This page last changed on Mar 13, 2011 by patrik_nordwall.

Sculptor Advanced Tutorial

This tutorial describes the features of Sculptor business tier. It presents how Sculptor works out-of-the-box, customization is often needed and that is the topic of the Developer's Guide.

Before you start you must follow the instructions in the Installation Guide. It is also recommended that you try the Hello World Tutorial to get a feeling of the environment.

Table of Contents:

Library Example

The example used in this tutorial is a simple system for a library of movies and books. It is "over designed" compared with the simple functionality it provides. The reason for this is to be able to illustrate many code generation ideas.

The core of the system is a Domain Model, see Figure 1. A Library consists of PhysicalMedia. Books and Movies are different types of Media, which are stored on a PhysicalMedia, e.g. DVD, VHS, paper books, eBooks on CD. A Media has Characters, e.g. James Bond, which can be played by a Person, e.g. Pierce Brosnan. A person can be involved (Engagement) in different Media, actually a Person can have several Engagements in the same Media. E.g. Quentin Tarantino is both actor and director in the movie 'Reservoir Dogs'.

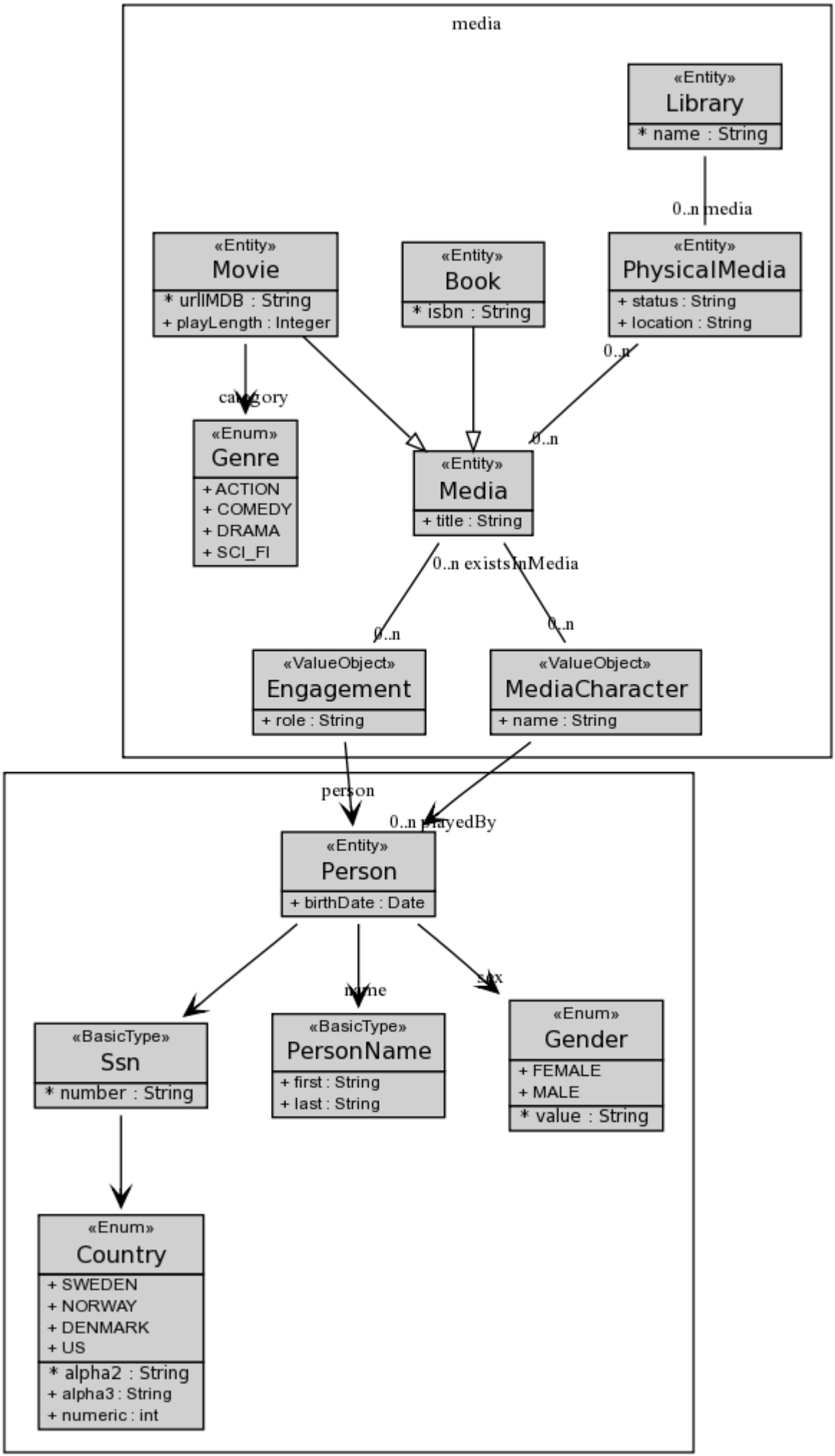
media
«Entity»
Library
* name : String
0..n
media
«Entity»
Movie
* urlIMDB : String
+ playLength : Integer
«Entity»
Book
* isbn : String
«Entity»
PhysicalMedia
+ status : String
+ location : String
0..n
category
«Enum»
Genre
+ ACTION
+ COMEDY
+ DRAMA
+ SCI_FI
0..n
«Entity»
Media
+ title : String
0..n existsInMedia
0..n
0..n
«ValueObject»
Engagement
+ role : String
«ValueObject»
MediaCharacter
+ name : String
person
0..n playedBy
«Entity»
Person
+ birthDate : Date
name
sex
«BasicType»
Ssn
* number : String
«BasicType»
PersonName
+ first : String
+ last : String
«Enum»
Gender
+ FEMALE
+ MALE
* value : String
«Enum»
Country
+ SWEDEN
+ NORWAY
+ DENMARK
+ US
* alpha2 : String
+ alpha3 : String
+ numeric : int

Figure 1. Domain model of the Library example. This diagram is generated by Sculptor

Setup Project

In this part we will setup the project structure for maven and eclipse.

try it!

1. Use the following command (one line) to create a maven pom and file structure.

mvn archetype:generate -DarchetypeGroupId=org.fornax.cartridges -DarchetypeArtifactId=fornax-cartridges-sculptor-archetype-standalone -DarchetypeVersion=2.0.0 -DarchetypeRepository=http://fornax-platform.org/nexus/content/repositories/public

Fill in groupId and archetypeId:

```
Define value for groupId: : org.library
Define value for artifactId: : library
Define value for version:  1.0-SNAPSHOT: :
Define value for package:  org.library: :
```

2. In the new directory, run `mvn eclipse:eclipse` to create an Eclipse project with the same dependencies as in the pom.

3. Open Eclipse and import the project.

Generate Code

In this part we will write a Sculptor DSL file and generate code from it.

try it!

1. Modify the file named `model.btdesign` in the folder `src/main/resources/model`

2. Open the design file with Scupltor DSL editor, double-click on it.
Add the following code to the design file. You can see that it defines two Modules, containing one Service each. The operations in the Services delegates directly to the Repositories. It defines the same Domain Objects, including attributes and references, as in Figure 1. The details will be explained further on.

```
Application Library {
   basePackage = org.library

   Module media {

      Service LibraryService {
        findLibraryByName => LibraryRepository.findLibraryByName;
        findMediaByName => MediaRepository.findMediaByName;
        findMediaByCharacter => MediaRepository.findMediaByCharacter;
        findPersonByName => PersonService.findPersonByName;
      }

      Entity Library {
```

```
  scaffold
  String name key
  - Set<@PhysicalMedia> media <-> library

  Repository LibraryRepository {
    findByQuery;
    @Library findLibraryByName(String name) throws LibraryNotFoundException;
  }
}

Entity PhysicalMedia {
  scaffold
  String status length="3"
  String location
  - @Library library nullable <-> media
  - Set<@Media> media <-> physicalMedia
}

Service MediaService {
  findAll => MediaRepository.findAll;
}

abstract Entity Media {
  gap
  String title !changeable
  - Set<@PhysicalMedia> physicalMedia inverse <-> media
  - Set<@Engagement> engagements cascade="all-delete-orphan" <-> media
  - Set<@MediaCharacter> mediaCharacters <-> existsInMedia

  Repository MediaRepository {
    > @MediaCharacterRepository
    int getNumberOfMovies(Long libraryId) => AccessObject;
    List<@Media> findMediaByCharacter(Long libraryId, String characterName);
    findById;
    save;
    findAll;
    findByQuery;
    protected findByKeys(Set<String> keys, String keyPropertyName, Class persistentClass);
    List<@Media> findMediaByName(Long libraryId, String name);
    Map<String, @Movie> findMovieByUrlIMDB(Set<String> keys);
  }
}

Entity Book extends @Media {
  !auditable
  String isbn key length="20"
}

Entity Movie extends @Media {
  !auditable
  String urlIMDB key
  Integer playLength
  - @Genre category nullable
}

enum Genre {
    ACTION,
    COMEDY,
    DRAMA,
    SCI_FI
}

ValueObject Engagement {
  String role
  - @Person person
  - @Media media <-> engagements
```

```
    }

    Service MediaCharacterService {
      findAll => MediaCharacterRepository.findAll;
    }

    ValueObject MediaCharacter {
      String name !changeable
      - Set<@Person> playedBy
      - Set<@Media> existsInMedia <-> mediaCharacters

      Repository MediaCharacterRepository {
        findByQuery;
        findAll;
      }
    }
}


Module person {
    Service PersonService {
      findPersonByName => PersonRepository.findPersonByName;
    }

    Entity Person {
      gap
      scaffold
      Date birthDate past
      - @Gender sex !changeable
      - @Ssn ssn key
      - @PersonName name

      Repository PersonRepository {
          List<@Person> findPersonByName(String name) => AccessObject;
          save;
          save(Collection<@Person> entities);
          findByQuery;
          findByExample;
          findByKeys;
      }
    }

    BasicType Ssn {
      String number key length="20"
      - @Country country key
    }

    BasicType PersonName {
      String first
      String last
    }

    enum Gender {
        FEMALE("F"),
        MALE("M")
    }

    enum Country {
        String alpha2 key
        String alpha3
        int numeric
        SWEDEN("SE", "SWE", "752"),
        NORWAY("NO", "NOR", "578"),
        DENMARK("DK", "DNK", "208"),
        US("US", "USA", "840")
    }
```

```
    }

}
```

3. Run `mvn clean install` to generate code and build. The JUnit test will fail.

Domain-Driven Design

Now it is time to explain the fundamental Domain-Driven Design concepts in the DSL. If you don't have the book you can download and read more in DDD Quickly. It is highly recommended that you study one of these books, at least the concepts mentioned in this part of the tutorial. The below quotes are from DDD Quickly.

Domain Objects

In the DSL there are three types of Domain Objects.

Entity

Entities have an identity and the state of the object may change during the lifecycle.

> "There is a category of objects which seem to have an identity, which remains the same throughout the states of the software. For these objects it is not the attributes which matter, but a thread of continuity and identity, which spans the life of a system and can extend beyond it. Such objects are called Entities."

ValueObject

For Value Objects the values of the attributes are interesting, and not which object it is. Value Objects are typically immutable.

> "There are cases when we need to contain some attributes of a domain element. We are not interested in which object it is, but what attributes it has. An object that is used to describe certain aspects of a domain, and which does not have identity, is named Value Object."

Note that Value Object is not the same as Data Transfer Object. Core J2EE patterns caused a lot of confusion when they used the term Value Object. They have renamed it to Transfer Object.

BasicType

BasicType is typically used for fundamental types, for example different quantities such as Money, or range of values.

BasicType is a ValueObject which is stored in the same table as the Domain Object referencing it. It corresponds to JPA @Embeddable.

Aggregate

> "An Aggregate is a group of associated objects which are considered as one unit with regard to data changes. The Aggregate is demarcated by a boundary which separates the objects inside from those outside. Each Aggregate has one root. The root is an Entity, and it is the only object accessible from outside. The root can hold references to any of the aggregate objects, and the other objects can hold references to each other, but an outside object can hold references only to the root object."

With Sculptor each Entity is by default an aggregate root, but you can use `!aggregateRoot` to define that it is not. Sculptor will validate the reference constraints described in the quote above.

Repository

A repository is used for:

- retrieving domain objects from the underlying database
- persisting new domain objects
- deleting domain objects

The interface of the Repository should always speak the Ubiquitous Language of the domain. It provides domain centric operations to the client. Repositories provide controlled access to the underlying data in the sense that it exposes only the Aggregate roots of the domain model.

> "Databases are part of the infrastructure. A poor solution is for the client to be aware of the details needed to access a database.
> ...
> Therefore, use a Repository, the purpose of which is to encapsulate all the logic needed to obtain object references. The domain objects won't have to deal with the infrastructure to get the needed references to other objects of the domain. They will just get them from the Repository and the model is regaining its clarity and focus."

You might feel confused regarding the difference between DAOs and repositories. The DAO is at a lower level of abstraction than the Repository and can contain plumbing codes to pull out data from the database. We typically have one DAO per database table, but one repository per domain type or aggregate.

I can recommend that you read Inject Repositories, not DAOs in Domain Entities.

Service

The Services act as a Service Layer around the domain model. It provides a well defined interface with a set of available operations to the clients.

The transaction boundary is at the service layer.

Module

> "Modules are used as a method of organizing related concepts and tasks in order to reduce complexity.
> ...
> Another reason for using modules is related to code quality. It is widely accepted that software code should have a high level of cohesion and a low level of coupling."

With Sculptor the Modules are realized as Java packages. By default they are subpackages of the application package, but it is possible to define another package by specifying the `basePackage` attribute of the Module.

Circular references between Modules are not allowed. Interaction between a Service in one Module and a Repository in another Module is not allowed, it must go via a Service. Sculptor will validate these constraints.

How to Generate Domain Objects

In the context of Sculptor, Domain Object is a common term for Entity, ValueObject and BasicType.

Gap Class

Separation of generated and manually written code is done by a generated base class and manually written subclass, a gap class. It is in the subclass you add methods to implement the behavior of the Domain Object. The subclass is also generated, but only once, it will never be overwritten by the generator. You can of course remove it to regenerate it.

The gap class is not generated initially. When you need a gap class you specify that in the DSL with `gap`.

```
Entity Person {
   gap
   Date birthDate
   - @Gender sex !changeable
   - @Ssn ssn key
   - @PersonName name
}
```

Note that in the gap class some annotations are specified, e.g. @Entity and @Table. Those are only generated once, and must be maintained manually. E.g. when changing the natural key attributes the @UniqueConstraint must be modified. In the JavaDoc of the generated base class the correct annotations are defined, for convenient copy to the hand written subclass.

It is possible to configure that gap classes are always to be generated, except when specified otherwise. Then you add the following property in `sculptor-generator.properties`:

```
generate.gapClass=true
```

In the DSL you can specify that a gap class is not needed:

```
BasicType PersonName {
   nogap
   String first
   String last
}
```

Operations

Domain Objects can of course contain behavior, otherwise it wouldn't be a rich domain model. However, the behavior logic is always written manually in the gap class or in a trait. You can write the methods directly in the java gap class. Optionally you can define the signature of important methods in `model.btdesign`. You still have to implement it manually in the gap class. The advantage of defining the operation in the model is that then it is included in generated documentation and diagrams.

Operations are defined with a java like syntax. To mark it as an operation you must use `def` or `*` in front of it. It must end with `;`.

```
Entity Person {
   Date birthDate
   def int age;
   - @PersonName name
   def changeName(String first, String last);
   - Set<@Person> children
   def List<@Person> getChildrenSortedByAge;
```

```
}
```

Traits

Traits provide a mixin composition mechanism that is missing in Java. Similar to interfaces in Java, traits are used to define object types by specifying the signature of the supported methods. Unlike interfaces, traits can be partially implemented; i.e. it is possible to define implementations for some methods. Sculptor has support for traits, inspired by Scala traits, but with some limitations compared to Scala traits.

Let us say that the Library system also should be used to sell products.

```
Entity Book extends Media with Product {
    String isbn
    int pages
}

Trait Product {
    String title
    def int price();
    def abstract int priceFactor();
}
```

This means that you can implement the logic for the price operation in one place, in the trait class. It will be available in classes that mixin the trait. `Book with Product`. The internal implementation is based on delegation, no magic.

Attributes and references in the trait will be mixed in to the domain object, i.e. stored together with the Book entity.

`abstract priceFactor` in the example above means that that operation has to be implemented in the domain object containing the trait, i.e. Book gap class.

Attributes, references and operations can be defined as protected if you for example don't want to expose them in the interface of the trait.

It is possible to mixin several traits.

```
Entity Person with Worker with Student {
}

Trait Worker {
    String company
    def boolean busy(Date day);
}

Trait Student {
    String school
    def boolean busy(Date day);
}
```

In the case the same operation, attribute or reference is defined in several of the traits the last one will dominate. In this example, if you invoke the `busy` method on a Person instance it will end up in the Student implementation. There is no support for stackable traits, i.e. to be able to call super from one trait and that will invoke next trait. In the above example that would have been useful, since you would probably want to check the schedule for both Student and Worker.

There is a special `self()` method that you can use if you from the trait need to get hold of the instance containing the trait.

Another example of traits is illustrated in this article.

Attributes and References

Domain Objects can have simple attributes and references to other Domain Objects and enums.

Primitive types or any fully qualified Java class can be used as type of the attributes. Built in types:

- String
- Integer (int), Long (long), Float (float), Double (double)
- BigDecimal
- Boolean (boolean)
- Date
- DateTime, Timestamp
- Clob
- Blob

It is easy to add your own DSL types and mapping to database and Java types. See Developer's Guide. Sculptor supports Joda Time instead of the Java date and time classes. It is also described in the Developer's Guide how to activate Joda Time.

To distinguish references from simple attributes, declarations of references starts with a `-`. In the same way as in other places you must also use an @ in front of the declaration when referring to a Domain Object. When the relation is one-to-many or many-to-many you define a collection as the type of the reference. Bidirectional associations are defined with the opposite `<->` syntax.

```
Entity PhysicalMedia {
   String status
   String location
   - @Library library <-> media
   - Set<@Media> media <-> physicalMedia
}
```

There is an alternative notation for references and bidirectional. Instead of `-` you can use `reference` and instead of `<->` you can use `opposite`.

Collections

Supported collection types:

- Set - unordered collection
- List - ordered collection using index column
- Bag - ordered collection using orderby sorting

For Bag collections you can specify `orderby`

```
ValueObject MediaCharacter {
   String name !changeable
   - Bag<@Person> playedBy orderby="birthDate asc"
   - Set<@Media> existsInMedia <-> characters
}
```

For bi-directional many-to-many associations it is possible to define the JPA/Hibernate inverse side.

```
abstract Entity Media {
   String title !changeable
   - Set<@PhysicalMedia> physicalMedia inverse <-> media
```

Associations with cardinality "many" (Set, Bag, List) that are not bidirectional are by default generated as many-to-many, with a separate relation table.

```
Entity Library {
   String name key
   - Set<@PhysicalMedia> physicalMedia
}
```

By defining the reference as `inverse` it will be generated as an ordinary foreign key in the child table, i.e. an one-to-many relation.

```
Entity Library {
   String name key
   - Set<@PhysicalMedia> physicalMedia inverse
}
```

Key

`equals` and `hashCode` requires some thought when used with JPA/Hibernate, see the discussion at the Hibernate site. Sculptor takes care of the details. You only have to mark the attributes that is the natural key of the Domain Object, or if there is no natural key Sculptor will generate a UUID automatically.

If the key is a composite key, consisting of several attributes, a separate internal key class will be generated.

It is also possible to use a separate BasicType object as key. This key object may have one or several attributes marked as key. Some keys are important objects in a domain model and may contain logic. It is also more clarifying to pass around a real type, instead of a String or Integer, e.g. AccountNumber.

```
Entity Person {
  Integer age
  - @Ssn ssn key
  - @PersonName name
}

BasicType Ssn {
  String number key
  String country key
}
```

All persistent Entities and Value Objects will also have a surrogate `id` attribute, which is the primary key in the database.

Changeable

Attributes and References can be defined as changeable or not. By default Entities are mutable, and ValueObjects and BasicType immutable. ValueObjects and BasicTypes can be defined as `!immutable` to

be changeable. Individual Attributes and References can be declared as `!changeable`. Natural keys are never changeable.

```
ValueObject ChangeableColor {
    !immutable
    String name !changeable;
    int red;
    int green;
    int blue;
}
```

Attributes and References that are defined as `!changeable` are included in the constructor and have no setter method.

There is an alternative notation for `!`. Instead of `!immutable` you can use `not immutable`.

Required

As a complement to `changeable` you can use `required` for Attributes and References that are to be included in the constructor, but still have a setter method.

```
ValueObject ChangeableColor {
    !immutable
    String name !changeable;
    int red required;
    int green required;
    int blue required;
}
```

Nullable

An Attribute can be defined as `nullable` which means that it may have null as value when stored in database, i.e. no `NOT NULL` declaration for the database column. By default Attributes are `not nullable`.

Copy mutator

It may feel inconvenient to use constructors with many parameters for immutable ValueObjects. Therefore copy mutator methods with a fluent interface are generated in immutable ValueObjects.

```
ValueObject Address {
    String street
    String streetNumber nullable
    String city nullable
    String zipCode nullable
    - @Country country nullable
}
```

The above Address may be constructed like this:

```
new Address("Drottninggatan")
    .withStreetNumber("17")
    .withCity("Stockholm")
    .withZipCode("10101")
```

```
        .withCountry(Country.SWEDEN);
```

Validation

Sculptor supports bean validation by adding Hibernate Validator annotations to the Domain Objects.

To validate a Domain Object add validation constraints to the model. You can add constraints to Entities, Attributes and References.

```
Entity Person {
   Date birthDate past
   String ssn key length="15"
   String country key length="2" pattern="'[DE|SE|US]'"
   Integer age nullable min="18,'must be an adult'"
   - Set<@Address> addresses notEmpty size="min=1,message='at least 1 address is needed'"
}
```

All built-in Hibernate validation constraints are supported by keywords. Each keyword can be used in several ways, e.g.

```
range="1"
range="1,10"
range="1,10,'must be between 1 - 10'"
```

or qualified like

```
range="min=1"
range="min='1',max=10"
range="min='1',max=10,message='must be between 1 - 10'"
```

If you want to omit a parameter you have to use the qualified constraint like

```
range="max=10"
```

It's also possible to define custom validation annotations. Create your own constraints and add it to the model via the validate keyword.

```
Entity Person {
   Date birthDate past validate="@customDateValidation(message='date violation')"
}
```

You can use fully qualified constraints or shortcuts. If want to use a shortcut add it to your sculptor-generator.properties, e.g.

```
validation.annotation.CustomDateValidation=foo.bar.CustomDateValidation
```

Of course combinations of built-in and custom constraints and multiple custom constraints are possible.

Sculptor will automatically validate Domain Objects on every save operation (create or update). In case of a constraint violation a ValidationException is thrown, which gives you detailed information about the cause. If you need more control over the validation process, a validation by hand is supported by using the frameworks DomainObjectValidator class.

It is also possible to define validation at the class level, which is useful for validation of related properties, associations, or aggregate of objects.

```
Entity Person {
    validate="@foo.bar.CustomValidation"
    String name
    Date birthDate
}
```

An alternative way of implementing class level validation is to trigger your own validation method using JPA @PrePersist and @PreUpdate annotations.

```
    @PreUpdate
    @PrePersist
    public void validatePlayLength() {
        if (getPlayLength() != null && Genre.SHORT.equals(getCategory()) && getPlayLength() > 15) {
            throw new ValidationException(
                    "Short movies should be less than 15 minutes");
        }
    }
```

Auditable

Entities are by default auditable, which means that when the objects are saved an interceptor will automatically update properties 'lastUpdated', 'lastUpdatedBy', 'createdDate' and 'createdBy'. These attributes are automatically added for auditable Domain Objects. You can turn off auditing for an Entity with `!auditable`.

```
Entity Book extends Media {
    !auditable
    String isbn key
}
```

Optimistic Locking

By default a `version` attribute is automatically added to each Entity and mutable persistent ValueObject. This is used for optimistic locking checks by Hibernate.
You can skip this feature by specifying `!optimisticLocking` for the Domain Object.

```
Entity Book extends Media {
    !optimisticLocking
    !auditable
    String isbn key
}
```

Aggregate

By default all Entities are considered to be aggregate roots, but you can use `!aggregateRoot` to specify that the Entity is part of an aggregate and not the root of it. The default values for `cascade` and `fetch` features take the aggregate into account.

Read more about the example below in the Domain-Driven Design book, page 134.

```
Entity PurchaseOrder {
   - @Money approvedLimit
   - List<@PurchaseOrderLineItem> items
}

Entity PurchaseOrderLineItem {
   !aggregateRoot
   Integer quantity
   - @Money price
   - @Part part
}

Entity Part {
   - @Money price
}
```

Basic Type

A `BasicType` corresponds to JPA @Embeddable / Hibernate Component, i.e. the properties are stored in the same table as the parent. It makes it easy to create object types.

try it!

In the Library example `PersonName` is an example of a BasicType. Look at annotations of the PersonName class.

A classical example is Money.

```
BasicType Money {
   String currency;
   BigDecimal amount;
}

Entity Account {
   String accountNumber key;
   - @Money balance;
}

ValueObject Transaction {
   DateTime timePoint;
   - @Money money;
   - @Account debitAccount;
   - @Account creditAccount;
}
```

Non-persistent ValueObject

It is possible to specify that a ValueObject is not persistent, i.e. not stored in database. This is for example useful for some parameters and return values for service operations. It can also be used for domain objects containing only logic, without any persistent state. E.g. an algorithm you want to make explicit.

```
Service PersonService {
```

```
    Set<@Person> findPersonsMatching(@PersonCriteria personCriteria);
}

ValueObject PersonCriteria {
    !persistent
    String name
    String country
    - @AgeInterval ageBetween
}

BasicType AgeInterval {
    Integer minAge
    Integer maxAge
}
```

Enum

Enums are defined like this:

```
enum Genre {
    ACTION,
    COMEDY,
    DRAMA,
    SCI_FI
}

Entity Movie extends @Media {
  !auditable
  String urlIMDB key
  Integer playLength
  - @Genre category nullable
}
```

An enum can have attributes. One attribute may be marked as `key` and that value will be stored in the database as identifier of the enum value. Otherwise the name of the enum values are stored.

```
enum Gender {
    String code key
    FEMALE("F"),
    MALE("M")
}
```

It is possible to define an implicit value without defining an attribute. This attribute is automatically named `'value'` and can be of String or int type. This attribute will be stored in database as identifier for the enum.

```
enum Gender {
    FEMALE("F"),
    MALE("M")
}
```

An example with several attributes:

```
enum WindSpeed {
   int beaufortNumber key;
   double minMps;
   double maxMps;

   CALM(1, "0", "0.2"),
   STORM(10, "24.5", "28.4"),
   HURRICAN(12, "32.7", "40.8");
}
```

Inheritance

An Entity may extend another Entity. A ValueObject may extend another ValueObject. Inheritance is not supported for BasicType. By default JOINED inheritance strategy is used, i.e. fields that are specific to a subclass are mapped to a separate table than the fields that are common to the parent class.

```
abstract Entity Project {
   String name key
}

Entity LargeProject extends @Project {
   BigDecimal budget nullable
}
```

You can also use SINGLE_TABLE strategy, i.e. a single table per class hierarchy.

```
abstract Entity Project {
   inheritanceType=SINGLE_TABLE
   String name key
}

Entity LargeProject extends @Project {
   BigDecimal budget nullable
}
```

When using SINGLE_TABLE you can also define several things to customize the mapping, similar to JPA.

```
abstract Entity Project {
   inheritanceType=SINGLE_TABLE
   discriminatorType=CHAR
   discriminatorColumn="TYPE"
   String name key
}

Entity LargeProject extends @Project {
   discriminatorValue="L"
   BigDecimal budget nullable
}
```

Package

By default the Domain Objects are located in a package named `domain`, which is a sub-package to the package of the module. However, it is possible to specify another package name for an individual Domain Object.

```
ValueObject PersonCriteria {
   package=param
   !persistent
   String name
   String country
   - @AgeInterval ageBetween
}
```

Cache

Domain Objects and collection references can be defined to be cached by JPA/Hibernate second level cache. EhCache is the default cache provider. It is described in the Developer's Guide how to change to another cache provider.

```
ValueObject Engagement {
  cache
  String role
  - @Person person
  - @Media media <-> engagements
}

abstract Entity Media {
  String title !changeable
  - Set<@PhysicalMedia> physicalMedia <-> media
  - Set<@Engagement> engagements cache <-> media
  - Set<@MediaCharacter> characters <-> existsInMedia
}
```

Database Definitions

The names of database tables and columns are normally derived from the names of the domain objects, but it is possible to specify other names by using `databaseTable` and `databaseColumn` attributes.

```
Entity PhysicalMedia {
   databaseTable="PHMED"
   String status databaseColumn="STAT"
   String location databaseColumn="LOC"
   - @Library library databaseColumn="LIB" <-> media
   - Set<@Media> media databaseColumn="MED" <-> physicalMedia
}
```

It is possible to define many-to-many join table with `databaseJoinTable` and its columns with `databaseColumn` at both sides of the bidirectional association:

```
    - Set<@Media> existsInMedia databaseJoinTable="MED_CHR" databaseColumn="CHR" <->
mediaCharacters
```

Those keywords are also useful for unidirectional to-many associations. Additionally `databaseJoinColumn` is used, since there is no opposite side to define the column on.

```
    - Set<@Person> playedBy databaseJoinTable="CHR_PERS" databaseColumn="PERS"
databaseJoinColumn="CHR"
```

For Attributes you can define `nullable`, `index`, `databaseColumn`, `databaseType` and `length`, which all relates to the database definition.

```
Entity Person {
    String ssn key length="15"
    String country key databaseType="CHAR" length="2"
    Integer age nullable
    - @PersonName name
}

BasicType PersonName {
    String first index
    String last index
}
```

Fetch

For References you can define `fetch` which corresponds to JPA/Hibernate feature.

```
ValueObject MediaCharacter {
    String name !changeable
    - List<@Person> playedBy fetch="eager"
    - Set<@Media> existsInMedia <-> characters
}
```

Possible values for fetch:

fetch=	Generated feature
eager	javax.persistence.FetchType.EAGER
join	javax.persistence.FetchType.EAGER
lazy	javax.persistence.FetchType.LAZY
subselect	org.hibernate.annotations.FetchMode.SUBSELECT

Cascade

For References you can define `cascade` which corresponds to JPA/Hibernate feature.

```
ValueObject MediaCharacter {
   String name !changeable
   - List<@Person> playedBy cascade="persist,merge"
   - Set<@Media> existsInMedia <-> characters
}
```

Possible values for cascade:

cascade=	Implementation type
persist	javax.persistence.CascadeType.PERSIST
merge	javax.persistence.CascadeType.MERGE
remove	javax.persistence.CascadeType.REMOVE
refresh	javax.persistence.CascadeType.REFRESH
all	javax.persistence.CascadeType.ALL
all-delete-orphan	javax.persistence.CascadeType.ALL, org.hibernate.annotations.CascadeType.DELETE_ORPHAN
delete-orphan	org.hibernate.annotations.CascadeType.DELETE_ORPHAN
delete	org.hibernate.annotations.CascadeType.DELETE
save-update	org.hibernate.annotations.CascadeType.SAVE_UPDATE
evict	org.hibernate.annotations.CascadeType.EVICT
replicate	org.hibernate.annotations.CascadeType.REPLICATE
lock	org.hibernate.annotations.CascadeType.LOCK

Several cascade types can be defined by separating them with comma, e.g. `cascade="persist,merge"`.

The default value for `cascade` and `fetch` takes the aggregate and module into account. When cascade is not explicitly defined in the DSL the convention is:

- `all-delete-orphan` is used if both Entities are in the same Aggregate.
- `all` is used if both Entities are in the same Module.

Default values for `cascade` can be defined in `sculptor-generator.properties`, see Developer's Guide.

try it!

Try the different features of the Domain Objects. Add a few more Entities and Value Objects to `model.btdesign`. Add different types of Attributes and References. For example you can add a new Module named customer, with a `Customer` entity, which has a Reference to rented `Media`. Maybe with a rental contract Value Object in between, containing time period and price.

Regenerate with `mvn generate-sources -Dfornax.generator.force.execution=true` and look at the generated Java code and JPA annotations.

Documentation of domain model

Sculptor generates summary documentation of domain model in clickable HTML format. The output is placed in src/generated/resources/DomainModelDoc*.html

The descriptions are defined in `model.btdesign`. Almost all elements in the model can have documentation. It is a simple quoted string before the element.

```
"Book is one of the products that..."
Entity Book extends @Media {
  "International Standard Book Number"
  String isbn key length="20"
}
```

The documentation is also included as JavaDoc of generated Java code.

Diagram of domain model

Sculptor generates several UML diagrams for the domain model. The above diagram for the Library domain is generated.
We are using Graphviz.
Sculptor generates textual Graphviz `.dot` files, which is then used to generate images.

There is a maven plugin fornax-graphviz-m2-plugin that generates images (.png) from the .dot files. This plugin is included in the pom.xml created by the archetypes, but you need to install Graphviz and have dot executable in path.

It is possible to mark some domain objects with `hint="umlgraph=core"` to generate a special diagram with those domain objects in focus.

```
Entity Book extends @Media {
  hint="umlgraph=core"
  String isbn key
}

Entity Movie extends @Media {
  hint="umlgraph=core"
  String urlIMDB key
  Integer playLength
  - @Genre category nullable
}
```

The colours are customizable as described in Developer's Guide

How to Generate Services

In the DSL an operation of a Service almost looks like an ordinary Java method with return type, parameters and throws declaration. You don't have to declare the visibility, it is public by default.

```
Service LibraryService {
   List<@Movie> findBestMovies(Long libraryId);
}
```

When referring to a Domain Object (Entity or ValueObject) you use an @ in front of the declaration. Primitive types or any fully qualified Java class can also be used as parameters and return types. The same built in types as can be used for the Domain Objects can be used in the service operations.

It is also possible to use Data Transfer Objects as parameters and return types.

```
Service LibraryFacade {
  List<BookDto> topTenBooks;
}

DataTransferObject BookDto {
    String title
    String isbn
    int rating
}
```

Collections use the ordinary Java generics syntax. Built in collection types:

- Set
- List
- Map
- Collection

You can easily delegate to an operation in a Repository or another Service. Then you only have to declare the name of the operation. Return type and parameters are "copied" from the delegate operation.

```
findLibraryByName => LibraryRepository.findLibraryByName;
```

Sometimes it can be convenient to declare a delegating operation with protected visibility. Such an operation is typically used by another manually coded operation in the Service.

```
@Library findLibraryByName(String name);
protected findLibraryByExample => LibraryRepository.findByExample;
```

There is an alternative notation for delegation. Instead of => you can use `delegates to`.

It is also possible to specify a dependency injection of a Repository or any other Spring bean, without having a generated delegation method.

```
Service LibraryService {
  > @MediaCharacterRepository
  addCharacter(@Person person, @Media media, String role);
}
```

The @ in front of the Repository name indicates that it is an internal reference and the DSL editor will validate that it exists. Skip the @ when you inject other Spring beans, which are not defined in the DSL model.
When you inject other beans you have to implement the setter method yourself in the Service implementation class. You define other Spring beans in `more.xml` or a Spring file imported from there.

There is an alternative notation for dependency injection. Instead of > you can use `inject`.

The generated code consists of:

- Interface, eg. `LibraryService`

- Implementation class, e.g. `LibraryServiceImpl`
- Base class for the implementation, e.g. `LibraryServiceImplBase`
- Test class, e.g. `LibraryServiceTest`
- Base class for the test, e.g. `LibraryServiceTestBase`

Separation of generated and manually written code is done by a generated base class and manually written subclass. The subclass is also generated, but only once, it will never be overwritten by the generator. You can of course remove it to regenerate it. There will not be any subclass, gap class, when the service only consists of operations that delegate to repositories and other other services.

If you have configured to use EJB there will also be Stateless session EJB and Client side proxy for the EJB.

try it!

Try to add one more operation to the PersonService, e.g. `findPersonsByCountry`. Generate with `mvn generate-sources -Dfornax.generator.force.execution=true`. Note that `PersonServiceImpl` is not overwritten and therefore you will get compilation errors for the new method. You have to add it manually in `PersonServiceImpl`. Tips: use ctrl+1 in Eclipse.

How to Generate Repositories

The default implementation of a Repository consists of an implementation class and Access Objects. The intention is a separation of concerns between the domain and the data layer. Repository is close to the business domain and Access Objects are close to the data layer. The JPA/Hibernate specific code is located in the Access Object, and not in the Repository.

Guidelines:

- The interface of the Repository should always speak the Ubiquitous Language of the domain. It provides domain centric operations to the client.
- Repositories provide controlled access to the underlying data in the sense that it exposes only the Aggregate roots of the domain model.
- The Repository operation can typically use several Access Objects or other Repositories to collect the final result.
- The Access Objects are at a lower level of abstraction than the Repository and can contain plumbing code to pull out data from the database. Even though JPA/Hibernate does a great job, there is often technical details of fetching, joining, caching, etc. that is not part of domain layer.

A Repository generated by Sculptor uses Access Objects, which implement the specialization needed for persistence. Access Objects are implemented as Commands and have separated interface and implementation. The JPA/Hibernate specific code is encapsulated in the implementation classes. A Factory is used to create instances of the Access Objects.

It is possible to use another design approach and implement everything directly in the Repository. This is described in the section *Hibernate Repository* in the Developer's Guide.

Generic Access Objects

Sculptor runtime framework provides generic access objects for the following operations:

- findById
- findAll
- findByExample
- findByQuery
- findByCondition
- findByCriteria (use findByCondition instead)
- findByKey
- findByKeys
- save

- delete
- countAll
- populateAssociations

findByExample, findByCondition and findByCriteria are currently only implemented for Hibernate JPA provider. We will implement findByCondition in future release using JPA 2.0 criteria API.

To use a generic Access Object you simply have to specify the name of it in the Repository in the DSL.

```
Repository LibraryRepository {
   findById;
   save;
   delete;
}
```

Note that you can define operations as protected to not expose them in the Repository interface and only use them from other methods with a more domain centric interface.

```
Repository PersonRepository {
   List<@Person> findPersonByName(String name);
   protected findByQuery;
}
```

See the source of the corresponding classes for more information of the functionality of these Access Objects.

Note that setter methods in the interface corresponds to method parameters in the repository operation. For example the findAll operation can be used with parameters for sorting and caching.

```
Repository LibraryRepository {
   findAll(String orderBy, boolean orderByAsc, boolean cache);
}
```

You would often like to define sorting and caching without exposing it as parameters, i.e. always use caching and always sort in a specific way. Therefore this is supported with hints.

```
Repository PersonRepository {
   findAll hint="orderBy=name.last, orderByAsc=false";
   findByKey hint="cache";
}
```

Custom Access Objects

You can also use Access Objects containing manually written code. You specify `=> AccessObject` in the DSL.

```
Repository MediaRepository {
   int getNumberOfMovies(Long libraryId) => AccessObject;
   ...
```

```
}
```

There is an alternative notation for delegation. Instead of => you can use `delegates to`.

Note that you define return type and parameters in the same way as for Service operations. When referring to a Domain Object (Entity or ValueObject) you use a @ in front of the declaration.

By default the Access Object has a similar name as the repository operation, but you can define another name.

```
Repository MediaRepository {
   int getNumberOfMovies(Long libraryId) => MovieCounter;
   ...
}
```

It is in the `performExecute` method in the AccessImpl class you place the hand written code.

```
public class GetNumberOfMoviesAccessImpl extends GetNumberOfMoviesAccessImplBase {
   public void performExecute() {
      DetachedCriteria criteria = DetachedCriteria.forClass(Movie.class);
      criteria.setProjection(Projections.rowCount());
      List result = getHibernateTemplate().findByCriteria(criteria);
      Number count = (Number) result.get(0);
      setResult(count.intValue());
   }
}
```

It is also possible to use, and specialize the generic Access Objects.

```
public class FindPersonByNameAccessImpl extends FindPersonByNameAccessImplBase {
   public void performExecute() {
      // use the generic FindByCriteria, but with a special restriction
      FindByCriteriaAccessImpl<Person> finder =
            new FindByCriteriaAccessImpl<Person>(Person.class) {
         protected void addRestrictions(Criteria criteria) {
            List names = Arrays.asList(getName().split(" "));
            criteria.add(Restrictions.or(
                  Restrictions.in(NAME + "." + FIRST, names),
                  Restrictions.in(NAME + "." + LAST, names)));
         }
      };
      finder.setSessionFactory(getSessionFactory());
      finder.setCache(true);
      finder.setOrderBy(NAME + "." + LAST);
      finder.execute();
      setResult(finder.getResult());
   }
}
```

Repository Operation without Access Object

There is also a third variant of Repository operations, which neither delegates to a generic nor a custom Access Object. It is manually written in the Repository implementation. To avoid automatic delegation you simply don't specify => `AccessObject` in the DSL.

```
Repository LibraryRepository {
   @Library findLibraryByName(String name);
   ...
}
```

It is possible to specify a dependency injection of a another Repository, which can be useful for this variant of Repository operation. It is used in the library example to implement `findMediaByCharacter` in the `MediaRepository`.

```
Repository MediaRepository {
  > @MediaCharacterRepository
  List<@Media> findMediaByCharacter(Long libraryId, String characterName);
  ...
}
```

The @ in front of the Repository name indicates that it is an internal reference and the DSL editor will validate that it exists. Skip the @ when you inject other Spring beans, which are not defined in the DSL model.
When you inject other beans you have to implement the setter method yourself in the Repository implementation class. You define other Spring beans by annotating the classes with `@Repository`, `@Service`, or `@Component`.

There is an alternative notation for dependency injection. Instead of > you can use `inject`.

try it!

Take a look at the Java code for `findLibraryByName` in `LibraryRepositoryImpl`.
As you can see you have to implement it manually, this can be done by something like this:

```
public Library findLibraryByName(String name) {
   Map<String, Object> parameters = new HashMap<String, Object>();
   parameters.put(LibraryNames.NAME, name);

   List<Library> result = findByQuery("Library.findLibraryByName", parameters);

   if (result.isEmpty()) {
      return null;
   } else {
      return result.get(0);
   }
}
```

Scaffold

It is possible to mark a Domain Object with `scaffold` to automatically generate some predefined CRUD operations in the Repository and corresponding Service.

```
Entity Person {
```

```
    scaffold
    String ssn key
    String name
}
```

The above DSL definition would result in the same as the following:

```
Service PersonService {
    findById => PersonRepository.findById;
    findAll => PersonRepository.findAll;
    save => PersonRepository.save;
    delete => PersonRepository.delete;
}

Entity Person {
    String ssn key
    String name

    Repository PersonRepository {
        findById;
        findAll;
        save;
        delete;
    }
}
```

The Repository and Service is also added automatically if they are not defined.

Which scaffolding operations to use can be defined in `sculptor-generator.properties`, see Developer's Guide.

Pagination

Paging of large result sets is supported for findAll, findByQuery, findByCriteria and custom access objects. Usage of paging is defined by adding a PagingParameter as parameter to a repository operation.

```
    findAll; // without paging
    findAll(PagingParameter pagingParameter); // with paging
```

When using a paged operation it looks like this:

```
        // fetch first page, and request for number of pages
        int page = 1;
        int pageSize = 25;
        boolean countTotalPages = true;
        PagingParameter pagingParameter = PagingParameter.pageAccess(pageSize, page, countTotalPages);
        PagedResult<Person> pagedResult = personRepository.findAll(pagingParameter);
        int totalPages = pagedResult.getTotalPages();
        List<Person> values = pagedResult.getValues();

        // later fetch another page, without calculating number of pages
        int page = 2;
        int pageSize = 25;
        PagingParameter pagingParameter = PagingParameter.pageAccess(pageSize, page);
        PagedResult<Person> pagedResult = personRepository.findAll(pagingParameter);
```

```
List<Person> values = pagedResult.getValues();
```

In the initial request you typically ask for number of available pages. To answer this the system must count total number of rows. For findAll there is a built in countAll access object. For findByQuery, findByCriteria custom access objects you must provide a count operation, which can be done in several ways.

For findByQuery the default is to use a naming convention of the named query for the counting query. `find` is replaced with `count`.
findByQuery on named query `"Person.findByCountry"` will use `"Person.countByCountry"`.

For findByCriteria and custom access objects you must define the count operation or named query to use. That is done with hint `countOperation` or `countQuery`. Operation parameters are passed to the count operation or count query. Try the following and look at the generated code. try it!

```
findByQuery(PagingParameter pagingParameter);

myPagedFindOperation(String searchFor, PagingParameter pagingParameter)
   hint="countQuery=Person.myPagedCountQuery"
   delegates to AccessObject;

findByCriteria(PagingParameter pagingParameter)
   hint="countOperation=findByCriteriaCountOperation";
protected int findByCriteriaCountOperation(Map<String, Object> restrictions);
```

Those hints are available for findAll and findByQuery also.

For findAll and findByCondition it is possible to use a config property in sculptor-generator.properties to say that paging should always be used, i.e. not necessary to add the PagingParameter to model.

```
findAll.paging=true
findByCondition.paging=true
```

findByCondition

findByCondition is one of the built in repository operations. It is used like this.

It takes a list of `ConditionalCriteria` objects as parameter. Use the fluent api of `ConditionalCriteriaBuilder` to define the criteria.

```
List<ConditionalCriteria> conditions = ConditionalCriteriaBuilder.criteriaFor(Person.class)
   .withProperty(PersonProperties.primaryAddress().city()).eq("Stockholm")
   .build();
```

Note that for each Domain Object a corresponding Properties class is generated. It contains definitions of all attributes and references of the the Domain Object. It serves two purposes. It is refactoring safe, since you will get compilation errors when something is changed. It provides convenient code completion (ctrl+space) support in your IDE.

You would typically use static imports to make the expression more compact and readable:

```
List<ConditionalCriteria> conditions = criteriaFor(Person.class)
   .withProperty(primaryAddress().city()).eq("Stockholm")
```

```
    .build();
```

The builder supports conditions such as `eq`, `like`, `between`, `lessThan`, `greaterThan`, `in`. The order of the result can be specified with `orderBy`. Use code completion (ctrl+space) to see all available methods. It also supports logical expressions `and`, `or` and `not`, which can be grouped with `lbrace` and `rbrace`.

```
List<ConditionalCriteria> conditions = criteriaFor(Person.class)
    .withProperty(ssn().country()).eq(Country.SWEDEN)
    .and().withProperty(birthDate()).between(new LocalDate("1970-01-01")).to(new LocalDate("1979-12-31"))
    .and().lbrace().withProperty(primaryAddress().city().eq("Stockholm")
        .or().withProperty(primaryAddress().city().eq("Malmö")).rbrace()
    .orderBy(name().last()).orderBy(name().first())
    .build();
```

It is also possible to construct the `ConditionalCriteria` without the builder, using the static factory methods in `ConditionalCriteria`.

```
List<ConditionalCriteria> conditions = new ArrayList<ConditionalCriteria>();
ConditionalCriteria conditionalCriteria =
    ConditionalCriteria.equal(PersonProperties.primaryAddress().city().toString(), "Stockholm");
conditions.add(conditionalCriteria);
```

JUnit

For each Service there is a generated JUnit test class that you have to implement. It extends IsolatedDatabaseTestCase which means that DBUnit it used to load the in memory HSQLDB database with test data from the XML file specified in the `getDataSetFile` method. The database is refreshed for each test method.

Spring beans are injected in the test with ordinary `@Autowired` annotations.

AbstractDbUnitJpaTests also provides a method to retrieve the ServiceContext, which is always passed in as the first parameter of the service methods.

```
List<Person> persons = personService.findPersonByName(
  getServiceContext(), "Skarsgård");
```

You can implement the same kind of JUnit tests for the Repositories.

Above tests are kind of integration tests and you should do ordinary unit tests for domain objects and other classes of importance.

Mocking

Sometimes it is useful to test the Services by stubbing dependencies, e.g. to Repositories.

It is possible to use Mockito or some other mocking framework together with Spring if you do as follows.

As example we have a MessageSender, which is implemented using JMS. The MessageSender implementation is normally injected using `@Autowired` annotation. It is this implementation we want to replace with a mock when testing. We can create the mock instance using the FactoryBean that is included in Sculptor so we only need to add the xml definition in more-test.xml:

```
<bean id="messageSenderMockFactory"
 class="org.fornax.cartridges.sculptor.framework.test.MockitoFactory"
 primary="true" >
   <property name="type" value="org.foo.MessageSender"/>
</bean>
```

In the junit test:

```
public class MyFacadeTest extends AbstractDbUnitJpaTests
   implements MyFacadeTestBase {

   private MessageSender messageSender;
   private MyFacade myFacade;

   @Autowired
   public void setMyFacade(MyFacade myFacade) {
      this.myFacade = myFacade;
   }

   @Autowired
   public void setMessageSender(MessageSender messageSender) {
      this.messageSender = messageSender;
   }

   @Before
   public void initMock() {
      when(messageSender.sendMessage(anyString()))
        .thenReturn(true);
   }


   @Test
   public void testDoSomething() throws Exception {
      int countBefore = countRowsInTable("SOMEDATA");
      myFacade.doSomething("17");
      int countAfter = countRowsInTable("SOMEDATA");

      assertEquals(countBefore + 1, countAfter);

      verify(messageSender).sendMessage(anyString());
   }
}
```

try it!

In the previous section about How to Generate Services you added the operation `findPersonsByCountry` in PersonService. Think about how to implement that method using one of the three different types of Repository operations. It is possible to do it with any of the three variants. Select the alternative you find most attractive and give it a try. Implement a test method also.

You might find the following DBUnit test data useful:

```
<?xml version="1.0" encoding="UTF-8"?>
<dataset>
```

```
  <PERSON ID="1" SSN_NUMBER="123456" SSN_COUNTRY="us" NAME_FIRST="Aaaa" NAME_LAST="Bbbb"
   BIRTHDATE="1963-01-01" SEX="M" VERSION="1"/>
  <PERSON ID="2" SSN_NUMBER="123456" SSN_COUNTRY="se" NAME_FIRST="Xxxx" NAME_LAST="Yyyy"
   BIRTHDATE="1964-01-01" SEX="F" VERSION="1"/>
  <PERSON ID="3" SSN_NUMBER="987654" SSN_COUNTRY="us" NAME_FIRST="Cccc" NAME_LAST="Dddd"
   BIRTHDATE="1965-01-01" SEX="F" VERSION="1"/>
</dataset>
```

Error Handling

Two types of exceptions are used, a checked exception called ApplicationException and a runtime exception called SystemException.

ApplicationException is used for faults that the system is designed to take care of, i.e. recoverable errors at the level of the application, e.g. validation failures.

System exceptions are used to indicate unrecoverable, unexpected errors that are outside the control of the application, e.g. database failures. The current processing can't continue and the transaction is rolled back when they occur.

Effective Java Exceptions is a good article describing similar error handling strategy.

Both ApplicationException and SystemException contain an error code, which is used by the client to translate to appropriate error message. Subclasses to these exceptions defines specific error situations and it is in these subclasses the error codes are defined, i.e. an exception class can define several error codes to indicate different flavours of the fault.

In the throws clause of operations for Services or Repositories you can define a comma separated list of exceptions. It can be fully qualified or unqualified class names. When it is unqualified name an ApplicationException subclass with that name will be generated.

```
Repository LibraryRepository {
   @Library findLibraryByName(String name) throws LibraryNotFoundException;
   protected findByQuery;
 }
```

A Spring advice will catch all exceptions that are thrown from the Services. This advice will log all SystemExceptions to the error log. ApplicationExceptions are logged at debug level. Log4j is used for the logging.

The ServiceContext class is needed to support logging and audit trail functionality through the tiers of an application. A ServiceContext object will typically be sent through all tiers, and represents the context in which a business service is called. It contains information about the user and ids for the session and request.

The first parameter of each method in the Services is a ServiceContext parameter. This is generated automatically. In front of the Services there is an advice, which stores this ServiceContext object in a thread local variable, ServiceContextStore, to make sure that it is available everywhere within that request in the tier. When calling remote methods it must be passed as a method parameter.

It is possible to skip the generation of ServiceContext, see Developer's Guide.

Alternative Notation

There is an alternative notation for some things in the DSL. It is more verbose but maybe easier to understand. You can mix the verbose and compact syntax as you wish.

Compact	Verbose
!	not
=>	delegates to
>	inject
<->	opposite
-	reference

```
Application Library {
   basePackage = org.library

   Module media {

      Service LibraryService {
        findLibraryByName delegates to LibraryRepository.findLibraryByName;
      }

      Entity Library {
        not auditable
        String name key
        reference Set<@PhysicalMedia> media opposite library

        Repository LibraryRepository {
          inject @MediaRepository
          @Library findLibraryByName(String name);
        }
      }
   }
}
```

Divide model into several files

It is possible to split model.btdesign and define one or more Modules in each file.

By using one file per module it is also 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`

Referenced files are imported with a URI syntax starting with `classpath:/` followed by classpath path to the `.btdesign` file to be imported.

```
import "classpath:/model-person.btdesign"

Application Library {
   basePackage = org.library

   Module media {

      // as usual ...
```

```
    }
}
```

```
ApplicationPart PersonPart {

    Module person {

        // as usual ...
    }
}
```

The main file doesn't have to define a Module.

```
import "classpath:/model-media.btdesign"
import "classpath:/model-person.btdesign"

Application Library {
    basePackage = org.library
}
```

Cross project references

In previous section it was described how the model could be separated into several files. It is also possible to define modules in a separate project to be used from other projects.

In that case the imported module must define a `basePackage` and use `ApplicationPart`.

```
ApplicationPart CommonPart {

    Module sharedtypes {
        basePackage=org.foo.common.sharedtypes

        BasicType Money {
            String currency;
            BigDecimal amount;
        }
    }
}
```

You don't want to generate code for the imported module in your project, which is going to use the sharedtypes from the common project. To skip generation of a module you define the module (sharedtypes in this example) in `sculptor-generator.properties`:

```
generate.module.sharedtypes=false
```

You import and use it as normal, i.e.

```
import "classpath:/model-sharedtypes.btdesign"

Application Bank {
```

```
    basePackage = org.bank

    Module accounting {
       Entity Account {
          String accountNumber key;
          - @Money balance;
       }
    }
}
```

The dependency to the classes must be added in `pom.xml`:

```
      <dependency>
         <groupId>org.foo.common</groupId>
         <artifactId>foo-common</artifactId>
         <version>1.0</version>
         <classifier>client</classifier>
      </dependency>
```

Note that if you use classifier `client` it will be dependency to the -client jar file, which was created by maven-jar-plugin with corresponding `client` classifier.

If you use ejb you will use <type>ejb-client</type> instead of classifier.

Source

The complete source code for this tutorial is available in Subversion.

Web Access (read only):
http://fisheye3.cenqua.com/browse/fornax/trunk/cartridges/sculptor/fornax-cartridges-sculptor-examples-library

Anonymous Access (read only):
https://fornax.svn.sourceforge.net/svnroot/fornax/trunk/cartridges/sculptor/fornax-cartridges-sculptor-examples-library

👎 2 3 4 5 6 7 8 9 👍