This page last changed on Mar 10, 2011 by patrik_nordwall.

Sculptor Developer's Guide

Sculptor is not an one-size-fits-all product. Even though it is a good start for many systems, sooner or later customization is always needed. This guide will give you an understanding of the internal design of Sculptor and explains how to do changes for some typical scenarios. Some things can easily be changed with properties or AOP, while other things requires more effort, since you need to setup the development environment. Some changes are staightforward and some requires more in depth understanding of Eclipse Xpand and Xtext (previously known as oAW).

Table of Contents:

Sculptor Internal Design

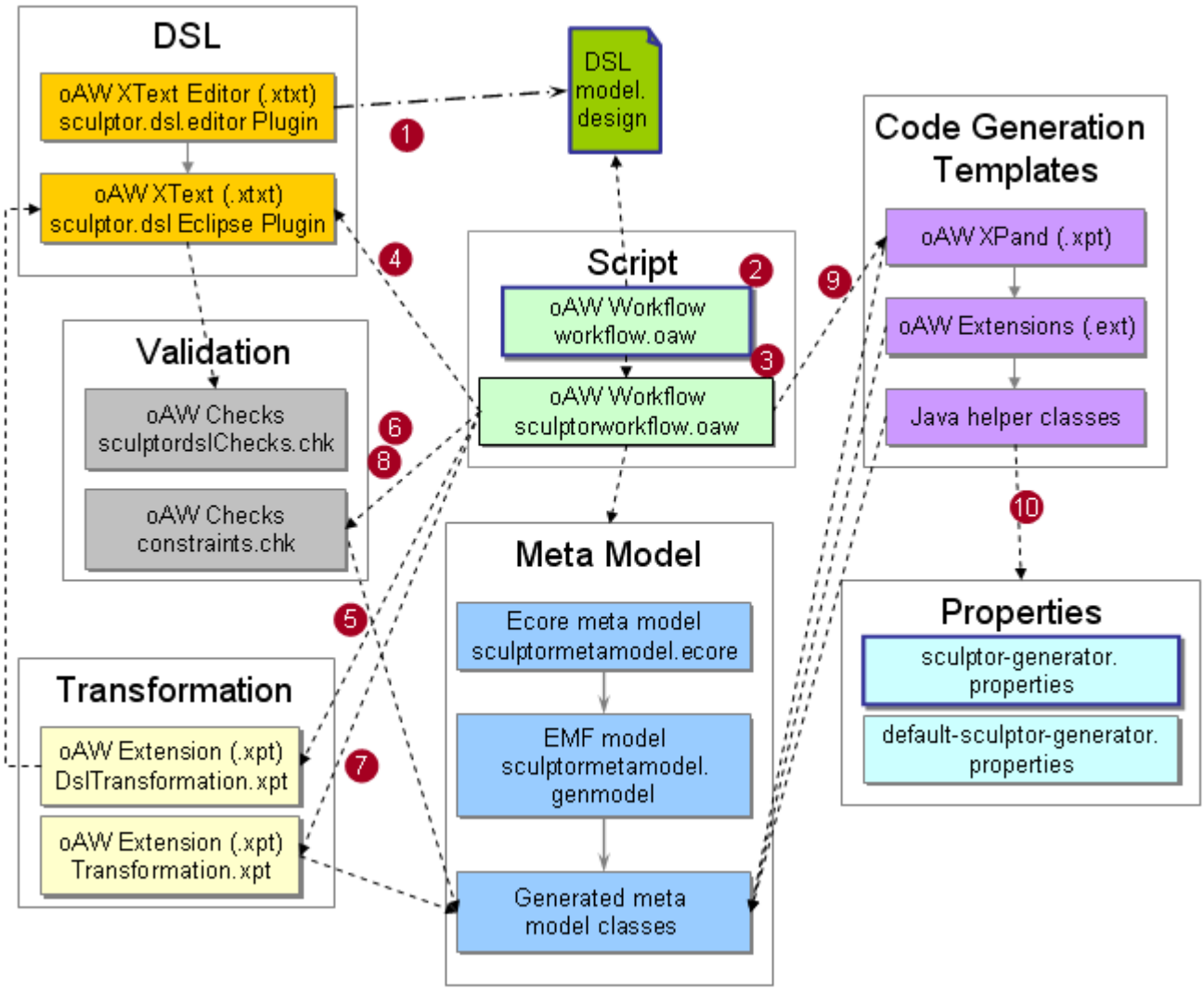


Figure 1. Internal Design of Sculptor

1. The developer is using the DSL Editor plugin to edit the application specific `model.btdesign`, i.e. the source for the concrete model that is the input to the code generation process. Constraints of the DSL is validated while editing.

2. When generating code the application specific `workflow.mwe` is executed. It doesn't contain much.

3. It invokes the `sculptorworkflow.mwe` which defines the flow of the code generation process.

4. It starts with parsing the `model.btdesign` file using the XText parser. Constraints of the DSL are checked.

5. The DSL model is transformed into a model of the type defined by the `sculptormetamodel.ecore` meta model.

6. Constraint validation is performed.

7. The model is transformed again. Now it is actually modified to add some default values.

8. Constraint validation again.

9. Now the actual generation of Java code and configuration files begin. It is done with code generation templates written in XPand language. The templates extract values from the model and uses Extensions and Java helper classes.

10. Properties of technical nature, which doesn't belong in the DSL or meta model, are used by the templates and the helpers.

Performance Tuning of Generator

By using one model file per module it is possible for Sculptor to do a partial generate of the changed modules and the ones depending on the changed modules. The file must be named the same as the module (media.btdesign, person.btdesign) or prefixed with model_ or model- (model-person.btdesign). This partial generation can shorten the generation time for large projects. `fornax-oaw-m2-plugin` will detect which model files has changed since previous generator run when using `mvn -o generate-sources`. Full generate will be done when using `-Dfornax.generator.force.execution=true` or `mvn clean generate-sources`

Note that maven -o (offline) option can reduce maven execution time a lot, if you know that you have everything locally.

I normally use `mvn -o generate-sources` when doing small changes, which is most of the time, and then `mvn clean generate-sources` or `mvn clean install`
when I have done major changes, or want to ensure that everything is working.

A major disadvantage of using `mvn clean` is that Eclipse will often not understand the changes and a separate (and annoying) Eclipse refresh and project clean is needed.

Note that Refresh in Eclipse is often time consuming. In the maven launcher you should specify Refresh "The project containing the selected resource" or "Specific resources". Don't use "The entire workspace".

`fornax-oaw-m2-plugin` runs the generator as a forked JVM process and therefore it can be necessary to define JVM parameters for large projects. This is done in the `<configuration>` section of `fornax-oaw-m2-plugin` in pom.xml

```
<jvmSettings>
    <jvmArgs>
        <jvmArg>-client</jvmArg>
        <jvmArg>-Xms256m</jvmArg>
        <jvmArg>-Xmx512m</jvmArg>
    </jvmArgs>
```

```
    </jvmSettings>
```

Generator Properties

There are a many things that can be easily customized with properties. The default properties are defined in `default-sculptor-generator.properties` in fornax-cartridges-sculptor-generator. You can override these properties by defining them in `sculptor-generator.properties` and `sculptor-gui-generator.properties`. You only have to define the ones that you need to change.

Sculptor will look for properties in the following order:

1. System.properties (only intended for temporary tests)
2. sculptor-gui-generator.properties
3. sculptor-generator.properties
4. default-sculptor-generator.properties

Project Nature

The main capabilities and responsibilities of the project is defined by the `project.nature` property. It is typically used to assign default values of more fine grained properties. E.g. the generated artifact of a presentation tier project is different from a business tier project.

The default project nature is `business-tier`. For a web project you specify:

```
project.nature=presentation-tier
```

From version 1.6, the jsp version of the presentation-tier have been removed, in favor of jsf.

A project may have several natures and the value of the property can be a comma separated list. E.g. to generate business and presentation tier in the same project you can specify:

```
project.nature=presentation-tier, business-tier
```

Framework Classes

You can replace the runtime framework with your own implementations by defining the class names in `sculptor-generator.properties`. The naming convention is that the property key is the last part, starting with `framework`, of the default Sculptor framework class.
E.g. to use your own base class for all domain objects:

```
framework.domain.AbstractDomainObject=org.myown.framework.MyAbstractDomainObject
```

Generic Access Objects

Sculptor provides generic access objects for operations such as findById, findAll, save, delete, etc. The implementation of these operations are located in the runtime framework. It is easy to replace those with your own implementation.

To replace all of them you define the following properties:

```
framework.accessimpl.package=org.mypkg.accessimpl
framework.accessimpl.prefix=
framework.accessapi.package=org.mypkg.accessapi
```

To replace a single implementation you define the class name like this:

```
framework.accessimpl.SaveAccessImpl=org.mypkg.accessimpl.SaveAccessImpl
```

If you need to replace the interface also:

```
framework.accessapi.SaveAccess=org.mypkg.accessapi.SaveAccess
framework.accessimpl.SaveAccessImpl=org.mypkg.accessimpl.SaveAccessImpl
```

When using Hibernate as JPA provider, which is the default, Hibernate specific implementations are used. For example findByCriteria is Hibernate specific. In case you want to use Hibernate as JPA provider, but stick to pure JPA implementations then you should add the following properties:

```
framework.accessimpl.prefix=Jpa
framework.accessimpl.package=org.fornax.cartridges.sculptor.framework.accessimpl.jpa
```

Database Product

MySQL, Oracle, PostgreSQL and HSQLDB-InMemory are supported out-of-the-box. You can select database with

```
db.product=mysql
#db.product=oracle
#db.product=postgresql
#db.product=hsqldb-inmemory
```

Contribution of support for Apache Derby has been posted in the Forum

The database properties are only used for generating the DDL and some Hibernate settings. You can also change the default type mapping:

```
db.mysql.type.Boolean=CHAR(1)
db.mysql.type.boolean=CHAR(1)
db.mysql.type.Integer=INTEGER
db.mysql.type.int=INTEGER
db.mysql.type.Long=BIGINT
db.mysql.type.long=BIGINT
db.mysql.type.Date=DATE
db.mysql.type.java.util.Date=TIMESTAMP
db.mysql.type.DateTime=DATETIME
db.mysql.type.Timestamp=TIMESTAMP
db.mysql.type.BigDecimal=DOUBLE
db.mysql.type.Double=DOUBLE
```

```
db.mysql.type.double=DOUBLE
db.mysql.type.String=VARCHAR
db.mysql.length.String=100
db.mysql.length.Enum=40
db.mysql.type.discriminatorType.STRING=VARCHAR
db.mysql.length.discriminatorType.STRING=31
db.mysql.type.discriminatorType.CHAR=CHAR(1)
db.mysql.type.discriminatorType.INTEGER=INTEGER
```

For parent child (one-to-many) relations the database can do cascaded delete of children when deleting parent. This is the default setting. It is possible to disable that and let Hibernate do the delete by setting the following property to `false`.

```
db.mysql.onDeleteCascade=false
```

The generated DDL SQL script contains drop statements. In case you don't wan't to generate those you can define:

```
generate.ddl.drop=false
```

Cascade

Default cascade values can be defined for different types of associations. References between different modules have no default cascade value. 'all' is default for references within the same module. You might want to change that to 'persist,merge' if you don't want delete to be propagated.

```
# default cascade values for aggregate references
cascade.aggregate=all
# bidirectional one-to-many for aggregate reference
cascade.aggregate.oneToMany=all-delete-orphan
# default cascade values for references within same module
# bidirectional many-to-many
cascade.manyToMany=all
# bidirectional one-to-many
cascade.oneToMany=all
# unidirectional to-many
cascade.toMany=all
# unidirectional to-one
cascade.toOne=all
```

Types

Mapping from simple types used in the DSL to generated types are defined with these properties:

```
javaType.Date=java.util.Date
javaType.DateTime=java.util.Date
javaType.Timestamp=java.util.Date
javaType.Map=java.util.Map
javaType.List=java.util.List
javaType.Set=java.util.Set
```

```
javaType.Collection=java.util.Collection
```

Note that the java.lang types, such as String and Integer are not defined, since they have the same name in the DSL and the Java code.

You can add your own types also, e.g.

```
javaType.AccountNumber=org.foobar.type.AccountNumber
javaType.Currency=String
javaType.Amount=BigDecimal
```

When you use your own classes, such as AccountNumber, you need to define the corresponding Hibernate user type.

```
hibernateType.AccountNumber=org.foobar.type.PersistentAccountNumber
```

If you define the above types you can use them directly in the DSL.

```
Entity Account {
  AccountNumber accountNumber
  - @Money balance
}
BasicType Money {
  Amount amount
  Currency currency
}
```

Another example, `ShortString`, which combines the database and java type properties.

```
db.mysql.type.ShortString=VARCHAR
db.mysql.length.ShortString=10
javaType.ShortString=String
```

The type of the automatically generated id attribute is defined as

```
id.type=Long
```

Joda Date and Time API

It is possible to use Joda Time instead of the Java date and time classes.

```
datetime.library=joda
```

Package Names

You can define the names of the generated packages.

```
package.serviceInterface=serviceapi
package.serviceImplementation=serviceimpl
package.serviceProxy=serviceproxy
package.consumer=consumer
package.xmlmapper=consumer
package.domain=domain
package.repositoryInterface=domain
package.exception=exception
package.repositoryImplementation=repositoryimpl
package.accessInterface=repositoryimpl
package.accessImplementation=accessimpl
package.dto=serviceapi
package.web=web
package.richClient=richclient
package.common=common
```

You can also use subpackages.

```
package.domain=domain
package.repositoryInterface=domain.repository
package.exception=domain.exception
```

Default Base Class

It is possible to define your own class to be use as extended base class for different types.

```
repository.extends=org.foo.SuperRepository
service.extends=org.foo.SuperService
entity.extends=org.foo.SuperEntity
valueObject.extends=org.foo.SuperValue
dataTransferObject.extends=org.foo.SuperDTO
basicType.extends=org.foo.SuperType
domainEvent.extends=org.foo.SuperEvent
commandEvent.extends=org.foo.SuperCommand
```

Note that extends can also be defined per domain object in model.btdesign:

```
Entity Movie extends @Media {
}

DataTransferObject SearchResponse extends org.foo.ResponseBase {
}
```

Database Naming

There are a few configuration options for different naming strategies of database elements.

Convert camel case to underscore in database names:

```
db.useUnderscoreNaming=true
```

Foreign key column names ending with _ID:

```
db.useIdSuffixInForeigKey=true
```

ID column prefixed with table name:

```
db.useTablePrefixedIdColumn=true
```

File Header

You can define a header to be generated in java files. The java comments are added automatically, but you need to include the newline character.

```
javaHeader=Copyright line 1 \n\
 line 2 \n\
 line 3
```

toString

ReflectionToStringBuilder is used for toString in DomainObjects. You can define style with property:

```
toStringStyle=SHORT_PREFIX_STYLE
#toStringStyle=MULTI_LINE_STYLE
#toStringStyle=NO_FIELD_NAMES_STYLE
#toStringStyle=SIMPLE_STYLE
```

JavaDoc of ToStringStyle describes the different formats.

You can also override this for individual DomainObject with hint

```
ValueObject Foo {
    hint="toStringStyle=MULTI_LINE_STYLE"
    String aaa
    String bbb
}
```

equals and hashCode

For composite natural keys an inner key class is generated. If you prefer to skip this inner class you can define this property:

```
generate.domainObject.compositeKeyClass=false
```

One drawback with not using key class is that `findByKeys` is not supported for composite keys.

Removing the generation of the ServiceContext

For some reason it might be required to switch off the generation of the ServiceContext - for example if you want to create a client without the created/updated functionality.
In this case it can be done easily with setting a property value in the sculptor-generator.properties file.

```
# specifies if ServiceContext is to be generated
generate.serviceContext=false
```

This turns off the generation of the ServiceContext. Please note that in this case the auditable feature does not work or must be implemented manually.

You can skip generation of auditable with:

```
generate.auditable=false
```

Null instead of NotFoundException

By default, some repository operations such as findById and findByKey throws NotFoundException if the query doesn't find any matching result. You can use the following generator property to return null instead of throwing exception:

```
generate.NotFoundException=false
```

XML configuration for Spring

If you prefer Spring bean definitions in xml files instead of annotations you can use this property:

```
generate.spring.annotation=false
```

XML mapping for Hibernate

If you prefer hibernate mapping in xml files (hbm) instead of JPA annotations you can use this property:

```
generate.jpa.annotation=false
```

Cache Provider

Cache provider settings are generated based on the property `cache.provider`.

```
cache.provider=EhCache
#cache.provider=JbossTreeCache
#cache.provider=DeployedTreeCache
#cache.provider=TreeCache
```

There are some built in cache providers.

1. EhCache - settings are defined in `ehcache.xml` file, skeleton is generated
2. JbossTreeCache - settings are defined as a JBoss mbean, EJB3EntityTreeCache
3. DeployedTreeCache - settings are defined as a JBoss mbean.
4. TreeCache - settings are defined in `treecache.xml` file

The choice also affects the cache `usage` attribute in the Hibernate mapping.

EhCache is the default cache provider, since it doesn't require any additional configuration. JBoss `JbossTreeCache` is recommended for (clustered) applications that require a transactional cache. Refer to JBoss documentation for information about how to configure the TreeCache.

Validation

In case you don't want annotations for Hibernate validator you can omit them with:

```
generate.validation.annotation=false
```

You can add your own validations or replace the implementation of the built in validators:

```
validation.annotation.AssertFalse=org.hibernate.validator.AssertFalse
validation.annotation.AssertTrue=org.hibernate.validator.AssertTrue
validation.annotation.CreditCardNumber=org.hibernate.validator.CreditCardNumber
validation.annotation.Digits=org.hibernate.validator.Digits
validation.annotation.EAN=org.hibernate.validator.EAN
validation.annotation.Email=org.hibernate.validator.Email
validation.annotation.Future=org.fornax.cartridges.sculptor.framework.validation.validator.Future
validation.annotation.Length=org.hibernate.validator.Length
validation.annotation.Max=org.hibernate.validator.Max
validation.annotation.Min=org.hibernate.validator.Min
validation.annotation.NotEmpty=org.hibernate.validator.NotEmpty
validation.annotation.NotNull=org.hibernate.validator.NotNull
validation.annotation.Past=org.fornax.cartridges.sculptor.framework.validation.validator.Past
validation.annotation.Pattern=org.hibernate.validator.Pattern
validation.annotation.Patterns=org.hibernate.validator.Patterns
validation.annotation.Range=org.hibernate.validator.Range
validation.annotation.Size=org.hibernate.validator.Size
validation.annotation.Valid=org.hibernate.validator.Valid
```

JPA annotation settings

If you prefer JPA annotations on getters instead of fields:

```
generate.jpa.annotation.onField=false
```

To generate annotations for database definition of column type:

```
generate.jpa.annotation.columnDefinition=true
```

JPA provider settings

If you prefer to use another JPA provider instead of Hibernate you can choose between EclipseLink and DataNucleus. With EclipseLink or DataNucleus you won't be able to use all features of sculptor, e.g. there is no support for Joda Date and Time API, Criteria API and limited support for enums.

To use EclipseLink add

```
jpa.provider=eclipselink
```

Replace the hibernate dependencies from pom.xml with

```
<dependency>
  <groupId>org.eclipse.persistence</groupId>
  <artifactId>eclipselink</artifactId>
  <version>1.2.0</version>
</dependency>
```

add the plugin to enhance the domain classes and the repository to resolve the dependencies

```
<plugin>
    <artifactId>maven-antrun-plugin</artifactId>
    <executions>
      <execution>
        <phase>process-classes</phase>
        <configuration>
          <tasks>
            <java classname="org.eclipse.persistence.tools.weaving.jpa.StaticWeave"
                 classpathref="maven.compile.classpath" fork="true">
              <arg line="-loglevel FINE target/classes target/classes"/>
            </java>
          </tasks>
        </configuration>
        <goals>
          <goal>run</goal>
        </goals>
      </execution>
    </executions>
</plugin>

<repository>
   <id>EclipseLink Repo</id>
   <url>http://www.eclipse.org/downloads/download.php?r=1&amp;nf=1&amp;file=/rt/eclipselink/
maven.repo</url>
</repository>
```

If you want to use DataNucleus add

```
jpa.provider=datanucleus
```

Replace the hibernate dependencies from pom.xml with

```
<dependency>
   <groupId>org.datanucleus</groupId>
   <artifactId>datanucleus-core</artifactId>
   <version>1.1.6</version>
</dependency>
<dependency>
   <groupId>org.datanucleus</groupId>
   <artifactId>datanucleus-rdbms</artifactId>
   <version>1.1.6</version>
</dependency>
<dependency>
   <groupId>org.datanucleus</groupId>
   <artifactId>datanucleus-jpa</artifactId>
   <version>1.1.5</version>
</dependency>
<dependency>
   <groupId>javax.jdo</groupId>
   <artifactId>jdo2-api</artifactId>
   <version>2.3-eb</version>
</dependency>
```

add the plugin to enhance the domain classes and the repository to resolve the dependencies

```
<plugin>
   <groupId>org.datanucleus</groupId>
   <artifactId>maven-datanucleus-plugin</artifactId>
   <version>${datanucleus.plugin.version}</version>
   <configuration>
      <mappingIncludes>**/domain/*.class</mappingIncludes>
      <log4jConfiguration>${basedir}/src/test/resources/log4j-test.properties</log4jConfiguration>
      <api>JPA</api>
      <verbose>false</verbose>
      <outputFile>${basedir}/target/datanucleus_ddl.sql</outputFile>
      <completeDdl>true</completeDdl>
   </configuration>
   <dependencies>
      <dependency>
         <groupId>org.hibernate</groupId>
         <artifactId>hibernate-validator</artifactId>
         <version>${hibernate.validator.version}</version>
      </dependency>
      <dependency>
         <groupId>hsqldb</groupId>
         <artifactId>hsqldb</artifactId>
         <version>1.8.0.4</version>
      </dependency>
  </dependencies>
  <executions>
      <execution>
         <id>enhance</id>
         <phase>process-classes</phase>
         <goals>
            <goal>enhance</goal>
```

```
            </goals>
        </execution>
        <execution>
            <id>test-schema-create</id>
            <phase>process-test-classes</phase>
            <goals>
                <goal>schema-create</goal>
            </goals>
            <configuration>
                <props>${basedir}/src/test/generated/resources/datanucleus-test.properties</props>
            </configuration>
        </execution>
    </executions>
</plugin>

<repository>
    <id>DataNucleus</id>
    <name>DataNucleus Repository</name>
    <url>http://www.datanucleus.org/downloads/maven2/</url>
</repository>
<pluginRepository>
    <id>DataNucleus</id>
    <url>http://www.datanucleus.org/downloads/maven2/</url>
    </pluginRepository>
</pluginRepositories
```

Business Component without Persistence

You can create a business component that is not using JPA or MongoDB by defining generator property:

```
jpa.provider=none
```

Remove persistence related dependencies in pom.xml

- joda-time-hibernate
- hsqldb
- dbunit
- commons-pool
- ehcache-core
- hibernate-validator
- hibernate-entitymanager
- hibernate-annotations
- hibernate-core
- persistence-api

If you are using pure-ejb3 you must add a classifier to the fornax-cartridges-sculptor-framework-test dependency:

```
            <dependency>
                <groupId>org.fornax.cartridges</groupId>
                <artifactId>fornax-cartridges-sculptor-framework-test</artifactId>
                <version>${sculptor.version}</version>
                <scope>test</scope>
                <classifier>without-jpa</classifier>
            </dependency>
```

The reason for using another jar file is that one of the test beans, JpaTestBean, should not be deployed.

Settings for Generated Diagrams

By default Sculptor generates a dot file for Graphviz for documentation purposes. This might not be needed for all projects. Please set the following property to drive the generation of this feature.

```
# specifies if umlgraph for Graphviz is to be generated
generate.umlgraph=false
```

You can change colours for the generated diagrams.

```
umlgraph.bgcolor.Module=CCCCFF
umlgraph.bgcolor.Service=CCFF99
umlgraph.bgcolor.Consumer=CCFFFF
umlgraph.bgcolor.BasicType=D0D0D0
umlgraph.bgcolor.Enum=E0E0E0
umlgraph.bgcolor.Entity=FFCC33
umlgraph.bgcolor.ValueObject=FFFF99
umlgraph.bgcolor.DataTransferObject=FFCC99
umlgraph.bgcolor.DomainEvent=FFCC99
umlgraph.bgcolor.CommandEvent=FFFF99

# Highlight core domain with other colors
#umlgraph.fontcolor.core.Module=mediumblue
umlgraph.fontcolor.core.Service=mediumblue
umlgraph.fontcolor.core.Consumer=mediumblue
umlgraph.fontcolor.core.BasicType=mediumblue
umlgraph.fontcolor.core.Enum=mediumblue
umlgraph.fontcolor.core.Entity=mediumblue
umlgraph.bgcolor.core.Entity=FFCC66
umlgraph.fontcolor.core.ValueObject=mediumblue
umlgraph.bgcolor.core.ValueObject=FFFF99
umlgraph.fontcolor.core.DataTransferObject=mediumblue
umlgraph.fontcolor.core.DomainEvent=mediumblue
umlgraph.fontcolor.core.CommandEvent=mediumblue
umlgraph.labeldistance=2.0
umlgraph.labelangle=-30
```

It is also possible to define color for individual elements using hint in model, hint="umlgraph.bgcolor=D0D0D0". Hide elements using hint="umlgraph=hide".

Deployment in Tomcat

By default Jetty is used to run the application. Instead of Jetty you can use Tomcat.
Deployment in Tomcat requires that you do the following.

1. Define the following property in `sculptor-generator.properties` in the business tier project.

```
deployment.applicationServer=Tomcat
```

2. The Tomcat definition of the datasource is located in `META-INF/context.xml` in the WAR. This file is generated once, but you might need to adjust the settings.

3. Copy the jdbc driver jar to tomcat lib directory.

4. Rebuild with `mvn clean install` in the parent directory.

5. The WAR located in the target directory of the web project can be deployed in Tomcat.

Deployment as EAR or WAR

Sculptor supports deployment as EAR or WAR. WAR is default when creating new projects with the archetypes. The design differences are:

- Services are exposed as EJBs when deployed as EAR, POJOs when deployed as WAR.
- Transaction management is done with JTA by the application server when deployed as EAR, by Spring when deployed as WAR.
- Consumers are not supported when deployed as WAR.

The Archetype Tutorial describes the steps how to convert WAR to EAR deployment. It also describes how to deploy in JBoss.

JAXB

JAXB annotations are by default only generated for DTOs. It can be turned on for other domain object types with these properties:

```
generate.xml.bind.annotation.dataTransferObject=true
generate.xml.bind.annotation.valueObject=false
generate.xml.bind.annotation.entity=false
generate.xml.bind.annotation.basicType=false
generate.xml.bind.annotation.domainEvent=false
generate.xml.bind.annotation.commandEvent=false
```

XStream

You should add the following property to your sculptor-generator.properties to avoid fully qualified class names the xml REST representations. @XStreamAlias will be generated.

```
generate.xstream.annotation=true
```

Scaffold

Scaffold operations are defined as a comma separated list.

```
scaffold.operations=findById,findAll,save,delete
```

Visibility of setters for not changeable attributes and references

By default the visibility is private for setter methods of not changeable attributes. Some tools might need visible setter methods and you can change the visibility with the following property. When you make these setters visible they will check that the value is not changed, once it has been set.

```
notChangeablePropertySetter.visibility=private
#notChangeablePropertySetter.visibility=protected
```

The visibility of setters for not changeable references are public. These setters will also check that the reference value is not changed, once it has been set. The reason for having the public setters is that

sometimes the referred object is not available at construction time. Then it is possible to pass in null in the constructor and then afterwards assign the reference. If you would like to hide these setters you can use the following property.

```
notChangeableReferenceSetter.visibility=private
```

Spring dispatcher servlet mapping

When generating the web client the default Spring dispatcher servlet mapping is /spring/*. To change it is a matter of changing a property:

```
gui.springServletMapping=foobarbaz
```

Now the mapping will be /foobarbaz/*

Spring Remoting Type

Spring remoting with RMI is used by default. The type of Spring remoting can be selected with properties in `sculptor-generator.properties`:

```
The type of remoting can be selected with
spring.remoting.type=rmi
#spring.remoting.type=hessian
#spring.remoting.type=burlap
#spring.remoting.type=httpInvoker
```

Hessian, Burlap and HttpInvoker requires some additional configuration in `web.xml`, as described in Spring remoting documentation

Spring Configuration

You can change the settings in `spring.properties`. It is runtime configuration of Spring.

Spring makes it possible to override bean definitions. The last definition wins. It is `applicationContext.xml` that is loaded first and it imports other configuration files. `more.xml` is imported last, which means that you can override generated bean definitions by specifying them in `more.xml`. This is also true for beans defined with annotations (@Repository, @Service, @Component). When running JUnit tests the corresponding file is `more-test.xml`.

This approach is not recommended, since it might be confusing to have multiple bean definitions. It is also easy to forget to change when something is renamed and that can have dangerous side effects. For testing or temporary modifications it might be an easy solution, but for permanent customization it is better to use one of the other alternatives presented in this article.

Change Generation Templates

You can change the code generation templates using the Aspect-Oriented Programming features in Xpand. It is described in Aspect-Oriented Programming in Xtend .

To use AOP you add a section in `workflow.mwe` in your application project.

```
        <component adviceTarget="generator" id="reflectionAdvice"
```

```
                class="org.eclipse.xpand2.GeneratorAdvice">
            <advices value="templates::SpecialCases" />
            <fileEncoding value="iso-8859-1" />
        </component>
```

Define your advice in `src/main/resources/templates/SpecialCases.xpt`. For example if you need to replace the UUID generation:

```
«IMPORT sculptormetamodel»
«EXTENSION extensions::helper»
«EXTENSION extensions::properties»

«AROUND templates::domain::DomainObjectAttribute::uuidAccessor FOR DomainObject»
    public String getUuid() {
        // lazy init of UUID
        if (uuid == null) {
            uuid = org.myorg.MyUUIDGenerator.generate().toString();
        }
        return uuid;
    }

    private void setUuid(String uuid) {
        this.uuid = uuid;
    }
«ENDAROUND»
```

You find the default templates in [fornaxcartridgessculptorgenerator/src/main/resources/templats](). Everything starts in Root.xpt, which you also can intercept to add more templates or exclude some of the existing templates.

Another alternative is to setup the development environment according to the next section and change the original templates and build a new version of fornax-cartridges-sculptor-generator.

Sometimes you need to override the extension functions used by the templates. This can also be done with the AOP features of Xtend.

Add extensionAdvices in the generator advice in `workflow.mwe` in your application project.

```
        <component adviceTarget="generator" id="reflectionAdvice"
                class="org.eclipse.xpand2.GeneratorAdvice">
            <advices value="templates::SpecialCases" />
            <extensionAdvices value="extensions::SpecialCases" />
            <fileEncoding value="iso-8859-1" />
        </component>
```

Define your advice in `src/main/resources/extensions/SpecialCases.ext`. For example if would like to change the location of spring resource files:

```
import sculptormetamodel;

extension extensions::helper;

around extensions::properties::getResourceDir(Application application, String name) :
  name == "spring" ?
    "spring/" :
    ctx.proceed();
```

```
around extensions::properties::getResourceDir(Module module, String name) :
 name == "spring" ?
  "spring-" + module.name + "/" :
  ctx.proceed();
```

hint

A very useful extension mechanism is available via the `hint` keyword in the model. It is possible to use a hint on almost any element in the model. It may contain any key/values, which can be used in `SpecialCases` to customize the code generation. For example, if you need to use a special database sequence for the ids of some entities. In the model you can use the hint:

```
   Entity Person {
      hint="idSequence=SEQ2"
```

In `SpecialCases.xpt`:

```
«AROUND templates::domain::DomainObjectAttributeAnnotation::idAnnotations FOR Attribute»
   «IF getDomainObject().hasHint("idSequence")»
      @javax.persistence.Id
      @javax.persistence.GeneratedValue(strategy=javax.persistence.GenerationType.SEQUENCE,
         generator="«getDomainObject().getHint('idSequence')»")
      @javax.persistence.Column(name="«getDatabaseName()»")
   «ELSE»
      «targetDef.proceed()»
   «ENDIF»
«ENDAROUND»
```

Several hints can be defined, separated with comma, and the hint may have a value or not.

```
      String someAttribute hint="aaa=A,bbb,ccc"
```

Setup Development Environment

To be able to do more adjustments you need to setup Eclipse for development of Sculptor. Please set default encoding in Eclipse to ISO-8859-1. Normally you can do it for whole eclipse in Window -> Preferences ->> General -> Workspace ->> Text file encoding -> ISO-8859-1.

Checkout from Subversion

Install a Subversion plugin, such as Subversive

Checkout the following projects from https://fornax.svn.sourceforge.net/svnroot/fornax/trunk/cartridges/sculptor/

- fornax-cartridges-sculptor-framework
- fornax-cartridges-sculptor-framework-web
- fornax-cartridges-sculptor-generator
- fornax-cartridges-sculptor-metamodel
- fornax-cartridges-sculptor-parent
- org.fornax.cartridges.sculptor.dsl

- org.fornax.cartridges.sculptor.dsl.ui
- org.fornax.cartridges.sculptor.gui.dsl
- org.fornax.cartridges.sculptor.gui.dsl.ui
- fornax-cartridges-sculptor-archetype
- fornax-cartridges-sculptor-archetype-ear
- fornax-cartridges-sculptor-archetype-jsf
- fornax-cartridges-sculptor-archetype-parent
- fornax-cartridges-sculptor-archetype-standalone
- org.fornax.cartridges.sculptor.framework.richclient
- org.fornax.cartridges.sculptor.richclient.lib
- org.fornax.cartridges.sculptor.wizard

Fornax Maven Launcher

Checkout Fornax maven-launcher as described in the Installation Guide (if you haven't already done that).

When you checkout/import the Fornax Maven Launcher into the workspace you can run maven inside Eclipse. The menu items for this are available in the external tools menu.

Try mvn

Verify that your environment is correct by running `Fornax [mvn-install]` for fornax-cartridges-sculptor-parent. Refresh all projects in Eclipse. Thereafter you should not have any red crosses (problems) in Eclipse. Sometimes, validation errors in code generation files (.xpt and .ext) must be cleaned. This is can be done with clean of the Eclipse project.

Checksum plugin

This maven plugin (fornax-checksum-m2-plugin) is responsible for deleting unchanged files from one-time-generate directories. Before when you changed for example Service name or method in your design file or SpecialCases.xpt you have to remove or change one-time-generated files manualy even if you don't touch them. This plugin care about this untouched files and they are removed before generation. New checksums are generated after generation. Everything is stored in 2 files. One is '.checksum.txt' which hold checksums and second is '.ignore-checksum.txt' which hold file names of changed (touched) files. Plugin is by default checking following directories: 'src/main/java,src/main/resources,src/test/java,src/test/resources'.

Some important notices:

- DO NOT put '.checksum.txt' and '.ignore-checksum.txt' to version system. It's not tragical mistake but you will have many problems with merging this files when updating. Plugin works nice without this. (For SVN add this files to svn:ignore)
- DO NOT change one-time-generated files after unsuccesful generation. This change will not be noticed by plugin and in next round, file will be deleted and regenerated. How to detect this? Check 'mvn' output. If you see line "ChecksumValidator: GENERATE CHECKSUMS" you are on safe side, you can change whatever you want. Maybe compilation crashed but generation went fine. If you don't see this line fix your model file or SpecialCases.xpt and restart generation 'mvn clean install'.
- DO NOT put all files under version control. Only changed files should go to version control system (files from src/generated should never go into version control system). How to achieve this, read following HOW-TO section.

HOW-TO:

- HOW TO START REGENERATION OF CHANGED FILE - remove file from filesystem and also from .ignore-checksum.txt
- HOW TO CHECK IF ALL CHANGED FILES ARE UNDER VERSION CONTROL - just to be 100% sure run 'mvn -Dmaven.test.skip=true clean install clean'. After this you can start your favorit version synchronisation tool. If you want to double check your tool. All files in .ignore-checksum.txt should be under your version control (don't forget to run 'clean install clean' before this check).

Reference Application

When you customize Sculptor it is important that you have a representative reference application. It is used to verify the changes to the code generator. It must be big enough to span most of the design concepts you use, but still small enough to be supple.

When you do refactoring of the code generator it is recommended that you first do manual refactoring of the reference application. Keep that code base as a baseline. When you have changed the code generation the generated code should correspond to the manual baseline. Use a diff tool to compare the results.

The library example can act as a reference application if you haven't developed one of your own. Checkout from Subversion:
https://fornax.svn.sourceforge.net/svnroot/fornax/trunk/cartridges/sculptor/fornax-cartridges-sculptor-examples-library

Run the JUnit tests to verify that it works before you start changing.

Eclipse project dependencies
It is convenient to use Eclipse project dependencies from the reference application to fornax-cartridges-sculptor-generator. Then you can do changes in the templates and try them directly, without having to build with maven. You can run the code generation directly inside Eclipse. In your reference application project, right click on `workflow.mwe` and select 'Run as MWE workflow'.

Constraint Validation

Check Language is used for validation of model constraints.

DSL Constraints

DSL constraints are validated directly when you edit the DSL file with the DSL editor (error highlight). DSL constraints are defined in `SculptordslJavaValidator.java` and `SculptordslChecks.chk` in org.fornax.cartridges.sculptor.dsl. If you change these you have to rebuild the plugins, but no other changes are needed.

Note that the constraints are both a help when editing the DSL and also a way to enforce certain design decisions.

Meta Model Constraints

Constraints of the ecore meta model is located in `constraints.chk` in fornax-cartridges-sculptor-generator.

Transformations

XTend is used for model transformations.

DSL Model Transformation

Two meta models are used. One for the DSL, which is generated from the XText grammar. Another meta model is used by the code generation templates. They are rather similar, but still different. They serve different purposes, which motivates why two meta models are needed.

It would also be possible to replace the XText DSL meta model with some other, e.g. a graphical DSL implemented with GMF.

A transformation converts the DSL model to the code generation model. This transformation is implemented in `DslTransformation.ext` in fornax-cartridges-sculptor-generator.

You can do a lot of powerful things in this transformation, without having to change code generation templates or its meta model. For example, the scaffold feature is implemented in the transformation. There is a property `scaffold` for `DslEntity`. It is like an ordinary `Entity` with attributes and references, but the `DslTransformation` automatically add `Repository` and `Service` with generic CRUD operations.

When you change the DSL, except for simple syntactic sugar, you often have to change the `DslTransformation` also.

Model Enrichment

There is one transformation that enrich the model with some useful default values. It is implemented in `Transformation.ext`. For example the id attribute of each DomainObject is added by this transformation.

You can add more features to this transformation when needed. If it is not specific to the concrete syntax of the DSL, it is better to add features to this transformation than to the `DslTransformation`, since they will be available even if the concrete syntax is replaced by some other DSL.

GUI Transformation

A separate model is used for generation of the CRUD GUI. A transformation takes the domain model as input and and creates a model that is better suited for generation of the user interface. This transformation is located in `GuiTransformation.ext`

Customize the Transformations

Extension advices can be used to customize the transformations.
Add the following section in `workflow.mwe` in your application project.

```
        <component adviceTarget="modelTransformation"
                        class="org.eclipse.internal.xtend.xtend.XtendAdvice">
            <extensionAdvice value="extensions::SpecialCases" />
        </component>

        <component adviceTarget="dslTransformation"
                        class="org.eclipse.internal.xtend.xtend.XtendAdvice">
            <extensionAdvice value="extensions::SpecialCases" />
        </component>
```

Define your advice in src/main/resources/extensions/SpecialCases.ext.
For example, to skip automatic addition of uuid property:

```
import sculptormetamodel;

around transformation::Transformation::modifyUuid(DomainObject domainObject) :
    null;
```

Meta Model

Input to the code generation is the model, which is structured according to the meta models `sculptormetamodel.ecore` and `sculptorguimetamodel.ecore`. The meta models are defined with Ecore.

Business Tier Model (domain model)

Figure 2. Meta Model for the code generation model (Click to view)

If you have installed Eclipse Graphical Modeling Framework SDK (Ganymede update site) you will be able to edit the ecore file with a graphical editor, see Figure 2. Open the `sculptormetamodel.ecore_diagram` with the editor.

> **MaxPermSize**
> To prevent OutOfMemoryError when you use the graphical ecore editor you can add `-XX:MaxPermSize=128m` in eclipse.ini, which is located in the Eclipse installation directory.

When you add a totally new concept you have to add it to this meta model.
When you change the meta model it is not enough to build with maven. First you must generate EMF model code.

1. An EMF genmodel is created from the ecore model. Normally you don't have to create a new genmodel, but when you do major changes you can remove the `sculptormetamodel.genmodel` and create a new from the ecore model using File > New > Eclipse Modeling Framework > EMF Model.
2. Open the genmodel and right click on the top node. Select Generate Model. It generates Java code to `src/main/java`. It is safe to remove that code and generate it again if needed.

Now you can package the meta model with `mvn install`.

GUI Model

Sculptor doesn't aim at implementing a general purpose GUI model that can be used for any type of GUI. We are focusing on the kind of CRUD GUI illustrated in the CRUD GUI Tutorial.

Figure 2. GUI Meta Model for the GUI parts of the code generation model (Click to view)

User Task
The model is arranged around User Tasks, The term UserTask is inspired by the presentation: From User Story to User Interface. In this presentation it is described that:

- Tasks require intentional action on behalf of a tool's user
- Tasks have an objective that can be completed
- Tasks decompose into smaller tasks

Sculptor's implementation with Spring WebFlow is that a UserTask corresponds to Flow. A conversation corresponds to a top level UserTask and its subtasks.

Code Generation Templates

Xpand Language is used for the templates.

The code generation starts in `Root.xpt`, which selects some course grained elements from the model and invokes other templates. In the templates you can access simple properties of the model objects and extension methods.

```
«DEFINE attributeTypeAndName FOR Attribute»
«getTypeName()» «name»
«ENDDEFINE»
```

The extension methods are defined with Xtend Lanuage in `helper.ext`. Some methods are implemented inside this file with the Expression Language and some delegate to Java helper classes.

Expression Language is also used in the templates to for example select elements from the model.

```
references.select(r | !r.many && (r.referenceType.metaType == BasicType))
```

How to

Some examples are implemented
Some of the examples presented here are already implemented in Sculptor. It was convenient to add documentation and using real examples while implementing new features.

How to exclude generation

For some special cases the default generation might not be appropriate and it is desirable to handle the special case with manual code instead. For example, assume we have a complex domain object and we need to do the JPA/Hibernate mapping manually.

This can be nicely solved with the Aspect-Oriented Programming feautures in Xpand. It is described in Aspect-Oriented Programming in Xtend .

To use AOP you add a section in `workflow.mwe` in your application project.

```
<component adviceTarget="generator" id="reflectionAdvice"
            class="org.eclipse.xpand2.GeneratorAdvice">
    <advices value="templates::SpecialCases" />
    <fileEncoding value="iso-8859-1" />
</component>
```

Define your advice in `src/main/resources/templates/SpecialCases.xpt`.

```
«IMPORT sculptormetamodel»
«EXTENSION extensions::helper»
«EXTENSION extensions::properties»

«REM»Skip createTable for Person«ENDREM»
«AROUND templates::db::OracleDDL::createTable FOR DomainObject»
  «IF name != "Person" -»
```

```
        «targetDef.proceed() -»
    «ENDIF -»
«ENDAROUND»

«REM»Skip dropTable for Person«ENDREM»
«AROUND templates::db::OracleDDL::dropTable FOR DomainObject»
    «IF name != "Person" -»
        «targetDef.proceed() -»
    «ENDIF -»
«ENDAROUND»

«REM»Skip DomainObject for Person«ENDREM»
«AROUND templates::db::DomainObject::domainObjectBase FOR DomainObject»
    «IF name != "Person" -»
        «targetDef.proceed() -»
    «ENDIF -»
«ENDAROUND»

«AROUND templates::db::DomainObject::domainObjectSubclass FOR DomainObject»
    «IF name != "Person" -»
        «targetDef.proceed() -»
    «ENDIF -»
«ENDAROUND»
```

Some of the generated artifacts can be controlled with properties in `sculptor-generator.properties`. Look in `default-sculptor-generator.properties` for properties named `generate.xxx`. For example, to skip generation of all junit tests:

```
generate.test=false
```

How to customize persistence.xml

To adjust the JPA/Hibernate properties in persistence.xml you can add the following in SpecialCases.xpt

```
«AROUND templates::jpa::JPA::persistenceUnitAdditionalProperties FOR Application»
    <property name="hibernate.show_sql" value="true" />
    <property name="hibernate.jdbc.batch_size" value="0" />
«ENDAROUND»
```

The above hibernate properties are useful when debugging.

How to use own Hibernate User Type

Hibernate User Type is powerful when you need conversion between domain and database representations.

You can define your own type in sculptor-generator.properties and also define its hibernateType. An example is to define a Time type like this:

```
javaType.Time=org.joda.time.LocalTime
hibernateType.Time=org.joda.time.contrib.hibernate.PersistentLocalTimeAsString
db.oracle.type.Time=VARCHAR2
```

```
db.oracle.length.Time=14
```

This means that you can use `Time` in model.btdesign and @Type(type = "org.joda.time.contrib.hibernate.PersistentLocalTimeAsString") annotation is generated in the java code.

How to change package names in the generated code

In the DSL you can define the root package for the `Application`. You can define the package of a `Module`, if the default is not satisfactory. The package name of individual Domain Objects can be specified, if the default is not appropriate.

```
Application Library {
   basePackage = com.mycorp.library

   Module common {
      basePackage = com.mycorp.common

      BasicType Address {
         package=types
         String street
         String city
      }
   }
}
```

You can change the names of the subpackages to the module by defining properties in `sculptor-generator.properties` as described in packageNames.

How to add support for another database

Generation of database schema (DDL) is specific for different database vendors. To add support for your database you can do like this.

1. In the application specific `sculptor-generator.properties` you define the database properties for your database. Note that the `db.product` value must be `custom`.

```
db.product=custom
db.custom.maxNameLength=27
db.custom.hibernate.dialect=org.hibernate.dialect.OracleDialect
db.custom.onDeleteCascade=true
db.custom.type.Boolean=CHAR(1)
db.custom.type.boolean=CHAR(1)
db.custom.type.Integer=NUMBER
db.custom.length.Integer=10
db.custom.type.int=NUMBER
db.custom.length.int=10
db.custom.type.Long=NUMBER
db.custom.length.Long=20
db.custom.type.long=NUMBER
db.custom.length.long=20
db.custom.type.Date=DATE
db.custom.type.java.util.Date=DATE
db.custom.type.DateTime=DATE
db.custom.type.Timestamp=DATE
db.custom.type.BigDecimal=NUMBER
db.custom.type.Double=NUMBER
db.custom.type.double=NUMBER
db.custom.type.String=VARCHAR2
db.custom.length.String=100
```

```
db.custom.length.Enum=40
db.custom.type.discriminatorType.STRING=VARCHAR
db.custom.length.discriminatorType.STRING=31
db.custom.type.discriminatorType.CHAR=CHAR(1)
db.custom.type.discriminatorType.INTEGER=INTEGER
```

2. Create a template for the DDL generation and locate it in `templates/CustomDDL.xpt` in the classpath, before the `fornax-cartridges-sculptor-generator` jar.

`OracleDDL.xpt` is a template you can mimic.

Contribution of support for Apache Derby has been posted in the Forum

How to remove the circular dependency check

Define the following in ((sculptor-generator.properties))

```
check.cyclicDependencies=false
```

How to add custom generic access object

The Sculptor runtime application framework provides some useful generic AccessObjects, such as `findById`, `findByQuery`, `save`. It is likely that you need to add other generic AccessObjects.

To add a new generic AccessObject you can do like this. Assume you need an AccessObject that can evict from second level cache.

```
/**
 * This AccessObject evicts objects from second level cache.
 */
public interface EvictAccess<T> {

    /**
     * Evict persistent objects. If {@link #setId(Long) id} is
     * not specified all objects of this class
     * be evicted.
     * @param persistentClass class of the persistent object
     */
    public void setPersistentClass(Class persistentClass);

    /**
     * Evict a specific object. Set
     * {@link #setPersistentClass(Class) persistentClass}
     * also.
     * @param id the id of the object to evict
     */
    public void setId(Long id);

    /**
     * Evict query cache with specified cache region
     */
    public void setQueryCacheRegion(String queryCacheRegion);

    public void execute();

}
```

```
public class EvictAccessImpl<T> extends JpaAccessBase implements EvictAccess<T> {

    private Class persistentClass;
    private Long id;
    private String queryCacheRegion;

    public void setPersistentClass(Class persistentClass) {
        this.persistentClass = persistentClass;
    }

    public void setId(Long id) {
        this.id = id;
    }

    public void setQueryCacheRegion(String queryCacheRegion) {
        this.queryCacheRegion = queryCacheRegion;
    }

    public void performExecute() throws PersistenceException {
        if (queryCacheRegion != null) {
            getHibernateSession().getSessionFactory().evictQueries(queryCacheRegion);
        }
        if (persistentClass != null) {
            if (id == null) {
                getHibernateSession().getSessionFactory().evict(persistentClass);
            } else {
                getHibernateSession().getSessionFactory().evict(persistentClass, id);
            }
        }
    }

}
```

There are a few conventions for the AccessObjects that the templates are based on:

- There must be an `execute` method. In above example it is implemented in `JpaAccessBase`, which invokes `performExecute`.
- There must be a `setEntityManager` method in the implementation. In above example it is implemented in `JpaAccessBase`.
- Input parameters are defined in the interface as setter methods. These corresponds to the parameters of the repository method.
- The result of the AccessObject must be available with `getResult` method, defined in the interface. This corresponds to the return type of the repository method. If void then no `getResult` is used.

Define the interface and implementation in `sculptor-generator.properties`:

```
framework.accessapi.EvictAccess=org.helloworld.common.accessapi.EvictAccess
framework.accessimpl.EvictAccessImpl=org.helloworld.common.accessimpl.EvictAccessImpl
```

The above addition of properties and classes means that you can use the evict AccessObject in your DSL model without any further changes of Sculptor.
E.g.

```
Repository PlanetRepository {
    findByExample;
    evict(String queryCacheRegion);
    evict(Class persistentClass);
    evict(Class persistentClass, Long id);
```

```
}
```

This would result in the following generated code in the Repository:

```
public void evict(String queryCacheRegion) {
    EvictAccess<Planet> ao = planetAccessFactory.createEvictAccess();
    ao.setQueryCacheRegion(queryCacheRegion);
    ao.execute();
}

public void evict(Class persistentClass) {
    EvictAccess<Planet> ao = planetAccessFactory.createEvictAccess();
    ao.setPersistentClass(persistentClass);
    ao.execute();
}

public void evict(Class persistentClass, Long id) {
    EvictAccess<Planet> ao = planetAccessFactory.createEvictAccess();
    ao.setPersistentClass(persistentClass);
    ao.setId(id);
    ao.execute();
}
```

The above is good enough in most cases, but it is also possible to define default parameters and return values for the RepositoryOperation, which means that you can skip those when using it in the DSL model. By this you can better support "convention over configuration".

For example assume that `evict(Class persistentClass)` is the most used operation and you would like that a simple `evict` declaration corresponds to this. Then you need to implement a `GenericAccessObjectStrategy` and register it in `sculptor-generator.properties`

```
public class EvictAccessObjectStrategy
        extends AbstractGenericAccessObjectStrategy
        implements GenericAccessObjectStrategy {

    public void addDefaultValues(RepositoryOperation operation) {
        if (operation.getParameters().isEmpty()) {
            addParameter(operation, "Class", "persistentClass");
        }
    }

    public String getGenericType(RepositoryOperation operation) {
        DomainObject aggregateRoot = operation.getRepository().getAggregateRoot();
        String aggregateRootName = GenerationHelper.getPackage(aggregateRoot) + "." +
            aggregateRoot.getName();
        return "<" + aggregateRootName + ">";
    }

    public boolean isPersistentClassConstructor() {
        return false;
    }

}
```

```
framework.accessapi.EvictAccess=org.helloworld.common.accessapi.EvictAccess
framework.accessimpl.EvictAccessImpl=org.helloworld.common.accessimpl.EvictAccessImpl
```

```
genericAccessObjectStrategy.evict=org.helloworld.common.accessimpl.EvictAccessObjectStrategy
```

To understand how to implement the strategy class you can have a look in `GenericAccessObjectManager` and the strategies implemented for the ordinary generic AccessObjects. It is located in `fornax-cartridges-sculptor-generator`. These strategies are only used by the code generator. They are not used in runtime.

The above means that you can use `evict` without any parameters in the DSL model.

```
Repository PlanetRepository {
   findByExample;
   evict;
}
```

and the generated Repository method would look like:

```
public void evict(Class persistentClass) {
   EvictAccess<Planet> ao = planetAccessFactory.createEvictAccess();
   ao.setPersistentClass(persistentClass);
   ao.execute();
}
```

For some rare special cases you might need to adjust the code generation template for the Repository, `genericBaseRepositoryMethod` in `Repository.xpt`.

You must place the genericAccessObjectStrategy class (EvictAccessObjectStrategy) in another library than your application project. It is used (only) by the code generator, and that is run before ordinary compilation, i.e. the class is not available in generate-sources phase if a clean has been done before.

How to add a transformation feature

Scaffold is a feature to be able to mark a Domain Object as scaffold and then automatically add some predefined operations (typically CRUD) to the corresponding Repository and Service. This is implemented in the transformation.

We add a boolean `scaffold` property to the DSL grammar for DslEntity. This is straightforward.

In the transformation `DslTransformation.ext` we add a Repository and Service if they doesn't already exist. Operations are added to these, if they don't exist.

```
scaffold(sculptormetamodel::DomainObject domainObject) :
   (domainObject.repository == null ?
      domainObject.addRepository() :
      null) ->
   domainObject.repository.addScaffoldOperations() ->
   (domainObject.module.services.exists(s | s.name == (domainObject.name + "Service")) ?
      null :
      domainObject.module.addService(domainObject.name + "Service")) ->
   domainObject.module.services.select(s | s.name == (domainObject.name + "Service")).
         addScaffoldOperations(domainObject.repository);
```

It is easiest to implement the actual creation and addition of model elements in Java. We implement the methods addRepository, addService and addScaffoldOperations in a Java helper class.

```
public static DomainObject addRepository(DomainObject domainObject) {
    SculptormetamodelFactory factory = SculptormetamodelFactoryImpl.eINSTANCE;

    Repository repository = factory.createRepository();
    repository.setName(domainObject.getName() + "Repository");
    domainObject.setRepository(repository);

    return domainObject;
}
```

How to change auditable column names

The auditable properties are added by a model transformation. You can modify that transformation in this way.

Add the following advice in `workflow.mwe` in your application project.

```
<component adviceTarget="modelTransformation"
                        class="org.eclipse.internal.xtend.xtend.XtendAdvice">
            <extensionAdvice value="extensions::TransformationSpecialCases" />
        </component>
```

Define your advice in `src/main/resources/extensions/TransformationSpecialCases.ext`.

```
import sculptormetamodel;

around transformation::Transformation::addAuditable(Entity entity) :
        addAuditDateAttribute(entity, "createdDate", "created_date") ->
        addAuditByAttribute(entity, "createdBy", "created_by") ->
        addAuditDateAttribute(entity, "lastUpdated", "updated_date") ->
        addAuditByAttribute(entity, "lastUpdatedBy", "updated_by");

create Attribute addAuditDateAttribute(Entity entity, String propertyName, String databaseName) :
setName(propertyName) ->
setDatabaseColumn(databaseName) ->
setType("java.util.Date") ->
setNullable(true) ->
        entity.attributes.add(this);

create Attribute addAuditByAttribute(Entity entity, String propertyName, String databaseName) :
setName(propertyName) ->
setDatabaseColumn(databaseName) ->
setType("String") ->
setLength("50") ->
setNullable(true) ->
        entity.attributes.add(this);
```

How to change syntactic sugar

It is rather easy to change the concrete syntax of the DSL. It is defined in the XText grammar `sculptordsl.xtxt`.

For example, as an alternative to `!` we would like to be able to use `not`.

Define a rule.

```
terminal NOT :
  ('!'|'not');
```

Use this rule instead of the "!" literal.

```
((notChangeable?=NOT "changeable") | ("changeable")) |
```

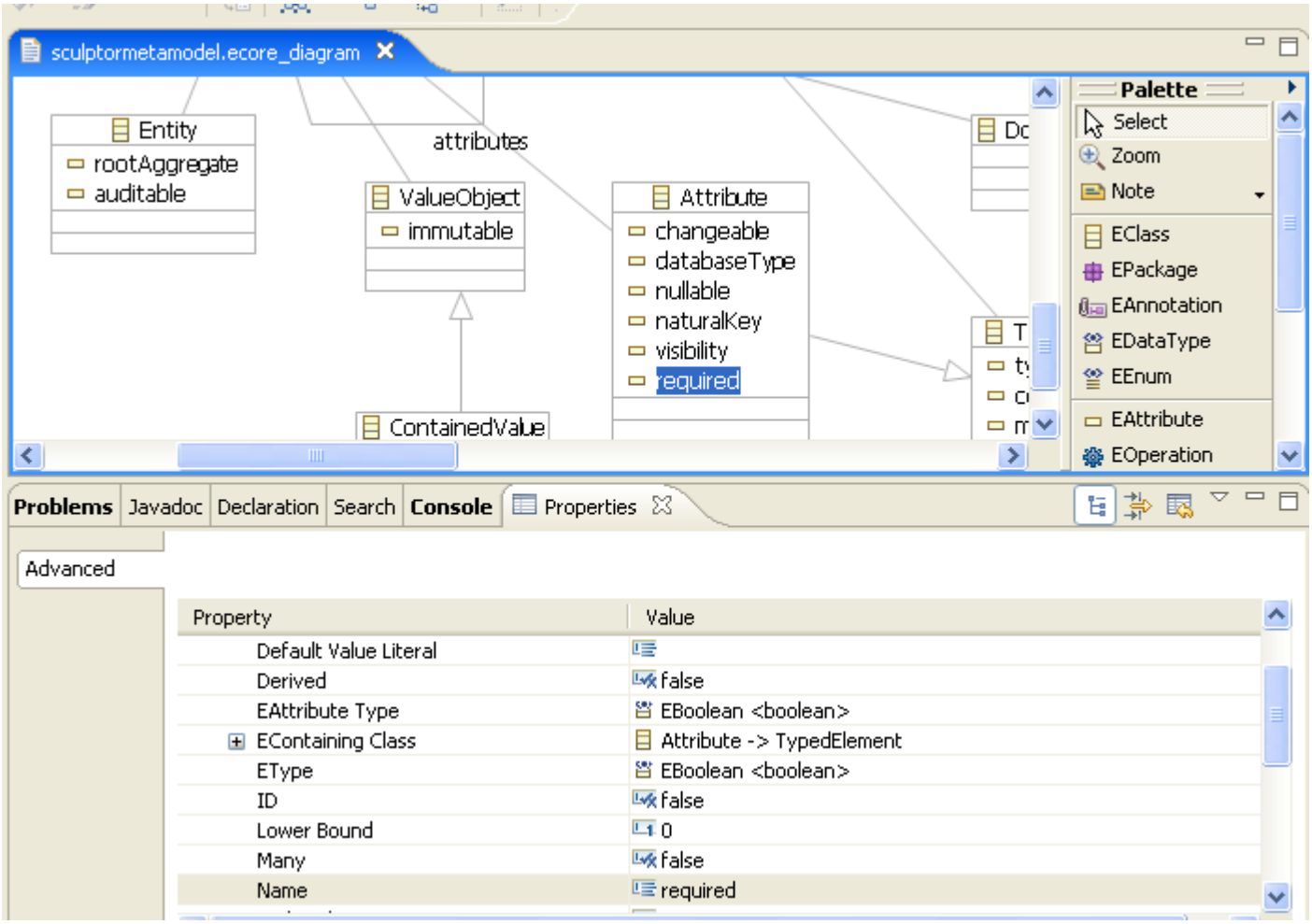
If you change the core elements of the language, such as DslService, DslEntity, DslAttribute, you have to change the transformation also, which is located in `DslTransformation.xpt` in the fornax-cartridges-sculptor-generator project.

Try DSL changes
When working with the DSL you generate the parser and editor by running the `GenerateSculptords.mwe` in the org.fornax.cartridges.sculptor.dsl project (Run as MWE Workflow). Thereafter you can try the DSL editor by launching a runtime workbench as an Eclipse Application, which includes the plugins in the workspace automatically.

How to add a new feature in the meta model

We need to be able to define an additional feature for the attributes of the Domain Objects. It should be possible to define that an attribute is included in the constructor, but still have setter method. A complement to the changeable feature. Let us call this new feature `required`.

1. Open the `sculptormetamodel.ecore_diagram` with the graphical editor.
Add a new EAttribute to the `Attribute` "class". Set the name to `required` in the Properties view. Select `EBoolean` in the `EAttribute Type`.



2. Open the `sculptormetamodel.genmodel` and right click on the top node. Select Generate Model.

3. Open `sculptordsl.xtxt` and add `required` in the same way as `nullable` in the `DslAttribute` section. Note that `required` should by default be treated as false when not specified.

```
(required?"required")?
```

4. Run `GenerateSculptordsl.mwe` in `org.fornax.cartridges.sculptor.dsl` project. This can be done by right clicking `GenerateSculptordsl.mwe` and selecting "Run as MWE Workflow".

5. Open `DslTransformation.ext` and add the "copy" of the `required` property in the same way as the `nullable` property in the create Attribute method.

```
setRequired(attribute.required) ->
```

6. Modify the code generation templates. In this case it was only needed to adjust the logic for the `constructorAttributes` method, which is located in `helper.ext`.

7. Build `fornax-cartridges-sculptor-parent` with `mvn clean install`.

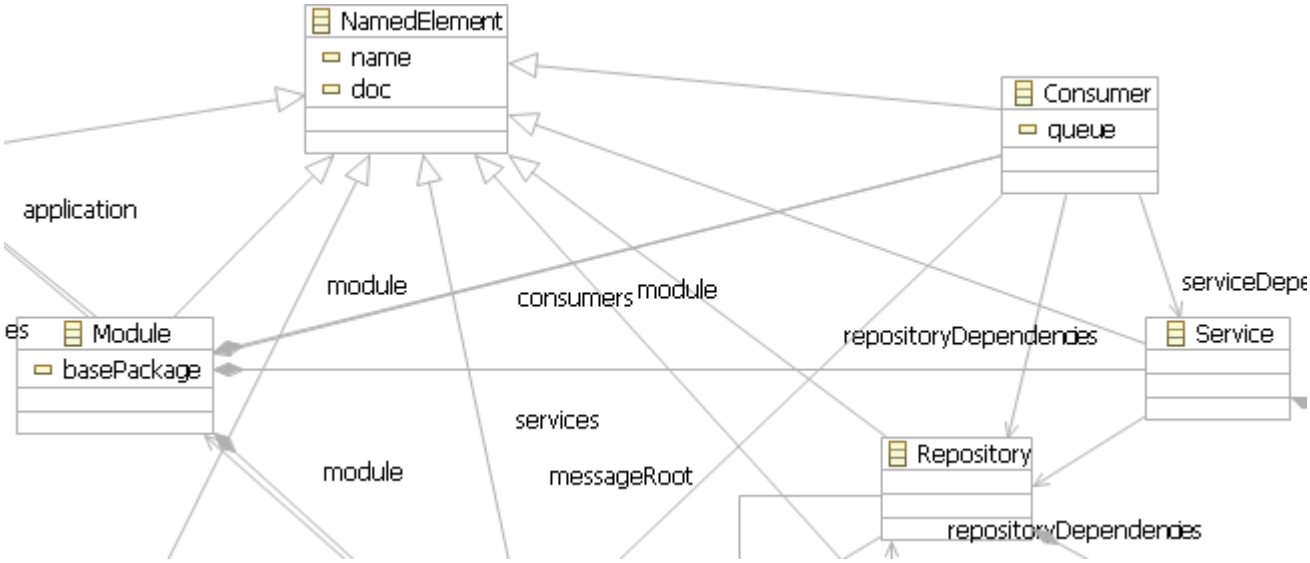
8. Install the DSL editor plugins and test the new feature with the reference application.

How to Add a New Core Concept

This section describes all steps of how to add a completely new concept, Consumer in this case. Consumer is implemented in Sculptor and since this description is very brief you have to look at the details in the source code to make any sense out of this.

1. Start with the Ecore meta model. Open the `sculptormetamodel.ecore_diagram` with the graphical editor.

- Add `Consumer` class.
- Add generalization link from `Consumer` to `NamedElement`, a Consumer has a name.
- Add bi-directional aggregate association between `Module` and `Consumer`, one Module can contain many Consumers, a Consumer belongs to one Module.
- Add association from `Consumer` to `Service`, `serviceDependencies`, used for dependency injection of Services into the Consumer.
- Add association from `Consumer` to `Repository`, `repositoryDependencies`, used for dependency injection of Repositories into the Consumer.



2. DSL grammar, located in `sculptordsl.xtxt`

- Define `DslConsumer`

```
DslConsumer :
    (doc=STRING)?
    "Consumer" name=ID "{"
      (dependencies+=DslDependency)*
      ("unmarshall" "to" (("@")?messageRoot=ID))?
      ("queueName" "=" queue=DslQueueIdentifier )?
    "}";
```

- Add `(consumers+=DslConsumer)*` to `DslModule`
- Add `DslDependency` in the same way as the existing DslRepositoryDependency
- Run `GenerateSculptordsl.mwe` in `org.fornax.cartridges.sculptor.dsl` to generate DSL meta model and DSL editor

3. DSL constraints checks, located in `SculptordslJavaValidator.java` and `SculptordslChecks.chk`.

- Add `DslServiceDependency` in the same way as the existing `DslRepositoryDependency`, `findService` is already implemented in `sculptordsl.ext`

4. Transformation of DSL model to Ecore meta model, located in `DslTransformation.ext` in `fornax-cartridges-sculptor-generator`.

- Add `consumers.addAll(module.consumers.transform()) ->` to the `Module` transformation.
- Add `create sculptormetamodel::Consumer this transform(DslConsumer consumer)...`
- Add `module.consumers.transformDependencies() ->` to the `Module` transformation.
- Add `transformDependencies(DslConsumer consumer)...`
- Add `sculptormetamodel::Service transformServiceDependency(DslDependency dependency)...`

5. Meta model constraints, located in `constraints.chk` in `fornax-cartridges-sculptor-generator`.

- Add `context Consumer ERROR "Not allowed to delegate to repository...`
- Add check of cyclic dependencies in `DependencyConstraints` class

6. Code generation template

- Add `Consumer.xpt` and everything to generate for the Consumers.
- Add properties for package and framework classes. This is done in `default-sculptor-generator.properties`, `properties.ext` and `helper.ext`.
- Add `EXPAND Consumer` in `Root.xpt`, you need `allConsumers()` in `helper.ext`, which can be implemented in the same way as `allServices`.
- Add Spring stuff for the Consumers in `Spring.xpt`.