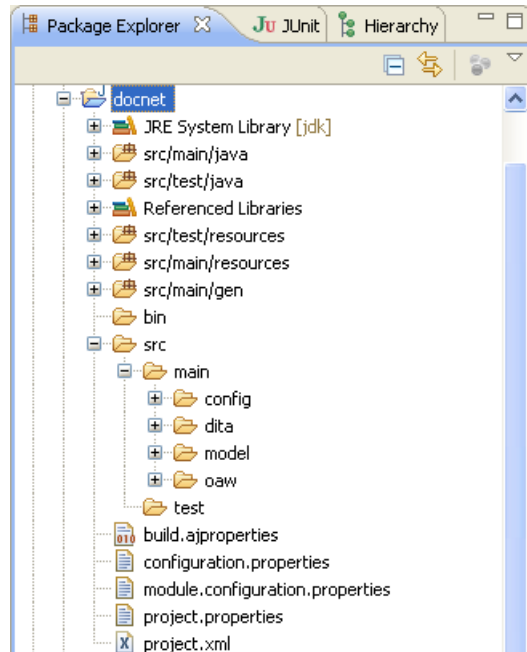


Figure 2: Initial Generated Project Structure

Now that we have our project folder structure we'll take a brief look at what we really have in the next section.

2.1.4 Project Directory Structure

As can be seen from [Figure 2](#), in the previous section, after the genapp process and the completion of the import into Eclipse, there are numerous directories and files created for us in our project. These are the basic files and directories for the generation, implementation and configuration of our new DocNet application module and its architecture stack as shown previously in [Figure 1](#).

The common configuration files for both Eclipse and Maven are located directly under the top level project folder, so in our case docnet. This includes `project.xml`, the Maven Project Object Model (POM), and `project.properties`, used to customize the behavior of Maven and its plug-ins.

Otherwise the following general folder structure is created for us:

- `src/main/config` - contains all configuration files that are parameterized.
- `src/main/dita` - stores the module's documentation in dita format.
- `src/main/model` & `src/main/oaw` - contain UML definitions and settings used by the code generator.

- `src/main/gen` - the output folder for the generator - both java and configuration files - this is under the control of the generator which means files here should not be modified by the developer (they will be replaced during the next generator run).
- `src/main/java` - the main java source directory that is under the control of the developer - any files stored here can be safely modified.
- `src/main/resources` - resource root directory.
- `src/test/java` - the main java test source directory.
- `src/test/resources` - test resource root directory - allows for additional test fragments without them being packaged in the resulting jar files. One example of this is the Spring context definition required for the tests.

2.2 DocNet Application Summary

Well this was a short chapter to get us up and running. That said we have used Maven, together with the genapp and eclipse plugins, to create a skeleton project structure that already has a lot of default configuration taken care of for us. This project structure follows the conventions used throughout the eHF, making it easy to integrate with the rest of the eHF at the appropriate time.

With this skeleton structure in place we are ready to go ahead and start to get our hands dirty with modeling, the generator and code.

3 Model and Generate

Having set up our project we are now ready to create our first model. Later on, we will use the model driven eHF generator to build the architecture stack, as shown previously in [Figure 1](#), for our DocNet application module.

[Table 3](#) shows the advantages of our model driven approach. In the context of eHF, the model driven code generation provides you with the same architecture stack in each module, transparently introducing security aspects as well as other eHF specific concepts.

Aspect	Advantage
Quality	<ul style="list-style-type: none"> • Increased quality of software • Homogenous architecture • Reduced sources of defects • Strictly adheres to conventions
Velocity	<ul style="list-style-type: none"> • Software development speed is increased
Separation of Concerns	<ul style="list-style-type: none"> • Domain-specific considerations and technical concerns are separated • Reduced redundancy (model as source code) • Improved maintainability/extensibility on both ends
Centralized expert knowledge	<ul style="list-style-type: none"> • In-depth technology knowledge, best practices, conventions are collected defining the generator • The gathered knowledge is used throughout the projects utilizing the generator.
Abstraction	<ul style="list-style-type: none"> • The model represents an abstraction from the implementation details and the system configuration
Flexibility	<ul style="list-style-type: none"> • Migration to other platforms is easier MDA paradigm (platform independence) • Optimizations can be applied in a centralized manner centralized expert knowledge

Table3. Advantages of Model Driven Software Development (MDSD)

To begin our first domain model we will create two domain objects, one called `Program` which will be our medical programs as described earlier, and the second will be `Appointment`. A `Program` instance can be related to any number of `Appointment` instances. While an `Appointment` instance can only belong to exactly one `Program` instance.

Here is the basic UML class model that we want to begin with:

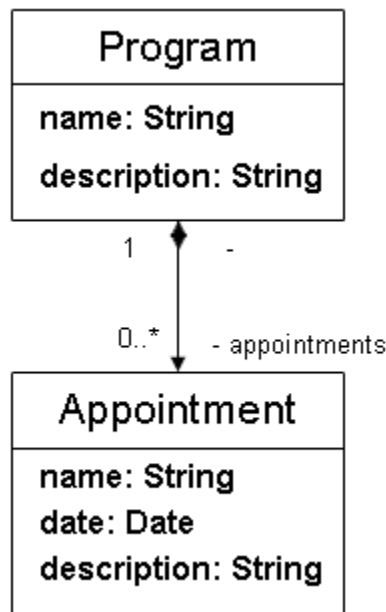


Figure 3: First Basic DocNet Model

Note: Creating UML2 Models

It should be noted from the beginning that the tool we have used for creating our models during this tutorial is the TopCased UML Editor plugin for Eclipse.



If you would prefer to use MagicDraw for creating your models you will need to use the `model.xml` file instead of the `model.uml` file referred to throughout the rest of this tutorial. Additionally when you save your model in MagicDraw you will need to make sure that you **export** your model to uml2 v2 format - this will then create the `model.uml` file for use with the eHF Generator. If you use a version of MagicDraw earlier than 12.5 you can only export the model in uml2 v1 format. This will rather confusingly save the file as `model.uml2`. It can still be used by the eHF Generator, you just need to rename the file as `model.uml`.

Next we will create this model with our UML tool. Following that we will then generate the architectural stack as shown previously in [Figure 1](#) and provide a detailed description of this architecture. Finally, we will familiarize ourselves with the generated Java classes by writing our first JUnit tests.

3.1 Creating the DocNet Model

In this section we will create our domain object model as shown in [Figure 3](#) using the TopCased UML Editor plugin for Eclipse.

Looking in the `src/main/model` folder of our DocNet module we can see that a default `model.uml` file has already been created for us by the Maven `genapp` plug-in. This contains all the model information required by the generator. What it does not contain is the diagram information - which is not required by the generator but makes it a lot easier for us humans to understand.

So first we need to create a diagram based on this `model.uml` file. Once we have our diagram we can then edit the model directly by using the diagram - the model will automatically be updated when we make and save changes in our diagram.

1. From Eclipse's main menu, select **Window -> Open Perspective -> Other...**, and in the window that appears select **Topcased Modeling**, which as you might expect should now open the Topcased Modeling perspective.

From the normal Eclipse Navigator, so NOT the Topcased Navigator, navigate to the `src/main/model` folder of the DocNet project. From this location right click on the `model.uml` file. From the shortcut menu that appears select **New -> Other...** and then in the window that appears select **UML Model with TOPCASED** from within the **Topcased/Topcased Diagrams** folder as shown below:

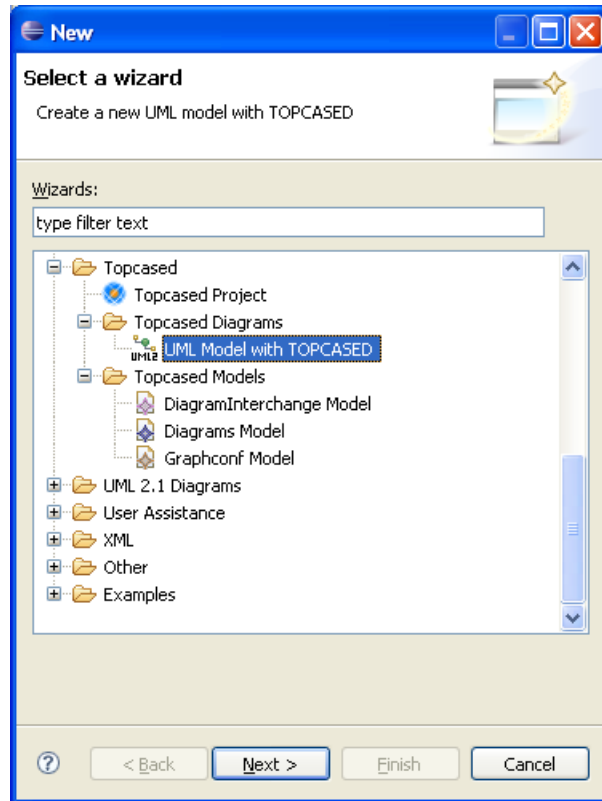


Figure 4: Selecting a new UML Diagram

2. In the "New UML model with TOPCASED" dialog window the radio button **Create from an existing Model** should already be selected, along with our `model.uml` file. From the **Root Diagram** drop down box select "**Class Diagram**", and deselect the option "**Initialize the diagram with existing model objects**" as shown below:

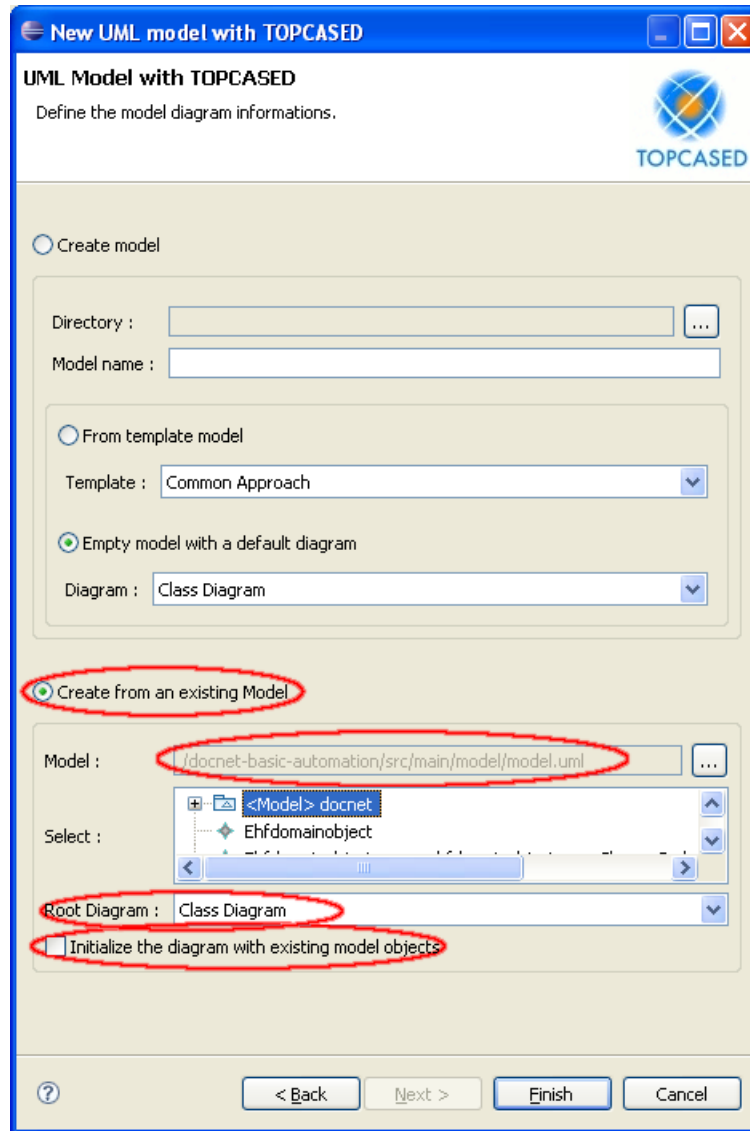


Figure 5: Completed "Create new UML diagrams" Dialog Window

Once all the options have been correctly set, click **Finish** to create a new blank diagram (this will be saved as `model.uml` in the `src/main/model` folder).

3. At the moment you won't see too much. In the main pane within Eclipse you should see the blank diagram. Select the **Outline** view and expand `<Model> docnet`. You should now see something similar to the following:

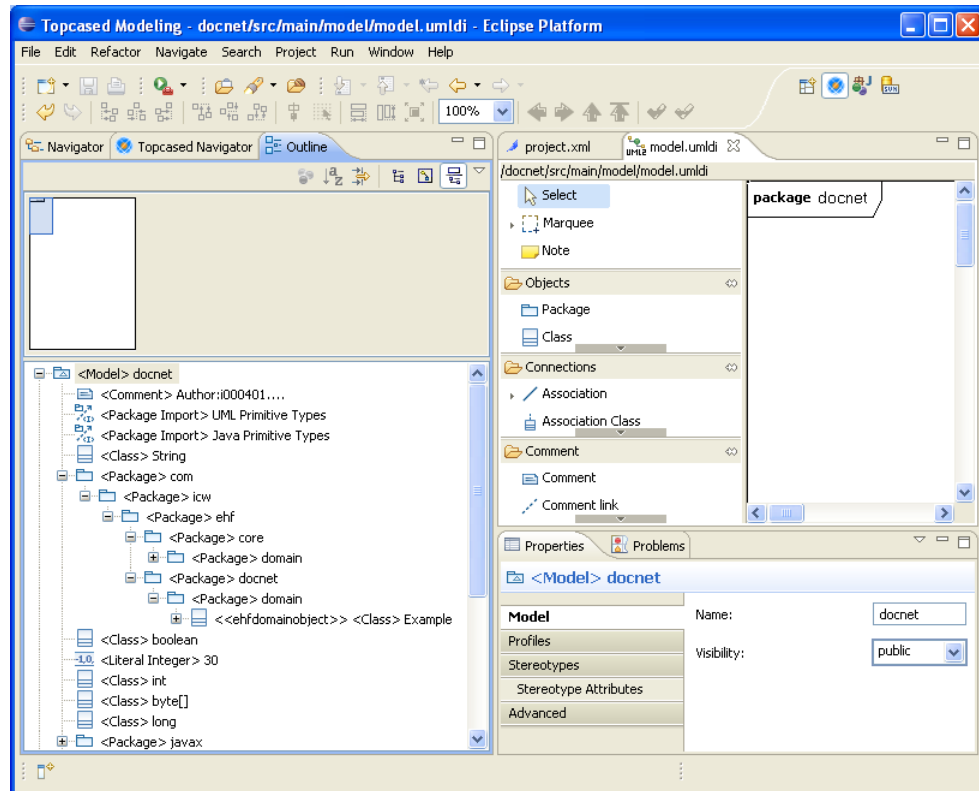


Figure 6: Initial UML Diagram View in Eclipse

4. The class diagram itself can be seen at the bottom of the **Outline** view tree as "**Class Diagram No name**". We don't actually need this diagram, so you can right click on it and select **Delete From Model** to get rid of it.
5. In [Figure 6](#), if we look at the **Outline** view we can see that the package `com.icw.ehf.docnet.domain` has already been created for us, along with the class `Example`.

We do not need the package `com.icw.ehf.docnet.domain` or the `Example` class and these can be safely deleted. Simply right click on the package `docnet` from the **Outline** view and select **Delete From Model** in the shortcut menu that appears. This will then delete the packages `docnet.domain` and the `Example` class in one go.

As well as the default `docnet` package you should also see that the model already contains a `com.icw.ehf.core.domain` package, along with the three classes: `Code`, `CodeSystem` and `Date`. They are marked with the stereotype *ehf-domainobject* and the related attribute *stub* is set to true. (This can be seen by first selecting one of the classes in the Outline view. Then select **Stereotypes** from the Properties window. The stereotype *ehf-domainobject* (from the *eHF_Profile*) is shown under **Applied Stereotypes**, while the *stub* attribute is marked as *true* under the section **Stereotype Attributes**).



Note: UML Stereotypes and their attributes

In UML class diagrams all classes have the same semantic, it's a class. Stereotypes add additional semantic to your UML model for distinguishing the different classes of your model. This information is evaluated by the eHF Generator (introduced later) that is responsible for building the Java based architecture out of your model.

Additionally, stereotypes can have attributes (often known as "tags") for fine-grained semantics.

Later in this document, you will see stereotypes such as *ehf-domainobject* and corresponding attributes like *crud*, *expose* and *export*.

As for *stubs* they basically define dependencies to other modules. Telling the generator not to generate new classes or persist them, but rather that they will be included as value objects within our new classes and therefore persisted from there.

We will be using these later during the development of our model, so we do not need to delete them.

Having set up Eclipse and deleted the example package, we are now ready to create our DocNet package and domain objects. Our main package will be `com.mycompany.docnet.domain`, while our two domain objects will be `Program` and `Appointment`. We'll start by creating the packages.

1. From within the **Outline** view, right click on the `com` package and from the drop down menu that appears select **Create child -> Packaged Element -> Package**. You should then get an empty **Package** element created as shown below:

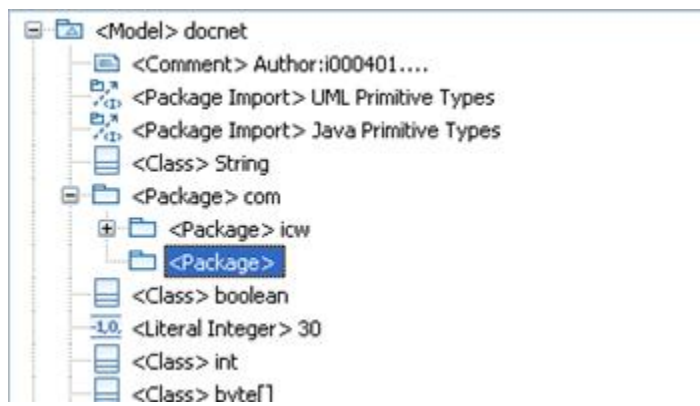


Figure 7: New Package Element in our Model

2. With the new package selected in the Outline view (as shown in [Figure 7](#)), move to the **Properties** window and select the **Model** section before entering the **Name** - `mycompany` as shown in the following figure:

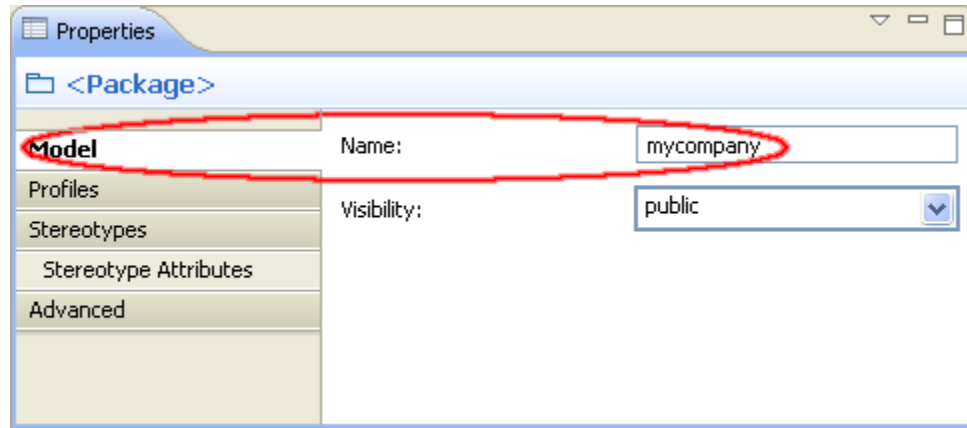


Figure 8: Setting the Package Name to `mycompany`

3. Repeat the previous two steps to create the packages `docnet` under `mycompany`, and `domain` under `docnet`.
4. Next we need to create the two classes `Program` and `Appointment`. Before we do this however we will create a new class diagram under our `domain` package. We can then use this diagram to create our classes. Anything we create in this diagram will then automatically be saved under the package `com.mycompany.docnet.domain`.

Right click on the new package `domain` from the **Outline** view and select **Add diagram -> Class Diagram**. This will then automatically be opened and shown in the main window with the title "**package domain**" as shown below:

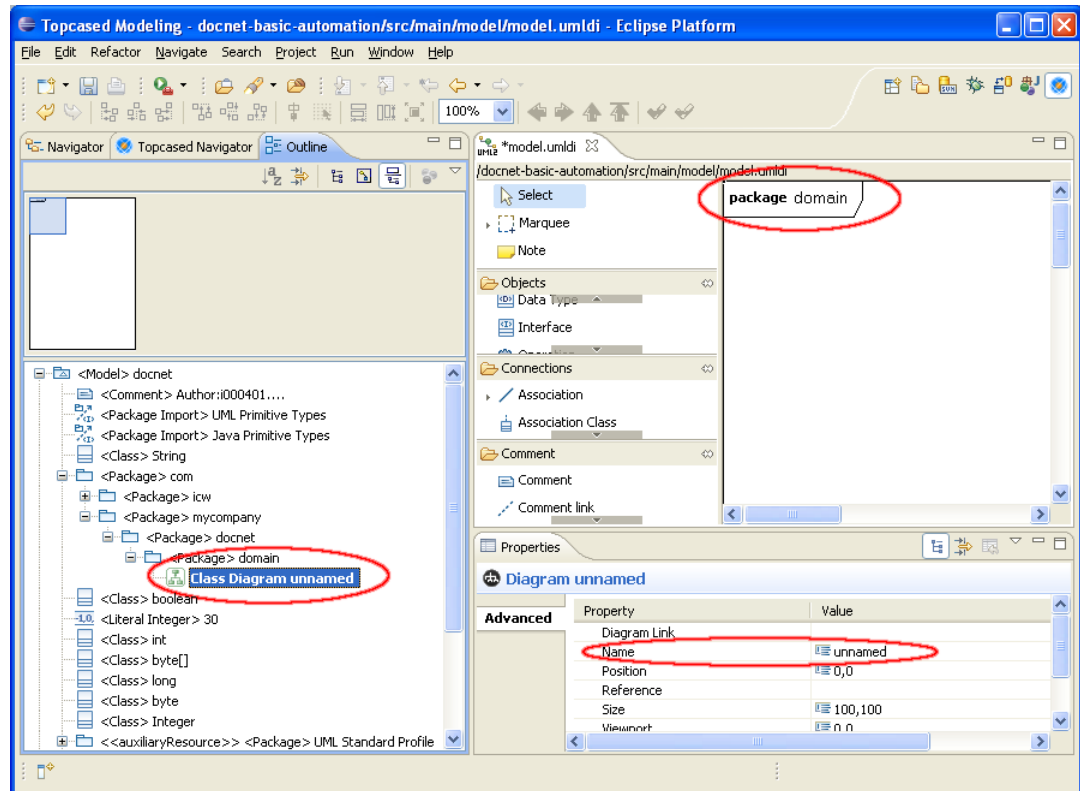


Figure 9: Newly Created Class Diagram

Click on the newly created **Class Diagram unnamed** in the **Outline** view and change its name to **docnet** in the **Properties** window, as highlighted in red in the previous figure (this isn't really necessary but makes the whole thing more readable for us).

- From the main modeling window click the **Class** button (highlighted in the following figure) and then click once anywhere in the diagram to create a new class (the mouse pointer should change to show a little plus symbol as soon as you move over the diagram after selecting the Class button).

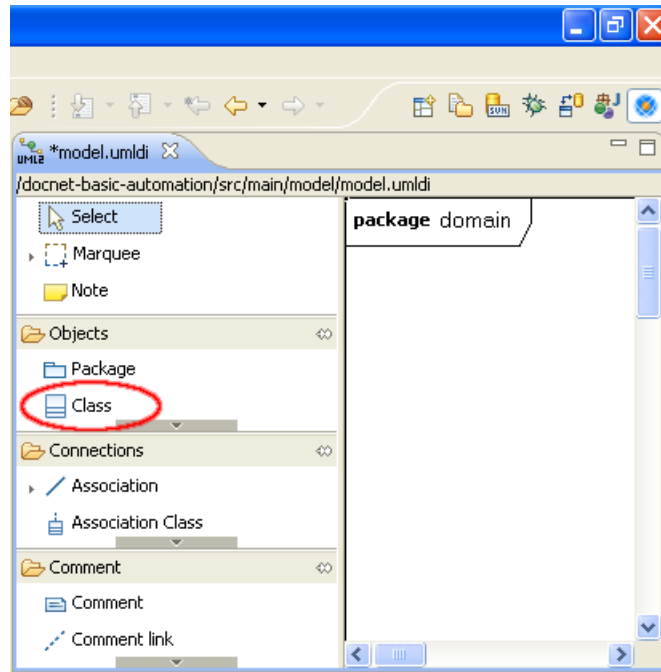


Figure 10: Creating a New Class in Topcased

Enter the name `Program`. When you first add the class to the diagram the name should automatically be highlighted in the diagram for you to edit it. Alternatively you can right click on the class in the diagram and select **Rename**, or you can select the class in the diagram and edit the name in the **Properties** window.

6. Repeat the previous step to create the class `Appointment`.
7. You should now have the same Outline view structure as shown in the following image:

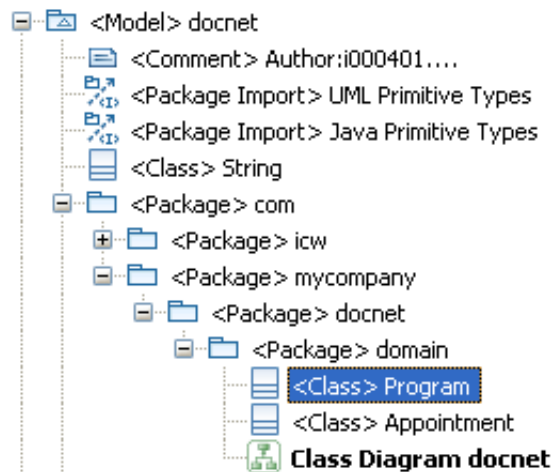


Figure 11: Outline view Showing Our New Docnet Domain

8. Next we need to apply the stereotype `ehf-domainobject` to our classes.

The `eHF_Profile` defines various stereotypes which can be used to identify UML model elements of specific interest to the eHF Generator. These stereotypes let the generator know what it needs to create for each of the marked objects during the generator run.

By marking our classes as *ehf-domainobjects* we are telling the generator that this class represents a standard domain object which in terms of the eHF means the generator will create an appropriate persistence architecture.

For complete details on this and all other available stereotypes, please see the Generator section of the eHF Reference Documentation.

- From within the **Outline** view, right click on the class `Program`, and from the drop down menu that appears select **Apply Stereotype**, in the window that appears select *eHF Profile::ehf-domainobject* in the left hand window and click **Add**. The stereotype will then be listed as a selected Feature as shown below:

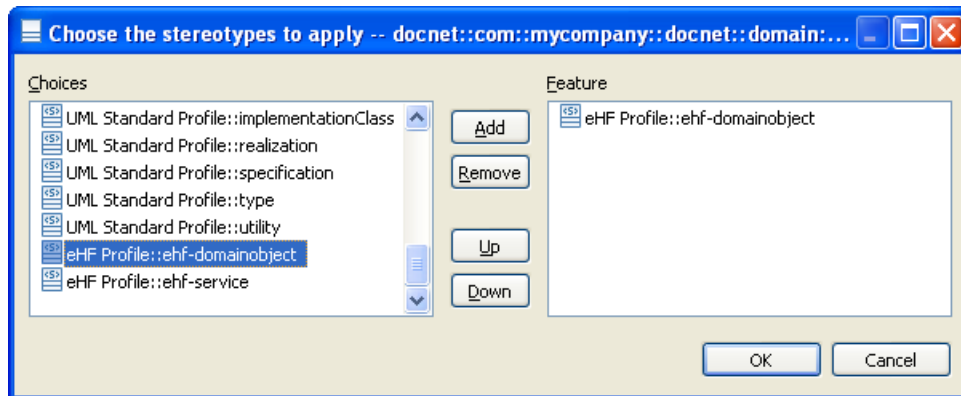


Figure 12: Selecting the *ehf-domainobject* Stereotype

Click **OK** to apply the stereotype. You should immediately see the change in the model diagram:



Figure 13: Program with *ehf-domainobject* Stereotype

Note: the `Program` class in the Outline view may not appear to be updated immediately - simply save your model diagram to have the change visible in the Outline view.

- Repeat the previous step for the class `Appointment`.



Note: Alternative Method for Applying Stereotype

An alternative method for applying the stereotype is to simply select the class in the Outline view and then set the stereotype via the Properties window.

- Our diagram should now look as shown in the following figure:



Figure 14: Initial DocNet Class Diagram

12. Next we want to create the `name` and `description` attributes for our `Program` class.
13. From the main modeling window click the **Property** button and move the mouse over the `Program` class in the diagram and click once with the left mouse button to add an attribute to the class - the mouse pointer should change to have a small plus symbol next to it when moved over the class in the diagram. The **Property** button is circled in the following image:

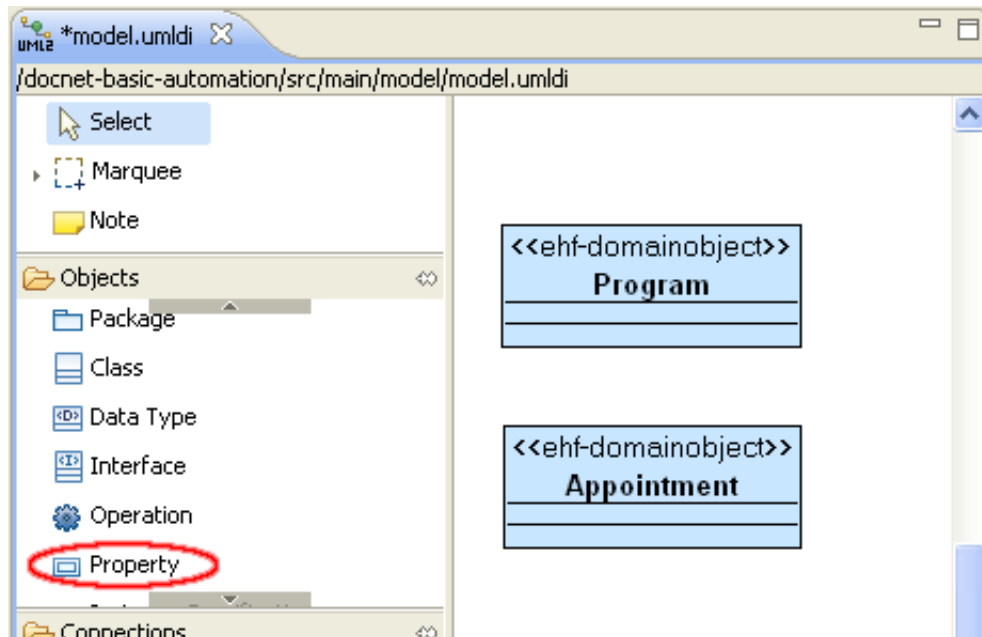


Figure 15: Adding a Property in Topcased

When you first add the **Property** it should automatically be editable so that you can modify the name of the property - set the **name** to be `name`.

Next click on the new `name` attribute of the `Program` in the class diagram to make sure the **Properties** window for the attribute is shown. (you can also set the name of the attribute to `name` via the Properties window) With the **Model** section selected set the **Visibility** to `private` and uncheck the **isUnique** checkbox.

Still in the Properties window of the `name` attribute we will now set the **Type** to be `<Class> String`. Click the **Type selection button** (circled in red in [Figure 16](#)) to open the **Object selection** dialog and select `<Class> String` from the list and click **OK**.

The Property window for the `name` attribute should now be as shown in the figure below:

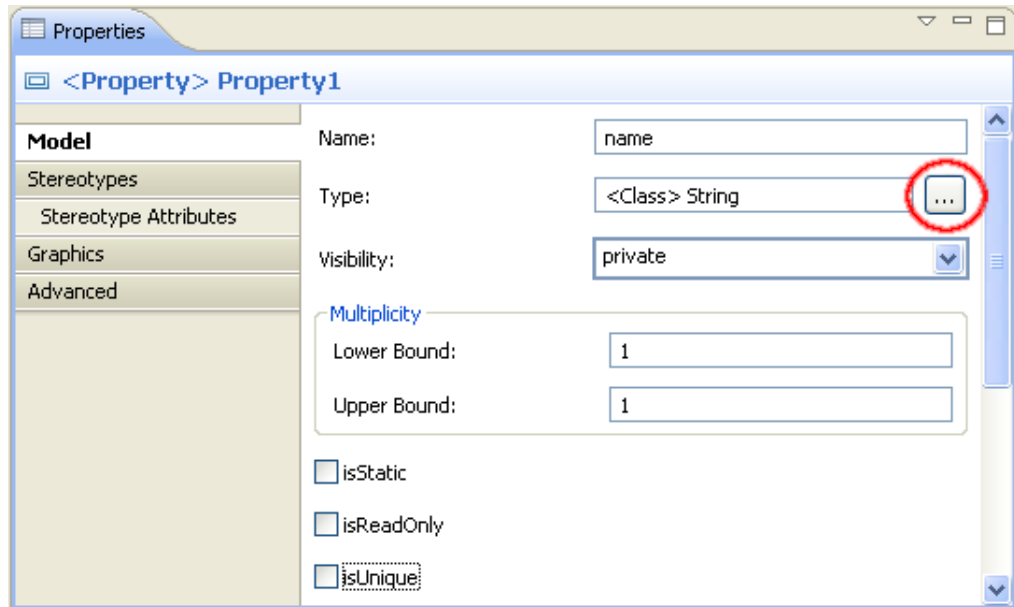


Figure 16: Program's name Attribute Properties Window (Type selection button highlighted in red)

14. Repeat the previous step to add a second attribute to the `Program` class with the following properties:
 - **Name** = `description`
 - **Type** = `<Class> String`
 - **Visibility** = `private`
 - **isUnique** = unchecked
15. In the same way as for the `Program` class we now need to add three new attributes to the `Appointment` class.

All three attributes should have a **Visibility** = `private`, and the **isUnique** checkbox should be **unchecked**.
16. The first attribute has the following details:
 - **Name** = `name`
 - **Type** = `<Class> String`
17. The second attribute has the following details:
 - **Name** = `description`
 - **Type** = `<Class> String`
18. The third attribute has the following details:
 - **Name** = `date`
 - **Type** = `<<ehfdomainobject>> <Class> Date`



Note: Date Type

The `Date` type `<<ehfdomainobject>> <Class> Date` is actually one of the stub classes that we automatically have defined for us in our model, it can be found under the package `com.icw.ehf.core.domain`.

19. Next we need to create an association between our `Program` and `Appointment` classes.

From the main modeling window click the **Association** button (shown in the following figure) and move the mouse pointer to be over the `Program` class in the diagram and click once with the left mouse button. Next move the mouse over the `Appointment` class and click once with the left mouse button. This will then create the association between the two classes.

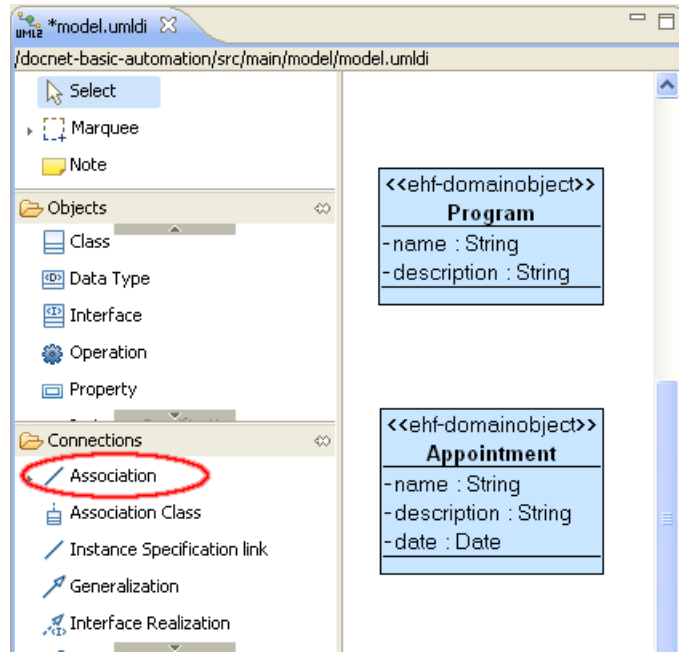


Figure 17: Creating an Association in Topcased

Click on the newly created association in the diagram to have its properties displayed in the **Properties** window. From there select the section **First End** and set the properties as follows (those properties not listed below can be left with their default settings):

Property	Value
Role	<<empty>> (will most likely have a default of "program" - simply delete this so that the field is empty)
isNavigable	unchecked
Association Type	none
Property Visibility	private
Multiplicity Lower Bound	1
Multiplicity Upper Bound	1
isUnique	unchecked

Table4. Association Between Program and Appointment First End Properties
 Still within the Properties window of our new association select the section **Second End** and set the following properties:

Property	Value
Role	appointments (will most likely have a default of "appointment" - simply change to be "appointments")
isNavigable	checked
Association Type	composite
Property Visibility	private
Multiplicity Lower Bound	0
Multiplicity Upper Bound	*
isUnique	unchecked

Table5. Association Between Program and Appointment Second End Properties
Finally, still within the Properties window select the section **Model** and delete the name entry. It will then be given a default name of `A_program_appointments` - this can be seen immediately in the Outline view.

20. The class diagram should now look as follows:

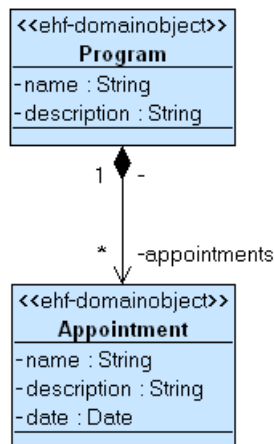


Figure 18: Final Class Diagram Including Association

21. Finally we need to set the *ehf-domainobject* stereotype's attribute *crud* to `true` for our `Program` class. This boolean flag states that a CRUD (Create, Read, Update, Delete) service implementation should be provided for the domain object by the generator.

There are numerous attributes available for use on objects marked as an *ehf-domainobject*. These attributes can be used to further qualify the characteristics of the domain object class instances, and therefore control the behavior of the generator. The CRUD attribute is just one example.

22. Select the `Program` class in the diagram to have its properties shown in the Properties window.

23. In the Properties window on the left hand side click on the section **Stereotype Attributes** and then in the right hand side click on the word `false` for the **Crud** attribute. You will then receive two options - select `true`, as shown below:

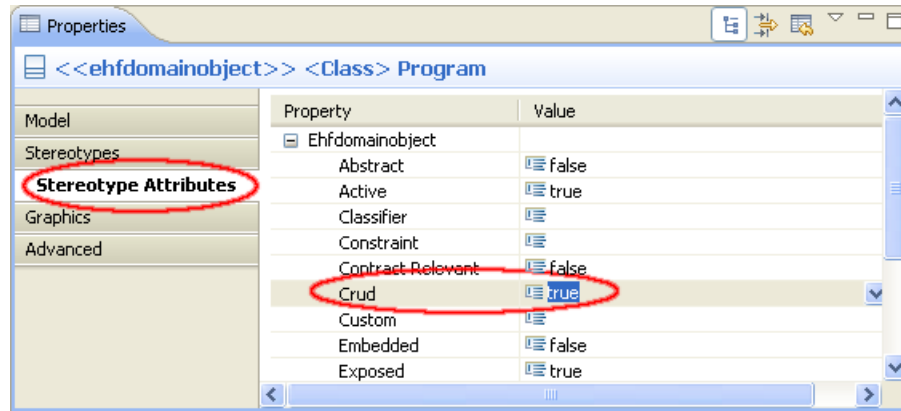


Figure 19: Setting the Crud Attribute to true for the Program Class



Note: No CRUD required for an Appointment

We do not need to set the `crud` attribute for the `Appointment` class. Due to the association between `Program` and `Appointment`, an `Appointment` can be created by adding the `Appointment` to the related `Program` via appropriate methods in the `ProgramService`.

There are numerous other attributes, other than `crud`, available for our `ehf-domainobject` stereotype. A complete list of these attributes and how they can be used can be found in the generator section of the eHF Reference Documentation.

24. With that our initial class diagram is now complete - make sure to save the changes you have made in the `model.umldi` file. This will also automatically save the required changes in the `model.uml` file as well.

3.2 The HSQL Database

Now that our class diagram is complete we are almost ready to generate our code with the help of Maven. Just before we do so though, we need to make sure that we have a suitable database in place. Maven will try to create the database schema as part of the build process, for use in the JUnit tests.

During eHF development we use the HSQL database (<http://hsqldb.org/>). HSQLDB is a java database which has a very small size and can run completely in-memory, making it ideally suited to development environments.

By default, through the project templates used with Maven's `genapp` plugin, our DocNet module has already been configured to use a HSQL database with the name "testdb". See the [HSQL Database Configuration](#) on page 103 appendix, for details on where and how the use of this HSQL database is actually configured in the DocNet module

If we start a HSQL database before building our module with Maven, then Maven will create the database schema in this running database while building the project. Maven will then leave the database running once it is finished. We can then use this database while creating and running our JUnit test cases in Eclipse.

3.2.1 Starting a HSQL Database

If you have installed the ICW IDE, you can simply start an appropriate HSQL database using the file *server-start.bat* in the `<<ICW_IDE_INSTALL_FOLDER>>/database/hsqldb/bin` directory.

If we don't explicitly start a database before building our project, Maven will simply start the HSQL database from the HSQL jar file, included in the classpath of our project, at the appropriate time (for when it runs the module's JUnit tests). Maven then stops the database once it has finished. However this means that you will be unable to look at the database structure afterwards, as by default all changes to the database will be lost once it is shut down, as it is running in-memory.

3.2.2 Shutting Down a HSQL Database

As with starting the HSQL database, if you have installed the ICW IDE, you can cleanly shutdown the database by simply using the *server-stop.bat* file in the `<<ICW_IDE_INSTALL_FOLDER>>/database/hsqldb/bin` directory.

As well as shutting down the database it will also persist any changes that have been made. Simply closing the command window that the database is running in or shutting down via the use of [Ctrl-C], will not necessarily persist the changes.

Alternatively, you can open the SQL database client of your choice, connect to the database and execute the command "shutdown" in the SQL command editor window.



Note: Not Using the ICW IDE?

If you have installed HSQLDB on your own (because you didn't install the ICW IDE) you will need to complete some extra configuration that is not included with the default install of HSQL. [Database Configuration](#) on page 103 provides details on this configuration.

3.3 Building DocNet

Having created our class diagram and made sure that Maven can find an appropriate database, we are now ready to build our project.

1. (Optional) Start your HSQL Database - this will then be used by the build process to initialize the database for us. So we will then automatically have a suitable database for use in our JUnit tests.
2. Open a console window, and navigate to the `docnet` folder in your workspace.
3. Enter the following command:

```
maven dev:build
```

Among other tasks the eHF Generator will automatically run as part of this Maven goal.

4. Once the build is complete, open Eclipse and refresh the **docnet** project view (simply select the `docnet` project in Eclipse's package Explorer and press **[F5]**). You should now be able to see what was produced by the generator, as shown in [Figure 20](#).

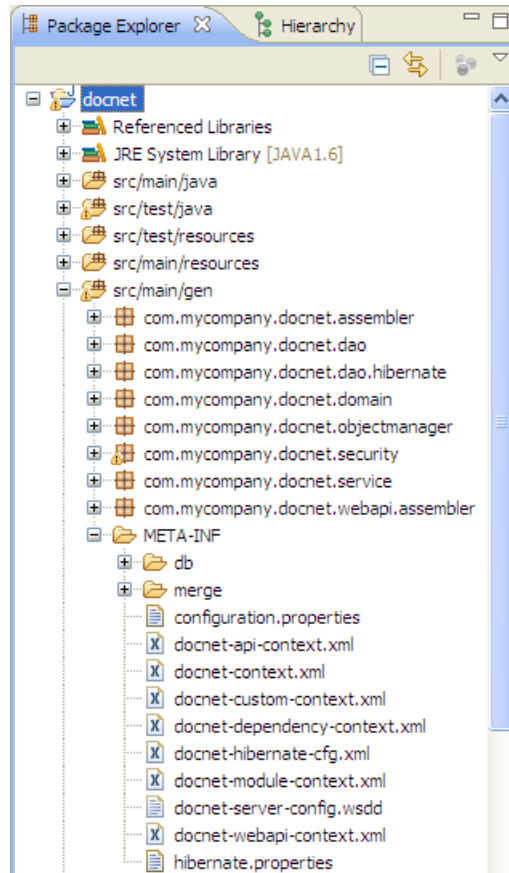


Figure 20: Folder Structure Showing what the Generator Creates After the First Run

We will describe the architecture behind these classes in section . However, at this point, we will describe some "technical" aspects of the generated artefacts:

- The generator created several Java files in `src/main/gen` and `src/main/java` (our two main locations for Java source files).
- Furthermore, there are several configuration files in the folders `src/main/gen/META-INF`, `src/main/resources/META-INF` and `src/main/config`.
- You need to make sure you do NOT edit any file under the folder `src/main/gen` as these files will be replaced by the next generator run. It is best practice not to add these generated files to your version control system like cvs or svn as these files can be generated at any time by simply executing `maven dev:build` from the project folder.
- You are allowed to edit files in the folders `src/main/java`, `src/main/resources/META-INF` and `src/main/config`. These files are only created during the first generator run. Future generator runs will detect the existence of these files and will not override them.
- Some files in `src/main/gen` contain the following comment: "THIS FILE WAS GENERATED FOR YOUR SUPPORT. FOR CUSTOMIZATION SIMPLY MOVE IT TO YOUR MAIN SOURCE. " You can move these files to the corresponding package in your `src/main/java` folder. The generator will detect this move and will not regenerate the file during the next run.
- The Hibernate produced SQL statements (as part of the `dev:build` process) for creating a database schema for our domain model, are in the directory `src/main/gen/META-INF/db/hsq1`. These statements are also automatically executed when running `maven dev:build`, as part of the step that executes the JUnit tests - they

are run before the JUnit tests to make sure an appropriate database exists should the JUnit tests require it.

Additionally if you explicitly started a HSQL database beforehand, you should be able to see that it has created a docnet schema, containing two tables; T_PROGRAM and T_APPOINTMENT.

3.4 The Generated Module Architecture

In this section we will take a look at the general architecture of an eHF module. This architecture is created for us by the eHF generator based on the UML model of the module. Elements of this architecture can then be extended by the developer.

Additionally through the model, and specifically the stereotypes (e.g. *ehf-domainobject*) of the *eHF Profile*, and their corresponding attributes (e.g. *crud*) we can control exactly which aspects of the architecture are actually created by the generator in each concrete case.

In this section we will give you an overview of the complete architecture stack that the generator can potentially produce. With this understanding we will then take a look at what we have actually received in the DocNet case and how we can then change this.

3.4.1 General Module Architecture Stack

Figure 21 shows the three layers of the module architecture stack:

1. Persistence
2. Service
3. Service Adapter

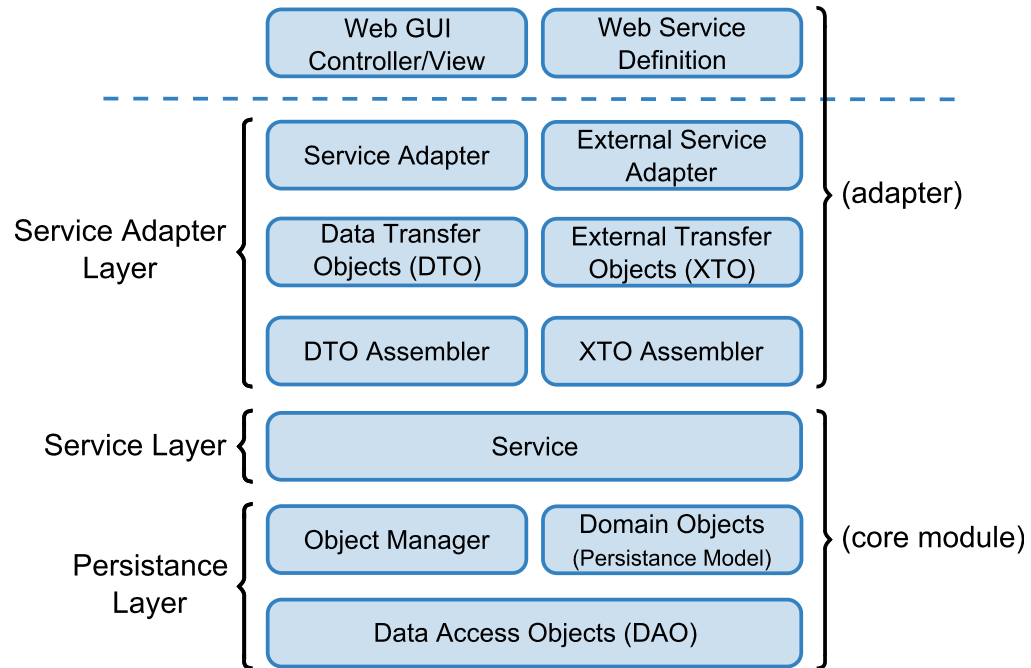


Figure 21: General Architecture of eHF modules

Persistence Layer

In the *Persistence Layer* the **Data Access Objects (DAO)** are responsible for all persistence oriented operations; creating, reading, updating and deleting **Domain Objects** (so the standard CRUD operations). The DAOs are implemented using Hibernate.

The **Object Manager** provides functionality to handle objects between the *Service Layer* and the *Persistence Layer*. It offers methods to supplement the integrity of a domain object graph before being persisted and deals with the overall consistency of the requested objects. For example, the Object Manager implements aspects for handling security and validation.

Service Layer

On top of the *Persistence Layer* we have the *Service Layer* which is where the module's logic is implemented. The *Service Layer* operates on **Domain Objects**.

The *Service Layer* is never directly accessed by other modules (such as GUI, web service or business modules).

Service Adapter Layer

Finally on top of the *Service Layer* we have the *Service Adapter Layer*. This forms the API of the module, and is what is then used by other modules.

There are two sides to the API:

1. **Service Adapter** - this is known as the *internal* API. It is basically API that is used when binding via Java. It is known as the "internal" API, as it is only available internally to other modules within the same JVM instance as our module's code. Clients can use this by adding the module's jar file to their classpath.
2. **External Service Adapter** - the *external* API. This is accessed via web service calls. It is the "external" API as it can be accessed by other external applications.

Both adapters always operate on transfer objects, **Data Transfer Object (DTO)** for the Service Adapter and **External Transfer Object (XTO)** for the External Service Adapter. To this end in the Service Adapter Layer we also have **DTO / XTO Assemblers** for assembling and disassembling the Domain Objects and Transfer Objects as they are passed between the *Service* and *Service Adapter* layers.

The advantage of having a *Service Adapter Layer* on top of the *Service layer* is that you can hide internal details of the **Service** and **Domain Objects**. Furthermore, there are differences if an external module accesses the module internally (via the Service Adapter) or externally (via the External Service Adapter). For example, Service Adapters are able to handle *Collections*, while External Service Adapters use *Arrays*.

There are two forms of the this general architecture stack: the **Object Service Stack** (which contains the CRUD operations for the given domain object, along with any custom methods defined for the object) and the **Domain Service Stack**. These will be examined in the following sections.

3.4.2 The Object/CRUD Service Stack

For each class in the UML class diagram that is marked with the stereotype *ehf-domainobject* and with the *crud* tag set to *true*, a so called *CrudService* (Service Layer) is generated - this can also be known as an *ObjectService*, so in our case *ProgramService*. It provides default methods for Creating, Reading, Updating and Deleting (CRUD) instances of the corresponding domain object. Furthermore, it provides methods for adding and deleting instances of associated domain objects.

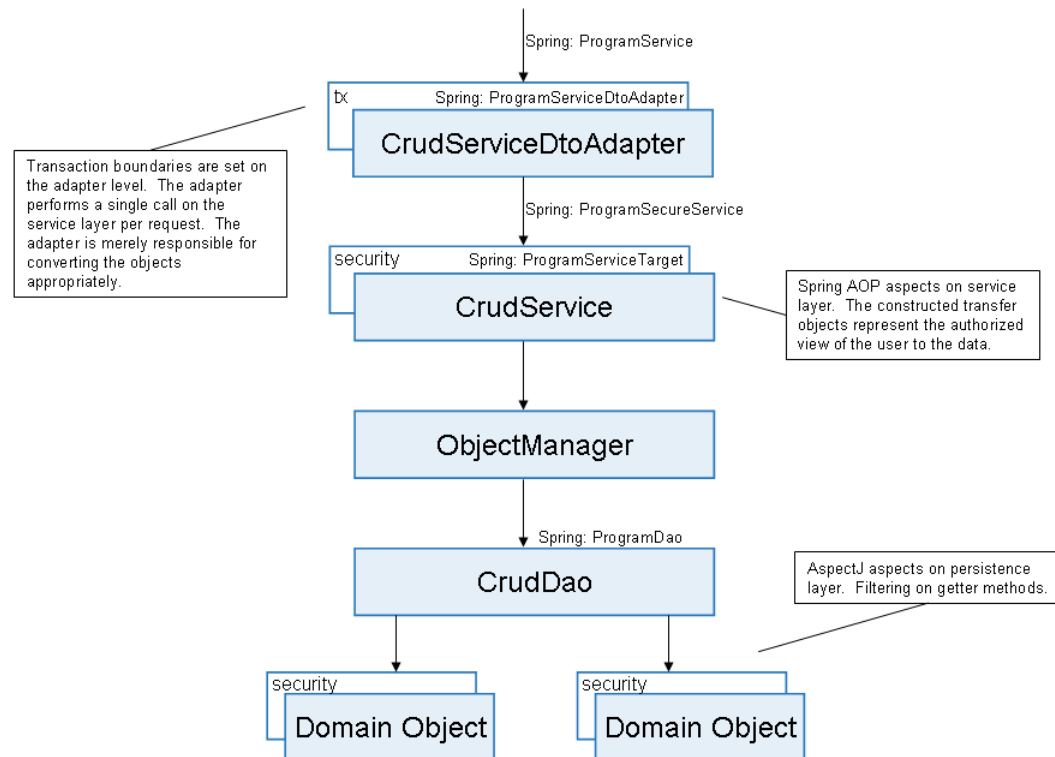


Figure 22: CRUD Service Architecture

A *CrudService* (e.g. *ProgramService* in our DocNet module) is located in the *Service Layer* of the previously described [General Module Architecture](#) on page 27. It uses an *ObjectManager* (e.g. *ProgramObjectManager*) from the *Persistence Layer* for

performing persistence operations. The Object Manager ([The Object Manager](#) on page) performs some additional steps and delegates the call to the `CrudDao` (e.g. `ProgramDAO`).

When marking a class in the UML diagram with the `ehf-domainobject'sexport` attribute set to `internal` (default has it set to `none`), the corresponding `CrudServiceDtoAdapter` (e.g. `ProgramServiceDtoAdapter`) in the *Service Adapter Layer* is created for usage by external modules.



Note: No External Service Adapter (so Web Service) for CRUD Services

The `CrudServiceDtoAdapter` is the internal service adapter, an external service adapter is not generated for CRUD services!

As shown in [Figure 22](#), security and transaction aspects are placed around some components.

Security

This mainly stands for authorization, which is the mechanism by which access to domain objects and system functions are either permitted or denied. Authorization is transparently introduced via aspects and interceptors. As shown previously in [Figure 22](#), there are security aspects in the *Service Layer* and the *Persistence Layer*.

These aspects are the entry points for authorization. They delegate the actual authorization decision to the so called **Security Service** which decides for a given domain object (e.g. a `Program` instance) if the current user is allowed to execute a given action (e.g. `READ` or `UPDATE`). The authorization aspects in the CRUD service stack react on negative decisions from the Security Service.

This will be clearer, if we show some example security aspects and their reaction on negative authorization decisions:

1. If the user is not allowed to carry out the `CREATE` action for a specific domain object then a listener in the `ObjectManager` will throw an `AccessDeniedException`.
2. If the CRUD service in the *Service Layer* returns a list of domain objects, the security aspect surrounding this CRUD service will remove all domain objects that the user is not allowed to read from the resulting list. This behavior is called *filtering*.
3. If the business logic operates on domain objects and navigates on the object graph using the related getter methods (e.g. `program.getAppointments()`), an aspect surrounding the domain object filters the returned domain object lists of the getter method as appropriate.

More complete details on authorization and its implementation can be found either in the Security Tutorial, or in the appropriate chapter of the eHF Reference Documentation.

Transactions

Each method of a Service Adapter (*Service Adapter Layer*) is surrounded by a transaction aspect. This is achieved via the `@Transactional` annotation from Spring, defined on a super class of the service adapter. So, when calling a method of this service adapter the default transaction handling of the Spring Framework is applied:

1. Starts a new transaction (if one does not already exist)
2. Commits the transaction on return or if a checked exception is thrown
3. Rolls back the transaction if a `RuntimeException` is thrown

Further details on Spring's transaction handling can be found in Spring's reference manual, chapter "[Transaction Management](#)".

The Object Manager

Throughout all of this we have the `ObjectManager` which augments the *Persistence Layer*. It provides an extended event mechanism for persistence and audit operations on object graphs. Furthermore, it manages security relevant attributes (discussed later) such as `scope`, `classifier` and `role` (this is called *endorsement*) and validates the objects according to constraints described on object level (using the Object Constraint Language (OCL)) or attribute level (e.g. `nullable`). (We'll take a look at how to define constraints for validation in [Adding Constraints](#) on page 56)

Also the `ObjectManager` handles so called **Links**. Links is an extended mechanism for making references on other objects. In contrast to normal foreign key constraints on database level, the Link mechanism allows for example for the separation of medical and administrative data for data protection reasons. When loading an object, the `ObjectManager` completes the object graph by loading linked objects. When persisting an object, the `ObjectManager` cascades the persistence operation to the linked objects. For more information about Links, consult the eHF Reference Documentation.

Lastly, the `ObjectManager` handles the task of updating object associations. This is required because the authorization module can filter information from the object graph. For example, when loading an object, authorization can delete (filter) associated objects, that the user is not allowed to see, from the object graph. When this object graph is to be updated, Hibernate would normally delete the filtered objects. So, the `ObjectManager` ensures that these filtered objects will not be deleted. For more information about authorization and filtering objects, see the eHF Security Tutorial.

CAUTION: Rules for the Using the `ObjectManager`

The `ObjectManager` is automatically used by the default CRUD operations created by the generator. However when you define your own service operations on CRUD services (see section 4.3) or when you create a Domain Service, then you have to manually ensure that the `ObjectManager` is used in these cases:



1. If you want to use validation, you have to use the `ObjectManager` when creating, updating and deleting objects.
2. When your objects have Links to other objects, you have to use the `ObjectManager` for each CRUD operation to handle these Links.
3. If you want to audit access to your data, you have to use the `ObjectManager`.

3.4.3 The Domain Service Stack

In [The Object/CRUD Service Stack](#) on page 29 we discussed how the generator creates an Object/CRUD Service stack for *each* domain object in our UML diagram with the stereotype *ehf-domainobject* and the related attribute *crud* set to `true`. Normally

though you can not solely base an application on these CRUD services (possible, but unlikely). We therefore need a suitable location where we can place custom application logic, integrate all the module's CRUD services and also communicate with other modules (module interdependencies).

So in addition to the Object Service Stacks, the general architecture stack, as described in [General Module Architecture Stack](#) on page 27, can also be generated for custom domain services. You can define these as interfaces in your model, and mark them with the *ehf-service* stereotype so that generator knows how to process them appropriately. Complete details on how to do this, along with what the generator actually creates for us is shown in



CAUTION: Interfaces

In the previous sections, we have only named the interfaces involved. There are further classes (implementing these interfaces, class hierarchies, technical aspects) in the named layers. They will be discussed in the [Java Classes](#) on page 32 section.

3.5 The Java Classes

As we have now discussed the general architecture of our new module, we will take a look at some of the specific classes generated in our DocNet project.

The generated java files are divided into two source folders:

1. `src/main/java` - under the developer's control
2. `src/main/gen` - under the generator's control

When discussing the generated files in the following sections, we will mark these files with the following tags:

- **D** - under the developer's control: the generator will not regenerate these classes as long as they exist in `src/main/java`.
- **G** - under the generator's control: the generator will regenerate these classes on each generator run. These files are placed in `src/main/gen`. Don't touch them! Any manual changes you make will be lost on the next generator run.
- **M** - movable files: the generator will regenerate these files as long as they are placed in `src/main/gen`. However if you move one of these files to `src/main/java`, then the generator will no longer regenerate it, so you can safely make manual changes to the file.



Note: Bringing Movable Files Back Under Generator Control

If you want to bring a movable file back under generator control, simply delete it from under `src/main/java`, before the next generator run and it will then automatically be created again by the generator. Obviously you would lose any custom code that you might have created.

Next, we will list the files generated out of our sample UML model and group them by the corresponding architectural layers. We will start with the classes of the *Persistence Layer*, before discussing the *Service Layer* and then ending with the *Service Adapter Layer*.



Note: Package Name Shortcuts

In the following sections we will make shortcuts for package names by omitting the `com.mycompany.docnet` part of the package name. For example for the package `com.mycompany.docnet.dao.hibernate` we will simply name it `dao.hibernate`.

3.5.1 The Persistence Layer

Domain Objects (Persistence Model)

For each class in our model that is marked with the stereotype *ehf-domainobject*, the generator creates a Domain Object as an EJB 3.0 entity bean:

- `domain.Program`, `domain.Appointment` [**M** - movable]

These classes are empty and can be moved to `src/main/java`. You can then extend them with your own logic. The class attributes you have specified in the UML model are not contained in these classes, but are instead located in abstract super classes (see the next point).

- `domain.ProgramBase`, `domain.AppointmentBase` [**G** - generator control]

These classes are under generator control and implement the class attributes specified in your UML model. These base classes are extended by the `Program` and `Appointment` classes respectively.

Furthermore these *base* classes implement the `domain.DocnetDomainObject` [**G**] interface which simply marks the domain objects as being related to our DocNet module.

Data Access Objects (DAO)

For each class in our UML model that is marked as *ehf-domainobject*, the generator will create these files as part of the CRUD Service Stack:

- A DAO interface (`dao.ProgramDao`, `dao.AppointmentDao` [**G**]). These interfaces are empty and extend the `CrudDao` interface which provides the CRUD persistence operations.
- A Hibernate DAO implementation (`dao.hibernate.HibernateProgramDao`, `dao.hibernate.HibernateAppointmentDao` [**M**]). These files are empty and extend the general `HibernateCrudDao` class - these are used to allow for defining custom DAO logic should it be required.

Object Manager

For each class in our UML class diagram marked as an *ehf-domainobject*, an object manager is created (`objectmanager.ProgramObjectManager`, etc [**G**]).

Additional Files

- The files `Program.xml` and `Appointment.xml` [**G**] contain object meta information, such as OCL (object constraint language) constraints and attribute meta data.

Defining constraints for validation is discussed in [Adding Constraints](#) on page 56.

- There are EMF (Eclipse Modeling Framework) based adapters created for the domain model (contained in the `emf` and `emf.model` packages) for accessing OCL constraints defined in the meta data.

3.5.2 The Service Layer

CRUD Service Stack

For each `CrudService`, there is an abstract and concrete service object. Furthermore there are interfaces for each service object. The abstract service is not changeable.

- `service.AbstractProgramService` [G],
 `service.AbstractProgramServiceImpl` [G]
- `service.ProgramService` [G],
 `service.ProgramServiceImpl` [M]

Security

For each Object/CRUD service, there is a method interceptor that is configured for handling security aspects. In our case there is a `security.ProgramServiceSecurity` [G, M] as soon as you define your own operations for the CRUD service].

Links

The eHF system provides *Links* as an extended association between objects. There are some classes to provide links. If your model does not require the use of dynamic links, you can safely ignore this class.



Note: Service Adapter Layer

For now we will not cover the Service Adapter Layer, but will instead revisit this when we implement our own API (and therefore actually generate this layer) in [Creating an API](#) on page 48.

3.5.3 External API

CRUD Service Stack

In the Crud Service Stack, there is no External Service Adapter.

Module Service Stack

On the module level, the `webapi.service.adapter.DocnetModuleServiceXtoAdapter` [D] interface defines the external API (implemented by `webapi.service.adapter.DocnetModuleServiceXtoAdapterImpl` [D]). The module service API is under the full control of the developer.

External Transfer Objects (XTO) and Assembler

Similar to the internal API, the domain model of the Persistence Layer is not a public model. Instead there are XTOs (`webapi.transfer.AppointmentXto` [M] and `webapi.transfer.ProgramXto` [M]). The related assembling / disassembling mechanism is located in the `webapi.assembler` package [G].

3.6 Spring Configuration

Spring plays a central role in the infrastructure of eHF Modules. The initial context configuration files are all created for you by the generator. Some of these stay under the control of the generator throughout, while others can be modified by the developer. [Figure 23](#), shows the various Spring configuration files used to define a module's complete Spring context - and those that stay under complete control of the generator at all times.

Spring plays a major role in our public API and modularization concept. The **Module Context** defines the module's Spring public API. It declaratively states the beans that the module **exports** to the platform as well as those that it needs to **import** from the platform. Only the beans exported in this file are available to the platform, all other beans defined in the rest of the module's spring context files are not available outside of the module (hence the little security logo next to the Module Context in [Figure 23](#)). This benefits the consumer in that they only have to worry about a small set of public beans, while it benefits the developers who can refactor internal beans without risking any incompatibilities with existing service consumers.

Don't worry about these files at the moment, we don't need to modify them just yet. For our purposes the default content as created by the generator is fine.

If you want to know more about how the module's normal and test Spring contexts are built, complete details can be found in the appendix [Spring Contexts](#) on page 105. While complete details on the Public API and Modularization concepts can be found in the eHF Reference Documentation.

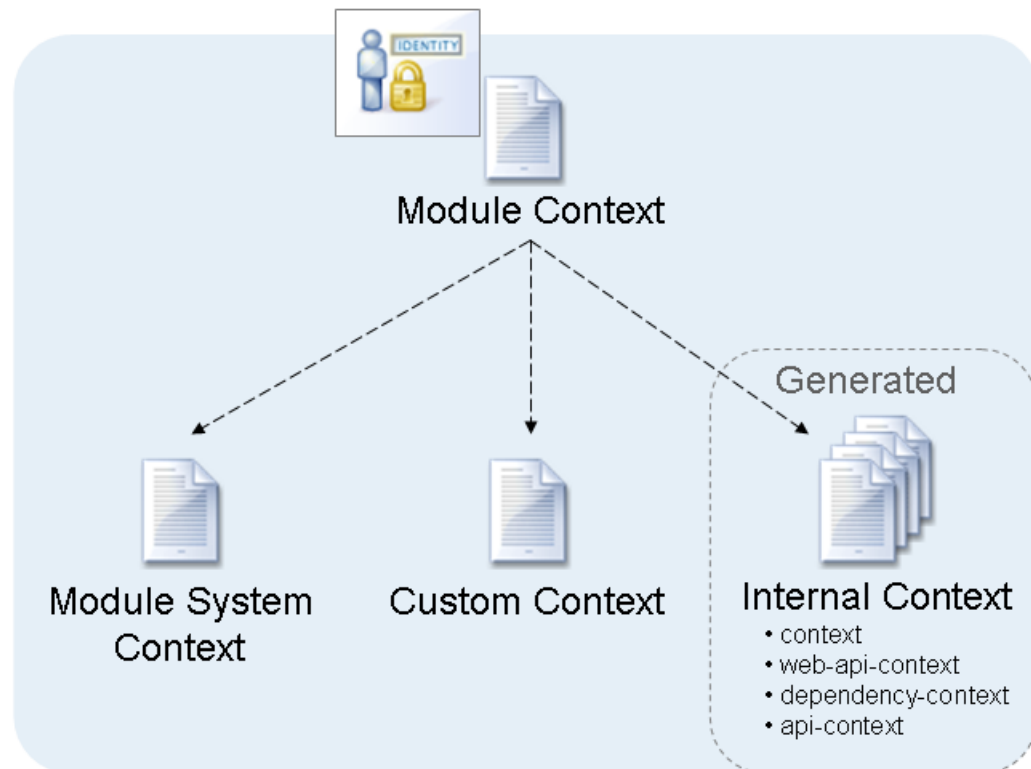


Figure 23: A Module's Spring Context

3.7 Configuration Files

As well as Spring there are various other configuration files generated for us by the build process. Among others these include Hibernate and Maven configuration files. To keep the tutorial as simple as we can, we will not discuss all of these configuration files here. Throughout the tutorial we detail them as much as is necessary to be able to move forward with the tutorial. At the appropriate points we will link to the appropriate eHF documentation which can provide more details.

3.8 Testing

As is best practice, we will now write some JUnit tests for testing our module. This is also a good way to show you how to use the generated services and other artifacts.

When you are developing your own module and start writing your tests, we recommend that you take 5 minutes to read the Testing Conventions section of the eHF Reference documentation. This will give you an idea of the best practices and conventions employed throughout the eHF.

Our first tests won't contain anything too complicated. We will simply test the CRUD service architecture stack that has been created for us by the eHF Generator for our `Program` class. Working on the *Service Layer* (Figure 21) we will create `Program` and `Appointment` domain objects and make sure that these are then successfully stored in the database.

3.8.1 Spring & JUnit Tests

We currently use JUnit 4.4 for our JUnit tests. As stated already the Spring Framework is used throughout the eHF, and our JUnit tests are no exception to this rule (where appropriate). For JUnit 4.4, we use the Spring TestContext Framework and the custom runner class that it provides. This allows us to use an annotation-drive approach to target our test context and populate fields.

We will use Spring to start a test system context. As well as initializing the required beans from our DocNet module, it will also *mimic* the platform context required for our tests. Such a test system context provides globally used beans (e.g. Hibernate session factory or transaction manager) and service beans contributed by other modules (e.g. security service), that our module would normally receive from the Platform Application's context. Often service beans defined in this test context are either Mocks or Stubs of the real implementations.

Normally in our tests we would import the Spring **Module Context** (Figure 23) of our Module. As previously discussed this Module Context only exports (so makes available) the specific beans from our module that form the Public API (by default only beans from the *Service Adapter* layer). When using this context in our tests we are basically carrying out a form of Black Box testing of our module. However for the tests that we are going to develop just now we will be working on the *Service layer* of our CRUD service architecture. Classes at this level in the architecture stack are by default not part of Public API. So we will need to use the **Internal Context** (Figure 23) in our tests. So for now we are carrying out White Box testing.

The beans that we would normally receive from the Platform Application's context are defined in the file `docnet-system-context.xml` under `src/test/resources/META-INF`. This System Context mimics the platform context and provides those beans that we require for our internal context.

Don't worry too much about the content of these files at the moment, we will define exactly which Spring context files you will need to use shortly.

3.8.2 The Default Module Test

With regard to our tests the generator has already created an example test class for us, so we can take a look at this now to see how we should write our tests.

Open the docnet project in Eclipse, and navigate to the folder `src/test/java`.

Under this folder you should see the package `com.mycompany.docnet` and the class `ModuleTest.java`. This has already been created for us by the generator. Taking a look at the contents you will see the following:

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations={"classpath:/META-INF/docnet-test-context.xml"})
public class ModuleTest {

    @Test
    public void testModule() {
        //TODO - Implement your test here
    }

    /**
     * 'Retro-fitting' the test for compatibility with older test runners,
     * for example the junit task in pre 1.7 versions of Ant.
     * @return adapter to older test runners
     */
    public static junit.framework.Test suite() {
        return new JUnit4TestAdapter(ModuleTest.class);
    }
}
```

The first two annotations for the class are what configure it to run with Spring:

- `@RunWith` - here we define the custom `Runner` class that we want to use.
- `@ContextConfiguration` - lets Spring know in which files the Spring context is defined. So in our case our **Test Context** (`docnet-test-context.xml`).

Looking at this Test Context file you will see that it simply imports the Module Context and a System Context. As mentioned previously the System Context is where we mimic the platform context and provide the beans to the Module Context that it would normally expect to receive from the platform. The Module Context is then our main module context as described earlier - so in this `ModuleTest` we would only be able to use (test) those beans that would be available to the platform. (Black Box testing)

If you want to know more about how the module's normal and test Spring contexts are built, complete details can be found in the appendix [Spring Contexts](#) on page 105.

You'll also notice that our test class does not need to extend any test class from either the Spring or JUnit frameworks. To tell JUnit that a method can be run as a test case we simply use the annotation `@Test` on the method.

The suite() Method

The only negative part to using JUnit 4.4 is that as part of the build process the eHF currently uses the Maven test plugin, which runs with earlier versions < 4.x of JUnit. Therefore, you have to declare the static method `suite()` which returns an adapted test case that the Maven plugin can then handle. Every concrete test class needs to contain an implementation of this method (replacing "`ModuleTest.class`" in the previous listing example with the name of the test class in question).

Having now taken a look at the example test class that has already been created for us we can go ahead and write our tests.

3.8.3 Creating our First Service Layer JUnit Tests

We will begin by first writing an abstract test class, which our concrete test classes can then extend. One of its main functions will be to deal with test bootstrapping.



Note: Missing JavaDoc?

To save space in all the listings shown throughout this tutorial, we do not always include complete JavaDoc comments on our classes or their members. Obviously we would normally recommend that you create suitable comments on all public classes and class members, but hopefully you will forgive us for skipping them in this instance.

The AbstractDocnetServiceTestCase

From within Eclipse, navigate to the docnet project and under the folder `src/test/java` create the package `com.mycompany.docnet.service`.

Within this package we need to create the abstract class `AbstractDocnetServiceTestCase`. The complete listing is shown below. We will go through and explain each part individually in the sections that follow.

```
package com.mycompany.docnet.service;

import org.junit.After;
import org.junit.Before;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;

import com.mycompany.docnet.domain.Program;

@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations = {
    "classpath:/META-INF/docnet-system-context.xml",
    "classpath:/META-INF/docnet-context.xml",
    "classpath:/META-INF/docnet-dependency-context.xml"})
public abstract class AbstractDocnetServiceTestCase {

    /**
     * Default scope for use in each JUnit test case.
     */
    protected static final String TEST_SCOPE = "TestScope";

    /**
     * Initial Program domain object available for each JUnit test case.
     */
    protected Program program;

    /**
     * ProgramService for use in each of the JUnit test cases
     * - will be autowired by Spring. docnetProgramServiceImpl
     * corresponds to a bean mapping in docnet-context.xml.
     */
    @Autowired
    protected ProgramService docnetProgramServiceImpl;

    /**
     * Initializes a new Program domain object for use in each test case.
     */
}
```

```

    */
    @Before
    public void onSetUp() {

        docnetProgramServiceImpl.deleteByScope(TEST_SCOPE);
        program = createProgram("Test Program",
                               "Better living with diabetics",
                               TEST_SCOPE);
    }

    /**
     * Deletes all objects from the database after each test case.
     */
    @After
    public void onTearDown() {
        docnetProgramServiceImpl.deleteByScope(TEST_SCOPE);
    }

    /**
     * Creates a new Program domain object based on the given field values.
     */
    protected Program createProgram(String name, String description, String scope)
    {

        Program newProgram = new Program();
        newProgram.setName(name);
        newProgram.setDescription(description);
        newProgram.setScope(scope);

        return newProgram;
    }
}

```

- To begin with we set up our tests for use with Spring, providing its configuration via the `@RunWith` and `@ContextConfiguration` annotations. For `@ContextConfiguration` we list the required system and internal Spring context files.

```

@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations = {
    "classpath:/META-INF/docnet-system-context.xml",
    "classpath:/META-INF/docnet-context.xml",
    "classpath:/META-INF/docnet-dependency-context.xml"})
public abstract class AbstractDocnetServiceTestCase {

```

- Next we declare two instance variables and one constant that will be used throughout our JUnit tests:

```

protected static final String TEST_SCOPE = "TestScope";

protected Program program;

@Autowired
protected ProgramService docnetProgramServiceImpl;

```

These variables are:

- `TEST_SCOPE` - By default all domain objects within the eHF automatically have a scope attribute. This attribute is not allowed to be null.

The `scope` is a mechanism by which it is possible to group objects. The objects have no relationship in terms of code, but rather ownership. For example a user could have numerous `Program` object's assigned to them. By giving them the same `scope` value we can build a logical record for this user. `Scope` is then normally used when defining permissions, such that you can give someone permissions on certain objects but only under a specific `scope`.

With our `TEST_SCOPE` constant we are defining a default `scope` for use in our tests. We are not concerned with grouping, ownership or permissions at the moment, we simply want to be able to create our objects.

- `program` - this is provided so that every test case will have a new fresh `Program` object with which to run.
- `docnetProgramServiceImpl` - As the `Program` is the main domain object that we use, and through which all our CRUD actions will take place, all our tests will require a `ProgramService` instance on which to call the appropriate methods. For this we define it as `docnetProgramServiceImpl`. This corresponds to a bean that is defined in the module's internal Spring context (`docnet-context.xml`). The use of the `@Autowired` annotation tells Spring to automatically populate this variable for us.
- Next we define the `onSetUp()` and `onTearDown()` methods to be run before and after each of our tests:

```
@Before
public void onSetUp() {

    docnetProgramServiceImpl.deleteByScope(TEST_SCOPE);
    program = createProgram("Test Program",
                           "Better living with diabetics",
                           TEST_SCOPE);
}

@After
public void onTearDown() {
    docnetProgramServiceImpl.deleteByScope(TEST_SCOPE);
}
```

As our class does not extend any specific JUnit class we use the `@Before` and `@After` annotations to tell JUnit that these methods should be called before and after each test case.

In both methods we clean up the database by deleting all objects with our `TEST_SCOPE` from the database (`docnetProgramServiceImpl.deleteByScope(TEST_SCOPE);`).

Additionally within `onSetUp` we instantiate our `program` instance variable so that all our test cases have a new `Program` domain object with which to work.

- The `createProgram()` method simply creates a new `Program` domain object for us based on the given name, description and `scope`:

```
protected Program createProgram(String name,
                                String description, String scope) {

    Program newProgram = new Program();
    newProgram.setName(name);
    newProgram.setDescription(description);
    newProgram.setScope(scope);

    return newProgram;
}
```

```
}

```

With the completion of our Abstract test class we are ready to write our concrete test classes. To begin with we will create two classes, one for the Program tests and one for the Appointment tests. We'll start with the Program tests.

The ProgramCrudServiceTest

Within the package `com.mycompany.docnet.service`, create the class `ProgramCrudServiceTest`:

```
package com.mycompany.docnet.service;

import static org.junit.Assert.*;
import junit.framework.JUnit4TestAdapter;

import org.junit.Test;

import com.icw.ehf.commons.exception.dao.ObjectNotFoundException;
import com.mycompany.docnet.domain.Program;

/**
 * Testing the basic functionality of the ProgramService.
 */
public class ProgramCrudServiceTest extends AbstractDocnetServiceTestCase {

    /**
     * Test creating, updating and deleting our first program in the database.
     */
    @Test
    public void testProgramCrudServices() {

        String programId = null;

        programId = testCreateProgramInDatabase();

        testUpdateProgramNameInDatabase(programId);

        testDeleteProgramFromDatabase(programId);

    }

    private String testCreateProgramInDatabase() {

        // confirm our initial program object has a null id -
        // only objects persisted in the database have an id
        String localProgramId = program.getId();
        assertNull(localProgramId);

        // EJB 3.0 does not reuse given java object but creates
        // a new one, hence we save this as createdProgram object
        Program createdProgram = docnetProgramServiceImpl.create(program);

        // confirm once persisted in the database our program
        // now has an id
        localProgramId = createdProgram.getId();
        assertNotNull(localProgramId);

        // return id for use in further tests
        return localProgramId;

    }

    private void testUpdateProgramNameInDatabase(String programId) {

        // first load program from database based on given id

```

```

        Program programToUpdate = docnetProgramServiceImpl.loadById(programId,
true);
        assertNotNull(programToUpdate);

        // change program name and update in database
        programToUpdate.setName("myname");
        docnetProgramServiceImpl.update(programToUpdate);

        // reload Program from the database based on given id
        programToUpdate = null;
        programToUpdate = docnetProgramServiceImpl.loadById(programId, true);

        assertEquals("myname", programToUpdate.getName());
    }

    private void testDeleteProgramFromDatabase(String programId) {

        // delete program from database based on given id
        docnetProgramServiceImpl.delete(programId);

        try {
            // attempt to load program from database - should fail
            Program deletedProgram = docnetProgramServiceImpl.loadById(programId,
true);
            fail("program should no longer exist in database");
        } catch (ObjectNotFoundException e) {
            // confirm failure was due to program no longer existing in database
            assertEquals(programId, e.getEntityId());
        }
    }

    /**
     * 'Retro-fitting' the test for compatibility with older test
     * runners, for example the junit task in pre 1.7 versions
     * of Ant.
     * @return adapter to older test runners
     */
    public static junit.framework.Test suite() {
        return new JUnit4TestAdapter(ProgramCrudServiceTest.class);
    }
}

```

Hopefully the code is fairly self explanatory. We have one public test case (testProgramCrudServices) which simply calls three private methods to test the Create, Update and Delete (as well as implicitly the Read) of our Program CRUD Services.

- We start by creating a Program in the database:

```

private String testCreateProgramInDatabase() {

    String localProgramId = program.getId();
    assertNull(localProgramId);

    Program createdProgram = docnetProgramServiceImpl.create(program);

    localProgramId = createdProgram.getId();
    assertNotNull(localProgramId);

    return localProgramId;
}

```

We assert that the program's ID is null before storing the program in the database. We then take the program object returned after being stored in the database and

assert that it now has an ID, as objects only become an ID after being stored in the database.

EJB3 Java Persistence API (JPA) does not reuse the given java object but instead creates a new one, hence we need to capture this as `createdProgram`.

We then finally return the ID of the newly created `Program` so that this (and therefore the newly created `Program`) can then be used in further tests if required.

- Next we attempt to update our newly created `Program` in the database.

```
private void testUpdateProgramNameInDatabase(String programId) {

    Program programToUpdate = docnetProgramServiceImpl.loadById(programId,
true);
    assertNotNull(programToUpdate);

    programToUpdate.setName("newName");
    docnetProgramServiceImpl.update(programToUpdate);

    programToUpdate = null;
    programToUpdate = docnetProgramServiceImpl.loadById(programId, true);

    assertEquals("newName", programToUpdate.getName());
}
```

We first load our `Program` from the database based on the given id, change its name and then persist this change in the database via the `update` method of the `Program Service` (`docnetProgramServiceImpl.update(programToUpdate)`). Finally we reload the `Program` from the database and make sure its name is now our `"newName"`.

- Lastly we attempt to delete our `Program` from the database.

```
private void testDeleteProgramFromDatabase(String programId) {

    docnetProgramServiceImpl.delete(programId);

    try {
        Program deletedProgram = docnetProgramServiceImpl.loadById(programId,
true);
        fail("program should no longer exist in database");
    } catch (ObjectNotFoundException e) {
        // confirm failure was due to program no longer existing in database
        assertEquals(programId, e.getEntityId());
    }
}
```

Once we have deleted the object based on the given id, we then try and reload it again making sure that this fails as it obviously should no longer be in the database.

If you remember from earlier, the static method `suite()` is also included to assure compatibility with our maven test plugin which relies on versions of JUnit earlier than 4.x. We need to include this method on all our concrete test classes.

The AppointmentServiceTest

Next create an `AppointmentServiceTest` class under the `com.mycompany.docnet.service` package. The complete listing is shown in the following. We'll then go through the individual parts in the sections that follow.

```

package com.mycompany.docnet.service;

import junit.framework.JUnit4TestAdapter;

import org.junit.Test;

import static org.junit.Assert.*;

import com.icw.ehf.core.domain.Date;
import com.mycompany.docnet.domain.Appointment;

public class AppointmentServiceTest extends AbstractDocnetServiceTestCase {

    private static final String DEFAULT_DATE = "2009-01-21T11:55+00:00";

    private static final String ZERO_TIMEZONE_OFFSET = "+00:00";

    private Appointment appointment;

    /**
     * creates a new appointment for use in each of our test cases.
     */
    public void setUp() {
        super.setUp();
        this.appointment = createAppointment(
            new Date(DEFAULT_DATE, ZERO_TIMEZONE_OFFSET),
            "1. Appointment",
            "my first appointment");
    }

    private Appointment createAppointment(
        Date date, String name, String description) {

        // create a new appointment.
        Appointment newAppointment = new Appointment();

        // no need to set scope as it will get this from program.
        newAppointment.setDate(date);
        newAppointment.setName(name);
        newAppointment.setDescription(description);

        return newAppointment;
    }

    /**
     * Add appointment to a program already persisted in database, using
     * the ProgramService's addAppointment method, then make
     * sure this appointment is correctly persisted.
     */
    @Test
    public void testAddAppointmentToExistingProgram() {

        program = docnetProgramServiceImpl.create(program);
        assertEquals(0, program.getAppointments().size());

        // add appointment using programService.
        docnetProgramServiceImpl.addAppointment(program, appointment);

        // reload program as it isn't updated in existing
        // Program instance.
        program = docnetProgramServiceImpl.loadById(program.getId(), true);
        assertEquals(1, program.getAppointments().size());
    }

    /**
     * Tests adding an Appointment to a non-persisted Program before trying
     * to persist both of them at the same time via the ProgramService.
     */
}

```

```

@Test
public void testAddAppointmentToNewProgram() {

    // add new appointment to program (before the program has
    // been persisted)
    program.addAppointment(appointment);

    program = docnetProgramServiceImpl.create(program);

    // make sure our program has really been stored in
    // the database (so has an ID)
    assertNotNull(program.getId());

    // make sure our persisted program also has an appointment
    assertEquals(1, program.getAppointments().size());

}

public static junit.framework.Test suite() {
    return new JUnit4TestAdapter(AppointmentServiceTest.class);
}
}

```

- To begin with we setup our Appointment tests:

```

private static final String DEFAULT_DATE = "2009-01-21T11:55+00:00";

private static final String ZERO_TIMEZONE_OFFSET = "+00:00";

private Appointment appointment;

public void onSetUp() {
    super.onSetUp();
    this.appointment = createAppointment(
        new Date(DEFAULT_DATE, ZERO_TIMEZONE_OFFSET),
        "1. Appointment",
        "my first appointment");
}

private Appointment createAppointment(
    Date date, String name, String description) {

    Appointment newAppointment = new Appointment();

    newAppointment.setDate(date);
    newAppointment.setName(name);
    newAppointment.setDescription(description);

    return newAppointment;
}

```

- **DEFAULT_DATE & ZERO_TIMEZONE_OFFSET** - these constants are simply used to create a valid Date object, for use when creating our Appointment object.
- **appointment** - we declare this instance variable so that every test case will have a new Appointment object with which to run.
- **onSetUp** - we override the onSetUp method from our AbstractDocnetServiceTestCase and additionally instantiate our appointment instance variable.
- **createAppointment** - simple creates a new Appointment object that we can use in our tests.

- Next we have our two tests which add an Appointment to a Program and store it in the database, `testAddAppointmentToExistingProgram` and `testAddAppointmentToNewProgram`.
- `testAddAppointmentToExistingProgram` is shown in the following listing

```
@Test
public void testAddAppointmentToExistingProgram() {

    program = docnetProgramServiceImpl.create(program);
    assertEquals(0, program.getAppointments().size());

    docnetProgramServiceImpl.addAppointment(program, appointment);

    program = docnetProgramServiceImpl.loadById(program.getId(), true);
    assertEquals(1, program.getAppointments().size());

}
```

In this test case we first store the Program in the database (`program = docnetProgramServiceImpl.create(program);`) then we create an Appointment object and add this new appointment to the program stored in the database via the ProgramService (`docnetProgramServiceImpl.addAppointment(program, appointment);`).

Next we reload the program, based on its ID. (`program = docnetProgramServiceImpl.loadById(program.getId(), true);`) We do this because the local program object we have a reference to is not updated when we add an appointment to it in the database. So we need to reload it first. Finally we assert that our program does indeed have an Appointment associated with it (`assertEquals(1, program.getAppointments().size());`).

The reason that the `docnetProgramServiceImpl.addAppointment()` method does not automatically update our program instance with the newly stored Appointment is for security reasons.

Security within the eHF can filter out associated Objects, based on permissions. For example we could give a program administrator access to all Programs, but we don't want them to be able to see the associated appointments as they could contain sensitive medical data that only a doctor is allowed to see. If this Program administrator loaded a program, the appointments would be filtered out in the program instance that they see. If they then modified this program and saved it to the database, a normal Hibernate persist would then delete the filtered Appointments as they are not currently listed with this particular program instance that the user is working with.

Therefore in our case we need to reload the program to be able to access the Appointment.

- `testAddAppointmentToNewProgram` is shown in the following:

```
@Test
public void testAddAppointmentToNewProgram() {

    program.addAppointment(appointment);

    program = docnetProgramServiceImpl.create(program);

    assertNotNull(program.getId());

    assertEquals(1, program.getAppointments().size());

}
```

```
}

```

In this test case we first add our new `Appointment` object to our new `Program` (`program.addAppointment(appointment);`) Then we persist the `program` in the database and at the same time return the newly persisted `program` (`program = createProgramInDatabase(program);`) Then we check to make sure that our `program` was really persisted by making sure it has an `id`. Finally we make sure that the persisted `program` has an `appointment`, to confirm that it too was persisted as part of the call to persist the original `program` object.

This test case demonstrates the cascading behavior when storing an object graph that has non-persisted objects. When we persisted the `program` object the non-persisted `appointments` in its object graph were also persisted at the same time. More details on this can be found in the eHF Generator section of the eHF Reference documentation.

- Lastly, as with the `ProgramCrudServiceTest` we also have a static `suite()` method for use by our maven test plugin in our `AppointmentServiceTest`. This method is only shown in the main listing as it is the same as used previously with only the class name different.

Before being able to successfully run the above JUnit tests you will need to make sure that you have a suitable HSQL database running, which already contains the `EHF_DOCNET` schema and the two tables: `T_PROGRAM` and `T_APPOINTMENT`. The simplest way to achieve this is to start the HSQL database before running `maven dev:build`, as this will then initialize the database for you.

Now, you can start the JUnit test. Right click on the test class in Eclipse's package explorer and choose **Run As -> JUnit Test**. The test starts and after finishing, (hopefully) the green bar signals the successful test run.

3.9 Model and Generate Summary

Phew, well we covered quite a bit in this chapter. We started by creating our basic model, and then letting the eHF Generator loose on it. This generated an architecture stack for us, with which we can work. We covered the basics of this generated architecture before taking a little detour into the world of Spring configuration. Finally we wrote a couple of simple tests to see if the whole lot worked. It did, and we were able to persist our created objects in the database.

Over the next chapters we'll expand our model and see what else the generator can do for us.

4 Creating an API

Having created a very simple domain model, let the eHF Generator work its magic and tested the code that it produced, we now need to consider our module's API.

If we look back at the General Architecture of eHF Modules, [Figure 21](#), we can see that the *Service Adapter Layer* is the layer that forms the API of our module.

In our previous tests we worked on the *Service Layer*, where our module's logic is implemented, and carried out a form of internal or "white box" testing on our `Program` and its associated CRUD Service. To test our module's API we need to work against the *Service Adapter Layer*. However we are unable to do this at the moment for the simple reason that we don't have one for our `Program`'s CRUD Service.

The lack of a *Service Adapter Layer* is because the eHF Generator never created one for us, and this is because we never told it to. We'll rectify that in this chapter and write some appropriate tests to make sure everything is in order.

To begin with we need to modify our model, so that the eHF Generator knows that it should create the appropriate *Service Adapter Layer* for us.

4.1 Service Adapter Layer Creation with the eHF Profile

If you remember from the General Architecture of eHF Modules, [General Module Architecture Stack](#) on page 27 the *Service Adapter Layer* has two sides to it:

- **Service Adapter** - the internal API. Clients can use this by adding the module's jar file to their classpath.
- **External Service Adapter** - the external API. This is accessed via web service calls.

The creation of either of these *Service Adapter* layer sides is controlled through the use of two attributes of the *ehf-domainobject* stereotype from the *eHF Profile* in our model. The two attributes in question are:

- **export** - regulates the visibility of domain object **services** on the adapter layer
- **expose** - regulates the visibility of modeled domain **objects** as transfer objects (either **Data Transfer Object (DTO)** for the Service Adapter and **External Transfer Object (XTO)** for the External Service Adapter)

Both of these *ehf-domainobject* stereotype attributes can have one of the following four values:

- **none** - no Service Adapter architecture is created (this is the default value)
- **internal** - only the internal Service Adapter architecture is created (e.g. when set for the *expose* tag, DTOs would be created).
- **external** - only the External Service Adapter architecture is created (e.g. for the *expose* tag, only XTOs would be created).
- **internal & external** - both internal and external Service Adapter architecture is created (e.g. for the *expose* tag, both DTO and XTOs are created).

For now we will only create the Service Adapter (so the internal API) for our `Program` CRUD Service. We will look at the External Service Adapter (so for use in Web Services) in section [Web Services](#) on page 80.

4.2 Modeling an Internal Service Adapter API

So to create the internal Service Adapter architecture for our `Program` CRUD Service we simply need to modify our model in the following way:

- For the `Program` class, set both the `export` and `expose` attributes of the `ehf-domainobject` stereotype to **internal**.
- For the `Appointment` class, set only the `expose` attribute to **internal**.

With the `Appointment` class we only need to set the `expose` attribute as `Appointment` does not have any services of its own that we need to `export`. However we do need to be able to work with appropriate `AppointmentDto` objects in our `Program` CRUD service.

1. From within Eclipse switch to the Topcased Modeling perspective, and then using the navigator, open the `model.uml` file from within the DocNet project under `src/main/model`.
2. In the DocNet class diagram, select the `Program` class to display its **Properties** window.
3. In the left hand menu, select **Stereotype Attributes**, and then in the right hand side change the **expose** and **export** attributes to **internal**.

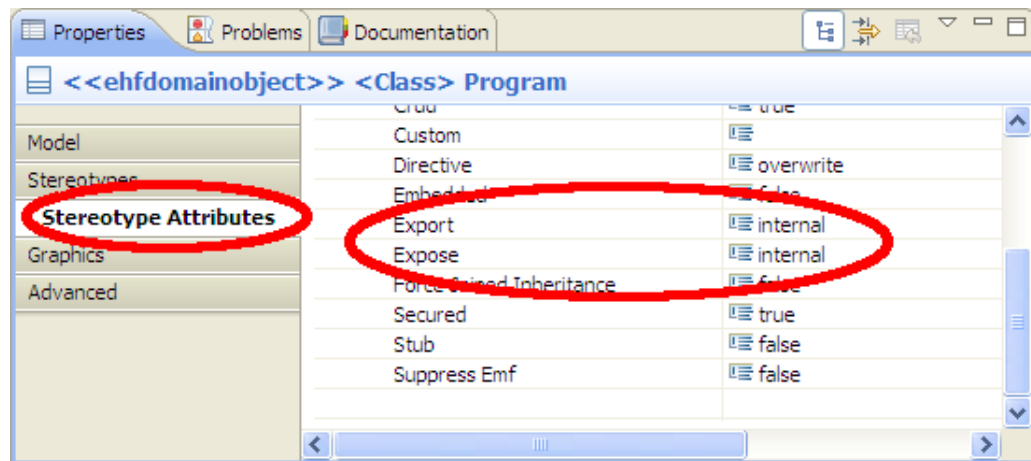


Figure 24: Setting Program's expose and export Attributes to internal

4. Similarly select the `Appointment` class in the class diagram to display its Properties window, and from there select the **Stereotype Attributes** and set **expose** to **internal**.

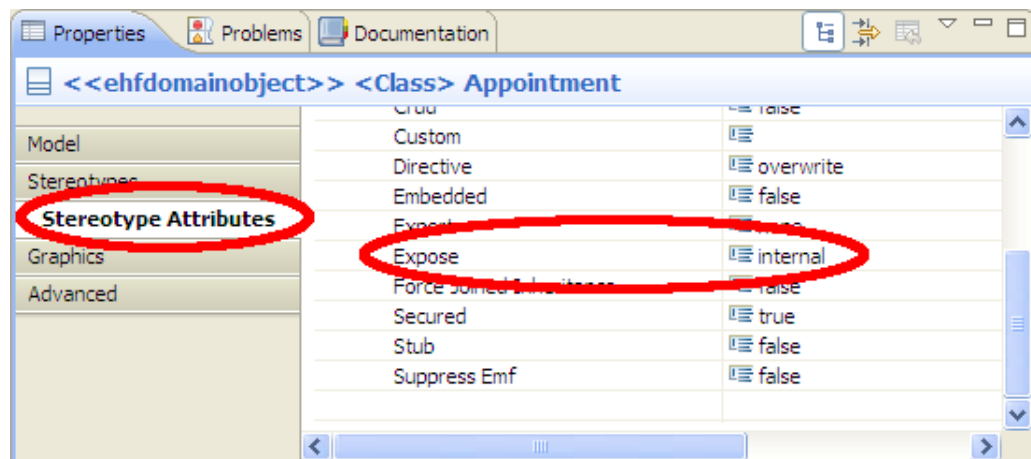


Figure 25: Setting Appointment's expose Attribute to internal

5. Remember to save your changes before building the project again (as described in section [Building DocNet](#) on page 25) using the command `maven dev:build`, and then refresh in Eclipse.

The eHF Generator has now created an appropriate *Service Adapter Layer* for our Program CRUD service for us. All that we need to do is go ahead and test it.

4.3 The Generated Service Adapter Layer

Before we test our new *Service Adapter Layer*, we'll take a quick look at what the generator has actually created for us.

If you remember from [The Object/CRUD Service Stack](#) on page 29, and specifically [Figure 22](#), well the generator has now completed this image for us and created the `CrudServiceDtoAdapter` for our *Program Service*.

So what does this mean in terms of the Java classes that are created for us?

Note: Who's in Control



Remember our key from before, when detailing who is in control of the generated Java classes:

- D - developer control (under `src/main/java`)
- G - generator control (under `src/main/gen`)
- M - movable (can be moved from `src/main/gen` to `src/main/java`)

Service Adapter Layer - Java Classes

The internal API for Object/CRUD services is defined by the `transfer.adapter.ProgramServiceDtoAdapter` [G] interface (implemented by `transfer.adapter.ProgramServiceDtoAdapterImpl` [G]). They extend the `transfer.adapter.AbstractProgramServiceDtoAdapter` [G] and the `transfer.adapter.AbstractProgramServiceDtoAdapterImpl` [G] respectively. As you can see, there is no possibility to extend this adapter as all files are under generator control. The only possibility to extend this interface is by adding custom methods to the UML model ().

Data Transfer Objects (DTO) and Assembler

The domain model of the Persistence Layer (`domain.Program` and `domain.Appointment`) is not the public model, instead there are DTOs (`transfer.AppointmentDto` [M] and `transfer.ProgramDto` [M]). An assembling / disassembling mechanism in the *Service Adapter Layer* (`assembler` package [G]) is responsible for transferring the data between the domain objects and the transfer objects.

4.4 Creating our First Service Adapter Layer JUnit Tests

Having modified our model, and had the generator create our API, and also looked at what exactly this meant in terms of Java classes, we can now go ahead and test this new layer.

Spring Context

As with the *Service Layer* JUnit tests, we will use Spring to start our test system context for us. However in our *Service Adapter Layer* tests we simply need to use one Spring context file instead of three as before. This is the file `docnet-test-context.xml` found under `src/test/resources/META-INF`.

If you take a look at this file you will see that it simply imports the **Module Context** and a **System Context**. As before the System Context is where we mimic the platform context

and provide the beans to the Module Context that it would normally expect to receive from the platform. The Module Context is then our main module context as described previously, [Spring Configuration](#) on page 34. The Module Context, only makes available to the platform those beans that are explicitly *exported* in this file. This means that we will only be able to use (and therefore test) these beans. You can think of this as a type of Black Box testing.

If you look in the file `docnet-module-context.xml` under `src/main/gen/META-INF` you will see that it exports the following bean:

```
<bean id="docnetProgramService" interface="com.mycompany.docnet.transfer.adapter.
ProgramServiceDtoAdapter" />
```

This is our new Service Adapter for our Program CRUD service, and is what we will use in our tests.

If you want to know more about how the module's test Spring context is built, complete details can be found in the appendix [A Module's Test Spring Context](#) on page 108.

The AbstractDocnetServiceAdapterTestCase

As with the Service JUnit tests, we will begin by first writing an abstract test class, which our concrete test classes can then extend.

From within Eclipse, navigate to the docnet project and under the folder `src/test/java` create the package `com.mycompany.docnet.service.adapter`.

Within this package we need to create the abstract class `AbstractDocnetServiceTestCase`. The complete listing is shown below.

```
package com.mycompany.docnet.service.adapter;

import java.util.List;

import org.junit.After;
import org.junit.Before;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;

import com.mycompany.docnet.transfer.ProgramDto;
import com.mycompany.docnet.transfer.adapter.ProgramServiceDtoAdapter;

@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations = {"classpath:/META-INF/docnet-test-context.xml"})
public abstract class AbstractDocnetServiceAdapterTestCase {

    /**
     * Default scope for use in each JUnit test case.
     */
    protected static final String TEST_SCOPE = "TestScope";

    /**
     * Initial ProgramDto object available for each JUnit test case.
     */
    protected ProgramDto programDto;

    /**
     * ProgramServiceDtoAdapter for use in each of the JUnit test
     * cases - will be autowired by Spring. docnetProgramService
     * corresponds to bean export in docnet-module-context.xml,
     * which corresponds to bean mapping in docnet-api-context.xml.
     */
    @Autowired
```

```

protected ProgramServiceDtoAdapter docnetProgramService;

/**
 * Initializes a new ProgramDto object for use in each
 * test case.
 */
@Before
public void setUp() {

    deleteAllObjects();
    programDto = createProgramDto("Test Program",
                                "Better living with diabetics",
                                TEST_SCOPE);
}

/**
 * Deletes all objects from the database after each test case.
 */
@After
public void tearDown() {
    deleteAllObjects();
}

/**
 * Creates a new ProgramDto object based on the given field values.
 * @param name of the ProgramDto object
 * @param description of the ProgramDto object
 * @param scope of the ProgramDto object
 *
 * @return the newly created ProgramDto object
 */
protected ProgramDto createProgramDto(String name, String description, String
scope) {

    ProgramDto newProgramDto = new ProgramDto();
    newProgramDto.setName(name);
    newProgramDto.setDescription(description);
    newProgramDto.setScope(scope);

    return newProgramDto;
}

/**
 * Deletes all objects with the defined TEST_SCOPE.
 * @see #TEST_SCOPE
 */
private void deleteAllObjects() {
    List<ProgramDto> objects =
        docnetProgramService.loadByScope(TEST_SCOPE, true);
    docnetProgramService.delete(objects);
}
}

```

You will notice that this class is almost identical to [The AbstractDocnetServiceTestCase](#) on page .

The main differences are as follows:

1. As we are working on the *Service Adapter Layer* instead of the *Service Layer* we need to use DTOs (*ProgramDto*) instead of domain objects (*Program*).
2. Also we are now using a *ProgramServiceDtoAdapter* instead of a *ProgramService*.
3. Lastly the *ProgramServiceDtoAdapter* does not provide a *deleteByScope()* method so we have created our own local method, *deleteAllObjects()* to take

care of this for us. We then simply call this instead in our `onSetUp` and `onTearDown` methods.

This method simply loads all `ProgramDtos`, based on our `TEST_SCOPE` and then calls the `delete()` method of the `ProgramServiceDtoAdapter` that takes a `List` of `ProgramDto` objects.

With the completion of our Abstract test class we can go ahead and write our concrete test classes.

The ProgramCrudServiceAdapterTest

Within the package `com.mycompany.docnet.service.adapter`, create the class `ProgramCrudServiceAdapterTest`:

```
package com.mycompany.docnet.service.adapter;

import static org.junit.Assert.*;

import junit.framework.JUnit4TestAdapter;
import org.junit.Test;

public class ProgramCrudServiceAdapterTest extends
    AbstractDocnetServiceAdapterTestCase {

    /**
     * Test creating our first program & storing it in the database.
     */
    @Test
    public void testCreateProgram() {
        assertNull(programDto.getId());
        programDto = docnetProgramService.create(programDto);
        // objects persisted in database automatically receive an id
        assertNotNull(programDto.getId());
    }

    public static junit.framework.Test suite() {
        return new JUnit4TestAdapter(ProgramCrudServiceAdapterTest.class);
    }
}
```

At the moment it is fairly simple - to the extreme. We assert that the `programDto`'s ID is null before passing it to the `ProgramServiceDtoAdapter`'s `create` method to store it in the database.

Through this method call the `ProgramDto` object is transformed into a `Program` domain object and passed to the `ProgramService` which then takes care of storing domain object in the database.

We then take the `programDto` object returned and assert that it now has an ID, as objects only receive an ID after being stored in the database.

Lastly if you remember from earlier, the static method `suite()` is simply included to assure compatibility with our maven test plugin which relies on versions of JUnit earlier than 4.x. We need to include this method on all our concrete test classes.

In this test we only make sure that the `ProgramServiceDtoAdapter` is configured through Spring, and that it is possible to store a `Program` in the database. As we didn't want to repeat ourselves too much, we have skipped the basic Read, Update and Delete methods for testing. However go ahead and create tests for these if you want. They will be similar to the corresponding tests in the `ProgramCrudServiceTest` that we wrote earlier, [The ProgramCrudServiceTest](#) on page .

The AppointmentServiceAdapterTest

Next create an AppointmentServiceAdapterTest class under the com.mycompany.docnet.service.adapter package. The complete listing is shown in the following:

```

package com.mycompany.docnet.service.adapter;

import static org.junit.Assert.*;

import org.junit.Test;

import junit.framework.JUnit4TestAdapter;

import com.icw.ehf.core.transfer.DateDto;
import com.mycompany.docnet.transfer.AppointmentDto;

public class AppointmentServiceAdapterTest extends
    AbstractDocnetServiceAdapterTestCase {
    private static final String DEFAULT_DATE =
        "2009-01-21T11:55+00:00";

    private static final String ZERO_TIMEZONE_OFFSET = "+00:00";

    private AppointmentDto appointmentDto;

    /**
     * creates a new appointment for use in each of our test cases.
     */
    public void setUp() {
        super.setUp();
        this.appointmentDto = createAppointmentDto(
            new DateDto(DEFAULT_DATE, ZERO_TIMEZONE_OFFSET),
            "1. Appointment",
            "my first appointment");
    }

    private AppointmentDto createAppointmentDto(
        DateDto date, String name, String description) {

        // create a new appointment.
        AppointmentDto newAppointmentDto = new AppointmentDto();

        // no need to set scope as it will get this from program.
        newAppointmentDto.setDate(date);
        newAppointmentDto.setName(name);
        newAppointmentDto.setDescription(description);

        return newAppointmentDto;
    }

    /**
     * Add appointment to a program already persisted in database,
     * using the docnetProgramService addAppointment method, then
     * make sure this appointment is correctly persisted.
     */
    @Test
    public void testAddAppointmentToExistingProgram() {

        programDto = docnetProgramService.create(programDto);
        assertEquals(0, programDto.getAppointments().size());

        // add appointment using ProgramServiceDtoAdapter.
        docnetProgramService.addAppointment(programDto.getId(),
            appointmentDto);
    }
}

```

```

        // reload program as it isn't updated in existing
        // Program instance.
        programDto = docnetProgramService
            .loadById(programDto.getId(), true);
        assertEquals(1, programDto.getAppointments().size());
    }

    /**
     * Tests adding an Appointment to a non-persisted Program
     * before trying to persist both of them at the same time
     * via the Program service.
     */
    @Test
    public void testAddAppointmentToNewProgram() {

        // add new appointmentDto to programDto (before the programDto has
        // been persisted)
        programDto.addAppointment(appointmentDto);

        programDto = docnetProgramService.create(programDto);

        // make sure our programDto has really been stored in
        // the database (so has an ID)
        assertNotNull(programDto.getId());

        // make sure our persisted programDto also has an appointment
        assertEquals(1, programDto.getAppointments().size());
    }

    public static junit.framework.Test suite() {
        return new JUnit4TestAdapter(AppointmentServiceAdapterTest.class);
    }
}

```

This code is almost exactly the same as that in [The AppointmentServiceTest](#) on page . The only difference is that we are working on the *Service Adapter Layer* so need to use DTOs (`AppointmentDto`) and are calling the methods on the `ProgramServiceDtoAdapter` instead of a `ProgramService`.

As before, you can start the JUnit tests. Right click on the appropriate test class in Eclipse's package explorer and choose **Run As -> JUnit Test**. The test starts and after finishing, (hopefully) the green bar signals the successful test run. Remember to make sure that an appropriate database is running in the background before executing the tests.

4.5 Creating an API Summary

This was a fairly short chapter. Through a couple of small changes in our model, the eHF Generator has expanded our `Program` CRUD Service stack to include the *Service Adapter Layer*, and specifically the internal API in the form of the `ProgramServiceDtoAdapter`. We then created a couple of tests to show that everything worked as expected.

Hopefully this has shown you that we can run the eHF Generator as often as we want, and that it is not a problem to incrementally expand our model.

In the following chapter we will take this a couple of steps further and see what else the generator can do for us.

5 Adding Constraints

At this point we now have a simple model, that defines our main CRUD Service and exports this to the *Service Adapter Layer*. We've let the eHF Generator generate our code based on this model, before testing that the basic code we received worked.

However we can define a lot more in our model than what we have done up until now. We might want to implement a custom method or domain service, or have some constraints on our object parameters – after all we don't want to allow users to add programs with a title the length of a book do we?

We'll take a look at adding constraints in this chapter, and see what the generator creates for us before testing the constraints.

5.1 Defining Constraints in the Model

First we are going to add the constraints to our model. We'll start with our `Program`, and add the following constraints to its attributes:

- **name** – must be unique (and therefore is automatically "not nullable"), and no longer than 30 characters in length.
- **description** – is not nullable.

Then we will set the following constraints on our `Appointment`:

- **name** – is not nullable, and must be no longer than 30 characters in length.
- **date** – is not nullable.

To begin with switch to the Topcased Modeling perspective in Eclipse and then, using the navigator, open the `model.uml` file from within the docnet project under `src/main/model`.

1. From within the DocNet class diagram, select the `name` attribute of the `Program` class to display its properties window.
2. In the left hand menu, select **Stereotypes**, and from the list displayed select `eHF Profile::ehf-attribute` and click **Add**. We can now use the attributes of this stereotype to define the required constraints for our `name` attribute of our `Program` class.

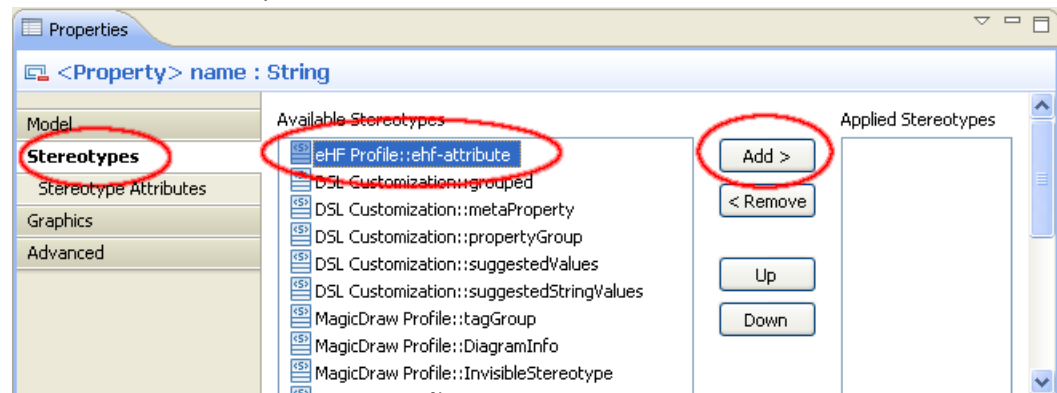


Figure 26: Setting the `ehf-attribute` Stereotype

3. Still within the Properties window of the `name` attribute of the `Program` class, from the left hand menu select **Stereotype Attributes**. A list of the available attributes for the `ehf-attribute` stereotype will be displayed. Now we need to change the two attributes *Unique* and *Max Length* to have the following values:

- Unique = true
- Max Length = 30

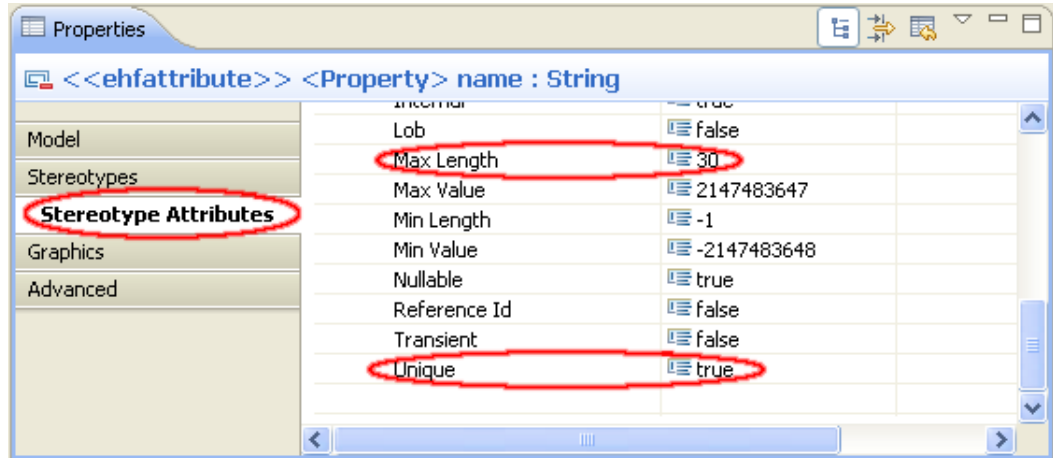


Figure 27: Setting the *Unique* and *Max Length* Attributes

Note: Setting a Class Attribute to be Unique



Some of you may have noticed that there is an *isUnique* checkbox available in the Properties window of a class attribute under the section Model. This is standard in the uml specification.

So you might ask the question why do we not use this, instead of the *Unique* attribute of the *ehf-attribute* stereotype when we want to specify that a class attribute must be unique? The simple reason is that at the moment the eHF Generator only supports the *ehf-attribute* stereotype option. This may change in the future.

4. Similarly in the class diagram select the *description* attribute of the *Program* class. Use the Properties window to set the *eHF Profile::ehf-attribute* stereotype for the *description*, via the **Stereotypes** section. Once the stereotype has been set move to the **Stereotype Attributes** section and change the *Nullable* attribute from its default of *true* to *false*.
5. In the same way add the required constraints to the *Appointment* class. Add the stereotype *eHF Profile::ehf-attribute* to the required attributes of the class, *name* and *date*, and then set the following Stereotype attributes for each:
 - *name*: *Nullable* = false, *Max Length* = 30
 - *date*: *Nullable* = false

Note: Setting the *ehf-attribute* Stereotype



You might have noticed that we have not set the *ehf-attribute* stereotype for the *description* attribute of the *Appointment* class.

By default for classes which have been marked with the *ehf-domainobject* stereotype, the eHF Generator treats all attributes of these classes as if they have been marked with the *ehf-attribute* stereotype.

We only need to explicitly set the *ehf-attribute* stereotype on an attribute when we want to set some of the stereotype's attributes to be different from their default values.

Make sure to save your changes before rebuilding the DocNet project ([Building DocNet](#) on page 25).

Class Metadata

For each class in our model the generator creates a corresponding metadata XML file, which includes the constraints for each attribute of the class. The files are generated into the same location as the classes and named appropriately.

So for our `Program` and `Appointment` classes the respective metadata files *Program.xml* and *Appointment.xml* are created. These can be found under `src/main/gen/` in the package `com.mycompany.docnet.domain`.

The following xml snippet is from the *Program.xml* metadata file, and shows the constraints for the `name` attribute:

```
<metadata>
...
  <attributes>
    ...
    <string>
      <attribute>name</attribute>
      <contractRelevant>false</contractRelevant>
      <modifiable>true</modifiable>
      <unique>true</unique>
      <persistent>true</persistent>
      <directive>overwrite</directive>
      <mandatory>true</mandatory>
      <maxLength>30</maxLength>
      <minLength>-1</minLength>
    </string>
    ...
  </attributes>
</metadata>
```

The metadata XML file details all the constraints for each attribute in the class, and is used for validation of instances during runtime. It's content is fairly self-explanatory, for example we can easily deduce from the above snippet that our `name` attribute is of type `String`, has a maximum length of 30 characters, is mandatory (i.e. not nullable), and must be unique - which thankfully matches with the constraints that we just defined in our model.

For more on Constraint validation and how it works see the eHF Generator section of the eHF Reference documentation.

That is all you have to do when adding constraints to your module. Simply add them to your model. In the *Persistence Layer* of our module's architecture ([General Module Architecture Stack](#) on page 27), there are validation aspects which automatically check these rules.

Next we'll expand our JUnit tests and see if these constraints really work or not.

5.2 Expanding the JUnit Tests to Include Constraints

Having added the constraints to our model and re-built the project we are ready to expand our `ProgramCrudServiceAdapterTest`, and the

AppointmentServiceAdapterTest, to see if these constraints really work. We will start by adding tests for the Program constraints and then finish with the Appointment constraints.

Note: Package Imports

In the following listings, we omitted the required `import` statements. Please add these imports on your own - you can use Eclipse's organize imports shortcut, **[Ctrl + Shift + O]** to help with this.



The full package names of the required imports in each test are given below.

For the `ProgramCrudServiceAdapterTest`:

- `com.icw.ehf.commons.exception.validation.ValidationException`
- `com.icw.ehf.commons.metadata.validator.ValidationErrorCodeConstants`
- `org.springframework.jdbc.UncategorizedSQLException`

For the `AppointmentServiceAdapterTest`:

- `com.icw.ehf.commons.exception.validation.ValidationException`

5.3 Expanding the `ProgramCrudServiceAdapterTest` for Constraints

First we will test the new constraints on the `name` and `description` attributes of the `Program` class.

For the `name` attribute we will add three test cases to our `ProgramCrudServiceAdapterTest`:

1. **`testProgramNameUniquenessConstraint`** - to test that it is not possible to store two `Program` objects in the database with the same name.

```
private static final String UNIQUE_NAME = "Unique Name";

@Test
public void testProgramNameUniquenessConstraint() {

    programDto = createProgramDto(UNIQUE_NAME, "Description", TEST_SCOPE);
    docnetProgramService.create(programDto);

    try {
        programDto = createProgramDto(UNIQUE_NAME, "Description",
TEST_SCOPE);
        docnetProgramService.create(programDto);
        fail("Only unique program names should be allowed");
    } catch (UncategorizedSQLException e) {
        assertNotNull(e);
    }
}
```

Currently the unique constraint of the `name` attribute can only be tested at the database/transaction level, not at the application level. For this reason the only exception we can catch is an `org.springframework.jdbc.UncategorizedSQLException` which is database independent.

2. **testProgramNameLengthConstraint** - to test that it is not possible to store a Program object in the database with a name that is longer than 30 characters in length.

```
private static final String THIRTY_ONE_CHARACTER_STRING = "exactly-31-
characters-in-length";

@Test
public void testProgramNameLengthConstraint() {

    programDto.setName(THIRTY_ONE_CHARACTER_STRING);
    try {
        docnetProgramService.create(programDto);
        fail("Program name cannot be longer than 30 characters");
    } catch (ValidationException e) {
        assertEquals(1, e.getExceptionDetails().size());
        assertEquals(ValidationErrorConstants.
ERROR_STRING_MAX_VALIDATION,
            e.getExceptionDetails().get(0).getCode());
    }
}
```

3. **testProgramNameNotNullConstraint** - to test that it is not possible to upload a Program with no name.

```
@Test
public void testProgramNameNotNullConstraint() {
    // check that the name is not allowed to be null
    programDto.setName(null);
    try {
        docnetProgramService.create(programDto);
        fail("A program must have a name");
    } catch (ValidationException e) {
        assertEquals(1, e.getExceptionDetails().size());
        assertEquals(ValidationErrorConstants.ERROR_MANDATORY_VALIDATION,

            e.getExceptionDetails().get(0).getCode());
    }
}
```

Next for the description attribute of the Program class we add the `testProgramDescriptionNotNullConstraint` test case, to simply make sure that it is not possible to create a Program without setting a description.

```
@Test
public void testProgramDescriptionNotNullConstraint() {

    programDto.setDescription(null);
    try {
        docnetProgramService.create(programDto);
        fail("A program must have a description");
    } catch (ValidationException e) {
        assertEquals(1, e.getExceptionDetails().size());
        assertEquals(ValidationErrorConstants.ERROR_MANDATORY_VALIDATION,
            e.getExceptionDetails().get(0).getCode());
    }
}
```

Finally in the `Program` class we test the `scope` attribute. As described earlier, each class marked as an *ehf-domainobject* in our model is automatically enhanced with the `scope` attribute. This `scope` is not allowed to be `null`. We will demonstrate this by adding the `testProgramScopeNotNullConstraint` test case.

```
@Test
public void testProgramScopeNotNullConstraint() {
    programDto.setScope(null);
    try {
        docnetProgramService.create(programDto);
        fail("A program must have a scope");
    } catch (ValidationException e) {
        assertEquals(1, e.getExceptionDetails().size());
        assertEquals(ValidationErrorConstants.ERROR_SCOPE_NULL,
            e.getExceptionDetails().get(0).getCode());
    }
}
```

5.3.1 ValidationException

In the previous test cases you will have hopefully noticed that we caught a `ValidationException` whenever we tried to create a `Program` that we knew would break the constraints that we had defined.

`ValidationException` is the input validation exception thrown by the eHF whenever a new object does not meet all the constraints set on the domain object.

Collecting all Error Messages

However upon receiving the first input validation error the eHF does not stop there. It carries on validating the complete object, and collects all the errors on the object together in one exception. This means the user can then be informed of all problems at the same time, instead of having to correct the first issue, only to run into the next problem during the next attempt to create the object.

For this reason the `ValidationException` can be thought of as a container for all validation error messages (`ValidationMessage`) reported on the object, holding a `List` of `ValidationMessage` objects. So in our JUnit tests we tested to make sure that the size of this `List` was what we expected in each case:

```
assertEquals(1, e.getExceptionDetails().size());
```

Two Validation Failures

To demonstrate this container behavior we can add the `testTwoConstraintFailures` test case to our `ProgramCrudServiceAdapter` test class:

```
@Test
public void testTwoConstraintFailures() {
    programDto.setName(null);
    programDto.setDescription(null);
    try {
        docnetProgramService.create(programDto);
        fail("A program must have both a name and a description");
    } catch (ValidationException e) {
        assertEquals(2, e.getExceptionDetails().size());
    }
}
```

```

        assertTrue(e.getMessage().contains("'Program.name' is mandatory but not
set. ")
                && e.getMessage().contains("'Program.description' is mandatory but
not set. "));
    }
}

```

Here we simply set both the name and description attributes to null and then try and create the Program. We are breaking two constraints and as such receive two ValidationMessages.

5.4 Expanding the AppointmentServiceAdapterTest for Constraints

As with the Program attributes, we now need to create tests for our Appointment attributes.

First we will test the name attribute by adding two test cases to our AppointmentServiceAdapterTest:

1. **testAppointmentNameNotNullConstraint** - to test that it is not possible to store an Appointment without a name:

```

@Test
public void testAppointmentNameNotNullConstraint() {
    programDto = docnetProgramService.create(programDto);

    appointmentDto.setName(null);
    try {
        docnetProgramService.addAppointment(programDto.getId(),
appointmentDto);
        fail("An Appointment must have a name");
    } catch (ValidationException e) {
        assertEquals(1, e.getExceptionDetails().size());
        assertEquals(ValidationErrorConstants.ERROR_MANDATORY_VALIDATION,

                e.getExceptionDetails().get(0).getCode());
    }
}

```

2. **testAppointmentNameLengthConstraint** - to test that the name of an Appointment is not allowed to be longer than 30 characters:

```

private static final String THIRTY_ONE_CHARACTER_STRING = "exactly-31-
characters-in-length";

@Test
public void testAppointmentNameLengthConstraint() {

    programDto = docnetProgramService.create(programDto);

    appointmentDto.setName(THIRTY_ONE_CHARACTER_STRING);
    try {
        docnetProgramService.addAppointment(programDto.getId(),
appointmentDto);
        fail("Appointment name must be 30 characters or less");
    } catch (ValidationException e) {
        assertEquals(1, e.getExceptionDetails().size());
        assertEquals(ValidationErrorConstants.
ERROR_STRING_MAX_VALIDATION,
                e.getExceptionDetails().get(0).getCode());
    }
}

```

```
}
}
```

Finally we will test the not null constraint of the `date` attribute of the `Appointment` class by adding the `testAppointmentDateNotNullConstraint` test case:

```
@Test
public void testAppointmentDateNotNullConstraint() {
    programDto = docnetProgramService.create(programDto);

    appointmentDto.setDate(null);
    try {
        docnetProgramService.addAppointment(programDto.getId(), appointmentDto);
        fail("Not allowed to insert an appointment without date");
    } catch (ValidationException e) {
        assertEquals(1, e.getExceptionDetails().size());
        assertEquals(ValidationErrorConstants.ERROR_MANDATORY_VALIDATION,
            e.getExceptionDetails().get(0).getCode());
    }
}
```

Note: What about the Appointment Scope?



For those of you who spotted it, in the previous tests, we never explicitly set the `scope` of our `Appointment` objects, yet we had previously stated that for all objects within our UML diagram marked as *ehf-domainobjects* the `scope` is automatically set to not nullable, and our `Appointment` class definitely has a `scope` attribute. So what is going on?

Due to the composition between a `Program` and an `Appointment`, an `Appointment` is automatically given the same `scope` as the `Program` to which it belongs. So we don't need to explicitly set it ourselves.

When running these new tests, as before, you need to make sure that you have a suitable database started and correctly configured. The simplest way to do this is to start the database before you build the module using `maven dev:build`.

5.5 Constraints Summary

In this chapter we were able to add some simple constraints to our model, and have then seen how these effect the validation, and creation of our objects.

It should be noted that the concept of constraints, and controlling suitable values of an attribute can be further advanced with the introduction of codesystems (a controlled vocabulary). This in turn can be used as part of an internationalization and localization strategy. This is provided for via the module eHF Codesystem. This is covered in further detail in the eHF Codesystem Tutorial.

Additionally it is possible to define more complex constraints on objects using OCL (Object Constraint Language). An example of such a constraint is shown here:

```
self.effectiveTime.beginDate.canonicDate < ${currentTime}
```

So for the object on which this constraint is defined the above basically means, that for an instance of this object (`self`), the `beginDate` of the `effectiveTime` field is not allowed to be before the current time in milliseconds.

Further details on all of this can be found in the eHF Reference Documentation.

6 Adding a Custom Method

Looking at the `ProgramServiceDtoAdapter`, we can see that we can currently only load a `Program` either by `id` or by `scope`. The easiest way to see this via Eclipse, is to open one of the test classes, and then within one of the methods start a new line and enter the text “`docnetProgramService.lo`” (without the quotes, but including the dot) and then press **[Ctrl+Space]** to get a list of method proposals. You should then see something similar to the following:

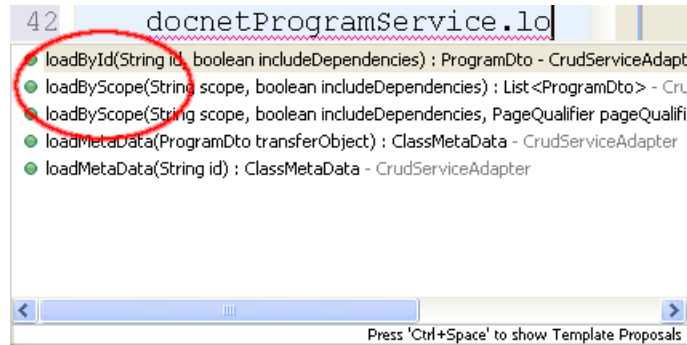


Figure 28: ProgramServiceDtoAdapter Method Proposals from Eclipse

Now while an `id` or `scope` is simple for a computer to deal with, it is not so obvious for us humans. As we have set the `name` attribute of our `Program` class to be unique it would be helpful if we could use this to load our `Program` objects – i.e. `loadByName()`. This is exactly what we will do next.

We'll add this custom method to our model, rebuild the project and take a look at what the generator creates for us, and see how much code we then have to add ourselves to implement our logic – just how much will the generator be able to read our minds and implement for us? We'll then write any custom code required and finally we'll extend our JUnit tests again to test this method.

6.1 Defining a Custom Method in the Model

To begin with switch to the Topcased Modeling perspective in Eclipse and then, using the navigator, open the `model.uml.di` file from within the `docnet` project under `src/main/model`.

1. From the main modeling window click the **Operation** button and then click once on the `Program` class in the diagram to add a method to the class – and give it the name `loadByName`. The Operation button is shown below:

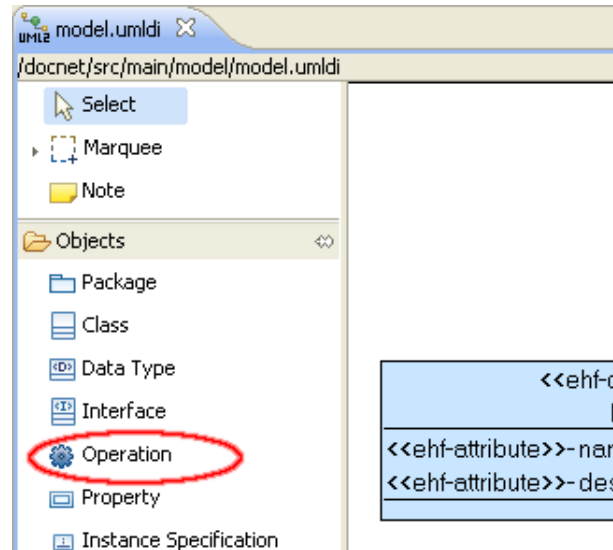


Figure 29: Creating a New Method in Topcased

2. Make sure that the new operation `loadByName` is selected in the diagram to display its **Properties** window. With the **Model** section of the Properties window selected:
 - check the **Return Type** check box
 - click the **Type Selection** button which will now be activated and from the list of types choose `<<ehfdomainobject>> <Class> Program`:

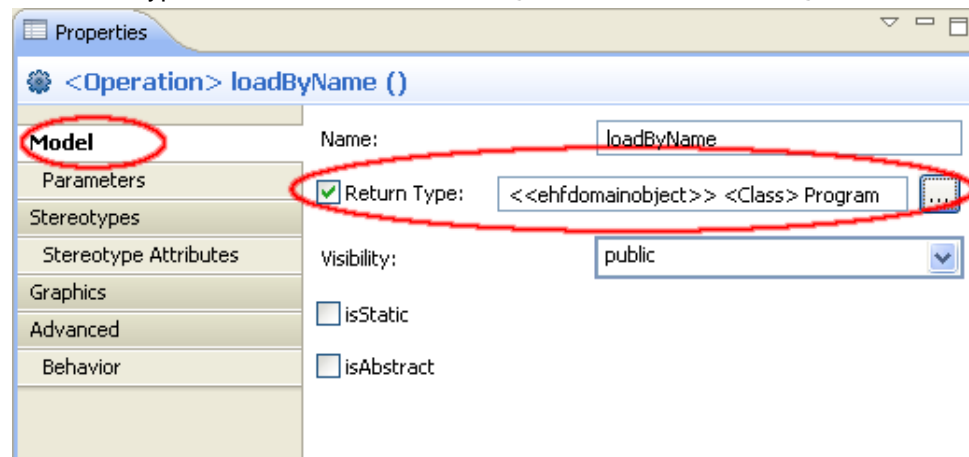


Figure 30: Setting the Return Type of the `loadByName` Method

3. Next select the **Parameters** section of the **Properties** window. Here you should see the return parameter already listed. We now need to add a new input parameter which will be the `String` holding the name of the `Program` object that we want to load. Click the **Add** button to create a new parameter and enter the following details:
 - *Name* - name
 - *Type* - `<Class> String`
 - *Visibility* - public (should be default)
 - *Direction* - in (should be default)
 - *Effect* - create (should be default)

The properties window should currently look as follows:

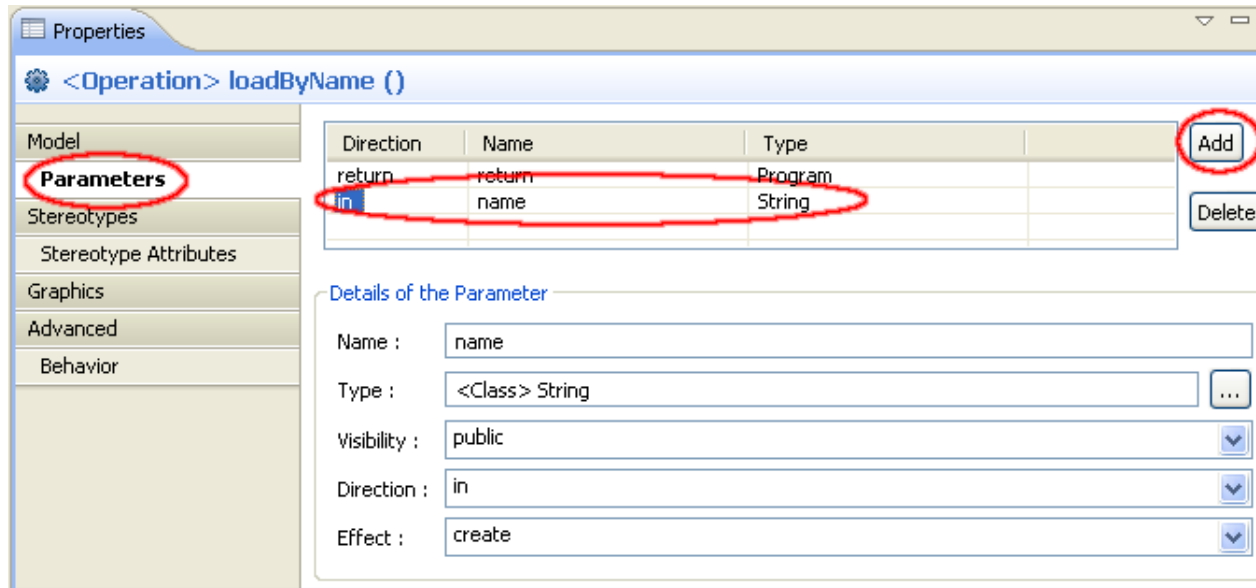


Figure 31: Properties Window for the loadByName Method's Parameters

- Next we need to consider where in our API we want to expose this new method.

By default the generator will not implement our method on the *Service Adapter Layer* of our architecture, so it won't be available in our API, either internal (Service Adapter) or external (External Service Adapter). As with [Creating an API](#) on page 48 we need to explicitly tell the generator what to do.

This is achieved through the use of the *expose* attribute of the *ehf-operation* stereotype from the *eHF Profile* in our model. This attribute controls the visibility of the modeled operation.

Still within the **Properties** window of the `loadByName` method, in the left hand menu, select **Stereotypes**, and from the list displayed select *eHF Profile::ehf-operation* and click **Add**. We can now use the attributes of this stereotype to *expose* the method in the *Service Adapter Layer* of our module.

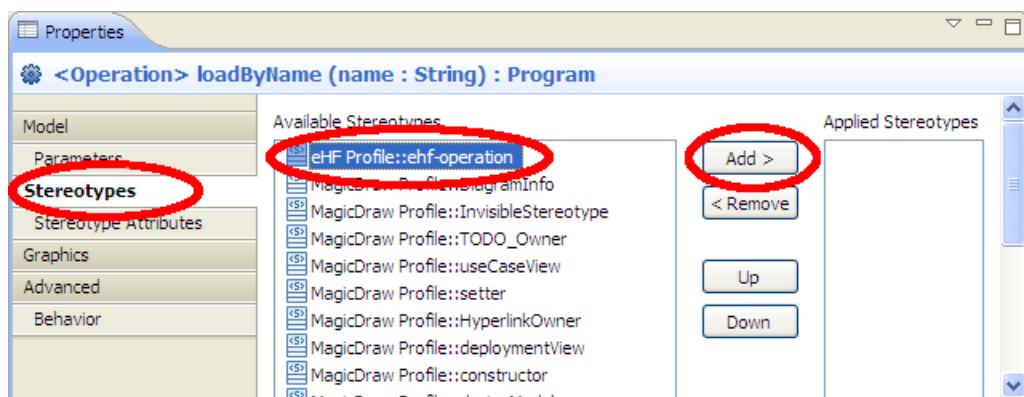


Figure 32: Setting the *ehf-operation* Stereotype

Then from the **Properties** window of the `loadByName` operation, from the left hand menu select **Stereotype Attributes**. A list of the available attributes for the *ehf-operation* stereotype will be displayed. Change the **expose** attribute to be *internal*.

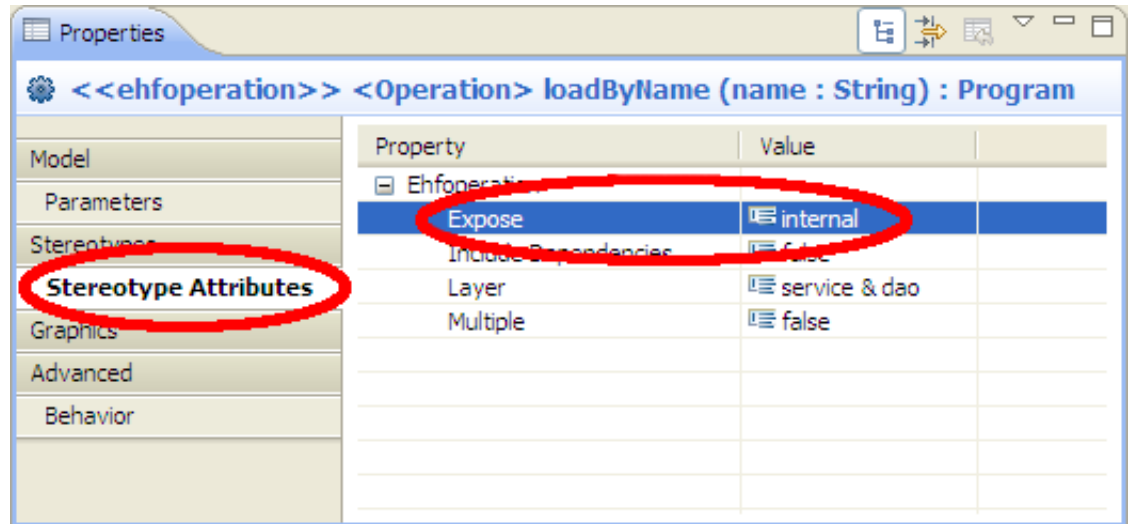


Figure 33: Setting the *ehf-operation* Stereotype

This tells the generator that our method should be included on the Service Adapter.

5. Finally our `Program` class diagram now looks as follows:

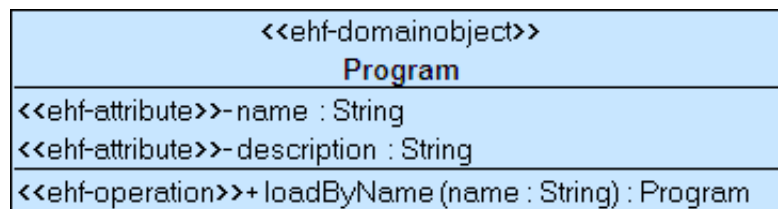


Figure 34: Program Class Diagram Including loadByName() Method

As ever make sure to save your changes before rebuilding the DocNet project ([Building DocNet](#) on page 25).

Before we write our custom method code, we'll take a look at what the generator has already put in place for us.

6.2 Custom Methods and the Generator

When we define a custom method in our Model, just what does the eHF Generator do for us.

Perhaps, you would have expected that the new method was added to the Domain Object `Program` (as you modeled the method in the UML model on the `Program` class). However we can see that our domain object (`com.mycompany.docnet.domain.Program` and `com.mycompany.docnet.domain.ProgramBase`, under `src/main/gen`), has not been modified. This is because by default all methods of a class, which has been marked with the *ehf-domainobject* stereotype, are treated as if they have been marked with the *ehf-operation* stereotype. This means that the eHF Generator knows not to create the method on the domain object, but rather to create it in the object's CRUD service stack (so in our `Program` CRUD service stack). We didn't need to explicitly set the *ehf-operation* stereotype on our method as we didn't need to change any of the default values for the stereotype's attributes.

Looking through `src/main/gen` we can see that the generator has indeed included our method in the `Program`'s CRUD service stack (see [Figure 22](#) for details):

1. In the *Service Adapter Layer*, the `com.mycompany.docnet.transfer.adapter.AbstractProgramServiceDtoAdapter(Impl)` contains the new method `loadByName`, which delegates to the *Service Layer* (see below).

Our method is included in the *Service Adapter Layer* after we set the `expose` attribute of the `ehf-operation` stereotype to the value `internal` in our model.

2. In the *Service Layer*, the `com.mycompany.docnet.service.AbstractProgramService(Impl)` contains the new method `loadByName` which in turn delegates to the *Persistence Layer* (see below). Furthermore, the `com.mycompany.docnet.security.ProgramServiceSecurity` class was extended by the `beforeProceed` and `afterProceed` methods. These methods are for adding security aspects for authorization – this is covered in the security tutorial.
3. In the *Persistence Layer*, the `com.mycompany.docnet.dao.ProgramDao` and `com.mycompany.docnet.dao.hibernate.HibernateProgramDao` contain the new method.

So the new modeled method results in changes on all levels of our CRUD service architecture stack. However, only the `HibernateProgramDao` and `ProgramServiceSecurity` classes are under the control of the developer. The first class is the place to implement the persistence logic of the method. The second one is for implementing security aspects. Both classes are placed in the `src/main/java` folder. The other named classes remain under the control of the generator and are located in the `src/main/gen` folder.

In the `HibernateProgramDao` class we can see the default implementation of the `loadByName` method as provided by the generator:

```
public Program loadByName(String name) {
    // TODO implement your logic here
    throw new UnsupportedOperationException();
}
```

The generator has handily added a TODO note to remind us that we need to actually write the method implementation ourselves. To stop anyone from trying use the method until then, the one line of code it has added for us is to throw an `UnsupportedOperationException`.

6.3 Implementing our Custom Method Logic

So the generator has taken care of most of the work for us and we simply need to add our method implementation to the `HibernateProgramDao`.

```
public Program loadByName(String name) {
    // Cast is okay as Hibernate will only return
    // Programs from the Program table
    @SuppressWarnings("unchecked")
    List<Program> result = (List<Program>) getHibernateTemplate()
        .find("from EHF_DOCNET.T_Program where name=?", name);
    if (result == null || result.size() == 0) {
        return null;
    }
}
```

```

    // name is unique, so we have max. one result
    return result.get(0);
}

```

This simply passes the HQL (Hibernate Query Language) script to Hibernate to retrieve the required `Program`. We then check the results to make sure that we actually have something returned; if not then we return `null`. However if we have a result, we simply return the first result (we should only ever get one as the name attribute of `Program` is unique, otherwise we would return a `List` of `Programs`).

As you have probably already noticed you will need to add an import statement for `java.util.List` to the `HibernateProgramDao` class.

Seven lines of code was all we needed to implement ourselves. The generator took care of everything else. With that we are now ready to test our new method.

6.4 Testing the loadByName Custom Method

Having implemented the `loadByName` method we can now add the `testProgramLoadByNameMethod` test case to our `ProgramCrudServiceTest`:

```

@Test
public void testProgramLoadByNameMethod() {

    programDto.setName(UNIQUE_NAME);
    docnetProgramService.create(programDto);

    ProgramDto programLoadedByName = null;
    programLoadedByName = docnetProgramService.loadByName(UNIQUE_NAME);

    assertEquals(UNIQUE_NAME, programLoadedByName.getName());
}

```

In this test we explicitly set the name of our `Program` before storing it in the database. We then create a new `ProgramDto` and set it using our new `loadByName` method, before asserting that we really did successfully load the `Program` from the database.

Our call to the service adapter layer (`docnetProgramService`) is passed through the *Service Layer* down to the *Persistence Layer* where our newly implemented Hibernate query is executed. The result of this query is passed back through the service stack and returned.

6.5 Returning More than One Program

In our custom method we only return a single `Program` object, however it is possible to have the method return a `Set` of `Program` objects (for example if the name attribute didn't have the unique constraint). This is very simply done by setting the *multiple* attribute of the *ehf-operation* stereotype on the method to `true` in the model, and then re-generating the code.

It could be that the build will fail the first time as you will need to modify your implementation in the `HibernateProgramDao` and your test classes to work with the new return type of a `Set<Program>` instead of just `Program`. However the generation of the main java classes will be successful. You can then refresh your project in Eclipse and use Eclipse's features

to help modify these classes, and then build the project again once everything has been updated.

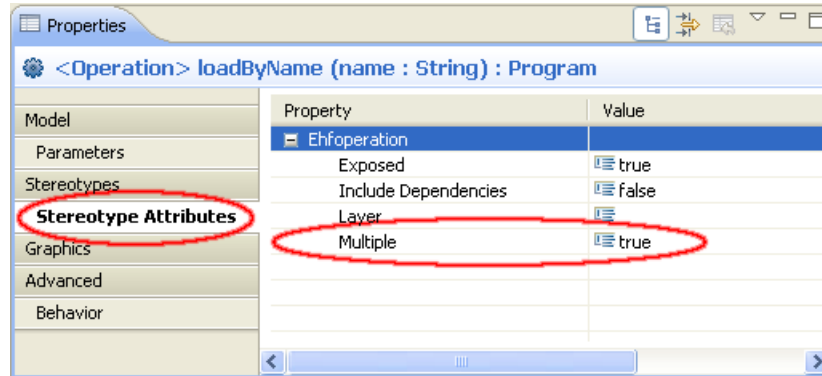


Figure 35: Setting *ehf-opertaion's* *Multiple* Attribute

As ever more information on the *multiple* attribute and all other available attributes can be found in the eHF Generator section of the eHF Reference Guide.

6.6 Custom Method Summary

Hopefully this chapter and [Adding Constraints](#) on page 56 has given you a taste for what is possible with the eHF Generator.

In this chapter we were able to define a custom method that the generator then automatically made available to us on all levels of the architecture stack. We only had to implement our business logic in one place. In fact our custom logic was only seven lines of code long.

However there is much more that the eHF Generator can achieve for us, from controlling the behavior of Hibernate (Cascade, Fetch) to creating links or index creation for specific attributes.

You can explore these options and more via the eHF Reference Documentation.

7 DocNet Assembly

Having successfully setup the basic DocNet domain model and implemented our first custom service and JUnit tests, we are going to leave the DocNet module to one side for the moment and create a DocNet Assembly for our project.

7.1 So What is an Assembly?

Firstly what is an Assembly, and why do we need one? As far as, what is an "assembly", it is basically a group of files, directories, and dependencies that are assembled into an archive format and distributed. We need one for our DocNet application, as we will use it to bring together our DocNet module along with all the various modules from the eHF that we will be using to create our whole application.

It means we can keep our dependencies to other eHF modules to a minimum in our DocNet module. Allowing it to concentrate on its own behavior and nothing else, and only including the dependencies on the other eHF modules in the assembly.

So basically it is how we are going to bring everything together.

7.2 Project Artifacts

Before we create our assembly, we are going to take a slight detour and cover project artifacts, or more precisely the various artifacts that can be created for us during a `maven dev:build`.

By default when we execute `maven dev:build` from within our DocNet project, not only does it download all the required dependencies, generate our code and so on, but it also creates a `docnet-<version>.jar` artifact for us within the `target` folder of our project (`<<version>>` is replaced with the value of the `<currentVersion>` tag in the `docnet project.xml`). So in our case it will create `docnet-SNAPSHOT.jar`, as that is the version set by default for new projects created via the module templates.

This `docnet-<version>.jar` contains the entire DocNet module. However we can have Maven create multiple artifacts for us during the build process, with each artifact containing specific parts of the DocNet module. The extra artifacts we can have Maven create for us are:

1. **docnet-api.jar** - contains all the java classes which represents the public interface. These, and only these, classes should be enough for the compilation of another module that is dependent on this module.
2. **docnet-bootstrap.jar** - includes all data files which are provided by a module. The functionality for reading, importing and handling these files is contained in the `module-runtime.jar`.
3. **docnet-config.jar** - contains all configuration files necessary for the usage of the module. Basically this contains all the Spring configuration xml files.
4. **docnet-documentation.jar** - The documentation artifact contains the module's documentation that we may have created (release notes etc).
5. **docnet-runtime.jar** - The runtime artifact contains all classes which are needed to use the full functionality of a module. The classes of the `module-api.jar` are included in this file.

The above is achieved by simply adding the respective lines to the `docnet project.properties` file:

```
maven.ehf.multiartifacts.api=true  
maven.ehf.multiartifacts.bootstrap=true  
maven.ehf.multiartifacts.documentation=true  
maven.ehf.multiartifacts.config=true  
maven.ehf.multiartifacts.runtime=true
```



Note: Setting the Runtime Property

If the `maven.ehf.multiartifacts.runtime` property is set to `true` then by default an **api** artifact will automatically be created as well.

For the DocNet module, we are going to have Maven create *runtime* (and therefore *api*) and *config* artifacts for us, by adding the following two lines to the `project.properties` file (these should already be set, but it is worth double checking):

```
maven.ehf.multiartifacts.runtime=true  
maven.ehf.multiartifacts.config=true
```

Now when we execute `maven dev:build` on the DocNet module, not only will we get a `docnet-<version>.jar` artifact created for us in the *target* folder, but we will also get `docnet-config-<version>.jar` and `docnet-runtime-<version>.jar` artifacts as well.

The advantage of this distribution is that the interface, implementation, configuration and bootstrap aspects are separated.

Note: Public API & Compile Time

The eHF Build Plugin for Maven only allows the use of *api* artifacts at compile time of a module (it removes any `runtime.jar`s from the classpath and then re-adds them once the module has been compiled - so that they are available for any test). This ensures that the module being built only has dependencies on classes that are defined in the public interface (so in the *api* artifacts) of the eHF modules.



For all eHF modules (and therefore our DocNet module) there are a default set of classes that are included in the *api* artifact of the module (i.e. one example is the `ProgramDto`). This is defined within the eHF Build Plugin. However this can be overridden, or extended, either by defining the Maven property `maven.ehf.multiartifacts.api.include.pattern` within the `project.properties` file of your module or by using the `@Public` annotation on your classes or packages (if you want to include an entire package).

The recommended form is to use the `@Public` annotation. More information can be found on this in the eHF Reference Guide.

7.3 Creating a DocNet Assembly

As we did when setting up our DocNet module for the first time in [Create the Project](#) on page 5, we will again use the *maven genapp plugin* in conjunction with a custom eHF assembly project template. As with the custom module project, the custom assembly project template is stored in the *ehf-genapp-maven-plugin*.

1. Open a console window, and navigate to your Eclipse <<workspace>> folder. (default location is C:\ICW_IDE\workspace)
2. Run the `maven genapp` plug-in in conjunction with the *ehf-assembly-template* by entering the following command:

```
maven genapp -Dmaven.genapp.template="ehf-assembly-template"
```

3. Answer the seven questions, as shown in the following output listing.



Note: The project's root directory does not need to exist beforehand.

If the project's root directory does not already exist, then the genapp plug-in will simply create it for you.

```

|_ \V/ |__ _Apache_ ___
| |\V| / _` \V / -_) ' \ ~ intelligent projects ~
|_| |_\_/_|\_\/\___|_|_| v. 1.1

build:start:

genapp:
Please specify the project root directory: [C:\ICW_IDE\workspace]
docnet-assembly
Please specify an id for your application: [ehf-product]
docnet-assembly
Please specify a groupId for your application: [ehf]
docnet-basic
Please specify a name for your application: [eHF Product]
DocNet Assembly
Please specify the package for your application: [com.icw.ehf]
com.mycompany.docnet
Please specify a module name for your application: [product]
docnet-assembly
Please specify the version of the eHF you would like to use: [SNAPSHOT]
X.X.X

```

These are basically the same questions we had to answer when we created the docnet module project, with the omission of the database schema question - there is no need for this in an assembly, as each module will define its own schema.

For the last question remember to make sure to replace X.X.X with the version of the eHF that you are currently working with.

7.3.1 Maven Eclipse:Eclipse

As with our DocNet module project we need to setup our new assembly project for use with Eclipse. We will again use the Maven Eclipse Plug-in to create the appropriate `.classpath` and `.project` files:

1. Open a console window, and navigate to the `<<workspace>>/docnet-assembly` folder
2. Run the eclipse plug-in by entering the following command:

```
maven eclipse:eclipse
```

3. This will create the `.classpath` and `.project` files for our assembly project, for use with Eclipse.

7.3.2 Import into Eclipse

As with the DocNet module, in [Import into Eclipse](#) on page 7, we now need to import the new DocNet Assembly project into Eclipse:

1. From within Eclipse select the menu **File -> Import...**
2. In the next window of the wizard, for the **root directory** use the **Browse** button to select the `<<workspace>>` folder. Docnet Assembly should then be listed in the list of available projects. Make sure it is selected and then click **Finish**.

The project "docnet-assembly" will now be listed in the package explorer of Eclipse.

3. Enter the project name `docnet-assembly` and then click **Finish**.

Having imported the docnet-assembly project into Eclipse you should see a similar folder structure to that, which was created for the docnet module, as detailed in [Project Directory Structure](#) on page 8. However there is one additional folder created for us when we generate an assembly project, namely: `src/webapp`, which is the web root directory. It in turn contains two folders:

1. `src/webapp/META-INF` - contains the web context configuration file for the web server
2. `src/webapp/WEB-INF` - contains standard configuration files

7.4 Incorporate DocNet Module into DocNet Assembly

While we now have a functioning assembly, our DocNet module is not actually included in the assembly just yet. We do this by adding a dependency to our DocNet module in the assembly:

1. From within Eclipse open the `project.xml` file of docnet-assembly project.
2. Add the following two dependencies to the file, within the `<dependencies>` tags (they can be placed anywhere within the tags, but for ease of maintainability, probably best placed towards the top):

```
<dependencies>
...
  <dependency>
    <groupId>docnet-basic</groupId>
    <artifactId>docnet-runtime</artifactId>
```

```

    <version>SNAPSHOT</version>
    <properties>
      <war.bundle>true</war.bundle>
    </properties>
    <type>jar</type>
  </dependency>

  <dependency>
    <groupId>docnet-basic</groupId>
    <artifactId>docnet-config</artifactId>
    <version>SNAPSHOT</version>
    <properties>
      <ehf-module>docnet</ehf-module>
      <ehf-persistence>true</ehf-persistence>
      <ehf-webapi>true</ehf-webapi>
    </properties>
    <type>jar</type>
  </dependency>
  ...
</dependencies>

```



Note: maven eclipse:eclipse

Having added the DocNet module dependencies to your assembly, you will need to execute `maven eclipse:eclipse`, as described in [Maven Eclipse:Eclipse](#) on page 74, so that Eclipse can find the appropriate classes.

3. Next we need to make sure that the Spring configuration for the DocNet module is included in the assembly.

We need to define the Spring configuration within the Module itself and then have the generator pull these out and include them automatically in the assembly for us, when we build the assembly.

Within the **DocNet module** we can create a `module.context.import.fragment` file within the `src/main/config/merge/assembly` folder. This is the location for all configuration files that should later be imported into an assembly. This `.fragment` file will contain the Spring configuration details for the DocNet Module.

Within `src/main/config` in the assembly, there is an `ehf-system-dependency-context.xml` file, which simply contains the following:

```

<beans>
  <import resource="classpath:/META-INF/ehf-commons-security-trusted-
certificates-context.xml"/>
  @@@module.context.import.fragment@@@
</beans>

```

The `@@@module.context.import.fragment@@@`, is a placeholder that will be replaced during the assembly build with the contents of the corresponding file from each module, where the file exists. (See the next section for more details on placeholders and parameterized configuration files) The resultant file that the build process creates for us can be seen under `src/main/gen/META-INF/ehf-system-dependency-context.xml`.

You will find that the `module.context.import.fragment` file has already been created for us within the `src/main/config/merge/assembly` folder of our DocNet module. If you open this file you will see that it contains one line (which simply means that our assembly context will import DocNet's module context):

```
<import resource="classpath:/META-INF/docnet-module-context.xml"/>
```

4. So, now that we have included the required dependencies to our DocNet module in our DocNet assembly, we can rebuild the assembly - open a command window, navigate to the DocNet assembly folder and execute `maven dev:build`.
5. While we haven't included any extra functionality just yet, the build should be successful and will produce a `docnet-assembly.war` file in the target folder of the DocNet Assembly project folder - refresh the project folder in Eclipse (select the folder and hit [F5]), to confirm the creation of the `.war` file.

7.4.1 Parameterized Configuration Files

There are different types of build process - e.g. local (for development) and master (for production) build. For this reason we occasionally need to duplicate some of the configuration files for these two cases.

To avoid having to maintain the same information in two places, configuration files are parameterized with placeholders, which will later be replaced with concrete values during the build process. There are two types of placeholders:

1. `@@@xxx.fragment@@@` - placeholder will be replaced with the complete content of the corresponding `xxx.fragment` files.
2. `@@xxx@@` - placeholder will be replaced with the specific `xxx` property value.

The actual values for the placeholders are defined in properties files or in so-called "fragment" files (they have a `.fragment` file extension). The fragment files contain composite definitions (more than one value); while in the standard property files (i.e. `configuration.properties` in the top level folder of a module/assembly), we only define a single value for a placeholder.

7.4.2 First DocNet Assembly Tests

Before we leave the DocNet assembly and return to expanding our DocNet module further, we will first write our initial JUnit tests for the assembly. These will be fairly basic for the moment, but will be expanded in advanced tutorials where we incorporate other eHF modules into our DocNet application, as there will be certain features that we can only test at the assembly level.

In our test we are simply going to include two attributes, `docnetProgramService` (of type `ProgramServiceDtoAdapter`) and `documentModuleService` (of type `DocumentModuleServiceDtoAdapter`) and make sure that they are correctly initialized by Spring and are not `null`.

While adding the dependencies for our DocNet module to the assembly in the `project.xml` file, you should hopefully have seen that there a lot of dependencies to other eHF modules. These are pre-configured for us. To confirm this we will add the second attribute `documentModuleService` from the eHF Document module to our test.

1. As before we will start by writing an abstract test class, `AbstractDocnetAssemblyTestCase`, in the package `com.mycompany.docnet`, under the folder `/src/test/java`.
2. As with our module test we will use the Spring TestContext Framework, and the custom runner class for Junit 4.4 that it provides, to help us with our tests in our assembly. We will define the `@RunWith` annotation to use the runner class and the `@ContextConfiguration` to define the spring context location. Unlike the module

tests we are no longer, using a test context rather we will use the real Spring context. In the assembly project this can be found under `src/main/gen/META-INF/ehf-system-context.xml`. The finished class looks as follows:

```
package com.mycompany.docnet;

import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;

import com.icw.ehf.document.service.adapter.DocumentModuleServiceDtoAdapter;
import com.mycompany.docnet.transfer.adapter.ProgramServiceDtoAdapter;

@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations={"classpath:/META-INF/ehf-system-context.xml"})
public abstract class AbstractDocnetAssemblyTestCase {

    @Autowired
    protected ProgramServiceDtoAdapter docnetProgramService;

    @Autowired
    protected DocumentModuleServiceDtoAdapter documentModuleService;

}
```

3. With our abstract test class in place we can now write our `DocnetIntegrationTest` class. In this class we will define one simple test, `testConfig`, which will assert that the DocNet and Document services are correctly configured by Spring and are not null.
4. As before we also need to remember to add the static `suite()` method to assure compatibility with our Maven test plugin.
5. The finished class looks as follows:

```
package com.mycompany.docnet;

import static org.junit.Assert.assertNotNull;
import junit.framework.JUnit4TestAdapter;

import org.junit.Test;

public class DocnetIntegrationTest
    extends AbstractDocnetAssemblyTestCase {

    /**
     * tests that the services are correctly initialized.
     */
    @Test
    public void testConfig() {
        assertNotNull(docnetProgramService);
        assertNotNull(documentModuleService);
    }

    public static junit.framework.Test suite() {
        return new JUnit4TestAdapter(DocnetIntegrationTest.class);
    }

}
```

If you now run the `DocnetIntegrationTest` as a JUnit test in Eclipse (right click on the file and select **Run As -> JUnit Test** from the drop down menu that appears), it should pass, showing that both the `docnetProgramService` and `documentModuleService` have been correctly contributed by the respective modules into our assembly and then therefore correctly configured for us by Spring.

7.5 DocNet Assembly Summary

That was a fairly short chapter. It may seem that we haven't done much but we have created the base of our DocNet assembly, and therefore the base for our complete DocNet application. We have also shown that without any specific configuration on our side we already have various eHF modules at our disposal.

We will return to the DocNet Assembly throughout the various additional eHF tutorials as we start to implement the various eHF modules at our disposal, and we will of course expand the JUnit tests so that they are a little more involved. However for now we have laid a good foundation on which we can further develop.

In this chapter we have very simply confirmed that our Java API - via the beans that we export in our module context - are available in our assembly, and therefore to other modules. However what about modules and applications that might want to communicate with us via web services rather than our Java API? Do we have any web services, and if not how do we define them? We'll look into that very topic in the next chapter.

8 Web Services

Modules can expose web service interfaces that can participate in an orchestrated service-oriented architecture (SOA) infrastructure. The Web Service layer allows external systems (service consumers) to access the eHF Business Service module's business logic in a programmatic fashion. For the eHF's web services we use SOAP over https.

Now several eHF modules such as Record, Document, Authorization and User Management already provide web service interfaces, but what about our DocNet module? Do we already have web services defined? If not, what do we need to do to create some?

8.1 Does DocNet Already Define Any Web Services?

Before we worry about defining our own web services, lets first take a look and see if the DocNet module already defines any by default. An easy way to find this out, is to simply deploy our assembly in Tomcat and then take a look via a web browser.

Note: ICW IDE - Apache & Tomcat Configuration



As detailed in [Integrated Development Environment](#) on page 2, for the purpose of this tutorial we have assumed that you have installed the ICW IDE. By default this includes suitable instances of Apache and Tomcat that have both been pre-configured for use with eHF based products.

If you have not installed the ICW IDE and wish to install Apache and Tomcat yourself, you will need to see the separate Infrastructure Setup Manual document which details the extra configuration that needs to be completed on top of the default install of both products.

8.1.1 Build & Deploy DocNet

To view our web services we need to build and deploy our DocNet application in Tomcat.

1. Start your HSQL database.

2. Build the DocNet Module

Open a command window, navigate to the DocNet module folder and execute `maven dev:build`.

3. Build the DocNet Assembly

Open a command window, navigate to the DocNet assembly folder and execute `maven dev:build`.

4. Copy DocNet Assembly to Tomcat

Copy the file `docnet-assembly.war` from the `target` folder of your DocNet assembly project to the following Tomcat folder: `<<ICW_IDE_INSTALL_FOLDER>>/tomcat_6.0.13.0/webapps/`.

5. Extend Apache's `mod_jk` configuration to include DocNet

This is the connector used to connect Tomcat with Apache.

Note: you can skip this step if you have previously done it during another deployment of the DocNet assembly.

- a. Open the Apache configuration file `<<ICW_IDE_INSTALL_FOLDER>>/apache/conf/httpd.conf`
- b. Add the following two lines of configuration to the end of the file:

```
JKMount /docnet-assembly node1
JKMount /docnet-assembly/* node1
```

This will then setup Apache to pass on appropriate requests to our DocNet application.

6. Start Apache

To start Apache, run the following file - `<<ICW_IDE_INSTALL_FOLDER>>/apache/bin/httpd.exe`.

7. Start Tomcat

To start Tomcat, run the following file - `<<ICW_IDE_INSTALL_FOLDER>>/tomcat_6.0.13.0/bin/startup.bat`.

After a couple of minutes you should hopefully see something similar to the following in the Tomcat console window (obviously the date and time will be different for you):

```
INFO: Server startup in 32559 ms: 2009-01-16 16:14:18,472
```

8. Browse to the DocNet Application

Open your particular browser of choice and navigate to: **<https://localhost/docnet-assembly/services>**. (You may receive a warning about an invalid certificate - simply accept the warning and have your browser continue to the page)

9. Authenticate with *user1*

You will then get prompted for authentication. By default some test data is already imported into our database during the assembly build. Among others this includes a number of test users, one of which we can use now. Use the following details to authenticate:

- User Name: `user1`
- Password: `user1`

Once authenticated you should then see a list of all the available web services currently offered by the DocNet application.

8.1.2 The Predefined DocNet Web Services

All the available web services for our application are broken down by each module that exposes web services. You should see that this list also includes our DocNet Module:

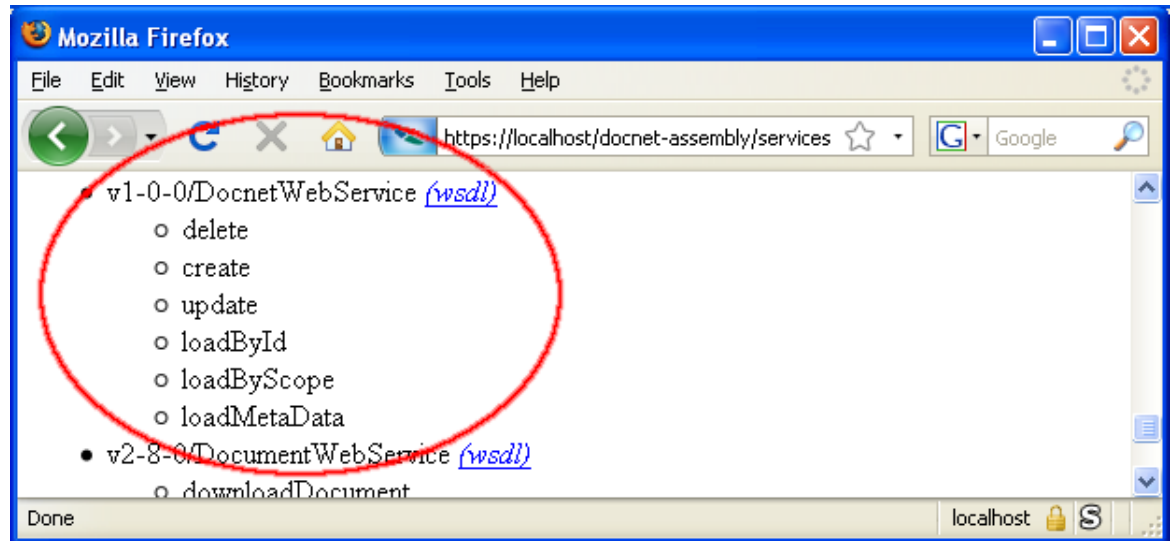


Figure 36: Default DocNet Web Services

So it seems that we do have web services for our DocNet module, but where do they come from and how can we modify them?

8.2 Configuring Web Services

When designing your module's web services, you first need to think about what methods you wish to make available, implement these in Java and then finally have your module expose them as web services. Once completed you then need to make sure that these are passed on to, and exposed by, your application's assembly. First we'll begin by worrying about our module's configuration.

8.2.1 Module Web Service Configuration

First we need to implement the Java logic for our web services, but where?

If you can remember in the module architecture stack, described in [General Module Architecture Stack](#) on page 27 previously, we have three architectural layers; *Persistence*, *Service* and *Service Adapter*. In the *Service Adapter Layer* we have two adapters; the *Service Adapter* that forms the internal API, and the *External Service Adapter* that forms the external web service API. These external service adapters are only generated for the module service stack and NOT for individual object's CRUD Service stacks. So this means that we need to implement our methods in the module's `ModuleServiceXtoAdapter`, or in our concrete case the `DocnetModuleServiceXtoAdapter`.

Once your methods have been implemented, you then need to expose them as web services by creating an appropriate Web Service Deployment Descriptor (WSDD) file. Thankfully for us, here the generator takes care of most of the hard work. All we need to do is list the names (without return types or input parameters) of the methods from the `ModuleServiceXtoAdapter` that we wish to expose in a simple fragment file - *method.wsdd.fragment*. The generator then takes this and the implementation from the `ModuleServiceXtoAdapter` and creates the appropriate WSDD file, *docnet-server-config.wsdd*.

Figure 37, shows graphically how all of this fits together, it also shows the location of the various files in question:

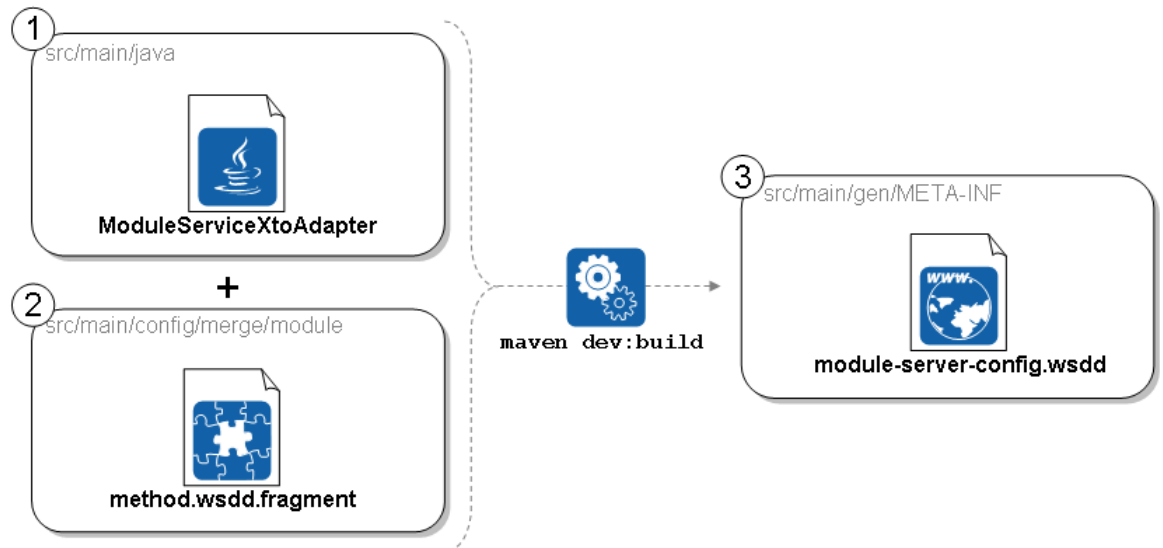


Figure 37: A Module's Web Service Configuration

So the steps involved are:

1. Define your web service methods in the `DocnetModuleServiceXtoAdapter` interface (and implement them in `DocnetModuleServiceXtoAdapterImpl`).
2. Add your method name to the list of methods contained within the fragment file `method.wsdd.fragment`
(See [Parameterized Configuration Files](#) on page 77 for more information on fragment files)
3. Build the module (`maven dev:build`), so that the generator produces the WSDD file `docnet-server-config.wsdd`.

From the module side you are basically done. All that is left is to incorporate your web services into your application (so assembly).

8.2.2 Assembly Web Service Configuration

The web services that an assembly exposes are defined in the file `axis-service-config.xml`, found under `src/main/config`. Each module simply needs to define the location of its WSDD file in this `axis-service-config.xml`.

However to save you from doing this by hand this is achieved via fragment file placeholder replacement. If you look in the `axis-service-config.xml` you will see it contains the following snippet:

```
<service-wsdd>
@@@axis.deploy.context.fragment@@@
</service-wsdd>
```

This token, `@@@axis.deploy.context.fragment@@@` is simply replaced with the content of the corresponding file from each module during the build of the assembly - with the completed `axis-service-config.xml`, landing in `src/main/gen/META-INF`.

This fragment file, *axis.deploy.context.fragment* can be found in the DocNet module under *src/main/config/merge/assembly*. If you take a look at this you will see that it already has the following content defined:

```
<value>/META-INF/docnet-server-config.wsdd</value>
```

So it already points to the location of our WSDD file. This means we do not actually need to do anything ourselves in the assembly to have our module's web services automatically exposed. Once we have added our module to our assembly's POM file (*project.xml*) we simply need to build the assembly (`maven dev:build`), and our module will be correctly included in the *axis-service-config.xml*.

Figure 38, shows graphically how all of this fits together, it also shows the location of the various files in question:

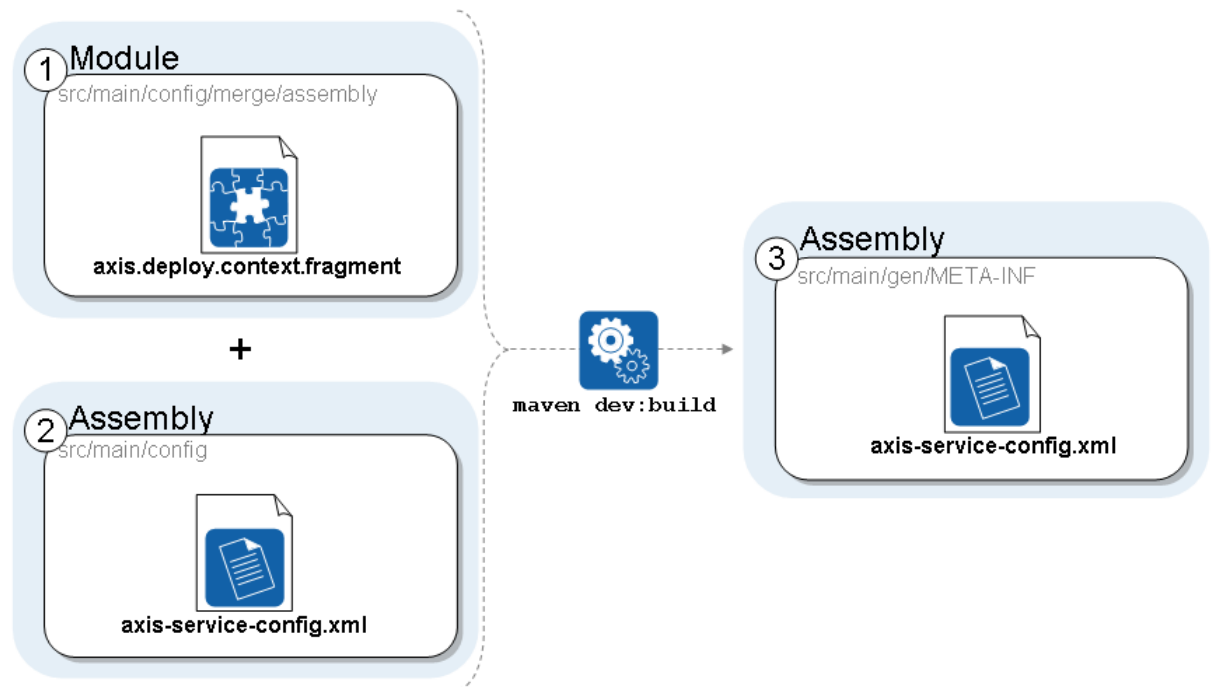


Figure 38: An Assembly's Web Service Configuration

8.3 Default Web Services

So having seen how the assembly part of web service configuration is handled we can see that any web services defined in our DocNet module will be exposed automatically. But we haven't defined any web services in our module so where did those web services from Figure 36, come from?

In [Module Web Service Configuration](#) on page 82 we said that our web service methods are implemented in `DocnetModuleServiceXtoAdapter`. If you take a look at this file you will see that it extends `ModuleServiceXtoAdapter` from eHF Commons, and it is here that these default services are actually defined. Also these methods are automatically listed in the fragment file *method.wsdd.fragment*. So we get these web services "out of the box", when we develop a new eHF based module.

If you do not want to expose these web services you can simply delete the methods from the list contained in the fragment file *method.wsdd.fragment*, and they will no longer be available.

So now that we know how to define the web services for our module, we'll take a look at put this new knowledge into practice by implementing our own new web service.

8.4 Defining a Custom Web Service

Having seen the theory we will now define our own web service for our DocNet Module.

We can see from the web services listed in [Figure 36](#) that we only have generic load methods, based on the `id` and `scope`, for all objects within our DocNet module.

Similarly to when we defined our own custom method on the *Service Adapter*, [Adding a Custom Method](#) on page 65, it would be great if we could create a `loadProgramByName` method for our web services.

8.4.1 Modeling External Transfer Objects (XTOs)

First we need to return to our model, and make a couple of small changes.

If you remember from [Modeling an Internal Service Adapter API](#) on page 48, we used the `expose` attribute of the *ehf-domainobject* stereotype to make our domain objects visible as transfer objects for the *Service Adapter*, as Data Transfer Objects (DTOs), by setting it to have the value `internal`.

Well if we want to work with our objects on the *External Service Adapter* we need to have them available as External Transfer Objects (XTOs). We do this by setting the `expose` attribute to have the value `internal & external`. This way the generator will create both DTOs and XTOs for our domain objects.

1. From within Eclipse switch to the Topcased Modeling perspective, and then using the navigator, open the *model.uml* file from within the DocNet project under *src/main/model*.
2. In the DocNet class diagram, select the `Program` class to display its **Properties** window.
3. In the left hand menu, select **Stereotype Attributes**, and then in the right hand side change the **expose** attribute from `internal` to `internal & external`.

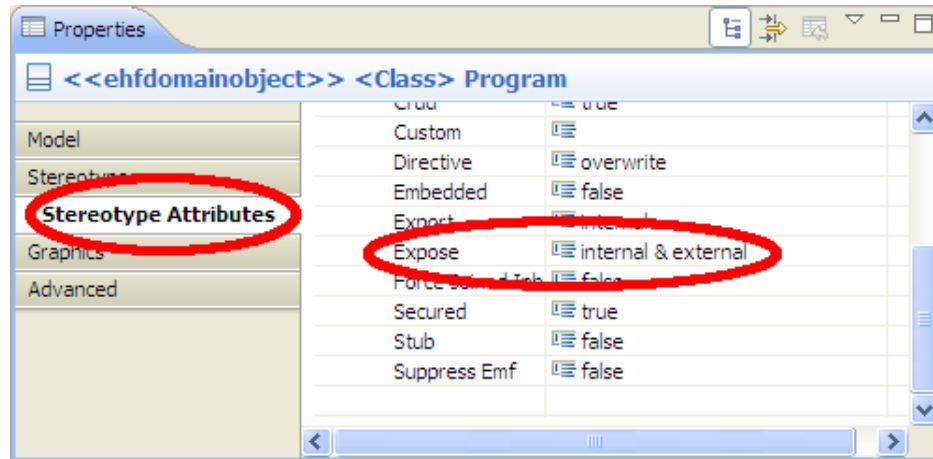


Figure 39: Setting Program's expose Attributes to internal & external

- Similarly select the `Appointment` class in the class diagram to display its **Properties** window, and from there select the **Stereotype Attributes** and set **expose** to internal & external.

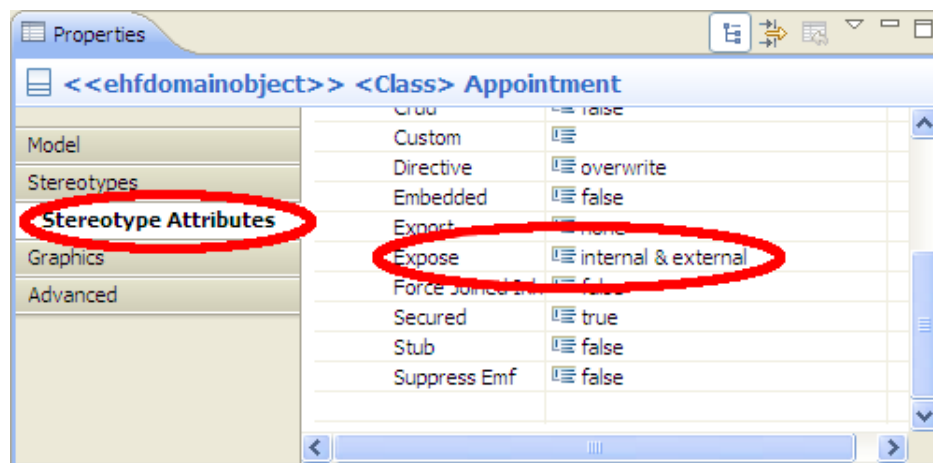


Figure 40: Setting Appointment's expose Attribute to internal & external

- Remember to save your changes before building the project again (as described in section [Building DocNet](#) on page 25) using the command `maven dev:build`, and then refresh in Eclipse.

The eHF Generator will now create appropriate XTO objects for our `Program` and `Appointment` classes, `ProgramXto` and `AppointmentXto` respectively, which we can then use in the *External Service Adapter Layer* (so our web services).

These new classes can be found in the package `com.mycompany.docnet.webapi.transfer` under `src/main/gen`.

8.4.2 Implementing a Custom Web Service

Next we will define our web service. We need to do this manually in Eclipse.

- From within Eclipse open the interface `com.mycompany.docnet.service.DocnetModuleService` under `src/main/java`, and add the following method signature:

```


```

```
Program loadProgramByName(String name);
```

Note that here we are working with plain `Program` domain objects as we are working on the *Service Layer* of our module.

2. Eclipse will now show an *Error* with the class `DocnetModuleServiceImpl`, found in the same package, that implements this interface. So we'd better fix that now. Simply implement our new method as follows:

```
public Program loadProgramByName(String name) {
    ProgramService programCrudService =
        (ProgramService) getCrudService(Program.class);
    return programCrudService.loadByName(name);
}
```

Here we simply delegate the call to `Program`'s CRUD service. First we fetch the `ProgramService`, `(ProgramService) getCrudService(Program.class);`, delegate the call to this service and return the `Program` returned by this service, `programCrudService.loadByName(name);`.

3. Having implemented our new method on the `DocnetModuleService` we can now go ahead and implement this on the `DocnetModuleServiceXtoAdapter`. Open this interface, from within the package `com.mycompany.docnet.webapi.service.adapter`, and add the following method signature:

```
ProgramXto loadProgramByName(String name);
```

Note here we are working with `ProgramXto` objects rather than the plain `Program` domain objects, as we are working on the *External Service Adapter Layer*.

4. This time Eclipse will now show an *Error* with the class `DocnetModuleServiceXtoAdapterImpl`, found in the same package, that implements this interface. Simply implement our new method in this class as follows:

```
public ProgramXto loadProgramByName(String name) {
    DocnetModuleService docnetModuleService = getModuleService();
    Program loadedProgram = docnetModuleService.loadProgramByName(name);
    return convertToTransferObject(loadedProgram, new
        AssemblerContext(true));
}
```

Here we simply fetch and delegate the method call to the `DocnetModuleService`. This returns a `Program` domain object, so finally we need to transform this into a `ProgramXto` external transfer object before returning it from our new method.

The second parameter of the `convertToTransferObject`, the `AssmeblerContext`, is used to control whether or not to include dependencies. Which in our case, we do.

5. Finally we simply need to add our new method name to the list of methods in the fragment file `method.wsdd.fragment` under `src/main/config/merge/module`. Simply add our method name `loadProgramByName` to the end of the list.

With that we are done. However before making sure that the web service is available when our DocNet Assembly is deployed in Tomcat, we'll write a JUnit test to make sure our newly implemented method in our `DocnetModuleServiceXtoAdapter` is working as expected.

8.4.3 Testing the DocnetModuleServiceXtoAdapter

As with our *Service Adapter Layer* tests, [Creating our First Service Adapter Layer JUnit Tests](#) on page 50 we will be carrying out "Black Box" testing. The `DocnetModuleServiceXtoAdapter` is exported as a bean in the DocNet **Module Context**.

Additionally as with our previous tests we will follow the same pattern and first create an Abstract test class (`AbstractDocnetModuleServiceXtoAdapterTestCase`), which is then extended by our concrete test class (`DocnetModuleServiceXtoAdapterTest`).

The AbstractDocnetModuleServiceXtoAdapterTestCase

From within Eclipse, navigate to the docnet project and under the folder `src/test/java` create the package `com.mycompany.docnet.webapi.service.adapter`.

Within this package we need to create the abstract class `AbstractDocnetModuleServiceXtoAdapterTestCase`. The complete listing is shown below.

```
package com.mycompany.docnet.webapi.service.adapter;

import javax.xml.namespace.QName;

import org.junit.After;
import org.junit.Before;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;

import com.icw.ehf.commons.transfer.TransferObject;
import com.mycompany.docnet.webapi.transfer.ProgramXto;

@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations = {"classpath:/META-INF/docnet-test-context.xml"})
public abstract class AbstractDocnetModuleServiceXtoAdapterTestCase {

    /**
     * Unique name for use in each JUnit test case, and in default programXto.
     */
    protected static final String UNIQUE_NAME = "UniqueName";

    /**
     * Default scope for use in each JUnit test case.
     */
    protected static final String TEST_SCOPE = "TestScope";
}
```

```

/**
 * Initial ProgramXto object available for each JUnit test case.
 */
protected ProgramXto programXto;

/**
 * DocnetModuleServiceXtoAdapter for use in each of the JUnit test
 * cases - will be autowired by Spring. docnetModuleServiceXtpAdapter
 * corresponds to bean export in docnet-module-context.xml,
 * which corresponds to bean mapping in docnet-webapi-context.xml.
 */
@Autowired
protected DocnetModuleServiceXtoAdapter docnetModuleServiceXtoAdapter;

/**
 * Initializes a new ProgramXto object for use in each
 * test case.
 */
@Before
public void onSetUp() {
    deleteAllObjects();

    programXto = new ProgramXto();

    programXto.setName(UNIQUE_NAME);
    programXto.setDescription("Description");
    programXto.setScope(TEST_SCOPE);
}

/**
 * Deletes all objects from the database after each test case.
 */
@After
public void onTearDown() {
    deleteAllObjects();
}

/**
 * Deletes all objects with the defined TEST_SCOPE.
 * @see #TEST_SCOPE
 */
private void deleteAllObjects() {
    TransferObject[] objects =
        docnetModuleServiceXtoAdapter.loadByScope(TEST_SCOPE,
            new QName("com.mycompany.docnet.webapi.transfer.ProgramXto"));
    docnetModuleServiceXtoAdapter.delete(objects);
}
}

```

You will notice that this class is almost identical to [The AbstractDocnetServiceAdapterTestCase](#) on page .

The main differences are as follows:

1. As we are working on the *External Service Adapter Layer* instead of the *Service Adapter Layer* we need to use XTOs (`ProgramXto`) instead of DTOs (`ProgramDto`) or plain domain objects (`Program`).
2. Also we are now using the `DocnetModuleServiceXtoAdapter` instead of the `ProgramServiceDtoAdapter`.

Otherwise, everything should look fairly familiar. We have an `onSetUp` and `onTearDown`, that provide a valid `ProgramXto` object, and clean up any data from the database after each test case has completed.

With the completion of our Abstract test class we can go ahead and write our concrete test class.

The ProgramCrudServiceAdapterTest

Within the package `com.mycompany.docnet.webapi.service.adapter`, create the class `DocnetModuleServiceXtoAdapterTest`:

```
package com.mycompany.docnet.webapi.service.adapter;

import static org.junit.Assert.*;

import org.junit.Test;
import com.mycompany.docnet.webapi.transfer.ProgramXto;

public class DocnetModuleServiceXtoAdapterTest extends
    AbstractDocnetModuleServiceXtoAdapterTestCase {

    /**
     * Store Program in the database, before then using loadProgramByName to
     * read it from the database.
     */
    @Test
    public void testLoadProgramByNameMethod() {

        assertNull(programXto.getId());

        // first store default programXto in the database
        ProgramXto[] programs = { programXto };
        docnetModuleServiceXtoAdapter.create(programs);

        // use loadProgramByName method to load the program
        ProgramXto loadedProgram =
            docnetModuleServiceXtoAdapter.loadProgramByName(UNIQUE_NAME);

        // objects persisted in database automatically receive an id
        assertNotNull(loadedProgram.getId());

    }

    public static junit.framework.Test suite() {
        return new JUnit4TestAdapter(DocnetModuleServiceXtoAdapterTest.class);
    }
}
```

As ever this is a fairly simple test case. We assert that the `programXto`'s ID is null before passing it to the `DocnetModuleServiceXtoAdapter`'s `create` method to store it in the database.

Unlike the `ProgramServiceDtoAdapter`'s `create` method, the `DocnetModuleServiceXtoAdapter`'s `create` method works on an Array of objects, so we first need to store our `programXto` in a suitable Array.

Through this `create` method call the `ProgramXto` object is transformed into a `Program` domain object and passed to the `ProgramService` which then takes care of storing domain object in the database.

We then create a new `ProgramXto` and set it using our new `loadProgramByName` method, before asserting that we really did successfully load the `Program` from the database.

Our call to the external service adapter layer (`docnetModuleServiceXtoAdapter`) is passed through the *Service Layer* (`DocnetModuleService` which delegates to `ProgramService`) down to the *Persistence Layer* (`HibernateProgramDao`). The result is then passed back through the architecture stack and returned.

Lastly if you remember from earlier, the static method `suite()` is simply included to assure compatibility with our maven test plugin which relies on versions of JUnit earlier than 4.x. We need to include this method on all our concrete test classes.

8.5 Testing Custom Web Service

Now that we have have defined and configured our custom web service in the DocNet module, we can go ahead and deploy our assembly in Tomcat and see if our new web service works as expected.

8.5.1 Authorization Detour

Before we can build and deploy our DocNet Module and Assembly to test our new web service, we need to take a small detour into the world of authorization.

As our DocNet assembly was created using the Maven `genapp` template, it already has a number of eHF modules pre-configured. This includes eHF Authentication, eHF Authorization and eHF User Management. Additionally the security they provide is turned on by default. This means that we need to give the users permissions to work with our DocNet domain objects.

To do this simply do the following:

- From within the DocNet Assembly project, open the file `src/main/resources/META-INF/ehf-assembly/bootstrap/authorization-assignments.xml`.
- This file has the following basic structure:

```
<authorization>
  <assignments>
    ...
    <assignment>
      ...
    </assignment>
    ...
    <assignment>
      ...
    </assignment>
    ...
    <!-- END OF BOOTSTRAP ASSIGNMENTS -->

  </assignments>
</authorization>
```

At the end of the file, just before the closing `</assignments>` tag add the following content to this file:

```
<assignment>
  <domainProfile>
    <profileReferences />
    <profileStale>false</profileStale>
  </profile>
```

```

<permission>
  <target>
    <type>com.mycompany.docnet</type>
    <role>*</role>
    <context>*</context>
    <identifier>*</identifier>
  </target>
  <actions>
    <action>CREATE</action>
    <action>READ</action>
    <action>UPDATE</action>
    <action>DELETE</action>
    <action>EXECUTE</action>
    <action>AUTH_CRUDE</action>
    <action>AUTH_CRUDE_DELEGATE</action>
    <action>AUTH_SEED</action>
  </actions>
</permission>
</profile>
<domain>
  <id>*</id>
</domain>
<restriction>false</restriction>
<dateRange></dateRange>
</domainProfile>
<principal class="role">
  <name>af8d3291-5a66-4bf8-993c-f8a9b5296b26</name>
  <alias>user</alias>
</principal>
</assignment>

```

We won't cover this in any detail here but basically in the above file we define a permission assignment which gives all users, defined in our application, all permissions on the DocNet domain.

This file is automatically included in the database bootstrap process.

For more information on this and other security modules and concepts, please see the eHF Reference documentation, or follow the separate eHF Security Tutorial.

8.5.2 Build & Deploy DocNet

To view our web services we need to build and deploy our DocNet application in Tomcat.

1. Start your HSQL database.

2. Build the DocNet Module

Open a command window, navigate to the DocNet module folder and execute `maven dev:build`.

3. Build the DocNet Assembly

Open a command window, navigate to the DocNet assembly folder and execute `maven dev:build`.

4. Copy DocNet Assembly to Tomcat

Copy the file `docnet-assembly.war` from the `target` folder of your DocNet assembly project to the following Tomcat folder: `<<ICW_IDE_INSTALL_FOLDER>>/tomcat_6.0.13.0/webapps/`.

5. Extend Apache's `mod_jk` configuration to include DocNet

This is the connector used to connect Tomcat with Apache.

Note: you can skip this step if you have previously done it during another deployment of the DocNet assembly.

- a. Open the Apache configuration file <<ICW_IDE_INSTALL_FOLDER>>/apache/conf/httpd.conf
- b. Add the following two lines of configuration to the end of the file:

```
JKMount /docnet-assembly node1
JKMount /docnet-assembly/* node1
```

This will then setup Apache to pass on appropriate requests to our DocNet application.

6. Start Apache

To start Apache, run the following file - <<ICW_IDE_INSTALL_FOLDER>>/apache/bin/httpd.exe.

7. Start Tomcat

To start Tomcat, run the following file - <<ICW_IDE_INSTALL_FOLDER>>/tomcat_6.0.13.0/bin/startup.bat.

After a couple of minutes you should hopefully see something similar to the following in the Tomcat console window (obviously the date and time will be different for you):

```
INFO: Server startup in 32559 ms: 2009-01-16 16:14:18,472
```

8. Browse to the DocNet Application

Open your particular browser of choice and navigate to: **https://localhost/docnet-assembly/services**. (You may receive a warning about an invalid certificate - simply accept the warning and have your browser continue to the page)

9. Authenticate with *user1*

You will then get prompted for authentication. By default some test data is already imported into our database during the assembly build. Among others this includes a number of test users, one of which we can use now. Use the following details to authenticate:

- User Name: *user1*
- Password: *user1*

Once authenticated you should then see a list of all the available web services currently offered by the DocNet application.

8.5.3 Creating and Sending SOAP Requests

Now that our DocNet application is deployed and running, you should hopefully see the following web services listed for our DocNet module, including our custom web service `loadProgramByName`.

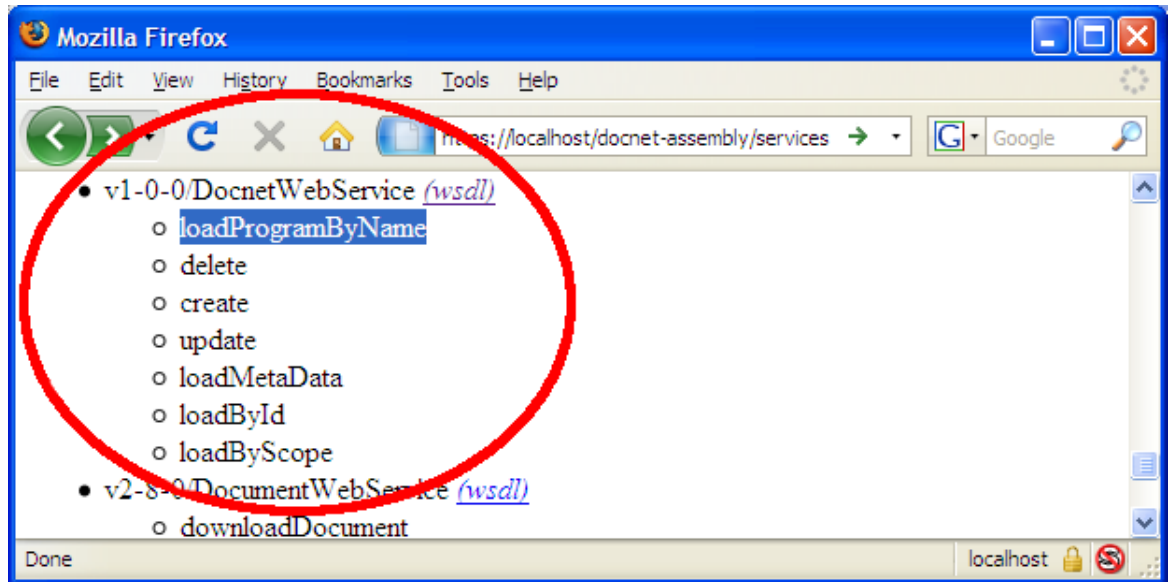


Figure 41: DocNet Web Services Including loadProgramByName

For testing purposes you need to use a web services client, to send the appropriate SOAP requests to the server. In the creation of this document we have used eviware's soapUI (<http://www.soapui.org>), but feel free to use your SOAP client of choice.

To test our `loadProgramByName` we will need to send two SOAP requests:

- `create` - to first store a `Program` in the database, something needs to exist before we can try and retrieve it.
- `loadProgramByName` - to see if we can retrieve our newly created `Program`.

Using soapUI

From within soapUI, we need to set up a new project and provide it with the WSDL for our DocNet project.

- From the main menu of soapUI select **File -> New soapUI Project**, give it the name *docnet-assembly* and click **OK**.
- From within the Navigator right click on the newly created *docnet-assembly* project and select **Add WSDL**. In the dialog that appears provide it with the url `https://localhost/docnet-assembly/services/v1-0-0/DocnetWebService?wsdl`, leave the checkbox *Create Requests* checked, and click **OK**.

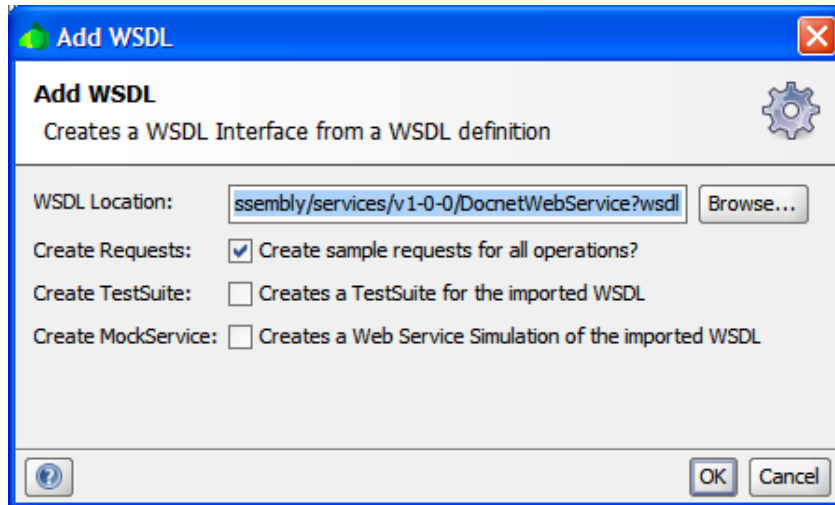


Figure 42: Configuring DocNet WSDL in soapUI

- soapUI will then attempt to load the definition for the DocNet WSDL. During this process you will be prompted for authentication. Use the following credentials:
 - Username: user1
 - Password: user1

You should then have all the possible requests listed in the soapUI Navigator:

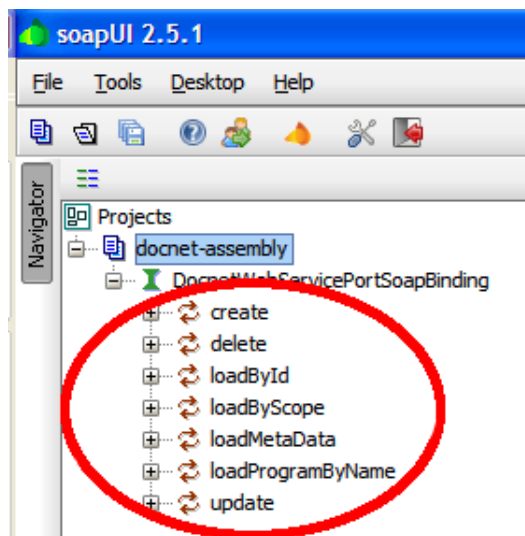


Figure 43: Listed DocNet Requests

We can now use this soapUI project to create the appropriate requests to test our `loadProgramByName` web service.

The create SOAP Request

First we need to *create* a *Program* in the database before we can attempt to retrieve it using `loadProgramByName`.

- From within the **soapUI Navigator**, right click on the **create** request and select **New request** and give it an appropriate name, "**Create Program**" (the name is not so important).
- soapUI will then attempt to create an appropriate SOAP request, unfortunately it doesn't get it quite right. Replace the contents with the following:

```

<soapenv:Envelope
  xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:ser="http://ehf.icw.com/docnet/v1-0-0/service"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <soapenv:Header/>
  <soapenv:Body>
    <ser:create>
      <!--1 or more repetitions:-->
      <ser:createObjects xsi:type="ser:Program">
        <ser:name>UniqueName</ser:name>
        <ser:description>Dummy Description</ser:description>
        <ser:scope>TestScope</ser:scope>
      </ser:createObjects>
    </ser:create>
  </soapenv:Body>
</soapenv:Envelope>

```

- Next we need to set the username and password for the request, by first selecting the *Create Program* request in the **Navigator**, and then from the **Request Properties** section of soapUI (normally found in the lower left hand corner of the main window), set both the **Username** and **Password** fields to `user1`.

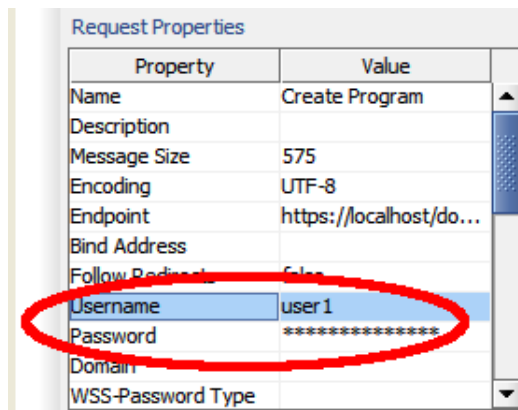


Figure 44: Setting Request Properties

- We can now execute our SOAP request by hitting the green play button on the toolbar of the *Create Program* window:

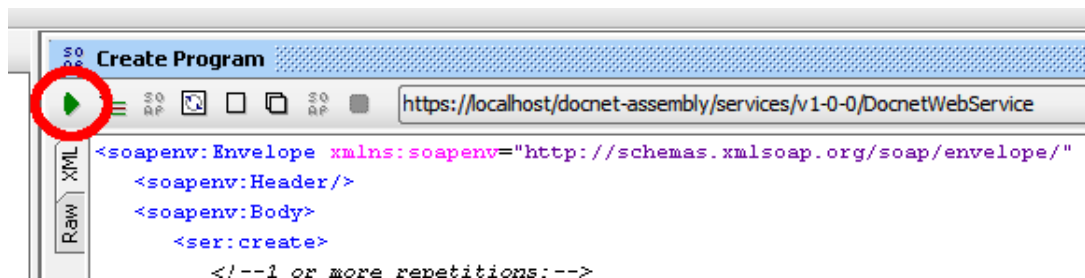


Figure 45: soapUI Play Button

- After a few seconds you should hopefully receive the following SOAP response in the right hand pane of the *Create Program* window.

```

<soapenv:Envelope
  xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"

```

```

xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<soapenv:Body>
  <createResponse xmlns="http://ehf.icw.com/docnet/v1-0-0/service">
    <createReturn xsi:type="ns1:Program"
      xmlns:ns1="http://ehf.icw.com/docnet/v1-0-0/service">
      <ns1:appointments/>
      <ns1:changeDate>2009-04-24T13:33:06.326Z</ns1:changeDate>
      <ns1:changerId>618fa909-d68e-4982-aab2-4ea90a3ef15c</
ns1:changerId>
      <ns1:createDate>2009-04-24T13:33:06.326Z</ns1:createDate>
      <ns1:creatorId>618fa909-d68e-4982-aab2-4ea90a3ef15c</
ns1:creatorId>
      <ns1:description>Dummy Description</ns1:description>
      <ns1:id>c44ea12d-2591-4b71-b6ca-67b788368c90</ns1:id>
      <ns1:name>UniqueName</ns1:name>
      <ns1:scope>TestScope</ns1:scope>
    </createReturn>
  </createResponse>
</soapenv:Body>
</soapenv:Envelope>

```

This response returns the newly created Program object, which we can see has an id (in the previous example `<ns1:id>c44ea12d-2591-4b71-b6ca-67b788368c90</ns1:id>`) so has successfully been saved in the database.

The create SOAP Request

Having successfully created our Program in the database we can now go ahead and test our loadProgramByName web service.

- From within the **soapUI Navigator**, right click on the **loadProgramByName** request and select **New request** and give it an appropriate name, "**Load Program By Name**" (the name is not so important).
- Again soapUI will attempt to create an appropriate SOAP request. This time it gets it right. In the `<ser:name>` tag simply replace the ? with **UniqueName**.

```

<soapenv:Envelope
  xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:ser="http://ehf.icw.com/docnet/v1-0-0/service">
  <soapenv:Header/>
  <soapenv:Body>
    <ser:loadProgramByName>
      <ser:name>UniqueName</ser:name>
    </ser:loadProgramByName>
  </soapenv:Body>
</soapenv:Envelope>

```

- As with the *Create Program* request, set the **username** and **password** properties for the request to user1.
- We can now execute our SOAP request by hitting the green play button on the toolbar of the *Load Program By Name* window.
- After a few seconds you should hopefully receive the following SOAP response in the right hand pane of the *Load Program By Name* window.

```

<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

```

```
<soapenv:Body>
  <loadProgramByNameResponse xmlns="http://ehf.icw.com/docnet/v1-0-0/
service">
    <loadProgramByNameReturn>
      <appointments/>
      <changeDate>2009-04-24T13:33:06.326Z</changeDate>
      <changerId>618fa909-d68e-4982-aab2-4ea90a3ef15c</changerId>
      <createDate>2009-04-24T13:33:06.326Z</createDate>
      <creatorId>618fa909-d68e-4982-aab2-4ea90a3ef15c</creatorId>
      <description>Dummy Description</description>
      <id>c44ea12d-2591-4b71-b6ca-67b788368c90</id>
      <name>UniqueName</name>
      <scope>TestScope</scope>
    </loadProgramByNameReturn>
  </loadProgramByNameResponse>
</soapenv:Body>
</soapenv:Envelope>
```

This response returns the requested Program object, confirming that our loadProgramByName web service works as expected.

8.6 Web Services Summary

In this chapter we have seen how it is possible to define our own custom web service, and how the web services are configured in the eHF.

We created our new method, loadProgramByName on the *External Service Adapter Layer* of our module's architecture (DocnetModuleServiceXtoAdapter), from which we ultimately delegated to our original custom method, loadByName of the ProgramService.

We then tested our new method via both a JUnit test and via a web service client.

This covers the basics of web services within the eHF. However there is a lot more including:

- Modifying the namespace url under which the web service is known
- Versioning of web services
- and the associated backwards compatibility of web services

As ever this information can be found in the eHF Reference documentation.

9 Summary & Outlook

Well, we've covered quite a bit in this tutorial through the creation of our DocNet Application.

- Starting from scratch we used maven, together with the genapp (`maven genapp`) and eclipse (`maven eclipse:eclipse`) plugins, and the eHF predefined project templates to create a skeleton project structure for our DocNet module. This took care of a lot of default configuration for us.
- From there we created our first UML model, and let the eHF Generator loose. While it was working its magic, we took a look at the three layered architecture (*Service Adapter Layer, Service Layer and Persistence Layer*) and the classes and configuration files that it was busy creating for us.
- We then expanded our model, by defining an API, constraints and a custom method. On each occasion we had the eHF Generator run again before testing our new functionality.
- At this point we took a small detour and created our DocNet assembly that integrates our DocNet module (business logic) together with the eHF modules and forms our deployable application.
- Upon returning to our DocNet module, we expanded it further to create a custom web service, and saw this in action when we deployed our complete DocNet application.

Through all of this you have hopefully gained a taste for how the eHF, and specifically the generator, can help you when creating a new module, as well as being introduced to the basic eHF Architecture design and concepts.

So what now? Well there are further tutorials available covering the various modules available from the eHF. In the following next sections, we will give you an outlook on some of these out-of-the-box eHF modules that simplify your application development.

9.1 Record

The eHF Record module delivers the basic foundation for an electronic health record, describing the central information in a medical system. This information is exposed by appropriate services that provide access to the following data:

1. **Administrative Data:** Administrative data comprises data on the subject # of the record, medical contacts (e.g. the family doctor or the primary pharmacy), and emergency contacts.
2. **Basic Medical Data:** Basic Medical Data primarily defines a common basic data set for medical information that is intended to be stored in a structured way within an electronic health record. The general data structures of Basic Medical Data are based on the HL7 v3 RIM, and aim at supporting basic data structures for the following main classes defined by HL7 v3 RIM:
 - Encounter
 - Observation (including measurements of different types, as well as diagnoses, health risks and allergies)
 - Substance administration (including medication)
 - Additional Medical Information like procedures and vaccinations

This module provides Create, Read, Update and Delete (CRUD) services for data access, and other services like calculation, filtering and validation for GUI and external web service consumers.

The eHF Record module comprises components which are dedicated to one of the previously mentioned data domains. An additional feature is the support of links:

- amongst basic medical data and
- external references to objects from other modules (e.g. documents).

The domain objects in this module are equipped with the standard persistence mechanisms created by the eHF Generator.

The Record module is composed of two sub modules separating the administrative from the medical data. Organizing information in sub modules enables to physically divide the storage of the persisted information. The organization of the module in sub modules supports central data protection and pseudonymization considerations.

9.2 eHF Code System

Many terms, nomenclatures, dictionaries, controlled vocabularies and classification systems have been introduced by standardization organizations. Within the eHealth Framework they are summarized as so-called code systems. The eHealth Framework portfolio includes the Code System Module. This allows you to store data using code systems, and to ultimately handle, validate, and resolve codes in different languages. The module offers dedicated flexibility to manage code systems and enables other eHF modules to address code-based tasks. Code systems strongly supplement the expressiveness of the domain models, and may include domain-specific code systems to describe administrative data, such as gender, country codes and medical codes (e.g. diagnosis and encounter type), as well as system-level code systems, such as error codes or validation messages.

Code-based validation of the data model instances ensures the consistency of the overall information stored in the system and supports specific demands regarding internationalization and localization through customized code systems. The interfaces exposed by this module provide an abstraction for different implementations of the code system. Such implementations can be delivered either directly by the ICW eHealth Framework or by adaptors utilizing of external catalogs and services.

The code systems provided by eHealth Framework are described in an XML format and can be imported into the database as part of the standard build or deployment procedure.

All of this and more is covered in the eHF Code System tutorial. It will show you how to expand your model to include codes, and then integrate eHF Code System into your application. It then covers the main aspects of eHF Code System's API, demonstrating how it can be used to find and resolve codes, for validation, translation and localization.

9.3 eHF Document

The Document module represents the central document storage of the eHealth Framework. It enables the storage of arbitrary content. The content can be in a binary format (e.g. pictures) or may be of textual nature, such as plain text or XML files. The module additionally supports schema validation. A web service interface is provided in order to upload or download health record relevant documents into or from the system. The module supports a specialized versioning mechanism. Updating or deleting a document will result in a new version number of the document, leaving previous versions unaltered.

The Document Management module provides two basic methods for uploading and downloading documents:

1. The **upload method** takes the given document or a composition of documents and stores it into the system. If the document does not exist as an entry in the system,

a new entry is created for it. Otherwise, the existing document is updated and the version number of the document is incremented.

2. The **download method** provides the functionality to download documents and document compositions.

All of this and more will be covered in the eHF Document tutorial.

9.4 Security

Medical data is the most precious content in health care applications. It is therefore mandatory for any health care application to protect their data from unauthorized access and tampering. The eHealth Framework (eHF) provides building blocks that facilitate the implementation of effective data security.

Application security comprises methods to verify who is interacting with the system, and to check whether this interaction is permitted. Additionally, application security attempts to prevent unauthorized system access through forgery or circumvention of security measures.

The eHealth Framework supports application security in the following areas:

- The **Authentication** module verifies the identity of interacting users, typically by username/password or PIN combinations.
- The **Security Token Service** supports authentication by providing additional authenticity validations based on WS-Trust security tokens.
- The **Certificate Validation** service manages authentication through X.509 certificates, including rejection of revoked certificates.
- The **Authorization** module ensures that an authenticated user may only perform permitted actions.
- The **CSRF Guard** prevents cross-site request forgery attacks by encoding unique request/response identifiers into the client/server communication.
- **Hardened Apache and Tomcat configurations** attempt to close loopholes that may be used by intruders to gain access to the system, for example through unused modules.

Data security comprises methods to protect an application's raw data. While application security works *within* the application, data security attempts to protect the data *outside* of the application's reach. There, it may be subject to access or tampering by unauthorized, yet privileged persons, for instance administrators.

The eHealth Framework supports data security in the following areas:

- At the lowest possible level, **Oracle Database Encryption** is used to render the raw database contents illegible unless the appropriate security credentials are provided.
- **Transport Level Encryption** (HTTPS, SSL) is used whenever data are transported over a network. This prevents sensitive data from being captured on the network layer.
- A **Content Scanner** can be used to filter malicious content, for example computer viruses or worms, which may gain access to servers or users' PCs.
- The **Audit** module logs security-related events such as authentication attempts or access to medical information.
- A hardened **Exception Handling** mechanism is employed to prevent stack traces from leaking out of the system boundary. They may contain sensitive information which may be used by intruders to attack the system.
- **Patches for the Axis web service library** are applied in order to prevent host names from being transmitted as part of web service responses. This information may be used by potential attackers.

Getting Started with the eHF

All of this is covered in detail in the eHF Reference documentation, while Authentication, Authorization, Audit, Exception Handling, Input Validation and CSRF Guard are covered in the eHF Security Tutorial.

HSQL Database Configuration

By default, when developing a module for the eHF a HSQL database (<http://hsqldb.org/>) is used. HSQL DB is a java database which has a very small size and can run completely in memory - which means it is ideally suited to development environments.

Module Configuration

The following details the two files that configure your module to use a HSQL database.

1. First your module needs to define a dependency to the appropriate database artifact. To this end you should find the following dependency to the appropriate **hsqldb jar** file in the module's *project.xml* file:

```
<dependency>
  <groupId>hsqldb</groupId>
  <artifactId>hsqldb</artifactId>
  <version>1.8.0.7</version>
  <type>jar</type>
</dependency>
```

2. Then the appropriate database connection details are defined in the *configuration.properties* file, located in the root folder of your module:

```
connection.dialect=org.hibernate.dialect.HSQLDialect
connection.driver=org.hsqldb.jdbcDriver
connection.url=jdbc:hsqldb:hsqldb://localhost:9999/testdb
connection.username=sa
connection.password=
```

The *configuration.properties* file is used by maven to incorporate local configuration properties at build time. Through the properties above, we are telling maven the Hibernate dialect, what JDBC driver to use, the URL of the database, and the username and password to connect with for our test Spring context. (If you're really interested, these properties are actually used to replace the corresponding tokens in the *dataSource* bean definition contained in the Spring context configuration file */src/test/resources/META-INF/docnet-system-context.xml*)

Manual Configuration of HSQLDB

If you have installed HSQLDB on your own, you need to carry out some additional configuration so that it matches the default settings defined in eHF modules. You need to go to the *bin* directory of your HSQLDB installation directory and create the following four files:

1. *server.properties* - defines the name of the database and on which port it should be available:

```
server.port=9999
server.database.0=testdb
server.dbname.0=testdb
```

2. *server-start.bat* - used to start the database on a Windows based machine:

```
@echo off
"%JAVA_HOME%\bin\java" -cp ../lib/hsqldb.jar org.hsqldb.Server
```

3. *server.sqltoolrc* - to correctly configure HSQL's SqlTool for use with our database defined previously in *server.properties*:

```
urlid testdb
url jdbc:hsqldb:hsqldb://localhost:9999/testdb
username sa
password
```

4. *server-stop.bat* - to cleanly shutdown the database on a Windows based machine:

```
@echo off
"%JAVA_HOME%\bin\java" -jar ../lib/hsqldb.jar --noinput --sql "shut-down;" --
rcfile server.sqltoolrc testdb
```

CAUTION: Shutting Down your HSQL Database



If you want to persist the changes in the database you need to make sure you shutdown the database properly by using the *server-stop.bat*.

Simply closing the command window that the HSQL database is currently running in or shutting down via the use of [Ctrl+C], will not necessarily persist any changes that have been made.

Alternatively, you can open the SQL database client of your choice, connect to the database and execute the command "shutdown" in the sql command editor window.

Spring Contexts

Spring is used throughout the eHF, complete details of which are covered in the Spring Context section of the Conventions chapter and the Spring section of the Concepts chapter in the eHF Reference documentation.

For the time being though we'll cover the fundamentals of the Spring configuration so as to provide a basic understanding.

There are two main contexts that we need to look at, a module's standard Spring context and its test Spring context.

A A Module's Standard Spring Context

When looking through various xml configuration files, it is very easy to feel lost. However with the eHF and our modularization concept the way the Spring configuration has been split up into various files should help make navigating them fairly easy. Each module's spring context has been split into numerous logical files, which in turn are stored in different folders depending on their scope, and who has control of them (either the developer or the generator).

Figure 46, shows the various Spring configuration files used to define a module's complete Spring context. The arrows between files shows the "imports" defined in the file from where the arrow originates (as we'll see shortly in the case of the Module Context it is not a classic Spring import hence the quotes). From the diagram we can see that the four Internal Context files are under the complete control of the Generator.

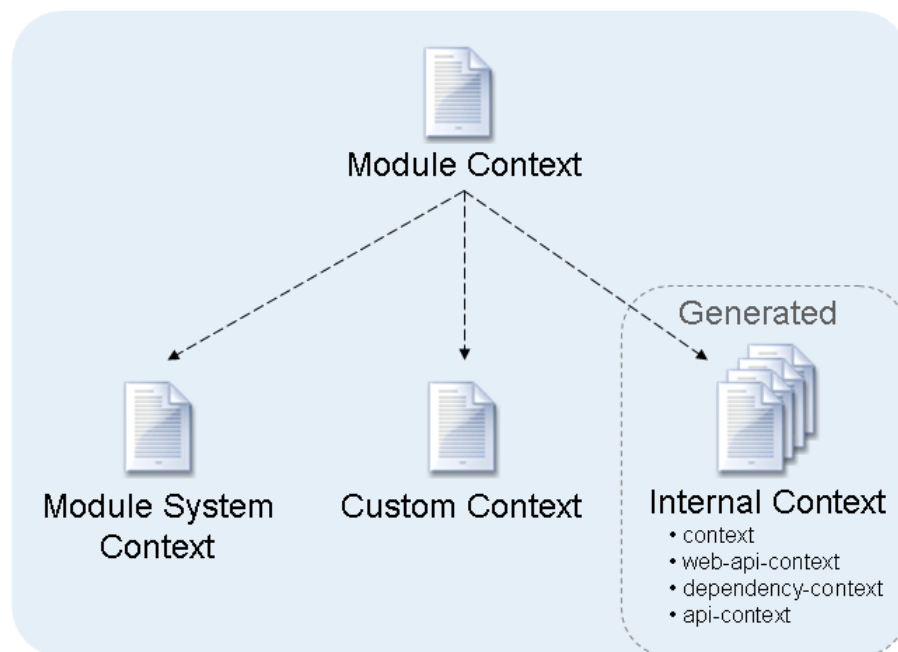


Figure 46: Standard Module's Spring Context

Note that the real file names have the module's ID appended to the front. So for example our DocNet module's "Module Context" file is named "*docnet-module-context.xml*".

The following table shows where these files can be found in your project and provides a short description on what is defined within each configuration file.

File	Location	Contains
Module Context	<p>Initially this is under the control of the generator and can therefore first be found under: <i>src/main/gen/META-INF</i>.</p> <p>You can however bring this file under developer control if you wish. After the first generator run, simply move it so that it is located under: <i>src/main/resources/META-INF</i>.</p> <p>Once moved to this location the generator will no longer create it.</p>	<p>This is the module's main context. It can also be thought of as the module's public interface for its Spring configuration.</p> <p><i>Although it imports the module system context and the internal context files it does so via a filter. This filter does not make any beans defined in these imported files available outside of the module unless they are explicitly marked as beans that should be exported.</i></p> <p>This means that only those beans that are exported in the Module Context are available to the platform context and other modules.</p> <p>This follows the concept of implementation hiding, and fits in with our public API and modularization concept. It also means that a consumer only has to focus on one set of public beans instead of looking through all the Spring context files provided by the module. Lastly it means that we can refactor all internal beans without risking incompatibility with existing consumers.</p> <p>Additionally in this file we also define what</p>

File	Location	Contains
		beans our module needs to import from the platform's context to be able to run i.e a Data Source.
Module System Context	src/main/resources/META-INF	This is for the definition of beans that are used internally by our module, but are of objects external to our module (for example the definition of a Spring interceptor) and that are not delivered by the platform.
Custom Context	src/main/config	This is the only internal Spring configuration file that is under the control of the developer. One use of this file is to allow us to define post processors to be able to modify the beans defined for us in the generator controlled Internal Context files.
Internal Context	src/main/gen/META-INF	These files define the runtime beans that are internal to our module. These spring configuration files are automatically created for us by the eHF Generator.

Table6. Standard Module's Spring Context File Contents

So the three main locations of interest when looking at the module's main Spring context are:

1. *src/main/resources/META-INF* - (if we want it to) holds the module's main context definition files. Files contained here are under the control of the developer.
2. *src/main/config* - contains the module's only internal context file that is under the control of the developer.
3. *src/main/gen/META-INF* - contains the rest of the module's internal context. These files are under the control of the generator and can not be modified by the developer directly.

All of these configuration files are then included in the module's jar file, and therefore is part of the overall platform context. Although as stated previously only those beans we

specifically mark as ones that should be exported in the Module Context file are actually made available to other modules via the platform context.



Note: Spring Context and the eHF Modularization Concept

The ability to export and import beans to and from the overall application's platform context is part of our overall Modularization concept. More details on this can be found in the Modularization section of the Concept chapter in the eHF Reference documentation.

B A Module's Test Spring Context

When we run our tests we want to be able test our module separately from both the overall platform and other modules. But how do we do this when we have dependencies to beans from the platform and other modules. This is the point where we need to define a separate **Test Context**.

To help us with our tests we use the Spring TestContext Framework (<http://static.springframework.org/spring/docs/2.5.x/reference/testing.html#testcontext-framework> ↗).

We use Spring to start a **test system context** which will *mimic* the platform context required for our tests. Such a test system context provides globally used beans (e.g. hibernate session factory or transaction manager) and service beans contributed by other modules (e.g. security service), that our module would normally receive from the Platform Application's context. Generally any service beans defined in this test context are either Mocks or Stubs of the real implementations.

Our Spring configuration file hierarchy for this test system context is shown in [Figure 47](#).

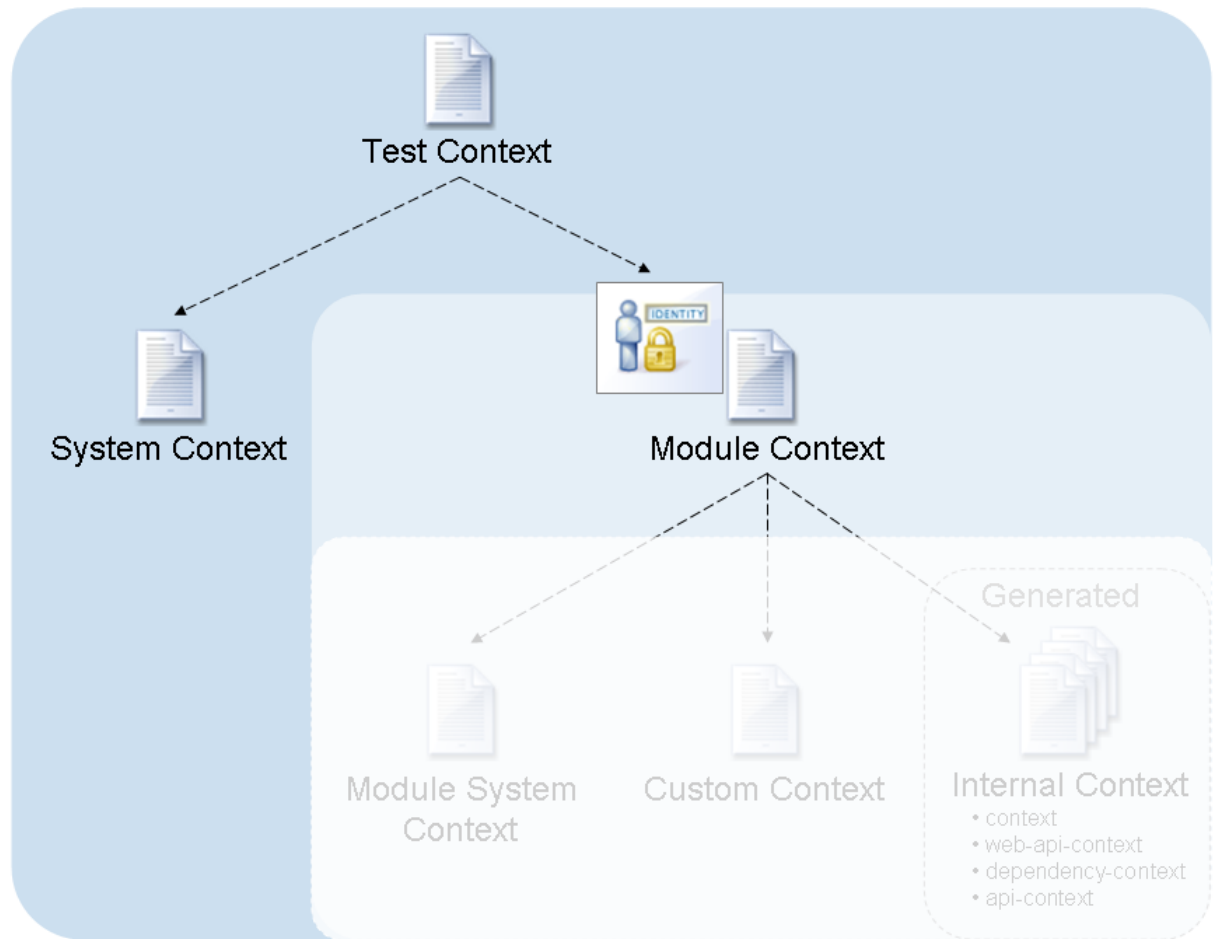


Figure 47: A Module's Spring Test Context Configuration

When compared to the module's main context definition we simply have two new files:

1. **Test Context** - this is the start of the main test context and is the only configuration file we need to use in our tests. By default it simply imports the System Context and the Module Context. It is also the location for any custom bean definitions that we might want to use in our tests but are not actually required by our module itself to run.
2. **System Context** - this is where we mimic the platform context for use in our tests. It is also where we will define beans of the mocks and stubs of service beans were the real implementation would normally be received from another module via the platform context.

Both of the above files are stored within our module under `src/test/resources/META-INF`. Unlike the other Spring configuration files these two files will NOT be included in the module's jar file. This also means that there is no problem with a conflict of bean names that might be defined in these files, when the module is part of the platform context (for example a `dataSource` definition pointing to a development database in the module and the same bean pointing to a production database in the platform).

As stated previously (you can tell this is important as we are repeating ourselves here) the **Module Context** file imports the rest of the module's context via a filter and by default does not make any bean available outside of the module - hence the security man stood in front of the Module Context when it is imported by the Test Context in [Figure 47](#). The Module Context can be thought of as the doorway to the rest of the module's Spring context and

you can only see what the Module Context makes available. This means that any beans defined in configuration files lower down in the hierarchy are **not available** to our **Test Context** (and would therefore also not be available to the platform later on). So we need to take any services (beans) that we want to make available outside of the module and explicitly export them in the Module Context.

By default the generator will export one bean for your module's main DTO service (so in our case the `DocnetModuleServiceDtoAdapter`) and one for the web service (so the `DocnetModuleServiceXtoAdapter`), and one for each exposed service defined in your model. In the DocNet case this is the `DocnetProgramService`.

You can see this by opening the file `docnet-module-context.xml`, either located under `src/main/gen/META-INF`, or under `src/main/resources/META-INF` depending on whether you have brought it under developer control or not. By default you should see that it currently exports three beans:

```
<export>
  <bean id="docnetModuleServiceXtoAdapter" interface=
    "com.mycompany.docnet.webapi.service.adapter.
DocnetModuleServiceXtoAdapter" />
  <bean id="docnetModuleService" interface=
    "com.mycompany.docnet.service.adapter.DocnetModuleServiceDtoAdapter" />
  <bean id="docnetProgramService" interface=
    "com.mycompany.docnet.transfer.adapter.ProgramServiceDtoAdapter" />
</export>
```

If you look at the rest of the `docnet-module-context.xml` file you can see that by default it defines four beans that our DocNet module needs to import (`securityService`, `sessionFactory`, `hibernateTemplate` and `transactionManager`) from the platform or other modules. These are the default beans that are required by all eHF application modules.

For those of you that are interested you will find that for our test context the module context receives these imported beans through the file `docnet-test-context.xml` which imports the configuration file `docnet-system-context.xml` which is where these beans are actually defined. So the `docnet-system-context.xml` is where we mimic our platform for our tests.

Okay so hopefully with all those configuration files we haven't lost you. The main things you need to take from all of this is that we have a module context where we define what beans we want to make available from outside of our module and we have a test context within our module that we use to mimic the platform for use in our JUnit tests.