

Learn **OpenXava** by example

JPA

JUnit

Liferay

Hibernate Validator

PostgreSQL

Javier Paniza

Eclipse

HtmlUnit

LITE

Learn
OpenXava
by example

EDITION 1.1 LITE

© 2011 *Javier Paniza*

About the book

This book is meant to teach you how to develop Java Enterprise applications with OpenXava as well as other Java related technologies, tools and frameworks. Together we will develop step by step a complete application from scratch.

Why a book about OpenXava? There is already plenty of free documentation available, which is very exhaustive and always up to date. However, some members of the OpenXava community kept asking me for a book, so eventually I was convinced that writing one would be a good idea.

The OpenXava documentation describes all syntax and associated semantics of OpenXava precisely. Without a doubt, it's an essential tool, and allows you to learn all the OpenXava secrets. But, if you are a newcomer to the Java Enterprise world, you might need more than reference documentation. You want a lot of code examples as well as a means to learn the other related technologies which are needed to create an advanced application.

In this book you'll learn not only about OpenXava, but JPA, Eclipse, PostgreSQL, JUnit, HtmlUnit, Hibernate Validator framework, Liferay, etc. as well. And even more important: you're going to learn techniques to fulfill the common and advanced requirements you will be facing in typical business applications.

Don't hesitate to contact me if you have any suggestions about the book at javierpaniza@yahoo.com.

A very special thanks to Amitabh Sinha, Franklin J. Alier López de Haro, Hans-Jörg Bauer, Olle Nilsson, Stephan Rusch and Sudharsanan Selvaraj for their proofreading work.

Contents

Chapter 1: Architecture & philosophy	1
1.1 The OpenXava concepts	2
Lightweight Model-Driven Development	2
Business Component	3
1.2 Application architecture	5
Application developer viewpoint	6
User viewpoint	7
Project layout	8
1.3 Flexibility	8
Editors	9
Custom view	9
1.4 Summary	10
Chapter 2: Java Persistence API	11
2.1 JPA Annotations	12
Entity	12
Properties	13
References	15
Embeddable classes	16
Collections	17
2.2 JPA API	19
2.3 Summary	21
Chapter 3: Annotations	22
3.1 Validation	23
Declarative validation	23
Built-in validations	23
Custom validation	25
Learn more about validations	26
3.2 User interface	27
The default user interface	27
The @View annotation	28
Refining member presentation	29
Learn more about the user interface	32
3.3 Other annotations	32
3.4 Summary	33

Chapter 4: Getting started with Eclipse and PostgreSQL	34
4.1 Our goal: A small Invoicing application	35
4.2 Installing PostgreSQL	35
4.3 Creating the project in Eclipse	37
Installing OpenXava	38
Create the project	38
Configuring Tomcat inside Eclipse	40
Creating your first entity	42
4.4 Preparing the database	44
Configuring persistence.xml	44
Update schema	45
4.5 Running the application	47
4.6 Modifying the application	48
4.7 Accessing a database from Eclipse	49
4.8 Summary	51
Chapter 5: Modeling with Java	52
5.1 Basic domain model	53
Reference (ManyToOne) as descriptions list (combo)	53
Stereotypes	55
Embeddable	56
Composite key	58
Calculating default values	60
Regular reference (ManyToOne)	62
Collection of dependent entities (ManyToOne with cascade)	63
5.2 Refining the user interface	65
Default user interface	66
Using @View for defining layout	66
Using @ReferenceView to refine the user interface for reference	67
Refining a collection item entry	68
Refined user interface	70
5.3 Agile development	70
5.4 Summary	73
Chapter 6: Automated testing	74
6.1 JUnit	75
6.2 ModuleTestBase for testing the modules	75
The code for the test	75
Executing the tests from Eclipse	77

6.3 Adding the JDBC driver to the Java Build Path	79
6.4 Creating test data using JPA	80
Using setUp() and tearDown() methods	80
Creating data with JPA	81
Removing data with JPA	83
Filtering data from list mode in a test	83
Using entity instances inside a test	84
6.5 Using existing data for testing	85
6.6 Testing collections	86
Breaking down tests in several methods	86
Asserting default values	88
Data entry	89
Verifying the data	90
6.7 Suite	91
6.8 Summary	92
Chapter 7: Inheritance	94
7.1 Inheriting from a mapped superclass	95
7.2 Entity inheritance	97
New Order entity	97
CommercialDocument as an abstract entity	98
Invoice refactored to use inheritance	99
Creating Order using inheritance	100
Naming convention and inheritance	101
Associating Order with Invoice	101
7.3 View inheritance	102
The extendsView attribute	102
View for Invoice using inheritance	103
View for Order using inheritance	105
Using @ReferenceView and @CollectionView to refine views	106
7.4 Inheritance in JUnit tests	108
Creating an abstract module test	108
Using the abstract module test to create concrete module tests	110
Adding new tests to the extended module test	110
7.5 Summary	112
Chapter 8: Basic business logic	114
8.1 Calculated properties	115
Simple calculated property	115
Using @DefaultValueCalculator	116

Calculated properties depending on a collection	119
Default value from a properties file	121
8.2 JPA callback methods	123
Multiuser safe default value calculation	123
Synchronizing persistent and calculated properties	124
8.3 Database logic (@Formula)	127
8.4 JUnit tests	129
Modifying existing tests	129
Testing default values and calculated properties	130
Testing calculated and persistent synchronized properties / @Formula	132
8.5 Summary	133
Chapter 9: Advanced validation	134
9.1 Validation alternatives	135
Adding delivered property to Order	135
Validating with @EntityValidator	136
Validating with a JPA callback method	138
Validating in the setter	138
Validating with Hibernate Validator	139
Validating on removal with @RemoveValidator	139
Validating on removal with a JPA callback method	141
What's the best way of validating?	141
9.2 Creating your own Hibernate Validator annotation	142
Using a Hibernate Validator from your entity	142
Defining your own ISBN annotation	143
Using Apache Commons Validator to implement the validation logic	143
Call to a REST web service to validate the ISBN	145
Adding attributes to your annotation	147
9.3 JUnit tests	148
Testing validation for adding to a collection	148
Testing validation assigning a reference and validation on removal	149
Testing the custom Hibernate Validator	150
9.4 Summary	151
Chapter 10: Refining the standard behavior	154
10.1 Custom actions	155
Typical controller	155
Refining the controller for a module	157
Writing your own action	158
10.2 Generic actions	161

MapFacade for generic code	161
Changing the default controller for all modules	162
Come back to the model a little	164
Metadata for more generic code	165
Chaining actions	167
Refining the default search action	168
10.3 List mode	172
Filtering tabular data	172
List actions	173
10.4 Reusing actions code	175
Using properties to create reusable actions	175
Custom modules	177
Several tabular data definitions by entity	178
Reusable obsession	178
10.5 JUnit tests	179
Testing the customized delete behavior	179
Testing several modules in the same test method	181
10.6 Summary	183
Chapter 11: Behavior & business logic	186
11.1 Business logic in detail mode	187
Creating an action for custom logic	188
Writing the real business logic in the entity	189
Write less code using Apache Commons BeanUtils	190
Copying a collection from entity to entity	191
Application exceptions	192
Validation from action	193
On change event to hide/show an action programmatically	194
11.2 Business logic from list mode	197
List action with custom logic	198
Business logic in the model over several entities	200
11.3 Changing module	202
Using IChangeModuleAction	202
Detail only module	203
Returning to the calling module	204
Global session object and on-init action	205
11.4 JUnit tests	208
Testing the detail mode action	208
Finding an entity for testing using list mode and JPA	210
Testing hiding of the action	211

Testing the list mode action	212
Asserting test data	214
Testing exceptional cases	214
11.5 Summary	215
Chapter 12: References & collections	216
12.1 Refining reference behavior	217
Validation is good, but not enough	217
Modifying default tabular data helps	218
Refining action for searching reference with list	219
Searching the reference typing in fields	221
Refining action for searching reference typing key	222
12.2 Refining collection behavior	225
Modifying default tabular data helps	225
Refining the list for adding elements to a collection	226
Refining the action to add elements to a collection	228
12.3 JUnit tests	231
Adapting OrderTest	231
Testing the @SearchAction	232
Testing the @OnChangeSearch	233
Adapting InvoiceTest	234
Testing the @NewAction	235
Testing the action to add elements to the collection	238
12.4 Summary	240
Chapter 13: Security & navigation with Liferay	242
13.1 Introduction to Java portals	243
13.2 Installing Liferay	244
13.3 Deploying our application on Liferay	245
13.4 Navigation	247
Adding the pages for the application and its modules	247
Filling an empty page with a Portlet	249
Adding left navigation menu	249
Copying page	250
Two modules in the same page	251
Using CMS	252
13.5 User management	255
Roles	255
Users	256
13.6 Access levels	257

Limiting access to the entire application	257
Limiting access to individual modules	259
Limiting functionality	260
Limiting list data visibility	260
Limiting detail data visibility	262
13.7 JUnit tests	264
13.8 Summary	266
Appendix A: Source code	268
Appendix B: Learn more	316

Architecture & philosophy

chapter 1

OpenXava is a framework for rapid development of business applications using Java. It is easy to learn and one can have an application up in no time. At the same time, OpenXava is extensible, customizable and the application code is structured in a very pure object oriented way, allowing you to develop arbitrarily complex applications.

The OpenXava approach for rapid development differs from those that use visual environments (like Visual Basic or Delphi) or scripting (like PHP). Instead, OpenXava uses a model-driven development approach, where the core of your application are Java classes that model your problem. This means you can stay productive while still maintaining a high level of encapsulation.

This chapter will show you the concepts behind OpenXava as well as an overview of its architecture.

1.1 The OpenXava concepts

Though OpenXava takes a very pragmatic approach to development, it is based on the refinement of two well known ideas: The very popular methodology of Model-Driven Development (MDD) and the concept of Business Component. Ideas from MDD are borrowed in a rather lightweight way. The Business Component, however, is at the very heart of OpenXava.

Let's look at these concepts in closer detail.

1.1.1 Lightweight Model-Driven Development

Basically, MDD states that only the model of an application needs to be developed, and that the rest is automatically generated. This is illustrated in figure 1.1.

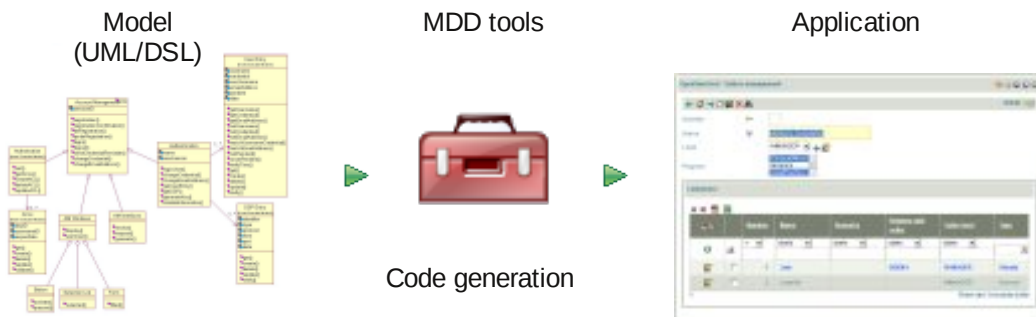


Figure 1.1 Model-Driven Development

In the context of MDD the model is the means of representing the data and the logic of the application. It can be either a graphical notation, such as UML, or a

3 Chapter 1: Architecture & philosophy

textual notation such as a Domain-Specific Language (DSL).

Unfortunately, using MDD is very complex. It requires a big investment of time, expertise, and tooling¹. Still the idea behind MDD is very good and hence OpenXava uses that idea in a simplified way. OpenXava uses plain annotated Java classes for defining the model, and instead of generating code, all functionalities are generated dynamically at runtime (table 1.1).

	Model definition	Application generation
Classic MDD	UML/DSL	Code generation
OpenXava	Simple Java classes	Dynamically at runtime

Table 1.1 MDD / OpenXava comparison

Figure 1.2 shows why we call OpenXava a *Lightweight Model-Driven Framework*.

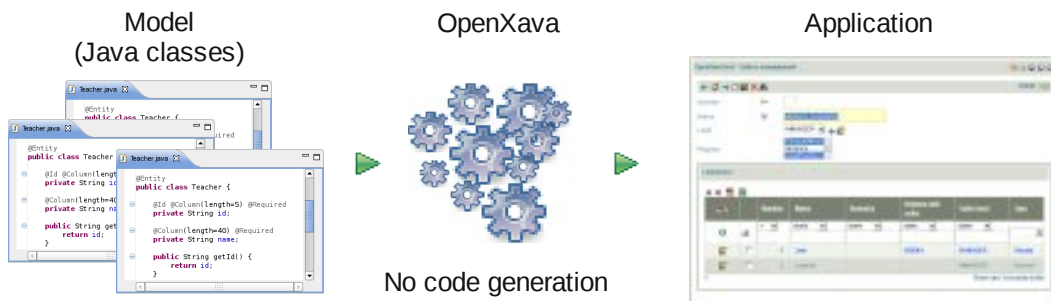


Figure 1.2 Lightweight Model-Driven Development in OpenXava

From just plain Java classes you obtain a full-fledged application. The next section about the Business Component concept will reveal some important details about the nature of these classes.

1.1.2 Business Component

A Business Component is a part of an application containing all the software artifacts related to some business concept (e.g., an invoice), it is merely a way of organizing software. The orthogonal way of developing software is the paradigm of MVC (Model-View-Controller) where the code is compartmentalized by data (Model), user interface (View), and logic (Controller).

¹ See the Better Software with Less Code white paper (<http://www.lulu.com/content/6544428>)

Figure 1.3 shows the organization of software artifacts in an MVC application. All code units associated with the creation of the user interface, like JSP pages, JSF, Swing, JavaFX, etc., are kept

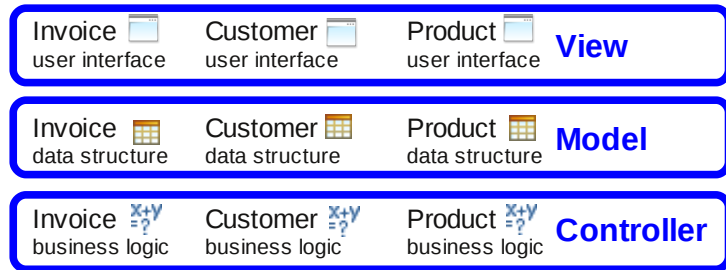


Figure 1.3 Model View Controller approach

closely together in the view layer, and likewise, for the model and controller layers. This contrasts with a business component architecture where the software artifacts are organized around business concepts. This is illustrated in figure 1.4.

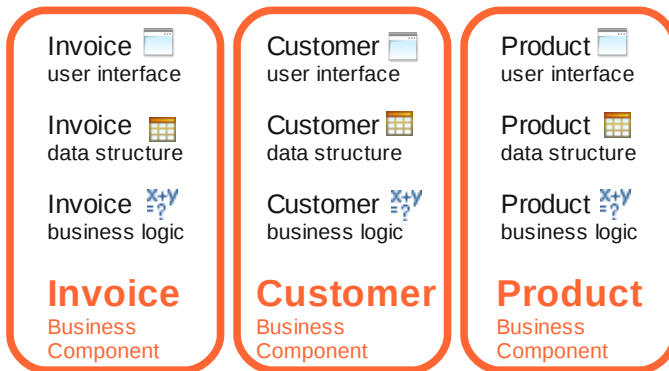


Figure 1.4 Business Component approach

Here, all software artifacts contributing to the Invoice concept, like user interface, database access, and business logic, are gathered in the same place.

Which paradigm to choose depends on your needs. If your data structures and business logic are likely to change frequently, then the

Business Component approach is very useful since all changes can be made in the same place instead of being scattered over multiple files.

In OpenXava the main part to develop is the Business Component, which is defined as a simple annotated Java class, exemplified in listing 1.1.

Listing 1.1 Invoice: A Java class for defining a business component

```
@Entity // Database
@Table(name="GSTFCT") // Database
@View(members= // User interface
    "year, number, date, paid;" +
    "customer, seller;" +
    "details;" +
    "amounts [ amountsSum, vatPercentage, vat ]"
)
public class Invoice {

    @Id // Database
    @Column(length=4) // Database
    @Max(9999) // Validation
    @Required // Validation
    @DefaultValueCalculator( // Declarative business logic
```

```

        CurrentYearCalculator.class
    )
    private int year; // Data structure (1)

    @ManyToOne(fetch=FetchType.LAZY) // Database
    @DescriptionsList // User interface
    private Seller seller; // Data structure

    public void applyDiscounts() { // Programmatic business logic (2)
        ...
    }

    ...
}

```

As you can see, everything to do with the concept of an Invoice is defined in a single place: the Invoice class. This class contains code dealing with persistence, data structures, business logic, user interface, validation, etc.

This is accomplished using the Java metadata facility, so-called annotations. Table 1.2 shows the annotations used in this example.

Facet	Metadata	Implemented by
Database	@Entity, @Table, @Id, @Column, @ManyToOne	JPA
User interface	@View, @DescriptionsList	OpenXava
Validation	@Max, @Required	Hibernate Validator, OpenXava
Business logic	@DefaultValueCalculator	OpenXava

Table 1.2 Metadata (annotations) used in Invoice business component

Thanks to metadata you can do most of the work in a declarative way and the dirty work is done for you by JPA, Hibernate Validator and OpenXava.

Moreover, the code you write is plain Java, like properties (year and seller, 1) for defining the data structure, and methods (applyDiscounts(), 2) for programmatic business logic.

All you need to write about invoice is *Invoice.java*. It is a Business Component. The magic of OpenXava is that it transforms this Business Component into a ready to use application.

1.2 Application architecture

You have seen how Business Components are the basic cells to construct in an OpenXava application. Indeed, a complete OpenXava application can be created using only Business Components. Nevertheless, there are plenty of additional ingredients available.

1.2.1 Application developer viewpoint

As stated above, a fully functional application can be built using only Business Components. Usually, however, it is necessary to add more functionalities in order to fit the behavior of the application to your needs. A complete OpenXava application has the shape of figure 1.5.

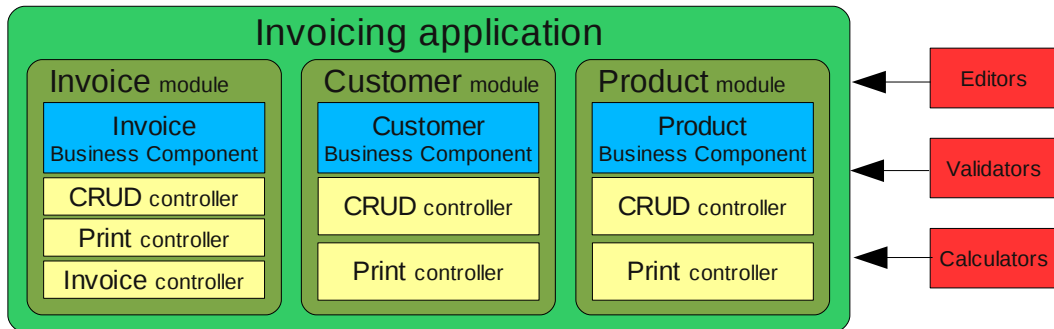


Figure 1.5 Shape of an OpenXava application

In figure 1.5, apart from Business Components, there are modules, controllers, editors, validators and calculators. Let's see what these things are:

- **Business components:** Java classes that describe all aspects of the business concepts. These classes are the only required pieces in an OpenXava application.
- **Modules:** A module is what the final user sees. It's the union of a Business Component and several controllers. You can omit the module definition, in which case a default module is used for each Business Component.
- **Controllers:** A controller is a collection of actions. From a user viewpoint, actions are buttons or links he can click; for the developer, they are the classes containing program logic to execute when those buttons are clicked. The controllers define the behavior of the application and can be reused in different modules of your application. OpenXava comes with a set of predefined controllers for many everyday tasks, and, of course, you can also define your own custom controllers.
- **Editors:** Editors are user interface components that specify how different members and attributes of the Business Component are displayed and edited. They provide a means for extending and customizing the user interface.
- **Validators:** Reusable validation logic that you can use in any Business Component.

- **Calculators:** Reusable business logic that can be used in various parts of Business Components, e.g., for generating default values.

1.2.2 User viewpoint

The standard way to access an OpenXava application is to point your browser to one of its modules, either by explicitly typing the URL or by navigating through a portal. Typically, the module will have a list mode for browsing through the objects (figure 1.6) and a detail mode for editing them (figure 1.7).

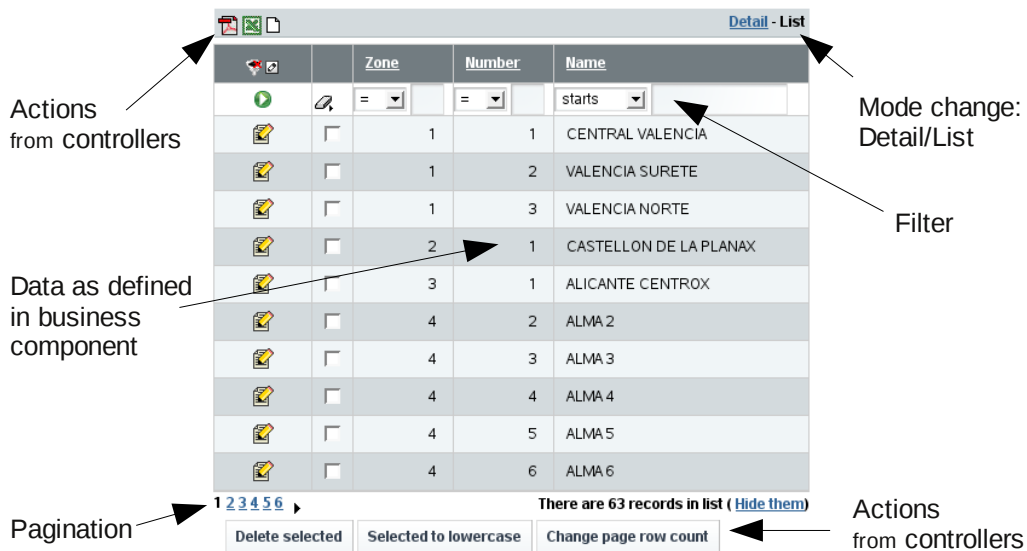


Figure 1.6 List mode of an OpenXava module

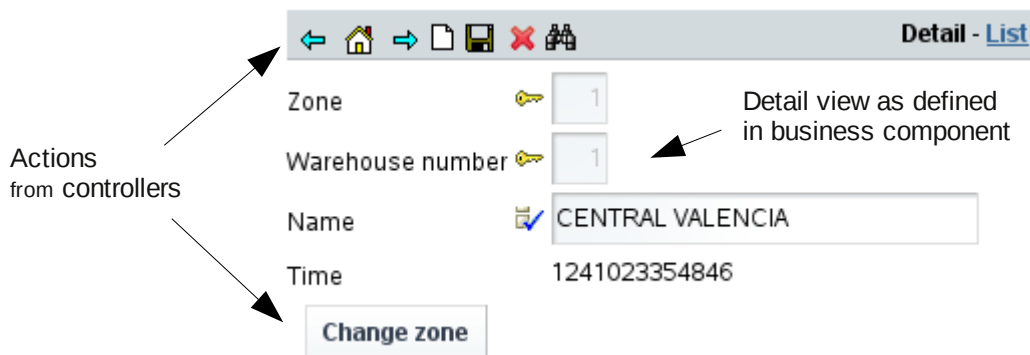


Figure 1.7 Detail mode of an OpenXava module

This shows you visually what a module really is: a functional piece of software generated from a Business Component (data and logic) together with its controllers (behavior).

1.2.3 Project layout

We have seen the concepts behind OpenXava as well as what it looks like to the end user. But what does OpenXava look like for the developer? Figure 1.8 shows the structure of a typical OpenXava project.

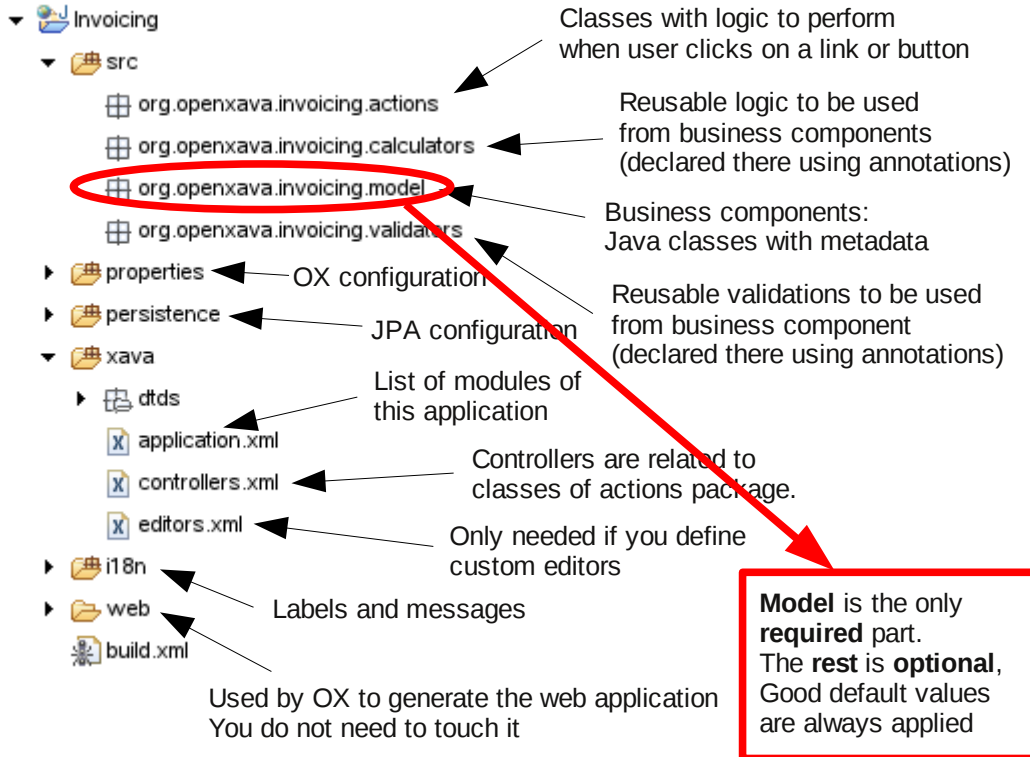


Figure 1.8 Structure of a typical OpenXava project

Only classes in the `model` package, the Business Components, are required. This is a bird's-eye view of an OpenXava project. You'll learn more details in the rest of the book.

1.3 Flexibility

As we have seen, OpenXava takes Java classes, annotated with metadata, and produces a complete application. This includes the automatic generation of the user interface. This might seem too automatic, and that it is quite likely that the resulting user interface will not be rich enough. Especially if your application has very specific requirements. This is not true. OpenXava annotations are flexible enough to generate very powerful user interfaces for most real world problems.

Nevertheless, OpenXava provides mechanisms to customize the user interface.

Below two of those are presented: Editors and Custom views.

1.3.1 Editors

Editors are the user interface elements used to view and edit the members of your business component. There are built-in editors for all the basic types that will satisfy the vast majority of cases, and OpenXava can readily be extended with your own editors when you need non standard ways of viewing and modifying elements.

Figure 1.9 shows how built-in editors are used for numbers and strings, but for the color property a custom editor is used. You can use JSP, JavaScript, HTML, AJAX or whatever web presentation technology you want, to create your custom editor, and then assign this editor to types or specific members.



This is a very reusable way to customize the user interface generation of your OpenXava application.

1.3.2 Custom view

Some Business Components might not render appropriately using the standard OpenXava view. Instead, a very particular interface may be needed like a map, photo gallery, or a calendar. In such cases, you can create your custom interface, using JavaScript, HTML, JSP, or similar, and then use it inside your OpenXava application. Figure 1.10 shows an OpenXava module that uses a custom view.

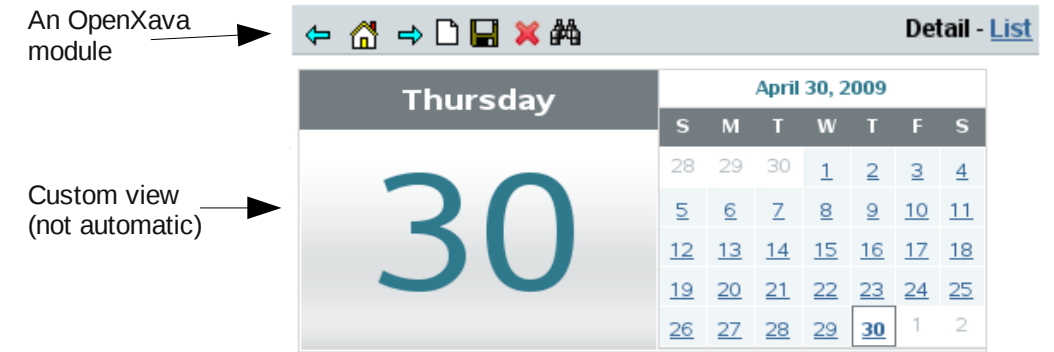


Figure 1.10 An OpenXava module that uses a custom view

In other words, OpenXava automatically generates a user interface for your

Business Components, but you always have the option of doing it yourself.

1.4 Summary

OpenXava uses a model-driven approach to rapid development, in which you produce a model and obtain a full application from it. The distinguishing feature of OpenXava is that the model consists of a Business Component.

The Business Component approach allows you to structure the application around business concepts. In OpenXava a plain annotated Java class defines the Business Component, making application development highly declarative.

Apart from business components an OpenXava application has modules, controllers, validators, calculators, etc. that you can optionally use to customize your application. You can even customize the way OpenXava generates the user interface using editors and custom views.

OpenXava is a pragmatic solution to Enterprise Java development. It generates a lot of automatic stuff, but it is flexible enough to be useful developing real life business applications.

At the end of the day you can develop applications just using simple Java classes with annotations. In the next two chapters, you'll learn more details about the annotations you can use with OpenXava.

Java
Persistence
API

chapter2

Java Persistence API (JPA) is the Java standard for object-relational mapping technique. Object-relational mapping allows you to access data in a relational database in an object-oriented fashion. In the Java applications you work only with objects. These objects are declared as persistent, and it's the JPA engine that is responsible for saving and reading objects from database to application.

JPA mitigates the so-called impedance mismatch problem caused due to the inherent difference between the structure of a relational database and that of the object oriented applications. A relational database structure consists of tables and columns with simple data whereas the object oriented applications have an absolutely different structure consisting of classes with references, collections, interfaces, inheritance, etc. In any Java application you use the Java classes for representing the business concept. You need to write a lot of SQL code to write the data from your objects to database and vice versa. JPA does it for you.

This chapter is an introduction to JPA. For a complete understanding of this standard technology, you would need a book dedicated to JPA. In fact, I have cited some of the books in the summary section of this chapter.

Please skip this chapter, if you already know JPA.

2.1 JPA Annotations

JPA has 2 aspects. The first one is a set of Java annotations to add to your classes. This is used to mark them as persistent and further give details about mapping the classes to the tables. The second aspect is that of an API to read and write objects from your application. Let's look at the annotations first.

2.1.1 Entity

In JPA nomenclature a persistent class is called an entity. In other words we can say that an entity is a class whose instances are saved in the database. Usually each entity represents a business concept of the domain. Therefore we use a JPA entity as the basis for defining a business component in OpenXava. In fact you can create a full OX application just from bare JPA entities. Listing 2.1 shows the way a JPA entity is defined.

Listing 2.1 Annotations for JPA entity definition

```
@Entity // To define this class as persistent
@Table(name="GSTCST") // To indicate the database table (optional)
public class Customer {
```

As you can see, you only have to mark your class with the `@Entity` annotation and optionally also with the `@Table` annotation. In this case we say that the

13 Chapter 2: Java Persistence API

Customer entity is saved in the table GSTCST of the database. From now on, JPA will store and retrieve the data between Customer objects in the application and the GSTCST table in the database, as shown in figure 2.1.

In OpenXava it is enough to mark Customer with `@Entity` annotation to recognize the object as a business component. In fact in OpenXava “entity” is synonymous to business component.

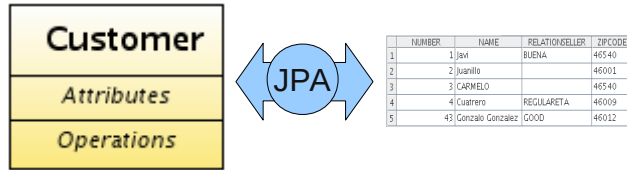


Figure 2.1 JPA maps classes to tables

2.1.2 Properties

The basic state of an entity is represented using properties. Entity properties are plain Java properties, with their respective getter and setter² methods (listing 2.2).

Listing 2.2 Entity property definition

```
private String name;

public String getName() {
    return name;
}

public void setName(String name) {
    this.name = name;
}
```

By default properties are persistent that is, JPA assumes that the property name (listing 2.2) is stored in a column named 'name' in the database table. If you do not want a property to be saved in the database you have to mark it as `@Transient` (listing 2.3). Note that we can not only annotate the field, but we can also annotate the getter if we want (listing 2.4).

This rule applies to all JPA annotations. You can annotate the field (field-base access) or the getter (property-base access). Do not mix the two styles in the same entity.

Listing 2.3 Transient property

```
@Transient // Marked as transient, it is not stored in the database
private String name;

public String getName() {
    return name;
}
```

2 To learn more options about properties and fields look at section 2.1.1 of JPA 1.0 Specification

```
public void setName(String name) {
    this.name = name;
}
```

Listing 2.4 Transient property using property-based access

```
private String name;

@Transient // We mark the getter, so all JPA annotations in this entity must be in getters
public String getName() {
    return name;
}

public void setName(String name) {
    this.name = name;
}
```

Other useful annotations for properties are `@Column` to specify the name and length of the table column and `@Id` to indicate which property is the key property. You can see the use of these annotations in an already mentioned `Customer` entity in listing 2.5.

Listing 2.5 The Customer entity with `@Id` and `@Column` annotations

```
@Entity
@Table(name="GSTCST")
public class Customer {

    @Id // Indicates that number is the key property (1)
    @Column(length=5) // Here @Column indicates only the length (2)
    private int number;

    @Column(name="CSTNAM", length=40) // name property is mapped to CSTNAM
    private String name; // column in database

    public int getNumber() {
        return number;
    }

    public void setNumber(int number) {
        this.number = number;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

It's mandatory that at least one property should be an id property (shown as 1). You have to mark the id property with `@Id` and usually it maps to the table key column. `@Column` can be used to indicate just the length without the column name

(shown as 2). The length is used by the JPA engine for schema generation. It's also used by OpenXava to know the size of the user interface editor. From the Customer entity of listing 2.5 OpenXava generates the user interface you can see in figure 2.2.

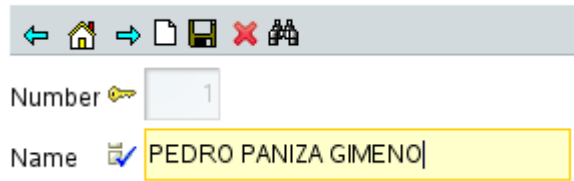


Figure 2.2 User interface for Customer entity with 2 properties

Now, you know how to define basic properties in your entity. Let's learn how to declare relationships between entities using references and collections.

2.1.3 References

An entity can reference another entity. You only have to define a regular Java reference annotated with the `@ManyToOne` JPA annotation, just as in listing 2.6.

Listing 2.6 A reference to another entity using `@ManyToOne`

```
@Entity
public class Invoice {

    @ManyToOne( // The reference is persisted as a database relationship (1)
        fetch=FetchType.LAZY, // The reference is loaded on demand (2)
        optional=false) // The reference must always have a value
    @JoinColumn(name="INVCST") // INVCST is the foreign key column (3)
    private Customer customer; // A regular Java reference (4)

    // Getter and setter for customer
```

As you can see in listing 2.6 we declare a reference to `Customer` inside `Invoice` in a plain Java style (shown as 4). The annotation `@ManyToOne` (shown as 1) is to indicate that this reference is stored as a many-to-one database relationship between the table for `Invoice` and the table for `Customer`, using the `INVCST` (shown as 3) column as foreign key. The annotation `@JoinColumn` (shown as 3) is optional. JPA assumes default values for join column (`CUSTOMER_NUMBER` in this case).

If you use `fetch=FetchType.LAZY` (shown as 2) the customer data is not loaded until you use it once. That is at the given moment that you use the customer reference, for example if you call the method `invoice.getCustomer().getName()` the data for the customer is loaded from the database at this point in time. It's advisable to always use lazy fetching.



**Figure 2.3 A reference to other entity:
A *..1 association in UML**

A regular Java reference usually corresponds to a ManyToOne relationship in JPA and to an association with *..1 multiplicity in UML notations (figure 2.3).

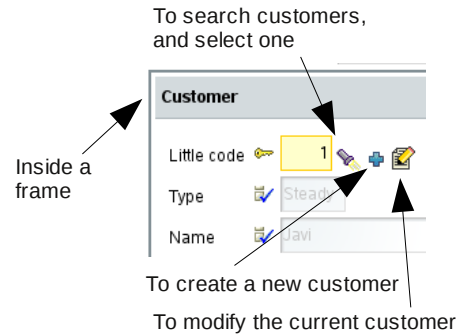


Figure 2.4 User interface for a reference

Figure 2.4 shows the user interface that OpenXava generates automatically for a reference. You have seen how to reference other entities. You can also reference other objects that are not entities, and even the embeddable objects.

2.1.4 Embeddable classes

In addition to entities you can also use embeddable classes to model some concepts of your domain. If you have an entity A that has a reference to B, you would model B as an embeddable class when:

- You can say that A has B.
- A is deleted then B is also deleted.
- B is not shared.

Sometimes the same concept can be modeled as embeddable or as an entity. For example, the concept of an address. If the address is shared by several persons then you must use a reference to an entity, while if each person has his own address then an embeddable object is a good option.

Let's model an address as an embeddable class. It's easy. Create a plain Java class and annotate it as `@Embeddable`, just as in listing 2.7.

Listing 2.7 Embeddable class definition

```

@Embeddable // To define this class as embeddable
public class Address {

    @Column(length=30) // You can use @Column as in an entity
    private String street;

    @Column(length=5)
    private int zipCode;
  
```

```

@Column(length=20)
private String city;

// Getters and setters
...
}

```

And now, creating a reference to Address from an entity is also easy. It's just a regular Java reference annotated as `@Embedded` (listing 2.8).

Listing 2.8 A reference to an embeddable class from an entity

```

@Entity
@Table(name="GSTCST")
public class Customer {

    @Embedded // References to an embeddable class
    private Address address; // A regular Java reference

    // Getter and setter for address
    ...
}

```

From a persistence viewpoint an embeddable object is stored in the same table as the container entity. In this case the street, zipcode and city columns are in the table for Customer. Address has no table for itself.

Figure 2.5 Embedded reference user interface

Figure 2.5 shows the user interface that OpenXava generates automatically for a reference to an embeddable class.

2.1.5 Collections

An entity can have a collection of entities. You only have to define a regular Java collection annotated as `@OneToMany` or `@ManyToMany` (listing 2.9).

Listing 2.9 A collection of entities using `@OneToMany`

```

@Entity
public class Customer {

    @OneToMany( // The collection is persistent (1)
        mappedBy="customer") // The customer reference from Invoice is used
                                // to map the relationship at the database level (2)
    private Collection<Invoice> invoices; // A regular Java collection (3)

    // Getter and setter for invoices
    ...
}

```

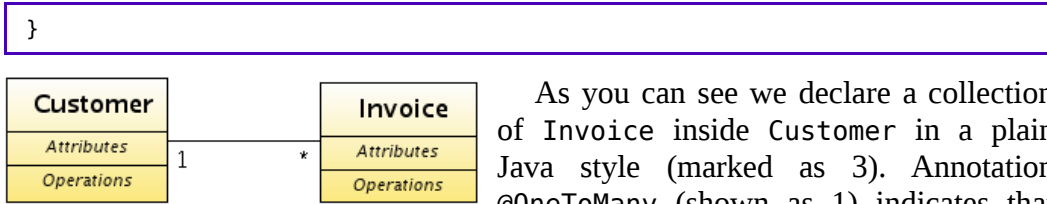


Figure 2.6 A collection of entities: A 1..* association in UML

As you can see we declare a collection of Invoice inside Customer in a plain Java style (marked as 3). Annotation `@OneToMany` (shown as 1) indicates that this collection is stored as a one-to-many

database relationship between the table for Customer and the table for Invoice, using the column called customer for mapping the reference in Invoice (usually a foreign key to Customer table from Invoice table).

A regular Java collection normally corresponds to a `OneToMany` or `ManyToMany` relationship in JPA and to an association with `1..*` or `*..*` multiplicity in UML notations (figure 2.6).

It's not possible to define a collection of embeddable objects in JPA (at least for v1.0), but it's easy to simulate the semantic of embeddable objects using a collection of entities with the `cascade` attribute of `@OneToMany`, as shown in listing 2.10.

Listing 2.10 A collection to entities with cascade REMOVE to simulate embedded

```

@Entity
public class Invoice {

    @OneToMany (mappedBy="invoice",
               cascade=CascadeType.REMOVE) // Cascade REMOVE to simulate embedded
    private Collection<InvoiceDetail> details;

    // Getter and setter for details
    ...
}

```

Thus, when an Invoice is removed its details are removed too. We can say that an invoice *has* details.

You can see the default user interface for a collection of entities in figure 2.7.

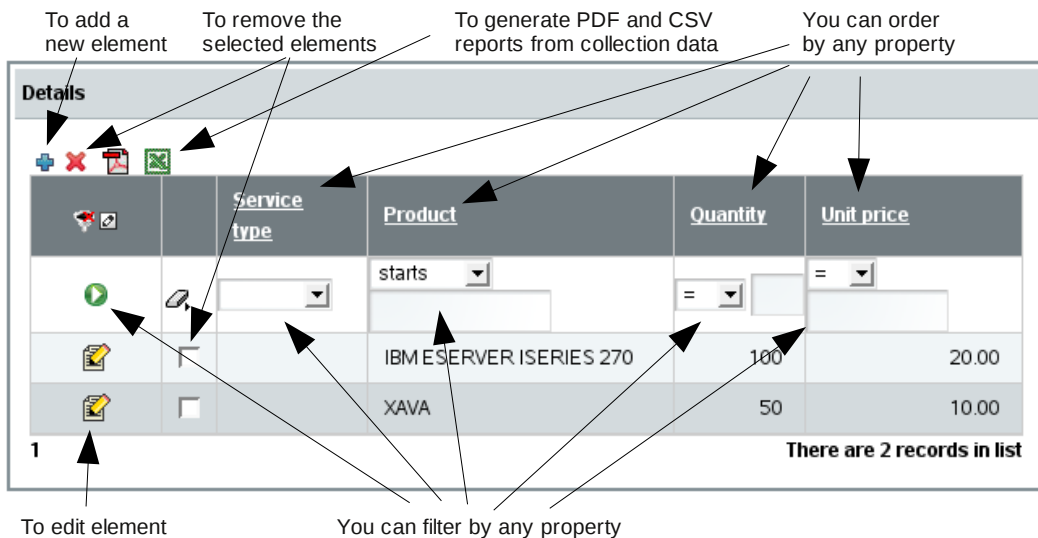


Figure 2.7 User interface for a collection of entities

There is a slight difference in the behavior of the user interface if you use cascade REMOVE or ALL. With cascade REMOVE or ALL when the user clicks to add a new element, he can actually enter all the data for the element. On the other hand if your collection is not cascade REMOVE or ALL, when the user clicks to add a new element a list of entities is shown for the user to choose.

You have seen how to write your entities using JPA annotations and how OpenXava interprets them in order to generate a suitable user interface. Now you are going to learn how to use the JPA API to write and read from the database programmatically.

2.2 JPA API

The key class to manage your entities with JPA is `javax.persistence.EntityManager`. An `EntityManager` allows you to save, modify and search entities programmatically.

Listing 2.11 shows the typical way of using JPA from a non-OpenXava application.

Listing 2.11 The typical way to use JPA API

```
EntityManagerFactory f = // You need an EntityManagerFactory to create a manager
    Persistence.createEntityManagerFactory("default");
```

```

EntityManager manager = f.createEntityManager(); // You create the manager
manager.getTransaction().begin(); // You have to start a transaction
Customer customer = new Customer(); // Now you create your entity
customer.setNumber(1); // and populate it
customer.setName("JAVI");
manager.persist(customer); // persist marks the object as persistent
manager.getTransaction().commit(); // On commit changes are done into database
manager.close(); // You have to close the manager

```

You can see how verbose this snippet of code is. Too much of bureaucratic code. If you wish you can write JPA code in this way inside an OpenXava application. On the other hand OpenXava offers you a much more succinct way to do it (listing 2.12).

Listing 2.12 JPA code to save an entity inside an OpenXava application

```

Customer customer = new Customer();
customer.setNumber(1);
customer.setName("PEDRO");
XPersistence.getManager().persist(customer); // This is enough (1)

```

Inside an OpenXava application you can obtain the manager by means of `org.openxava.jpa.XPersistent` class. You don't need to close the manager, start and commit a transaction. This dirty work is taken care of by OpenXava for you. The code you see in listing 2.12 is enough to save a new entity to the database (shown as 1).

If you want to modify an existing entity you have to do so as shown in listing 2.13.

Listing 2.13 Modifying an object using JPA

```

Customer customer = XPersistence.getManager()
    .find(Customer.class, 1); // First, you search the object to modify (1)
customer.setName("PEDRITO"); // Then, you change the object state. No more

```

In order to modify an object you only need to find and modify it. JPA is responsible for saving the changes to the database on transaction commit (sometime sooner) and OpenXava commits the JPA transaction automatically.

In listing 2.13 you have seen how to search by primary key using `find()`. But JPA allows you to use queries as shown in the listing 2.14.

Listing 2.14 Using queries to search entities

```

Customer pedro = (Customer) XPersistence.getManager()
    .createQuery(
        "from Customer c where c.name = 'PEDRO'") // JPQL query (1)
    .getSingleResult(); // To obtain one single entity (2)

List pedros = XPersistence.getManager()
    .createQuery(
        "from Customer c where c.name like 'PEDRO%') // JPQL query

```

```
.getResultList(); // To obtain a collection of entities (3)
```

You can use the Java Persistence Query Language (JPQL, shown as 1) to create complex queries on your database and to obtain a single entity object by means of `getSingleResult()` method (shown as 2), or a collection of entities by means of `getResultList()` method (shown as 3).

2.3 Summary

This chapter has been a brief introduction to the JPA technology. Unfortunately, many interesting things about JPA remains unsaid such as inheritance, polymorphism, composite keys, OneToOne relationships, ManyToMany relationship, unidirectional relationships, callback methods, advanced queries etc. In fact, there are more than 60 annotations in JPA 1.0. We would need lot of time to learn all the details of JPA.

Fortunately you will have the chance to learn some advanced JPA use cases while going through this book. If you still want to learn more just read some other books and references. For example:

- *Java Persistence API Specification* by Linda DeMichiel, Michael Keith and others (available at jcp.org as part of JSR-220).
- *Pro EJB 3: Java Persistence API* by Mike Keith and Merrick Schincariol.
- *Java Persistence with Hibernate* by Christian Bauer and Gavin King.

JPA is an undisputed technology in the Java Enterprise universe. All the assets of knowledge and code around JPA is always a good investment.

Apart from the standard JPA annotations you have seen in this chapter, there are other useful annotations you can use in your entities. Let's see them in the next chapter.

Annotations

chapter3

23 Chapter 3: Annotations

Annotations are the tool that Java provides to define metadata in your applications. In other words, it is the way to do declarative development in Java, where you refer to “what” and not the “how” part.

In chapter 2 you have seen how to use the JPA annotations to do the object-relational mapping. In this chapter you will see the annotations you can use inside an OpenXava application to define validations, user interface and some other aspects to fit your application to your needs.

The goal of this chapter is to introduce you to these annotations. On the other hand this chapter does not show you all the intricacies and use cases of all the annotations. The comprehensive coverage of all the annotations is out of the scope of this book.

3.1 Validation

OpenXava includes an easy to use and extensible validation framework. Moreover OpenXava supports Hibernate Validator.

3.1.1 Declarative validation

The preferred way to do validation in OpenXava is by means of annotations, i.e. in a declarative way. For example, you only have to mark a property as `@Required` (listing 3.1).

Listing 3.1 A declarative validation: Marking a property as required

```
@Required // This forces to validate this property as required on save
private String name;
```

And OpenXava will do the appropriate validation on save (figure 3.1).

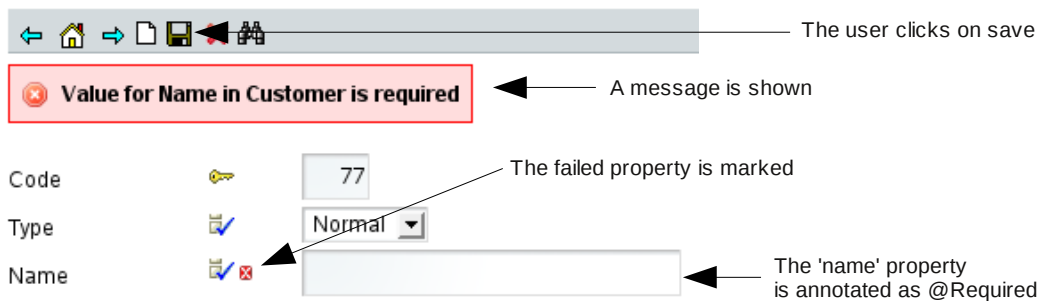


Figure 3.1 Visual effect of a validation

3.1.2 Built-in validations

The validation annotations that OpenXava provides out of the box (table 3.1) are defined using the Hibernate Validator (except for `@RemoveValidator`).

Hibernate Validator is a very popular validation framework (it has nothing to do with the Hibernate persistence framework).

Annotation	Apply on	Checking
@Required	Property	Checks if the property has a value
@PropertyValidator	Property	Allows you to define custom validation logic
@EntityValidator	Entity	Allows you to define a custom validation logic
@RemoveValidator	Entity	Allows you to define a custom validation logic when removing

Table 3.1 Built-in OpenXava validations

The Hibernate Validator annotations are recognized by OpenXava, so you can use all the built-in Hibernate Validator annotations in your OpenXava applications (table 3.2).

Annotation	Apply on	Checking
@Length(min=, max=)	Property (String)	Check if the string length matches the range
@Max(value=)	Property (numeric or string representation of a numeric)	Check if the value is less than or equal to max
@Min(value=)	Property (numeric or string representation of a numeric)	Check if the value is greater than or equals to min
@NotNull	Property	Check if the value is not null
@NotEmpty	Property	Check if the string is not null nor empty
@Past	Property (date or calendar)	Check if the date is in the past
@Future	Property (date or calendar)	Check if the date is in the future
@Pattern(regex="reg exp", flag=)*	Property (string)	Check if the property match the regular expression given a match flag

Table 3.2 Built-in Hibernate validations. They can be used from OpenXava

Annotation	Apply on	Checking
@Range(min=, max=)	Property (numeric or string representation of a numeric)	Check if the value is between min and max (included)
@Size(min=, max=)	Property (array, collection, map)	Check if the element size is between min and max (included)
@AssertFalse	Property	Check that the method evaluates to false (useful for constraints expressed in code rather than annotations)
@AssertTrue	Property	Check that the method evaluates to true (useful for constraints expressed in code rather than annotations)
@Valid	Property (object)	Perform validation recursively on the associated object. If the object is a Collection or an array, the elements are validated recursively. If the object is a Map, the value elements are validated recursively
@Email	Property (String)	Check whether the string conforms to the email address specification
@CreditCardNumber	Property (String)	Check whether the string is a well formatted credit card number (derivative of the Luhn algorithm)
@Digits	Property (numeric or string representation of a numeric)	Check whether the property is a number having up to integerDigits integer digits and fractionalDigits fractional digits define column precision and scale
@EAN	Property (string)	Check whether the string is a properly formatted EAN or UPC-A code

Table 3.2(cont.) Built-in Hibernate validations. They can be used from OpenXava

3.1.3 Custom validation

It is very easy to add your own validation logic to your entity because the @PropertyValidator, @EntityValidator and @RemoveValidator annotations allows you to indicate a class (the validator) with the validation logic.

For example, if you want a custom logic to validate an unitPrice property, you have to write something like the code in listing 3.2.

Listing 3.2 Defining a custom validation with @PropertyValidator

```
@PropertyValidator(UnitPriceValidator.class) // Contains the validation logic
private BigDecimal unitPrice;
```

And now you can write the logic you want inside `UnitPriceValidator` class, as you see in listing 3.3.

Listing 3.3 Custom validation logic in a validator class

```
public class UnitPriceValidator
    implements IPropertyValidator { // Must implement IPropertyValidator (1)

    public void validate( // Required because of IPropertyValidator (2)
        Messages errors, // Here you add the error messages (3)
        Object object, // The value to validate
        String objectName, // The entity name, usually to use in message
        String propertyName) // The property name, usually to use in message
    {
        if (object == null) return;
        if (!(object instanceof BigDecimal)) {
            errors.add( // If you add an error the validation fails
                "expected_type", // Message id in i18n file
                propertyName, // Arguments for i18n message
                objectName,
                "bigdecimal");
            return;
        }
        BigDecimal n = (BigDecimal) object;
        if (n.intValue() > 1000) {
            errors.add("not_greater_1000"); // Message id in i18n file
        }
    }
}
```

As you can see your validator class must implement `IPropertyValidator` (shown as 1), this forces you to have a `validate()` (shown as 2) method that receives a `Messages` object, that we call `errors` (shown as 3). It is just a container of error messages. You only need to add a message to `errors` in order to provoke the validation to fail.

This is a simple way to do custom validation, moreover the validation logic in your validator can be reused across your application. Although, if you want to create a reusable validation a better option is to create your own validation annotation using `Hibernate Validator`. It's more verbose than using a validator class but it's more elegant if you reuse the validation many times. We'll create a custom `Hibernate Validator` in chapter 9.

3.1.4 Learn more about validations

This section is only a brief introduction to validation in `OpenXava`. You can learn more details about this topic in chapter 3 (Model) of the *OpenXava Reference Guide* and in the *Reference Documentation of Hibernate Validator*.

Additionally, you'll learn more advanced cases of validation in chapter 9 of this book.

3.2 User interface

Though OpenXava automatically generates a crude but valid user interface from a naked JPA entity, this is only useful for very basic cases. In real life applications it is required to refine the ways the user interface is generated. In OpenXava this is done by annotations that, in a very abstract way, defines the user interface shape.

3.2.1 The default user interface

By default, OpenXava generates a user interface that shows all members of the entity sequentially. If you have a Seller entity as in listing 3.4

Listing 3.4 A naked entity, without view annotations

```
@Entity
public class Seller {

    @Id @Column(length=3)
    private int number;

    @Column(length=40) @Required
    private String name;

    @OneToMany(mappedBy="seller")
    private Collection<Customer> customers;

    // Getters and setters
    ...
}
```

OpenXava will produce for you the user interface as shown in figure 3.2. As you see, it shows the members (number, name and customers in this case) in the same order as they are declared in the Java source. OpenXava uses the JPA and

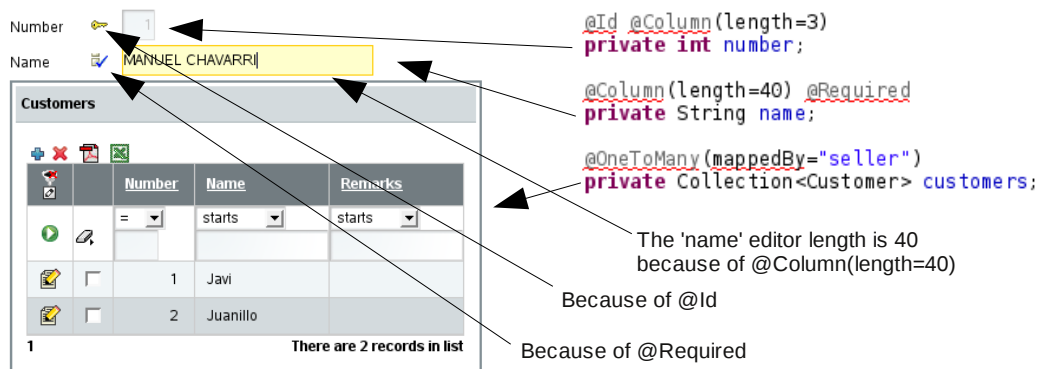


Figure 3.2 Default user interface from an entity

validation annotations to generate a better user interface for example it determines the size of the editors from `@Column(length)`, shows a key icon for

the @Id property and shows an icon to indicate that it is required if the property is marked as @Required, and so on.

This default interface is useful for simple cases, but for a more advanced user interface you need a way to customize it. OpenXava provides you with annotations to do so, such as @View for customizing the layout of the members.

3.2.2 The @View annotation

@View annotation is used to define the layout of the members in the user interface. It is defined at the entity level. Look at the example in listing 3.5 and the resulting user interface in figure 3.3.

Listing 3.5 @View annotation for defining user interface layout

```
@Entity
@View(members=
    "year, number, date, paid;" + // Comma means in the same line
    "discounts [" + // Between square brackets means inside a frame
    "    customerDiscount, yearDiscount;" +
    "]" +
    "comment;" + // Semicolon means new line
    "customer { customer }" + // Between curly brackets means inside a tab
    "details { details }" +
    "amounts { amountsSum; vatPercentage; vat }" +
    "deliveries { deliveries }"
)
public class Invoice {
```

As you can see, defining the layout of the member is easy. You only need to enumerate them inside a string using commas to separate elements, semicolons for new line, square brackets for groups (frames), curly brackets for sections (tabs) and so on.

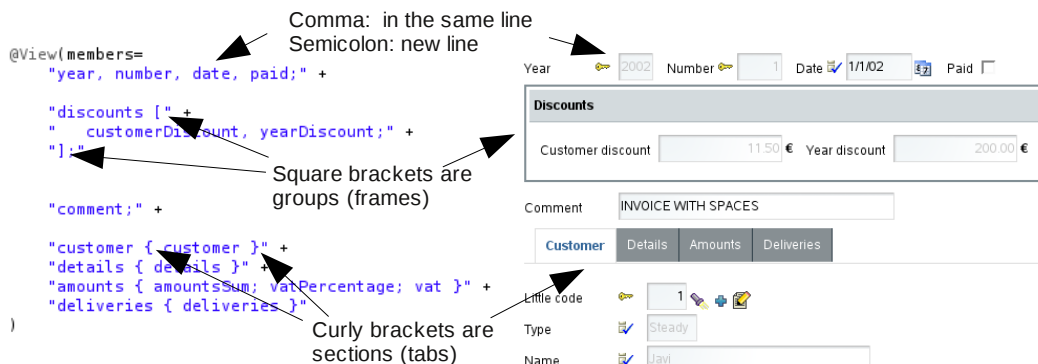


Figure 3.3 @View annotation for defining user interface layout

You may have several views for each entity. For that purpose the @Views

annotation gives a name to each view (listing 3.6).

You can leave a view without a name which will result in a default view. The view names are used from other parts of the application to choose which view to use.

Listing 3.6 Several views for an entity

```
@Entity
@Views({
    @View(
        members="number, name; address; invoices"
    ),
    @View(
        name="Simple", members="number, name"
    )
})
public class Customer {
```

Though with a `@View` annotation you can define the layout but you also need to define the way each member is displayed. OpenXava takes care to provide you a lot of useful annotations which you will see in the next section.

3.2.3 Refining member presentation

OpenXava allows you to refine the user interface for any property, reference or collection in nearly infinite ways. You only need to add the corresponding annotation. For example, by default a reference (a `@ManyToOne` relationship) is displayed using a frame with a detailed view. If you want to show that reference using a combo you only have to annotate the reference with `@DescriptionsList` (figure 3.4).

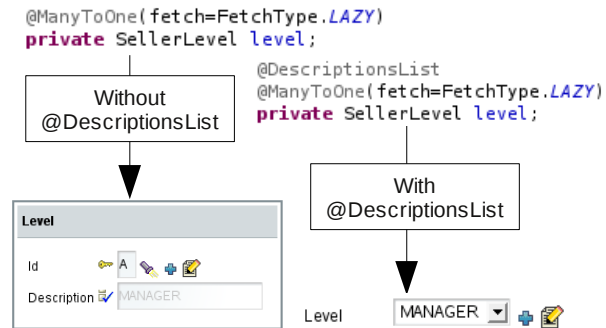


Figure 3.4 Effect of `@DescriptionsList` in a `@ManyToOne` relationship

If you want the effect of the annotation only for some views you can do so by using the attribute `forViews` available in all user interface annotations (listing 3.7).

Listing 3.7 Using `forViews` to narrow the effect of an annotation to some views

```
@Views ({ // You have several views for Seller
    @View(members=" ... "),
    @View(name="Simplest", members=" ... "),
    @View(name="Simple", members=" ... "),
    @View(name="Complete", members=" ... ")
```

```

})
public class Seller {

    @DescriptionsList(forViews="Simplest, Simple") // Combo only will be used for
    @ManyToOne(fetch=FetchType.LAZY)           // 'level' in Simplest and Simple views
    private SellerLevel level;

```

Table 3.3 shows all OpenXava annotations for customizing the user interface of entity members.

Annotation	Description	Apply on
@Action	Associates an action to a property or reference in the view	Properties and references
@AsEmbedded	Makes the behavior of a view for a reference (or collection) to an entity be the same as an embedded object (or collection of entities with CascadeType.REMOVE)	References and collections
@CollectionView	The view of the referenced object (each collection element) which is used to display the detail	Collections
@Condition	Restricts the elements that appear in the collection	Collections
@DescriptionsList	To visualize a reference as a descriptions list (actually a combo)	References
@DetailAction	Adds an action to the detail that is being edited in a collection	Collections
@DisplaySize	The size in characters of the editor in the User Interface used to display this property	Properties
@EditAction	Allows you to define your custom action to edit a collection element	Collections
@EditOnly	The final user can modify existing elements in the collection, but not add or remove collection elements	Collections
@Editor	Name of the editor to use for displaying the member in this view	Properties, references and collections.
@HideDetailAction	In a collection it allows you to define your custom action to hide the detail view	Collections
@LabelFormat	Format to display the label of this property or reference (displayed as descriptions list)	Properties and references

Table 3.3 Annotations for customizing the user interface of entity members

31 Chapter 3: Annotations

Annotation	Description	Apply on
@ListAction	To add actions to the list in a collection	Collections
@ListProperties	Properties to show in the list for visualization of a collection	Collections
@NewAction	Allows you to define your custom action to start adding a new element to a collection	Collections
@NoCreate	The final user cannot create new objects of the referenced type from here	References and collections
@NoFrame	The reference is not displayed inside a frame	References
@NoModify	The final user cannot modify the current referenced object from here	References and collections
@NoSearch	The user will not have a link to make searches with a list, filters, etc.	References
@OnChange	Action to execute when the value of this property/reference changes	Properties and references
@OnChangeSearch	Action to execute to do the search of a reference when the user types the key value	References
@OnSelectElementAction	Allows you to define an action to be executed when an element of the collection is selected or unselected	Collections
@ReadOnly	The member will never be editable by the final user in the indicated views	Properties, references and collection
@ReferenceView	View of the referenced object used to display it in a reference	References
@RemoveAction	Allows you to define a custom action to remove the element from the collection	Collections
@RemoveSelectedAction	Allows you to define your custom action to remove the selected elements from the collection	Collections
@RowStyle	For indicating the row style for list and collections	Entity (by means of @Tab) and collections
@SaveAction	Allows you to define a custom action to save the collection element	Collections

Table 3.3(cont.) Annotations for customizing the user interface of entity members

Annotation	Description	Apply on
@SearchAction	Allows you to specify your own action for searching	References
@ViewAction	Allows you to define your custom action to view a collection element	Collections
@XOrderBy	The eXtended version of @OrderBy (JPA)	Collections

Table 3.3(cont.) Annotations for customizing the user interface of entity members

You may think that these are a lot of annotations covered so far in table 3.3. But to your surprise there are more yet because most of these annotations have a plural version that allow you to declare different values for different views (listing 3.8).

Listing 3.8 Plural annotations allows to define several times the same annotation

```

@DisplaySizes({ // To use several @DisplaySize
    @DisplaySize(forViews="Simple", value=20), // name has 20 as display
                                                    // size in view Simple
    @DisplaySize(forViews="Complete", value=40) // name has 40 as display
                                                    // size in view Complete
})
private String name;

```

Don't worry if you don't know how to use all these annotations yet. You'll learn them as you develop OpenXava applications.

3.2.4 Learn more about the user interface

This section introduces you briefly to the user interface with OpenXava. Unfortunately, many interesting concepts still remain untouched such as nested groups and sections, actions in views, table layout for groups and sections, view inheritance, details about how to use all UI annotations, etc.

Through this book you'll learn advanced topics about user interface. Additionally, you can learn more details on this topic in chapter 4 (View) of *OpenXava Reference Guide* and in *OpenXava API Doc* of org.openxava.annotations.

3.3 Other annotations

Apart from validation and user interface OpenXava has some other useful annotations. You can see them in table 3.4.

We'll use most of these annotations to develop the examples of this book. Also you can find exhaustive explanations of them in the *OpenXava Reference Guide*

and the *OpenXava API Doc* of *org.openxava.annotations*.

Annotation	Description	Apply on
@DefaultValueCalculator	For calculating the initial value	Properties and references
@Hidden	A hidden property has a meaning for the developer but not for the user	Properties
@Depends	Declares that a property depends on other one(s)	Properties
@Stereotype	A stereotype is the way to determine a specific behavior of a type	Properties
@Tab	Defines the behavior for tabular data presentation (list mode)	Entities
@SearchKey	A search key property or reference is used by the user to search	Properties and references

Table 3.4 Miscellaneous annotations

3.4 Summary

You have seen how to use Java annotations to do declarative programming with OpenXava. Things such as the user interface or validation that are typically programming stuff, can be done just by annotating our code.

You might be overwhelmed by so many annotations. Don't worry. The best way to learn is by example. The rest of this book is illustrative enough with pure examples that will show you how to use these annotations.

Let's start learning.

This LITE edition of the book only includes the first three chapters

Get the full book here:

<http://www.openxava.org/buy-book>

Use the coupon LITE310IN9148 to get a 10% discount